OpenCV 中文参考手册

资料来源:OpenCV中文站 www.opencv.org.cn

制作时间:2010/10/31

说明

资料来源:OpenCV中文站 www.opencv.org.cn

本文最初版本(0.9.6/beta4)由如下人员翻译：Y.Q.Zhang, J.H.Tan, X.C.Qin, M.Liu 和 Z.T.Fan。全文由 Hunnish 做统一修改校正。

以下人员又对中文版提供了修改

* 于仕琪，中科院自动化所自由软件协会
* 张兆翔，中科院自动化所自由软件协会

友情支持: www.historycreator.com

2010/10/31 夜

OpenCV参考手册

- OpenCV 编程简介（矩阵/图像/视频的基本读写操作）入门必读
- OpenCV概述
- FAQ中文
- CxCore中文参考手册
    1. 基础结构
    2. 数组操作
    3. 动态结构
    4. 绘图函数
    5. 数据保存和运行时类型信息
    6. 其它混合函数
    7. 错误处理和系统函数
- 机器学习中文参考手册
- CvAux中文参考手册
- CvvImage类参考手册
- CvImage类参考手册
- CvImage中的陷阱和BUG
- Cv中文参考手册
    1. 图像处理
    2. 结构分析
    3. 运动分析与对象跟踪
    4. 模式识别
    5. 照相机定标和三维重建
- HighGUI中文参考手册
    1. HighGUI概述
    2. 简单图形界面
    3. 读取与保存图像
    4. 视频读写函数
    5. 实用函数与系统函数
- OpenCV 编码样式指南（阅读OpenCV代码前必读）
- OpenCV 的Python接口

OpenCV 编程简介（矩阵/图像/视频的基本读写操作）

Wikipedia，自由的百科全书

Introduction to programming with OpenCV

OpenCV编程简介

作者： **Gady Agam**

- Department of Computer Science
- January 27, 2006
- Illinois Institute of Technology

- URL: http://www.cs.iit.edu/~agam/cs512/lect-notes/opencv-intro/opencv-intro.html#SECTION00040000000000000000

翻译： **chenyusiyuan**

- January 26, 2010
- http://blog.csdn.net/chenyusiyuan/archive/2010/01/26/5259060.aspx

---

摘要： 本文旨在帮助读者快速入门OpenCV，而无需阅读冗长的参考手册。掌握了OpenCV的以下基础知识后，有需要的话再查阅相关的参考手册。

目录

[隐藏]

[编辑]

一、简介

[编辑]

1、OpenCV的特点

[编辑]

（1） 总体描述

- OpenCV是一个基于C/C++语言的开源图像处理函数库
- 其代码都经过优化，可用于实时处理图像
- 具有良好的可移植性
- 可以进行图像/视频载入、保存和采集的常规操作
- 具有低级和高级的应用程序接口（API）
- 提供了面向Intel IPP高效多媒体函数库的接口，可针对你使用的Intel CPU优化代码，提高程序性能（译注：OpenCV 2.0版的代码已显著优化，无需IPP来提升性能，故2.0版不再提供IPP接口）

[编辑]

（**2**） 功能

- 图像数据操作（内存分配与释放，图像复制、设定和转换）

  Image data manipulation (allocation, release, copying, setting, conversion).

- 图像/视频的输入输出（支持文件或摄像头的输入，图像/视频文件的输出）

  Image and video I/O (file and camera based input, image/video file output).

- 矩阵/向量数据操作及线性代数运算（矩阵乘积、矩阵方程求解、特征值、奇异值分解）

  Matrix and vector manipulation and linear algebra routines (products, solvers, eigenvalues, SVD).

- 支持多种动态数据结构（链表、队列、数据集、树、图）

  Various dynamic data structures (lists, queues, sets, trees, graphs).

- 基本图像处理（去噪、边缘检测、角点检测、采样与插值、色彩变换、形态学处理、直方图、图像金字塔结构）

  Basic image processing (filtering, edge detection, corner detection, sampling and interpolation, color conversion, morphological operations, histograms, image pyramids).

- 结构分析（连通域/分支、轮廓处理、距离转换、图像矩、模板匹配、霍夫变换、多项式逼近、曲线拟合、椭圆拟合、狄劳尼三角化）

  Structural analysis (connected components, contour processing, distance transform, various moments, template matching, Hough transform, polygonal approximation, line fitting, ellipse fitting, Delaunay triangulation).

- 摄像头定标（寻找和跟踪定标模式、参数定标、基本矩阵估计、单应矩阵估计、立体视觉匹配）

  Camera calibration (finding and tracking calibration patterns, calibration, fundamental matrix estimation, homography estimation, stereo correspondence).

- 运动分析（光流、动作分割、目标跟踪）

  Motion analysis (optical flow, motion segmentation, tracking).

- 目标识别（特征方法、HMM模型）

  Object recognition (eigen-methods, HMM).

- 基本的GUI（显示图像/视频、键盘/鼠标操作、滑动条）

  Basic GUI (display image/video, keyboard and mouse handling, scroll-bars).

- 图像标注（直线、曲线、多边形、文本标注）

  Image labeling (line, conic, polygon, text drawing)

[编辑]

（**3**） **OpenCV**模块

- cv – 核心函数库
- cvaux – 辅助函数库
- cxcore – 数据结构与线性代数库
- highgui – GUI函数库
- ml – 机器学习函数库

[编辑]

2、有用的学习资源

[编辑]

（**1**） 参考手册:

- <opencv-root>/docs/index.htm （译注：在你的OpenCV安装目录<opencv-root>内）

[编辑]

（**2**） 网络资源:

- 官方网站: http://www.intel.com/technology/computing/opencv/
- 软件下载: http://sourceforge.net/projects/opencvlibrary/

[编辑]

（**3**） 书籍:

- Open Source Computer Vision Library

by Gary R. Bradski, Vadim Pisarevsky, and Jean-Yves Bouguet, Springer, 1st ed. (June, 2006).
chenyusiyuan: 补充以下书籍

- Learning OpenCV - Computer Vision with the OpenCV Library

by Gary Bradski & Adrian Kaehler, O'Reilly Media, 1 st ed. (September, 2008).

- OpenCV教程——基础篇

作者：刘瑞祯 于仕琪，北京航空航天大学出版社，出版日期：200706

[编辑]

（**4**） 视频处理例程**(**在 **<opencv-root>/samples/c/):**

- 颜色跟踪: camshiftdemo
- 点跟踪: lkdemo
- 动作分割: motempl
- 边缘检测: laplace

[编辑]

（**5**） 图像处理例程 **(**在 **<opencv-root>/samples/c/):**

- 边缘检测: edge
- 图像分割: pyramid_segmentation
- 形态学: morphology
- 直方图: demhist
- 距离变换: distrans
- 椭圆拟合: fitellipse

[编辑]

3、OpenCV 命名规则

[编辑]

（1） 函数名:

```
cvActionTargetMod(...)

Action = 核心功能（core functionality） (e.g. set, create)
Target = 目标图像区域（target image area） (e.g. contour, polygon)
Mod    = （可选的）调整语（optional modifiers） (e.g. argument type)
```

[编辑]

（2） 矩阵数据类型:

```
CV_<bit_depth>(S|U|F)C<number_of_channels>

 S = 符号整型
U = 无符号整型
F = 浮点型

E.g.: CV_8UC1 是指一个8位无符号整型单通道矩阵,
          CV_32FC2是指一个32位浮点型双通道矩阵.
```

[编辑]

（3） 图像数据类型:

```
IPL_DEPTH_<bit_depth>(S|U|F)

E.g.: IPL_DEPTH_8U 图像像素数据是8位无符号整型.
      IPL_DEPTH_32F图像像素数据是32位浮点型.
```

[编辑]

（4） 头文件:

```
#include <cv.h>
    #include <cvaux.h>
    #include <highgui.h>
    #include <ml.h>
    #include <cxcore.h>    // 一般不需要，cv.h 内已包含该头文件
```

[编辑]

4、编译建议

[编辑]

（1） Linux:

```
g++ hello-world.cpp -o hello-world \
    -I /usr/local/include/opencv -L /usr/local/lib  \
    -lm -lcv -lhighgui -lcvaux
```

[编辑]

（2） Windows:

在Visual Studio的‘选项’和‘项目’中设置好OpenCV相关文件的路径。

[编辑]

5、C例程

```
//////////////////////////////////////////////////////////////////////////
```

```
//
// hello-world.cpp
//
// 该程序从文件中读入一幅图像，将之反色，然后显示出来.
//
////////////////////////////////////////////////////////////////////////
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <cv.h>
#include <highgui.h>

int main(int argc, char *argv[])
{
  IplImage* img = 0;
  int height,width,step,channels;
  uchar *data;
  int i,j,k;

  if(argc<2){
    printf("Usage: main <image-file-name>\n\7");
    exit(0);
  }

  // load an image
  img=cvLoadImage(argv[1]);
  if(!img){
    printf("Could not load image file: %s\n",argv[1]);
    exit(0);
  }

  // get the image data
  height    = img->height;
  width     = img->width;
  step      = img->widthStep;
  channels  = img->nChannels;
  data      = (uchar *)img->imageData;
  printf("Processing a %dx%d image with %d channels\n",height,width,channels);

  // create a window
  cvNamedWindow("mainWin", CV_WINDOW_AUTOSIZE);
  cvMoveWindow("mainWin", 100, 100);

  // invert the image
  // 相当于 cvNot(img);
  for(i=0;i<height;i++) for(j=0;j<width;j++) for(k=0;k<channels;k++)
    data[i*step+j*channels+k]=255-data[i*step+j*channels+k];

  // show the image
  cvShowImage("mainWin", img );

  // wait for a key
  cvWaitKey(0);

  // release the image
  cvReleaseImage(&img );
  return 0;
}
```

[编辑]

二、GUI 指令

[编辑]

1、窗口管理

[编辑]

（1） 创建和定位一个新窗口:

```
cvNamedWindow("win1", CV_WINDOW_AUTOSIZE);
  cvMoveWindow("win1", 100, 100); // offset from the UL corner of the screen
```

[编辑]

（2） 载入图像:

```
IplImage* img=0;
  img=cvLoadImage(fileName);
  if(!img) printf("Could not load image file: %s\n",fileName);
```

[编辑]

（3） 显示图像:

```
cvShowImage("win1",img);
```

该函数可以显示彩色或灰度的字节型/浮点型图像。字节型图像像素值范围为[0-255]；浮点型图像像素值范围为[0-1]。彩色图像的三色元素按BGR（蓝-红-绿）顺序存储。

[编辑]

（4） 关闭窗口:

```
cvDestroyWindow("win1");
```

[编辑]

（5） 改变窗口大小:

```
cvResizeWindow("win1",100,100); // new width/heigh in pixels
```

[编辑]

2、输入处理

[编辑]

（1） 处理鼠标事件:

- 定义一个鼠标处理程序:

```
void mouseHandler(int event, int x, int y, int flags, void* param)
  {
    switch(event){
      case CV_EVENT_LBUTTONDOWN:
        if(flags & CV_EVENT_FLAG_CTRLKEY)
          printf("Left button down with CTRL pressed\n");
        break;

      case CV_EVENT_LBUTTONUP:
        printf("Left button up\n");
        break;
    }
  }
```

```
 x,y:    相对于左上角的像素坐标

 event: CV_EVENT_LBUTTONDOWN,    CV_EVENT_RBUTTONDOWN,    CV_EVENT_MBUTTONDOWN,
        CV_EVENT_LBUTTONUP,      CV_EVENT_RBUTTONUP,      CV_EVENT_MBUTTONUP,
        CV_EVENT_LBUTTONDBLCLK,  CV_EVENT_RBUTTONDBLCLK,  CV_EVENT_MBUTTONDBLCLK,
        CV_EVENT_MOUSEMOVE:

 flags: CV_EVENT_FLAG_CTRLKEY, CV_EVENT_FLAG_SHIFTKEY, CV_EVENT_FLAG_ALTKEY,
        CV_EVENT_FLAG_LBUTTON, CV_EVENT_FLAG_RBUTTON,  CV_EVENT_FLAG_MBUTTON
```

- 注册该事件处理程序:

```
mouseParam=5;
  cvSetMouseCallback("win1",mouseHandler,&mouseParam);
```

[编辑]

（2） 处理键盘事件:

- 实际上对于键盘输入并没有专门的事件处理程序.
- 按一定间隔检测键盘输入（适用于循环体中）：

```
int key;
  key=cvWaitKey(10); // wait 10ms for input
```

- 中止程序等待键盘输入:

```
int key;
  key=cvWaitKey(0); // wait indefinitely for input
```

- 键盘输入的循环处理程序:

```
while(1){
    key=cvWaitKey(10);
    if(key==27) break;

    switch(key){
      case 'h':
        ...
        break;
      case 'i':
        ...
        break;
    }
  }
```

[编辑]

（**3**） 处理滑动条事件**:**

- 定义一个滑动条处理程序:

```
void trackbarHandler(int pos)
  {
    printf("Trackbar position: %d\n",pos);
  }
```

- 注册该事件处理程序:

```
int trackbarVal=25;
  int maxVal=100;
  cvCreateTrackbar("bar1", "win1", &trackbarVal ,maxVal , trackbarHandler);
```

- 获取当前的滑动条位置:

```
int pos = cvGetTrackbarPos("bar1","win1");
```

- 设置滑动条位置:

```
cvSetTrackbarPos("bar1", "win1", 25);
```

[编辑]

三、**OpenCV**的基本数据结构

（译注：OpenCV 1.1、1.2或2.0版本中各数据结构的结构体元素有所调整，以下仅作参考）

[编辑]

1、图像数据结构

[编辑]

1 IPL :

（ ）　　图像

```
IplImage
   |-- int  nChannels;      // 颜色通道数目 (1,2,3,4)
   |-- int  depth;          // 像素的位深:
   |                        //   IPL_DEPTH_8U, IPL_DEPTH_8S,
   |                        //   IPL_DEPTH_16U,IPL_DEPTH_16S,
   |                        //   IPL_DEPTH_32S,IPL_DEPTH_32F,
   |                        //   IPL_DEPTH_64F
   |-- int  width;          // 图像宽度（像素为单位）
   |-- int  height;         // 图像高度
   |-- char* imageData;     // 图像数据指针
   |                        // 注意彩色图像按BGR顺序存储数据
   |-- int  dataOrder;      // 0 - 将像素点不同通道的值交错排在一起，形成单一像素平面
   |                        // 1 - 把所有像素同通道值排在一起，形成若干个通道平面，再把平面排列起来
   |                        // cvCreateImage 只能创建像素交错排列式的图像
   |-- int  origin;         // 0 – 像素原点为左上角,
   |                        // 1 – 像素原点为左下角 (Windows bitmaps style)
   |-- int  widthStep;      // 相邻行的同列点之间的字节数
   |-- int  imageSize;      // 图像的大小（字节为单位） = height*widthStep
   |-- struct _IplROI *roi;// 图像的感兴趣区域（ROI）. ROI非空时对图像的
   |                        // 处理仅限于ROI区域.
   |-- char *imageDataOrigin; // 图像数据未对齐时的数据原点指针
   |                        // (需要正确地重新分配图像内存 )
   |                        // (needed for correct image deallocation)
   |-- int  align;          // 图像数据的行对齐: 4 or 8 byte alignment
   |                        // OpenCV 中无此项，采用widthStep代替
   |-- char colorModel[4]; // 颜色模型 – OpenCV中忽略此项
```

[编辑]

2、矩阵与向量

[编辑]

（1） 矩阵:

```
CvMat                       // 2D 矩阵
   |-- int   type;          // 元素类型 (uchar,short,int,float,double) 与标志
   |-- int   step;          // 整行长度字节数
   |-- int   rows, cols;    // 行、列数
   |-- int   height, width; // 矩阵高度、宽度，与rows、cols对应
   |-- union data;
       |-- uchar*  ptr;     // data pointer for an unsigned char matrix
       |-- short*  s;       // data pointer for a short matrix
       |-- int*    i;       // data pointer for an integer matrix
       |-- float*  fl;      // data pointer for a float matrix
       |-- double* db;      // data pointer for a double matrix

CvMatND                     // N-维矩阵
   |-- int   type;          // 元素类型 (uchar,short,int,float,double) 与标志
   |-- int   dims;          // 矩阵维数
   |-- union data;
   |   |-- uchar*  ptr;     // data pointer for an unsigned char matrix
   |   |-- short*  s;       // data pointer for a short matrix
   |   |-- int*    i;       // data pointer for an integer matrix
   |   |-- float*  fl;      // data pointer for a float matrix
   |   |-- double* db;      // data pointer for a double matrix
   |
   |-- struct dim[];        // 各维信息
       |-- size;            // 元素数目
       |-- step;            // 元素间距（字节为单位）

CvSparseMat // N-维稀疏矩阵
```

[编辑]

（2） 一般矩阵:

```
CvArr*      // 仅作为函数定义的参数使用,
            // 表明函数可以接受不同类型的矩阵作为参数,
            // 例如: IplImage*, CvMat* 甚至是 CvSeq*.
            // 矩阵的类型通过矩阵头的前4个字节信息来确定
```

[编辑]

（**3**） 标量**:**

```
CvScalar
  |-- double val[4]; //4D 向量
```

初始化函数:

```
CvScalar s = cvScalar(double val0, double val1=0, double val2=0, double val3=0);
// Example:
CvScalar s = cvScalar(20.0);
s.val[0]=20.0;
```

注意该初始化函数的函数名与对应的结构体名称几乎同名，差别仅在于函数名第一个字母是小写的，而结构体名第一个字母是大写的。它并不是一个 C++ 构造函数。（译注：类似的还有 cvMat 与 CvMat、cvPoint 与 CvPoint 等等）

[编辑]

3、其它结构类型

[编辑]

（**1**） 点**:**

```
CvPoint      p = cvPoint(int x, int y);
CvPoint2D32f p = cvPoint2D32f(float x, float y);
CvPoint3D32f p = cvPoint3D32f(float x, float y, float z);
//E.g.:
p.x=5.0;
p.y=5.0;
```

[编辑]

（**2**） 矩形框大小（以像素为精度）**:**

```
CvSize       r = cvSize(int width, int height);
CvSize2D32f  r = cvSize2D32f(float width, float height);
```

[编辑]

（**3**） 矩形框的偏置和大小**:**

```
CvRect       r = cvRect(int x, int y, int width, int height);
```

[编辑]

四、图像处理

[编辑]

1、图像的内存分配与释放

[编辑]

（**1**） 分配内存给一幅新图像**:**

```
IplImage* cvCreateImage(CvSize size, int depth, int channels);
```

size: cvSize(width,height);

depth: 像素深度: IPL_DEPTH_8U, IPL_DEPTH_8S, IPL_DEPTH_16U,

IPL_DEPTH_16S, IPL_DEPTH_32S, IPL_DEPTH_32F, IPL_DEPTH_64F

channels: 像素通道数. Can be 1, 2, 3 or 4.

各通道是交错排列的. 一幅彩色图像的数据排列格式如下:

b0 g0 r0 b1 g1 r1 ...

示例:

```
// Allocate a 1-channel byte image
IplImage* img1=cvCreateImage(cvSize(640,480),IPL_DEPTH_8U,1);

// Allocate a 3-channel float image
IplImage* img2=cvCreateImage(cvSize(640,480),IPL_DEPTH_32F,3);
```

[编辑]

（**2**） 释放图像**:**

```
IplImage* img=cvCreateImage(cvSize(640,480),IPL_DEPTH_8U,1);
cvReleaseImage(&img);
```

[编辑]

（**3**） 复制图像**:**

```
IplImage* img1=cvCreateImage(cvSize(640,480),IPL_DEPTH_8U,1);
IplImage* img2;
img2=cvCloneImage(img1);  // 注意通过cvCloneImage得到的图像
                          // 也要用 cvReleaseImage 释放, 否则容易产生内存泄漏
```

[编辑]

（**4**） 设置**/**获取感兴趣区域**ROI:**

```
void  cvSetImageROI(IplImage* image, CvRect rect);
void  cvResetImageROI(IplImage* image);
CvRect cvGetImageROI(const IplImage* image);
```

大多数OpenCV函数都支持 ROI.

[编辑]

（**5**） 设置**/**获取感兴趣通道**COI:**

```
void cvSetImageCOI(IplImage* image, int coi); // 0=all
int cvGetImageCOI(const IplImage* image);
```

大多数OpenCV函数不支持 COI.

[编辑]

2、图像读写

[编辑]

（**1**） 从文件中读入图像**:**

```
IplImage* img=0;
  img=cvLoadImage(fileName);
  if(!img) printf("Could not load image file: %s\n",fileName);
```

支持的图像格式: BMP, DIB, JPEG, JPG, JPE, PNG, PBM, PGM, PPM,
SR, RAS, TIFF, TIF

OpenCV默认将读入的图像强制转换为一幅三通道彩色图像. 不过可以按以下方法修改读入方式:

```
img=cvLoadImage(fileName,flag);
```

flag: >0 将读入的图像强制转换为一幅三通道彩色图像

```
=0 将读入的图像强制转换为一幅单通道灰度图像
<0 读入的图像通道数与所读入的文件相同.
```

[编辑]

（**2**） 保存图像**:**

```
if(!cvSaveImage(outFileName,img)) printf("Could not save: %s\n", outFileName);
```

保存的图像格式由 outFileName 中的扩展名确定.

[编辑]

3、访问图像像素

[编辑]

（**1**） 假设你要访问第**k**通道、第**i**行、第**j**列的像素。

[编辑]

（**2**） 间接访问**: (**通用，但效率低，可访问任意格式的图像**)**

- 对于单通道字节型图像:

```
IplImage* img=cvCreateImage(cvSize(640,480),IPL_DEPTH_8U,1);
CvScalar s;
s=cvGet2D(img,i,j); // get the (i,j) pixel value
printf("intensity=%f\n",s.val[0]);
s.val[0]=111;
cvSet2D(img,i,j,s); // set the (i,j) pixel value
```

- 对于多通道字节型/浮点型图像:

```
IplImage* img=cvCreateImage(cvSize(640,480),IPL_DEPTH_32F,3);
CvScalar s;
s=cvGet2D(img,i,j); // get the (i,j) pixel value
printf("B=%f, G=%f, R=%f\n",s.val[0],s.val[1],s.val[2]);
s.val[0]=111;
s.val[1]=111;
s.val[2]=111;
cvSet2D(img,i,j,s); // set the (i,j) pixel value
```

[编辑]

（**3**） 直接访问**: (**效率高，但容易出错**)**

- 对于单通道字节型图像:

```
IplImage* img=cvCreateImage(cvSize(640,480),IPL_DEPTH_8U,1);
((uchar *)(img->imageData + i*img->widthStep))[j]=111;
```

- 对于多通道字节型图像:

```
IplImage* img=cvCreateImage(cvSize(640,480),IPL_DEPTH_8U,3);
((uchar *)(img->imageData + i*img->widthStep))[j*img->nChannels + 0]=111; // B
((uchar *)(img->imageData + i*img->widthStep))[j*img->nChannels + 1]=112; // G
((uchar *)(img->imageData + i*img->widthStep))[j*img->nChannels + 2]=113; // R
```

- 对于多通道浮点型图像:

```
IplImage* img=cvCreateImage(cvSize(640,480),IPL_DEPTH_32F,3);
((float *)(img->imageData + i*img->widthStep))[j*img->nChannels + 0]=111; // B
((float *)(img->imageData + i*img->widthStep))[j*img->nChannels + 1]=112; // G
((float *)(img->imageData + i*img->widthStep))[j*img->nChannels + 2]=113; // R
```

[编辑]

（**4**） 基于指针的直接访问**: (**简单高效**)**

- 对于单通道字节型图像:

```
IplImage* img  = cvCreateImage(cvSize(640,480),IPL_DEPTH_8U,1);
int height     = img->height;
int width      = img->width;
int step       = img->widthStep/sizeof(uchar);
uchar* data    = (uchar *)img->imageData;
data[i*step+j] = 111;
```

- 对于多通道字节型图像:

```
IplImage* img  = cvCreateImage(cvSize(640,480),IPL_DEPTH_8U,3);
int height     = img->height;
int width      = img->width;
int step       = img->widthStep/sizeof(uchar);
int channels   = img->nChannels;
uchar* data    = (uchar *)img->imageData;
data[i*step+j*channels+k] = 111;
```

- 对于多通道浮点型图像（假设图像数据采用4字节（32位）行对齐方式）:

```
IplImage* img  = cvCreateImage(cvSize(640,480),IPL_DEPTH_32F,3);
int height     = img->height;
int width      = img->width;
int step       = img->widthStep/sizeof(float);
int channels   = img->nChannels;
float * data    = (float *)img->imageData;
data[i*step+j*channels+k] = 111;
```

[编辑]

（**5**） 基于 **c++ wrapper** 的直接访问**:** （更简单高效）

- 首先定义一个 c++ wrapper ‘Image’，然后基于Image定义不同类型的图像:

```
template<class T> class Image
{
  private:
  IplImage* imgp;
  public:
  Image(IplImage* img=0) {imgp=img;}
  ~Image(){imgp=0;}
  void operator=(IplImage* img) {imgp=img;}
  inline T* operator[](const int rowIndx) {
    return ((T *)(imgp->imageData + rowIndx*imgp->widthStep));}
};

typedef struct{
  unsigned char b,g,r;
} RgbPixel;

typedef struct{
  float b,g,r;
} RgbPixelFloat;

typedef Image<RgbPixel>       RgbImage;
typedef Image<RgbPixelFloat>  RgbImageFloat;
typedef Image<unsigned char>  BwImage;
typedef Image<float>          BwImageFloat;
```

- 对于单通道字节型图像:

```
IplImage* img=cvCreateImage(cvSize(640,480),IPL_DEPTH_8U,1);
BwImage imgA(img);
imgA[i][j] = 111;
```

- 对于多通道字节型图像:

```
IplImage* img=cvCreateImage(cvSize(640,480),IPL_DEPTH_8U,3);
RgbImage  imgA(img);
```

```
imgA[i][j].b = 111;
imgA[i][j].g = 111;
imgA[i][j].r = 111;
```

- 对于多通道浮点型图像:

```
IplImage* img=cvCreateImage(cvSize(640,480),IPL_DEPTH_32F,3);
RgbImageFloat imgA(img);
imgA[i][j].b = 111;
imgA[i][j].g = 111;
imgA[i][j].r = 111;
```

[编辑]

4、图像转换

[编辑]

（1） 字节型图像的灰度-彩色转换:

```
cvConvertImage(src, dst, flags=0);

 src = float/byte grayscale/color image
 dst = byte grayscale/color image
 flags = CV_CVTIMG_FLIP     (垂直翻转图像)
         CV_CVTIMG_SWAP_RB  (置换 R 和 B 通道)
```

[编辑]

（2） 彩色图像->灰度图像:

```
// Using the OpenCV conversion:
cvCvtColor(cimg,gimg,CV_BGR2GRAY); // cimg -> gimg

// Using a direct conversion:
for(i=0;i<cimg->height;i++) for(j=0;j<cimg->width;j++)
  gimgA[i][j]= (uchar)(cimgA[i][j].b*0.114 +
                       cimgA[i][j].g*0.587 +
                       cimgA[i][j].r*0.299);
```

[编辑]

（3） 不同彩色空间之间的转换:

```
cvCvtColor(src,dst,code); // src -> dst

 code    = CV_<X>2<Y>
 <X>/<Y> = RGB, BGR, GRAY, HSV, YCrCb, XYZ, Lab, Luv, HLS
```

e.g.: CV_BGR2GRAY, CV_BGR2HSV, CV_BGR2Lab

[编辑]

5、绘图指令

[编辑]

（1） 绘制矩形:

```
// 在点 (100,100) 和 (200,200) 之间绘制一矩形, 边线用红色、宽度为 1
cvRectangle(img, cvPoint(100,100), cvPoint(200,200), cvScalar(255,0,0), 1);
```

[编辑]

（2） 绘制圆形:

```
// 圆心为(100,100)、半径为20. 圆周绿色、宽度为1
cvCircle(img, cvPoint(100,100), 20, cvScalar(0,255,0), 1);
```

[编辑]

（3） 绘制线段:

```
// 在 (100,100) 和 (200,200) 之间、线宽为 1 的绿色线段
cvLine(img, cvPoint(100,100), cvPoint(200,200), cvScalar(0,255,0), 1);
```

[编辑]

（4） 绘制一组线段:

```
CvPoint  curve1[]={10,10,  10,100,  100,100,  100,10};
CvPoint  curve2[]={30,30,  30,130,  130,130,  130,30,  150,10};
CvPoint* curveArr[2]={curve1, curve2};
int      nCurvePts[2]={4,5};
int      nCurves=2;
int      isCurveClosed=1;
int      lineWidth=1;

cvPolyLine(img,curveArr,nCurvePts,nCurves,isCurveClosed,cvScalar(0,255,255),lineWidth);

void cvPolyLine( CvArr* img, CvPoint** pts, int* npts, int contours, int is_closed,
                 CvScalar color, int thickness=1, int line_type=8, int shift=0 );
img        图像。
pts        折线的顶点指针数组。
npts       折线的定点个数数组。也可以认为是pts指针数组的大小
contours   折线的线段数量。
is_closed  指出多边形是否封闭。如果封闭，函数将起始点和结束点连线。
color          折线的颜色。
thickness  线条的粗细程度。
line_type  线段的类型。参见cvLine。
shift          顶点的小数点位数
```

[编辑]

（5） 绘制一组填充颜色的多边形:

```
cvFillPoly(img,curveArr,nCurvePts,nCurves,cvScalar(0,255,255));

cvFillPoly用于一个单独被多边形轮廓所限定的区域内进行填充。函数可以填充复杂的区域,例如，有漏洞的区域和有交叉点的区域等等。
void cvFillPoly( CvArr* img, CvPoint** pts, int* npts, int contours,CvScalar color, int
line_type=8, int shift=0 );
img            图像。
pts            指向多边形的数组指针。
npts          多边形的顶点个数的数组。
contours   组成填充区域的线段的数量。
color          多边形的颜色。
line_type  组成多边形的线条的类型。
shift          顶点坐标的小数点位数。
```

[编辑]

（6） 文本标注:

```
CvFont font;
double hScale=1.0;
double vScale=1.0;
int    lineWidth=1;
cvInitFont(&font,CV_FONT_HERSHEY_SIMPLEX|CV_FONT_ITALIC, hScale,vScale,0,lineWidth);

cvPutText (img,"My comment",cvPoint(200,400), &font, cvScalar(255,255,0));
```

其它可用的字体类型有: CV_FONT_HERSHEY_SIMPLEX, CV_FONT_HERSHEY_PLAIN, CV_FONT_HERSHEY_DUPLEX, CV_FONT_HERSHEY_COMPLEX, CV_FONT_HERSHEY_TRIPLEX, CV_FONT_HERSHEY_COMPLEX_SMALL, CV_FONT_HERSHEY_SCRIPT_SIMPLEX, CV_FONT_HERSHEY_SCRIPT_COMPLEX,

---

[编辑]

五、矩阵处理

[编辑]

1、矩阵的内存分配与释放

[编辑]

（1） 总体上:

- OpenCV 使用C语言来进行矩阵操作。不过实际上有很多C++语言的替代方案可以更高效地完成。
- 在OpenCV中向量被当做是有一个维数为1的N维矩阵.
- 矩阵按行-行方式存储，每行以4字节（32位）对齐.

[编辑]

（2） 为新矩阵分配内存:

```
CvMat* cvCreateMat(int rows, int cols, int type);

 type: 矩阵元素类型.
 按CV_<bit_depth>(S|U|F)C<number_of_channels> 方式指定.  例如: CV_8UC1 、CV_32SC2.
 示例:

CvMat* M = cvCreateMat(4,4,CV_32FC1);
```

[编辑]

（3） 释放矩阵内存:

```
CvMat* M = cvCreateMat(4,4,CV_32FC1);
cvReleaseMat(&M);
```

[编辑]

（4） 复制矩阵:

```
CvMat* M1 = cvCreateMat(4,4,CV_32FC1);
CvMat* M2;
M2=cvCloneMat(M1);
```

[编辑]

（5） 初始化矩阵:

```
double a[] = { 1,  2,  3,  4,
               5,  6,  7,  8,
               9, 10, 11, 12 };
CvMat Ma=cvMat(3, 4, CV_64FC1, a);

//等价于:
CvMat Ma;
cvInitMatHeader(&Ma, 3, 4, CV_64FC1, a);
```

[编辑]

（6） 初始化矩阵为单位矩阵:

```
CvMat* M = cvCreateMat(4,4,CV_32FC1);
cvSetIdentity(M); // does not seem to be working properly
```

[编辑]

2、访问矩阵元素

[编辑]

（1） 假设需要访问一个2D浮点型矩阵的第（i, j）个单元.

[编辑]

（2） 间接访问:

```
cvmSet(M,i,j,2.0); // Set M(i,j)
t = cvmGet(M,i,j); // Get M(i,j)
```

[编辑]

（**3**） 直接访问（假设矩阵数据按**4**字节行对齐）**:**

```
CvMat* M    = cvCreateMat(4,4,CV_32FC1);
int n       = M->cols;
float *data = M->data.fl;
data[i*n+j] = 3.0;
```

[编辑]

（**4**） 直接访问（当数据的行对齐可能存在间隙时 **possible alignment gaps**）**:**

```
CvMat* M    = cvCreateMat(4,4,CV_32FC1);
int   step  = M->step/sizeof(float);
float *data = M->data.fl;
(data+i*step)[j] = 3.0;
```

[编辑]

（**5**） 对于初始化后的矩阵进行直接访问**:**

```
double a[16];
CvMat Ma = cvMat(3, 4, CV_64FC1, a);
a[i*4+j] = 2.0; // Ma(i,j)=2.0;
```

[编辑]

3、矩阵**/**向量运算

[编辑]

（**1**） 矩阵之间的运算**:**

```
CvMat *Ma, *Mb, *Mc;
cvAdd(Ma, Mb, Mc);       // Ma+Mb    -> Mc
cvSub(Ma, Mb, Mc);       // Ma-Mb    -> Mc
cvMatMul(Ma, Mb, Mc);    // Ma*Mb    -> Mc
```

[编辑]

（**2**） 矩阵之间的元素级运算**:**

```
CvMat *Ma, *Mb, *Mc;
cvMul(Ma, Mb, Mc);       // Ma.*Mb   -> Mc
cvDiv(Ma, Mb, Mc);       // Ma./Mb   -> Mc
cvAddS(Ma, cvScalar(-10.0), Mc); // Ma.-10 -> Mc
```

[编辑]

（**3**） 向量乘积**:**

```
double va[] = {1, 2, 3};
double vb[] = {0, 0, 1};
double vc[3];

CvMat Va=cvMat(3, 1, CV_64FC1, va);
CvMat Vb=cvMat(3, 1, CV_64FC1, vb);
CvMat Vc=cvMat(3, 1, CV_64FC1, vc);

double res=cvDotProduct(&Va,&Vb); // 向量点乘:   Va . Vb -> res
cvCrossProduct(&Va, &Vb, &Vc);    // 向量叉乘:   Va x Vb -> Vc
```

注意在进行叉乘运算时，Va, Vb, Vc 必须是仅有3个元素的向量.

[编辑]

（**4**） 单一矩阵的运算**:**

```
CvMat *Ma, *Mb;
cvTranspose(Ma, Mb);       // 转置: transpose(Ma) -> Mb (注意转置阵不能返回给Ma本身)
CvScalar t = cvTrace(Ma); // 迹: trace(Ma) -> t.val[0]
double d = cvDet(Ma);      // 行列式: det(Ma) -> d
cvInvert(Ma, Mb);          // 逆矩阵: inv(Ma) -> Mb
```

[编辑]

（**5**） 非齐次线性方程求解**:**

```
CvMat* A  = cvCreateMat(3,3,CV_32FC1);
CvMat* x  = cvCreateMat(3,1,CV_32FC1);
CvMat* b  = cvCreateMat(3,1,CV_32FC1);
cvSolve(&A, &b, &x);    // solve (Ax=b) for x
```

[编辑]

（**6**） 特征值与特征向量 **(**矩阵为方阵**):**

```
CvMat* A  = cvCreateMat(3,3,CV_32FC1);
CvMat* E  = cvCreateMat(3,3,CV_32FC1);
CvMat* l  = cvCreateMat(3,1,CV_32FC1);
cvEigenVV(&A, &E, &l);  // l = A 的特征值(递减顺序)
                        // E = 对应的特征向量 (行向量)
```

（7） 奇异值分解（SVD）:====

```
CvMat* A  = cvCreateMat(3,3,CV_32FC1);
CvMat* U  = cvCreateMat(3,3,CV_32FC1);
CvMat* D  = cvCreateMat(3,3,CV_32FC1);
CvMat* V  = cvCreateMat(3,3,CV_32FC1);
cvSVD(A, D, U, V, CV_SVD_U_T|CV_SVD_V_T); // A = U D V^T
```

标志位使矩阵U或V按转置形式返回 (若不转置可能运算出错).

[编辑]

六、视频处理

[编辑]

1、从视频流中捕捉一帧画面

[编辑]

（**1**） **OpenCV** 支持从摄像头或视频文件（**AVI**格式）中捕捉帧画面**.**

[编辑]

（**2**） 初始化一个摄像头捕捉器**:**

```
CvCapture* capture = cvCaptureFromCAM(0); // capture from video device #0
```

[编辑]

（**3**） 初始化一个视频文件捕捉器**:**

```
CvCapture* capture = cvCaptureFromAVI("infile.avi");
```

[编辑]

（**4**） 捕捉一帧画面**:**

```
IplImage* img = 0;
if(!cvGrabFrame(capture)){              // capture a frame
  printf("Could not grab a frame\n\7");
  exit(0);
```

```
}
img=cvRetrieveFrame(capture);           // retrieve the captured frame
```

若要从多个摄像头中同步捕捉画面，则须首先从每个摄像头中抓取一帧，紧接着要将被捕捉的帧画面恢复到一个IplImage*型图像中。（译注：这一过程其实可以用 cvQueryFrame() 函数一步完成）

[编辑]

（**5**） 释放视频流捕捉器**:**

```
cvReleaseCapture(&capture);
```

注意由视频流捕捉器得到的图像是由捕捉器分配和释放内存的，不需要单独对图像进行释放内存的操作。

[编辑]

2、获取**/**设置视频流信息

[编辑]

（**1**） 获取视频流设备信息**:**

```
cvQueryFrame(capture); // 在读取视频流信息前，要先执行此操作
int frameH    = (int) cvGetCaptureProperty(capture, CV_CAP_PROP_FRAME_HEIGHT);
int frameW    = (int) cvGetCaptureProperty(capture, CV_CAP_PROP_FRAME_WIDTH);
int fps       = (int) cvGetCaptureProperty(capture, CV_CAP_PROP_FPS);
int numFrames = (int) cvGetCaptureProperty(capture,  CV_CAP_PROP_FRAME_COUNT);
```

统计总帧数仅对视频文件有效，但似乎不太准确（译注：也许OpenCV2.0中此问题已解决）

[编辑]

（**2**） 获取帧图信息**:**

```
float posMsec    =       cvGetCaptureProperty(capture, CV_CAP_PROP_POS_MSEC);
int posFrames    = (int) cvGetCaptureProperty(capture, CV_CAP_PROP_POS_FRAMES);
float posRatio   =       cvGetCaptureProperty(capture, CV_CAP_PROP_POS_AVI_RATIO);
```

所抓取的帧的位置有三种表达方式：距离第一帧画面的时间间隔（毫秒为单位），或者距离第一帧画面（序列号为0）的序列数. 第三种方式是按相对比率，第一帧的相对比率为0，最后一帧的相对比率为1. 此方式仅对读取视频文件时有效.

[编辑]

（**3**） 设置从视频文件抓取的第一帧画面的位置**:**

```
// start capturing from a relative position of 0.9 of a video file
cvSetCaptureProperty(capture, CV_CAP_PROP_POS_AVI_RATIO, (double)0.9);
```

注意此方法定位并不准确。

[编辑]

3、保存视频文件

[编辑]

（**1**） 初始化视频编写器**:**

```
CvVideoWriter *writer = 0;
int isColor = 1;
int fps     = 25;  // or 30
int frameW  = 640; // 744 for firewire cameras
int frameH  = 480; // 480 for firewire cameras
writer=cvCreateVideoWriter("out.avi",CV_FOURCC('P','I','M','1'),
                           fps,cvSize(frameW,frameH),isColor);
```

其它的编码器代号包括: CV_FOURCC('P','I','M','1') = MPEG-1 codec CV_FOURCC('M','J','P','G') = motion-jpeg codec (does not work well) CV_FOURCC('M', 'P', '4', '2') = MPEG-4.2 codec CV_FOURCC('D', 'I', 'V', '3') = MPEG-4.3 codec CV_FOURCC('D', 'I', 'V', 'X') = MPEG-4 codec CV_FOURCC('U', '2', '6', '3') = H263 codec CV_FOURCC('I', '2', '6', '3') = H263I codec CV_FOURCC('F', 'L', 'V', '1') = FLV1 codec 若编码器代号为 -1，则运行时会弹出一个编码器选择框.

[编辑]

（**2**） 保持视频文件**:**

```
IplImage* img = 0;
int nFrames = 50;
for(i=0;i<nFrames;i++){
  cvGrabFrame(capture);          // capture a frame
  img=cvRetrieveFrame(capture);  // retrieve the captured frame
  // img = cvQueryFrame(capture);
  cvWriteFrame(writer,img);      // add the frame to the file
}
```

要查看所抓取到的帧画面，可以在循环中加入以下语句:

```
cvShowImage("mainWin", img);
key=cvWaitKey(20);            // wait 20 ms
```

注意 cvWaitKey 参数应该不小于 20 ms，否则画面的显示可能出错.

[编辑]

（**3**） 释放视频编写器**:**

```
cvReleaseVideoWriter(&writer);
```

By Gady Agam 2006-03-31 翻译：chenyusiyuan 2010-1-26

OpenCV概述

Wikipedia，自由的百科全书

Intel® 开源计算机视觉库OpenCV

目录

[隐藏]

[编辑]

什么是**OpenCV**

OpenCV是Intel®开源计算机视觉库。它由一系列 C 函数和少量 C++ 类构成，实现了图像处理和计算机视觉方面的很多通用算法。

[编辑]

重要特性

OpenCV 拥有包括 300 多个C函数的跨平台的中、高层 API。它不依赖于其它的外部库——尽管也可以使用某些外部库。

OpenCV 对非商业应用和商业应用都是免费（FREE）的。（细节参考 BSD License）。

OpenCV 为Intel® Integrated Performance Primitives (IPP) 提供了透明接口。 这意味着如果有为特定处理器优化的的 IPP 库， OpenCV 将在运行时自动加载这些库。 更多关于 IPP 的信息请参考: http://www.intel.com/software/products/ipp/index.htm

[编辑]

谁创建了它

作者列表可以在文件AUTHORS中找到。

此外，还有很多人给出了建议、补丁、BUG 报告等等。一个不太完整的列表在文件THANKS中。

[编辑]

新特征

请参考OpenCVChangeLog。

[编辑]

从哪里下载 **OpenCV**

访问 http://www.sourceforge.net/projects/opencvlibrary 。您可以直接下载exe文件的安装包（windows用户），或者您可以通过代码管理工具（比如SVN或者CVS等下载openCV最新的源代码），您甚至不需要安装任何代码管理工具，直接通过sourceforge网站上提供的打包下载源代码的办法下载，具体可以参考Mingw编译最新版本的OpenCV代码。

如果在使用OpenCV存在问题，在 Google (http://www.google.com)中输入 "OpenCV" 和相关问题的关键字进行搜索。也可以到本网站的论坛上面发帖来咨询。论坛地址是：opencv中文论坛。

[编辑]

如果在安装/运行/使用 **OpenCV** 中遇到问题

1. 阅读FAQ中文
2. 在 OpenCV 邮件列表 www.yahoogroups.com （http://groups.yahoo.com/group/OpenCV/ ）中搜索。
3. 加入到 yahoo group 上的 OpenCV 邮件列表中（如何加入请参考 FAQs），并发送你的问题到邮件列表中。（这个邮件列表可能会迁移到OpenCV's SourceForge site）
4. 参考 OpenCV 的例子代码，阅读参考手册 :)

[编辑]

OpenCV参考手册

- CxCore中文参考手册
- Cv中文参考手册
- CvAux中文参考手册
- HighGUI中文参考手册

[编辑]

中文翻译者

- 于仕琪，中科院自动化所自由软件协会
- HUNNISH，阿须数码

FAQ中文

Wikipedia,自由的百科全书

目录

[隐藏]

[编辑]

安装配置问题

[编辑]

缺少**highgui100.dll**

是由于highgui100.dll所在目录(一般为C:\Program Files\OpenCV\bin)没有添加到系统环境变量所致,请参考VC6下安装与配置#配置Windows环境变量。

[编辑]

使用库的技术问题

[编辑]

视频读写出现问题

请参考:视频读写概述。

[编辑]

怎么访问图像像素

(坐标是从0开始的,并且是相对图像原点的位置。图像原点或者是左上角 (img->origin=IPL_ORIGIN_TL) 或者是左下角 (img->origin=IPL_ORIGIN_BL))

- 假设有 8-bit 1-通道的图像 I (IplImage* img):

```
I(x,y) ~ ((uchar*)(img->imageData + img->widthStep*y))[x]
```

- 假设有 8-bit 3-通道的图像 I (IplImage* img):

```
I(x,y)blue ~ ((uchar*)(img->imageData + img->widthStep*y))[x*3]
I(x,y)green ~ ((uchar*)(img->imageData + img->widthStep*y))[x*3+1]
I(x,y)red ~ ((uchar*)(img->imageData + img->widthStep*y))[x*3+2]
```

例如,给点 (100,100) 的亮度增加 30 ,那么可以这样做:

```
CvPoint pt = {100,100};
((uchar*)(img->imageData + img->widthStep*pt.y))[pt.x*3] += 30;
((uchar*)(img->imageData + img->widthStep*pt.y))[pt.x*3+1] += 30;
((uchar*)(img->imageData + img->widthStep*pt.y))[pt.x*3+2] += 30;
```

或者更高效地:

```
CvPoint pt = {100,100};
uchar* temp_ptr = &((uchar*)(img->imageData + img->widthStep*pt.y))[pt.x*3];
temp_ptr[0] += 30;
temp_ptr[1] += 30;
temp_ptr[2] += 30;
```

- 假设有 32-bit 浮点数, 1-通道 图像 I (IplImage* img):

```
I(x,y) ~ ((float*)(img->imageData + img->widthStep*y))[x]
```

- 现在,一般的情况下,假设有 N-通道,类型为 T 的图像:

```
I(x,y)c ~ ((T*)(img->imageData + img->widthStep*y))[x*N + c]
```

你可以使用宏 CV_IMAGE_ELEM(image_header, elemtype, y, x_Nc)

```
I(x,y)c ~ CV_IMAGE_ELEM( img, T, y, x*N + c )
```

也有针对各种图像(包括 4 通道图像)和矩阵的函数(cvGet2D, cvSet2D), 但是它们非常慢。

[编辑]

如何访问矩阵元素?

方法是类似的(下面的例子都是针对 0 起点的列和行)

- 设有 32-bit 浮点数的实数矩阵 M (CvMat* mat):

```
M(i,j) ~ ((float*)(mat->data.ptr + mat->step*i))[j]
```

- 设有 64-bit 浮点数的复数矩阵 M (CvMat* mat):

```
Re M(i,j) ~ ((double*)(mat->data.ptr + mat->step*i))[j*2]
Im M(i,j) ~ ((double*)(mat->data.ptr + mat->step*i))[j*2+1]
```

- 对单通道矩阵,有宏 CV_MAT_ELEM(matrix, elemtype, row, col), 例如对 32-bit 浮点数的实数矩阵:

```
M(i,j) ~ CV_MAT_ELEM( mat, float, i, j ),
```

例如,这儿是一个 3x3 单位矩阵的初始化:

```
CV_MAT_ELEM( mat, float, 0, 0 ) = 1.f;
CV_MAT_ELEM( mat, float, 0, 1 ) = 0.f;
CV_MAT_ELEM( mat, float, 0, 2 ) = 0.f;
CV_MAT_ELEM( mat, float, 1, 0 ) = 0.f;
CV_MAT_ELEM( mat, float, 1, 1 ) = 1.f;
CV_MAT_ELEM( mat, float, 1, 2 ) = 0.f;
CV_MAT_ELEM( mat, float, 2, 0 ) = 0.f;
CV_MAT_ELEM( mat, float, 2, 1 ) = 0.f;
CV_MAT_ELEM( mat, float, 2, 2 ) = 1.f;
```

[编辑]

如何在 **OpenCV** 中处理我自己的数据

设你有 300x200 32-bit 浮点数 image/array, 也就是对一个有 60000 个元素的数组。

```
int cols = 300, rows = 200;
float* myarr = new float[rows*cols];
// 第一步,初始化 CvMat 头
CvMat mat = cvMat( rows, cols,
                   CV_32FC1, // 32 位浮点单通道类型
                   myarr // 用户数据指针(数据没有被复制)
                   );
// 第二步,使用 cv 函数, 例如计算 l2 (Frobenius) 模
double norm = cvNorm( &mat, 0, CV_L2 );
...
delete myarr;
```

其它情况在参考手册中有描述。 见 cvCreateMatHeader,cvInitMatHeader,cvCreateImageHeader, cvSetData 等

[编辑]

如何读入和显示图像

```
/* usage: prog <image_name> */
#include "cv.h"
#include "highgui.h"

int main( int argc, char** argv )
{
    IplImage* img;
    if( argc == 2 && (img = cvLoadImage( argv[1], 1)) != 0 )
    {
        cvNamedWindow( "Image view", 1 );
        cvShowImage( "Image view", img );
        cvWaitKey(0); // 非常重要,内部包含事件处理循环
        cvDestroyWindow( "Image view" );
        cvReleaseImage( &img );
        return 0;
    }
    return -1;
}
```

[编辑]

如何检测和处理轮廓线

参考 squares demo

```
/*
在程序里找寻矩形
*/
#ifdef _CH_
#pragma package <opencv>
#endif

#ifndef _EiC
#include "cv.h"
#include "highgui.h"
#include <stdio.h>
#include <math.h>
#include <string.h>
#endif

int thresh = 50;
IplImage* img = 0;
IplImage* img0 = 0;
CvMemStorage* storage = 0;
CvPoint pt[4];
const char* wndname = "Square Detection Demo";

// helper function:
// finds a cosine of angle between vectors
// from pt0->pt1 and from pt0->pt2
double angle( CvPoint* pt1, CvPoint* pt2, CvPoint* pt0 )
{
    double dx1 = pt1->x - pt0->x;
    double dy1 = pt1->y - pt0->y;
```

```
    double dx2 = pt2->x - pt0->x;
    double dy2 = pt2->y - pt0->y;
    return (dx1*dx2 + dy1*dy2)/sqrt((dx1*dx1 + dy1*dy1)*(dx2*dx2 + dy2*dy2) + 1e-10);
}

// returns sequence of squares detected on the image.
// the sequence is stored in the specified memory storage
CvSeq* findSquares4( IplImage* img, CvMemStorage* storage )
{
    CvSeq* contours;
    int i, c, l, N = 11;
    CvSize sz = cvSize( img->width & -2, img->height & -2 );
    IplImage* timg = cvCloneImage( img ); // make a copy of input image
    IplImage* gray = cvCreateImage( sz, 8, 1 );
    IplImage* pyr = cvCreateImage( cvSize(sz.width/2, sz.height/2), 8, 3 );
    IplImage* tgray;
    CvSeq* result;
    double s, t;
    // create empty sequence that will contain points -
    // 4 points per square (the square's vertices)
    CvSeq* squares = cvCreateSeq( 0, sizeof(CvSeq), sizeof(CvPoint), storage );

    // select the maximum ROI in the image
    // with the width and height divisible by 2
    cvSetImageROI( timg, cvRect( 0, 0, sz.width, sz.height ));

    // down-scale and upscale the image to filter out the noise
    cvPyrDown( timg, pyr, 7 );
    cvPyrUp( pyr, timg, 7 );
    tgray = cvCreateImage( sz, 8, 1 );

    // find squares in every color plane of the image
    for( c = 0; c < 3; c++ )
    {
        // extract the c-th color plane
        cvSetImageCOI( timg, c+1 );
        cvCopy( timg, tgray, 0 );

        // try several threshold levels
        for( l = 0; l < N; l++ )
        {
            // hack: use Canny instead of zero threshold level.
            // Canny helps to catch squares with gradient shading
            if( l == 0 )
            {
                // apply Canny. Take the upper threshold from slider
                // and set the lower to 0 (which forces edges merging)
                cvCanny( tgray, gray, 0, thresh, 5 );
                // dilate canny output to remove potential
                // holes between edge segments
                cvDilate( gray, gray, 0, 1 );
            }
            else
            {
                // apply threshold if l!=0:
                //     tgray(x,y) = gray(x,y) < (l+1)*255/N ? 255 : 0
                cvThreshold( tgray, gray, (l+1)*255/N, 255, CV_THRESH_BINARY );
            }

            // find contours and store them all as a list
            cvFindContours( gray, storage, &contours, sizeof(CvContour),
                CV_RETR_LIST, CV_CHAIN_APPROX_SIMPLE, cvPoint(0,0) );

            // test each contour
            while( contours )
            {
                // approximate contour with accuracy proportional
                // to the contour perimeter
                result = cvApproxPoly( contours, sizeof(CvContour), storage,
                    CV_POLY_APPROX_DP, cvContourPerimeter(contours)*0.02, 0 );
                // square contours should have 4 vertices after approximation
                // relatively large area (to filter out noisy contours)
                // and be convex.
                // Note: absolute value of an area is used because
                // area may be positive or negative - in accordance with the
                // contour orientation
                if( result->total == 4 &&
                    fabs(cvContourArea(result,CV_WHOLE_SEQ)) > 1000 &&
                    cvCheckContourConvexity(result) )
                {
                    s = 0;
```

```
                    for( i = 0; i < 5; i++ )
                    {
                        // find minimum angle between joint
                        // edges (maximum of cosine)
                        if( i >= 2 )
                        {
                            t = fabs(angle(
                            (CvPoint*)cvGetSeqElem( result, i ),
                            (CvPoint*)cvGetSeqElem( result, i-2 ),
                            (CvPoint*)cvGetSeqElem( result, i-1 )));
                            s = s > t ? s : t;
                        }
                    }

                    // if cosines of all angles are small
                    // (all angles are ~90 degree) then write quandrange
                    // vertices to resultant sequence
                    if( s < 0.3 )
                        for( i = 0; i < 4; i++ )
                            cvSeqPush( squares,
                                (CvPoint*)cvGetSeqElem( result, i ));
                }

                // take the next contour
                contours = contours->h_next;
            }
        }
    }

    // release all the temporary images
    cvReleaseImage( &gray );
    cvReleaseImage( &pyr );
    cvReleaseImage( &tgray );
    cvReleaseImage( &timg );

    return squares;
}


// the function draws all the squares in the image
void drawSquares( IplImage* img, CvSeq* squares )
{
    CvSeqReader reader;
    IplImage* cpy = cvCloneImage( img );
    int i;

    // initialize reader of the sequence
    cvStartReadSeq( squares, &reader, 0 );

    // read 4 sequence elements at a time (all vertices of a square)
    for( i = 0; i < squares->total; i += 4 )
    {
        CvPoint* rect = pt;
        int count = 4;

        // read 4 vertices
        memcpy( pt, reader.ptr, squares->elem_size );
        CV_NEXT_SEQ_ELEM( squares->elem_size, reader );
        memcpy( pt + 1, reader.ptr, squares->elem_size );
        CV_NEXT_SEQ_ELEM( squares->elem_size, reader );
        memcpy( pt + 2, reader.ptr, squares->elem_size );
        CV_NEXT_SEQ_ELEM( squares->elem_size, reader );
        memcpy( pt + 3, reader.ptr, squares->elem_size );
        CV_NEXT_SEQ_ELEM( squares->elem_size, reader );

        // draw the square as a closed polyline
        cvPolyLine( cpy, &rect, &count, 1, 1, CV_RGB(0,255,0), 3, CV_AA, 0 );
    }

    // show the resultant image
    cvShowImage( wndname, cpy );
    cvReleaseImage( &cpy );
}


void on_trackbar( int a )
{
    if( img )
        drawSquares( img, findSquares4( img, storage ) );
}
```

```
char* names[] = { "pic1.png", "pic2.png", "pic3.png",
                  "pic4.png", "pic5.png", "pic6.png", 0 };

int main(int argc, char** argv)
{
    int i, c;
    // create memory storage that will contain all the dynamic data
    storage = cvCreateMemStorage(0);

    for( i = 0; names[i] != 0; i++ )
    {
        // load i-th image
        img0 = cvLoadImage( names[i], 1 );
        if( !img0 )
        {
            printf("Couldn't load %s\n", names[i] );
            continue;
        }
        img = cvCloneImage( img0 );

        // create window and a trackbar (slider) with parent "image" and set callback
        // (the slider regulates upper threshold, passed to Canny edge detector)
        cvNamedWindow( wndname, 1 );
        cvCreateTrackbar( "canny thresh", wndname, &thresh, 1000, on_trackbar );

        // force the image processing
        on_trackbar(0);
        // wait for key.
        // Also the function cvWaitKey takes care of event processing
        c = cvWaitKey(0);
        // release both images
        cvReleaseImage( &img );
        cvReleaseImage( &img0 );
        // clear memory storage - reset free space position
        cvClearMemStorage( storage );
        if( c == 27 )
            break;
    }

    cvDestroyWindow( wndname );

    return 0;
}

#ifdef _EiC
main(1,"squares.c");
#endif
```


如何用 **OpenCV** 来定标摄像机

可以使用\OpenCV\samples\c目录下的calibration.cpp这个程序,程序的输入支持USB摄像机,avi文件或者图片
1。使用说明

a.输入为图片时:
// example command line (for copy-n-paste):
// calibration -w 6 -h 8 -s 2 -n 10 -o camera.yml -op -oe [<list_of_views.txt>]

```
/* The list of views may look as following (discard the starting and ending ------ separators):
--------------------
view000.png
view001.png
#view002.png
view003.png
view010.png
one_extra_view.jpg
--------------------
```

that is, the file will contain 6 lines, view002.png will not be used for calibration,
other ones will be (those, in which the chessboard pattern will be found)

b.输入为摄像机或者avi文件时

```
"When the live video from camera is used as input, the following hot-keys may be used:\n"
    "  <ESC>, 'q' - quit the program\n"
    "  'g' - start capturing images\n"
    "  'u' - switch undistortion on/off\n";
```

c。输入参数说明

```
"Usage: calibration\n"
"     -w <board_width>         # the number of inner corners per one of board dimension\n"
"     -h <board_height>        # the number of inner corners per another board dimension\n"
"     [-n <number_of_frames>]  # the number of frames to use for calibration\n"
"                              # (if not specified, it will be set to the number\n"
"                              #  of board views actually available)\n"
"     [-d <delay>]             # a minimum delay in ms between subsequent attempts to capture a next view\n"
"                              # (used only for video capturing)\n"
"     [-s <square_size>]       # square size in some user-defined units (1 by default)\n"
"     [-o <out_camera_params>] # the output filename for intrinsic [and extrinsic] parameters\n"
"     [-op]                    # write detected feature points\n"
"     [-oe]                    # write extrinsic parameters\n"
"     [-zt]                    # assume zero tangential distortion\n"
"     [-a <aspect_ratio>]      # fix aspect ratio (fx/fy)\n"
"     [-p]                     # fix the principal point at the center\n"
"     [-v]                     # flip the captured images around the horizontal axis\n"
"     [input_data]             # input data, one of the following:\n"
"                              #  - text file with a list of the images of the board\n"
"                              #  - name of video file with a video of the board\n"
"                              # if input_data not specified, a live view from the camera is used\n"
```

2.经多次使用发现,不指定 -p参数时计算的结果误差较大,主要表现在对u0,v0的估计误差较大,因此建议使用时加上-p参数

[编辑]

中文翻译者

- 于仕琪,中科院自动化所自由软件协会
- HUNNISH,阿须数码

CxCore中文参考手册

Wikipedia，自由的百科全书

[编辑]

CxCore库的框架概述

为使得OpenCV的整个库便于管理和扩充，将整个库分成若干子库，CxCore是最重要的一个子库，从“core核心"名字可以看出，该库提供了 所有OpenCV运行时的一些最基本的数据结构，包括矩阵，数组的基本运算，包括出错处理的一些基本函数。具体分为下面若干部分。

1. 基础结构
2. 数组操作
3. 动态结构
4. 绘图函数
5. 数据保存和运行时类型信息
6. 其它混合函数
7. 错误处理和系统函数

Cxcore基础结构

Wikipedia，自由的百科全书

目录

[隐藏]

[编辑]

CvPoint

二维坐标系下的点，类型为整型

```
typedef struct CvPoint
 {
  int x; /* X坐标, 通常以0为基点 */
  int y; /* y坐标, 通常以0为基点 */
 }
CvPoint;

/* 构造函数 */
inline CvPoint cvPoint( int x, int y );

/* 从 CvPoint2D32f类型转换得来 */
inline CvPoint cvPointFrom32f( CvPoint2D32f point )
```

ddd

[编辑]

CvPoint2D32f

二维坐标下的点，类型为浮点

```
typedef struct CvPoint2D32f
 {
  float x; /* X坐标, 通常以0为基点*/
  float y; /* Y坐标, 通常以0为基点*/
 }
CvPoint2D32f;

/* 构造函数 */
inline CvPoint2D32f cvPoint2D32f( double x, double y );
```

```
/* 从CvPoint转换来 */
inline CvPoint2D32f cvPointTo32f( CvPoint point );
```

[编辑]

CvPoint3D32f

三维坐标下的点，类型为浮点

```
typedef struct CvPoint3D32f
 {
  float x; /* x-坐标，通常基于0 */
  float y; /* y-坐标，通常基于0 */
  float z; /* z-坐标，通常基于0 */
 }
 CvPoint3D32f;

/* 构造函数 */
inline CvPoint3D32f cvPoint3D32f( double x, double y, double z );
```

[编辑]

CvSize

矩形框大小，以像素为精度

```
typedef struct CvSize
 {
  int width;  /* 矩形宽 */
  int height; /* 矩形高 */
 }
 CvSize;

/* 构造函数 */
inline CvSize cvSize( int width, int height );
```

注意：构造函数的cv是小写！

[编辑]

CvSize2D32f

以亚像素精度标量矩形框大小

```
typedef struct CvSize2D32f
 {
   float width;  /* 矩形宽 */
   float height; /* 矩形高 */
 }
 CvSize2D32f;

/* 构造函数*/
inline CvSize2D32f cvSize2D32f( double width, double height );
{
    CvSize2D32f s;

    s.width = (float)width;
    s.height = (float)height;

    return s;
}
```

[编辑]

CvRect

矩形框的偏移和大小

```
typedef struct CvRect
 {
```

```
  int x; /* 方形的最左角的x-坐标  */
  int y; /* 方形的最上或者最下角的y-坐标 */
  int width; /* 宽 */
  int height; /* 高 */
 }
 CvRect;

/* 构造函数*/
inline CvRect cvRect( int x, int y, int width, int height );
   { CvRect os;

      os.x = x;
      os.y = y;
      os.width = width;
      os.height = heigth;

      reture os;}
```

[编辑]

CvScalar

可存放在1-，2-，3-，4-TUPLE类型的捆绑数据的容器

```
typedef struct CvScalar
 {
  double val[4]
 }
 CvScalar;

/* 构造函数: 用val0初始化val[0]用val1初始化val[1], 以此类推*/
inline CvScalar cvScalar( double val0, double val1,
                          double val2, double val3);

{  CvScalar  arr;

    arr.val[4] = {val0,val1,val2,val3};

    reture arr;}

/* 构造函数: 用val0123初始化所有val[0]...val[3] */
inline CvScalar cvScalarAll( double val0123 );

  { CvScalar arr;

     arr.val[4] = {val0123,val0123,val0123,val0123,};

     reture arr;}

/* 构造函数: 用val0初始化val[0],用0初始化val[1],val[2],val[3] */
inline CvScalar cvRealScalar( double val0 );

    { CvScalar arr;

       arr.val[4] = {val0};

       reture arr;}
```

http://doc.blueruby.mydns.jp/opencv/classes/OpenCV/CvScalar.html

[编辑]

CvTermCriteria

迭代算法的终止准则

```
#define CV_TERMCRIT_ITER    1
#define CV_TERMCRIT_NUMBER  CV_TERMCRIT_ITER
#define CV_TERMCRIT_EPS     2

typedef struct CvTermCriteria
 {
  int    type;  /* CV_TERMCRIT_ITER 和CV_TERMCRIT_EPS二值之一，或者二者的组合 */
  int    max_iter; /* 最大迭代次数 */
```

```
  double epsilon; /* 结果的精确性 */
 }
 CvTermCriteria;

/* 构造函数 */
inline  CvTermCriteria  cvTermCriteria( int type, int max_iter, double epsilon );

/* 在满足max_iter和epsilon的条件下检查终止准则并将其转换使得type=CV_TERMCRIT_ITER+CV_TERMCRIT_EPS */
CvTermCriteria cvCheckTermCriteria( CvTermCriteria criteria,
                                    double default_eps,
                                    int default_max_iters );
```

[编辑]

CvMat

多通道矩阵

```
typedef struct CvMat
 {
  int type; /* CvMat 标识 (CV_MAT_MAGIC_VAL), 元素类型和标记 */
  int step; /* 以字节为单位的行数据长度*/
  int* refcount; /* 数据引用计数 */
  union
   {
    uchar* ptr;
    short* s;
    int* i;
    float* fl;
    double* db;
   } data; /* data 指针 */
  #ifdef __cplusplus
  union
   {
     int rows;
     int height;
   };
  union
   {
     int cols;
     int width;
   };
  #else
   int rows; /* 行数 */
   int cols; /* 列数*/
  #endif
 } CvMat;
```

[编辑]

CvMatND

多维、多通道密集数组

```
typedef struct CvMatND
    {
        int type; /* CvMatND 标识(CV_MATND_MAGIC_VAL), 元素类型和标号*/
        int dims; /* 数组维数 */

        int* refcount; /* 数据参考计数 */

        union
        {
            uchar* ptr;
            short* s;
            int* i;
            float* fl;
            double* db;
        } data; /* data 指针*/

        /* 每维的数据结构 (元素号,以字节为单位的元素之间的距离)是配套定义的 */
        struct
        {
            int size;
            int step;
        }
```

```
        dim[CV_MAX_DIM];
    } CvMatND;
```

[编辑]

CvSparseMat

多维、多通道稀疏数组

```
typedef struct CvSparseMat
    {
        int type; /* CvSparseMat 标识 (CV_SPARSE_MAT_MAGIC_VAL), 元素类型和标号 */
        int dims; /* 维数 */
        int* refcount; /* 参考数量 - 未用 */
        struct CvSet* heap; /* HASH表节点池 */
        void** hashtable; /* HASH表:每个入口有一个节点列表，有相同的 "以HASH大小为模板的HASH值" */
        int hashsize; /* HASH表大小 */
        int total; /* 稀疏数组的节点数 */
        int valoffset; /* 数组节点值在字节中的偏移 */
        int idxoffset; /* 数组节点索引在字节中的偏移 */
        int size[CV_MAX_DIM]; /*维大小 */

    } CvSparseMat;
```

[编辑]

IplImage

IPL 图像头

```
typedef struct _IplImage
    {
        int  nSize;         /* IplImage大小，=sizeof(IplImage)*/
        int  ID;            /* 版本 (=0)*/
        int  nChannels;     /* 大多数OPENCV函数支持1,2,3 或 4 个通道 */
        int  alphaChannel;  /* 被OpenCV忽略 */
        int  depth;         /* 像素的位深度: IPL_DEPTH_8U, IPL_DEPTH_8S, IPL_DEPTH_16U,
                               IPL_DEPTH_16S, IPL_DEPTH_32S, IPL_DEPTH_32F and IPL_DEPTH_64F 可支
持 */
        char colorModel[4]; /* 被OpenCV忽略 */
        char channelSeq[4]; /* 被OpenCV忽略 */
        int  dataOrder;     /* 0 - 交叉存取颜色通道，对三通道RGB图像，像素存储顺序为BGR BGR BGR ... BGR;
                                   1 - 分开的颜色通道，对三通道RGB图像，像素存储顺序为RRR...R GGG...G
BBB...B。
                               cvCreateImage只能创建交叉存取图像 */
        int  origin;        /* 0 - 顶-左结构,
                               1 - 底-左结构 (Windows bitmaps 风格) */
        int  align;         /* 图像行排列 (4 or 8). OpenCV 忽略它，使用 widthStep 代替 */
        int  width;         /* 图像宽像素数 */
        int  height;        /* 图像高像素数*/
        struct _IplROI *roi;/* 图像感兴趣区域. 当该值非空只对该区域进行处理 */
        struct _IplImage *maskROI; /* 在 OpenCV中必须置NULL */
        void  *imageId;     /* 同上*/
        struct _IplTileInfo *tileInfo; /*同上*/
        int  imageSize;     /* 图像数据大小(在交叉存取格式下imageSize=image->height*image-
>widthStep)，单位字节*/
        char *imageData;  /* 指向排列的图像数据 */
        int  widthStep;   /* 排列的图像行大小，以字节为单位 */
        int  BorderMode[4]; /* 边际结束模式，被OpenCV忽略 */
        int  BorderConst[4]; /* 同上 */
        char *imageDataOrigin; /* 指针指向一个不同的图像数据结构（不是必须排列的），是为了纠正图像内存分配准备
的 */
    }
    IplImage;
```

IplImage结构来自于 Intel Image Processing Library（是其本身所具有的）。OpenCV 只支持其中的一个子集:

- alphaChannel 在OpenCV中被忽略。
- colorModel 和channelSeq 被OpenCV忽略。OpenCV颜色转换的唯一函数 cvCvtColor把原图像的颜色空间的目标图像的颜色空间作为一个参数。
- dataOrder 必须是IPL_DATA_ORDER_PIXEL (颜色通道是交叉存取)，然而平面图像的被选择通道可以被

处理，就像COI（感兴趣的通道）被设置过一样。
- align 是被OpenCV忽略的，而用 widthStep 去访问后继的图像行。
- 不支持maskROI 。处理MASK的函数把他当作一个分离的参数。MASK在 OpenCV 里是 8-bit，然而在IPL他是 1-bit。
- tileInfo 不支持。
- BorderMode和BorderConst是不支持的。每个 OpenCV 函数处理像素的邻近的像素，通常使用单一的固定代码边际模式。

除了上述限制，OpenCV处理ROI有不同的要求。要求原图像和目标图像的尺寸或 ROI的尺寸必须（根据不同的操作，例如cvPyrDown 目标图像的宽（高）必须等于原图像的宽（高）除以2 ±1)精确匹配，而IPL处理交叉区域，如图像的大小或ROI大小可能是完全独立的。

[编辑]

CvArr

不确定数组

```
typedef void CvArr;
```

CvArr* 仅仅是被用于作函数的参数，用于指示函数接收的数组类型可以不止一个，如 IplImage*, CvMat* 甚至 CvSeq*. 最终的数组类型是在运行时通过分析数组头的前4 个字节判断。

Cxcore数组操作

Wikipedia，自由的百科全书

目录

[隐藏]

[编辑]

初始化

[编辑]

CreateImage

创建头并分配数据

```
IplImage* cvCreateImage( CvSize size, int depth, int channels );
```

size
　　图像宽、高.
depth
　　图像元素的位深度，可以是下面的其中之一：

　　IPL_DEPTH_8U - 无符号8位整型
　　IPL_DEPTH_8S - 有符号8位整型
　　IPL_DEPTH_16U - 无符号16位整型
　　IPL_DEPTH_16S - 有符号16位整型
　　IPL_DEPTH_32S - 有符号32位整型
　　IPL_DEPTH_32F - 单精度浮点数
　　IPL_DEPTH_64F - 双精度浮点数

channels
　　每个元素（像素）的颜色通道数量.可以是 1, 2, 3 或 4.通道是交叉存取的，例如通常的彩色图像数据排列是：

　　b0 g0 r0 b1 g1 r1 ...

　　虽然通常 IPL 图象格式可以存贮非交叉存取的图像，并且一些OpenCV 也能处理他, 但是这个函数只能创建交叉存取图像.

函数 cvCreateImage 创建头并分配数据，这个函数是下列的缩写型式

```
header = cvCreateImageHeader(size,depth,channels);
cvCreateData(header); //只是创建空间，并不会初始化空间内的数据
```


CreateImageHeader

分配，初始化，并且返回 IplImage结构

```
IplImage* cvCreateImageHeader( CvSize size, int depth, int channels );
```

size
　　图像宽、高.
depth
　　像深 (见 CreateImage).
channels
　　通道数 (见 CreateImage).

函数 cvCreateImageHeader 分配, 初始化, 并且返回 IplImage结构. 这个函数相似于：

```
  iplCreateImageHeader( channels, 0, depth,
                        channels == 1 ? "GRAY" : "RGB",
                        channels == 1 ? "GRAY" : channels == 3 ? "BGR" :
                        channels == 4 ? "BGRA" : "",
                        IPL_DATA_ORDER_PIXEL, IPL_ORIGIN_TL, 4,
                        size.width, size.height,
                        0,0,0,0);
```

然而IPL函数不是作为默认的 (见 CV_TURN_ON_IPL_COMPATIBILITY 宏)

[编辑]

ReleaseImageHeader

释放头

```
void cvReleaseImageHeader( IplImage** image );
```

image
　　双指针指向头内存分配单元.

函数 cvReleaseImageHeader 释放头. 相似于

```
if( image )
{
    iplDeallocate( *image, IPL_IMAGE_HEADER | IPL_IMAGE_ROI );
    *image = 0;
}
```

然而IPL函数不是作为默认的 (见 CV_TURN_ON_IPL_COMPATIBILITY 宏)

[编辑]

ReleaseImage

释放头和图像数据

```
void cvReleaseImage( IplImage** image );
```

image
　　双指针指向图像内存分配单元。

函数 cvReleaseImage 释放头和图像数据，相似于:

```
if( *image )
{
    cvReleaseData( *image );
    cvReleaseImageHeader( image );
}
```

[编辑]

InitImageHeader

初始化被用图分配的图像头

```
IplImage* cvInitImageHeader( IplImage* image, CvSize size, int depth,
                             int channels, int origin=0, int align=4 );
```

image
　　被初始化的图像头.
size
　　图像的宽高.
depth
　　像深(见 CreateImage).
channels
　　通道数(见 CreateImage).
origin
　　IPL_ORIGIN_TL 或 IPL_ORIGIN_BL.
align
　　图像行排列, 典型的 4 或 8 字节.

函数 cvInitImageHeader 初始化图像头结构, 指向用户指定的图像并且返回这个指针。

[编辑]

CloneImage

制作图像的完整拷贝

```
IplImage* cvCloneImage( const IplImage* image );
```

image
　　原图像.

函数 cvCloneImage 制作图像的完整拷贝包括头、ROI和数据

[编辑]

SetImageCOI

基于给定的值设置感兴趣通道

```
void cvSetImageCOI( IplImage* image, int coi );
```

image
　　图像头.
coi
　　感兴趣通道.

函数 cvSetImageCOI 基于给定的值设置感兴趣的通道。值 0 意味着所有的通道都被选定, 1 意味着第一个通道被选定等等。如果 ROI 是 NULL 并且COI!= 0, ROI 被分配. 然而大多数的 OpenCV 函数不支持 COI, 对于这种状况当处理分离图像/矩阵通道时，可以拷贝（通过 cvCopy 或cvSplit) 通道来分离图像/矩阵，处理后如果需要可再拷贝（通过cvCopy 或 cvCvtPlaneToPix）回来.

[编辑]

GetImageCOI

返回感兴趣通道号

```
int cvGetImageCOI( const IplImage* image );
```

image
　　图像头.

函数cvGetImageCOI 返回图像的感兴趣通道（当所有的通道都被选中返回值是0).

[编辑]

SetImageROI

基于给定的矩形设置'感兴趣'区域

```
void cvSetImageROI( IplImage* image, CvRect rect );
```

image
　　图像.
rect
　　ROI 矩形.

函数 cvSetImageROI 基于给定的矩形设置图像的 ROI（感兴趣区域）. 如果ROI是NULL 并且参数RECT的值不等于整个图像, ROI被分配. 不像 COI, 大多数的 OpenCV 函数支持 ROI 并且处理它就像它是一个分离的图像

(例如, 所有的像素坐标从ROI的左上角或左下角（基于图像的结构）计算。

[编辑]

ResetImageROI

释放图像的ROI

```
void cvResetImageROI( IplImage* image );
```

image
　　图像头.

函数 cvResetImageROI 释放图像 ROI. 释放之后整个图像被认为是全部被选中的。相似的结果可以通过下述办法

```
cvSetImageROI( image, cvRect( 0, 0, image->width, image->height ));
cvSetImageCOI( image, 0 );
```

但是后者的变量不分配 image->roi.

[编辑]

GetImageROI

返回图像的 ROI 坐标

```
CvRect cvGetImageROI( const IplImage* image );
```

image
　　图像头.

函数 cvGetImageROI 返回图像ROI 坐标. 如果没有ROI则返回矩形值为 cvRect(0,0,image->width,image->height)

[编辑]

CreateMat

创建矩阵

```
CvMat* cvCreateMat( int rows, int cols, int type );
```

rows
　　矩阵行数。
cols
　　矩阵列数。
type
　　矩阵元素类型。 通常以 CV_<比特数>(S|U|F)C<通道数>型式描述， 例如:

　　`CV_8UC1` 意思是一个`8-bit` 无符号单通道矩阵, `CV_32SC2` 意思是一个`32-bit` 有符号二个通道的矩阵。

函数 cvCreateMat 为新的矩阵分配头和下面的数据，并且返回一个指向新创建的矩阵的指针。是下列操作的缩写型式:

```
CvMat* mat = cvCreateMatHeader( rows, cols, type );
cvCreateData( mat );
```

矩阵按行存贮。所有的行以4个字节对齐。

[编辑]

CreateMatHeader

创建新的矩阵头

```
CvMat* cvCreateMatHeader( int rows, int cols, int type );
```

rows

矩阵行数.

cols
矩阵列数.

type
矩阵元素类型(见 cvCreateMat).

函数 cvCreateMatHeader 分配新的矩阵头并且返回指向它的指针. 矩阵数据可被进一步的分配，使用cvCreateData 或通过 cvSetData明确的分配数据.

[编辑]

ReleaseMat

删除矩阵

```
void cvReleaseMat( CvMat** mat );
```

mat
双指针指向矩阵.

函数cvReleaseMat 缩减矩阵数据参考计数并且释放矩阵头：

```
if( *mat )
    cvDecRefData( *mat );
cvFree( (void**)mat );
```

[编辑]

InitMatHeader

初始化矩阵头

```
CvMat* cvInitMatHeader( CvMat* mat, int rows, int cols, int type,
                        void* data=NULL, int step=CV_AUTOSTEP );
```

mat
指针指向要被初始化的矩阵头.

rows
矩阵的行数.

cols
矩阵的列数.

type
矩阵元素类型.

data
可选的，将指向数据指针分配给矩阵头.

step
排列后的数据的整个行宽，默认状态下，使用STEP的最小可能值。也就是说默认情况下假定矩阵的行与行之间无隙.

函数 cvInitMatHeader 初始化已经分配了的 CvMat 结构. 它可以被OpenCV矩阵函数用于处理原始数据。

例如, 下面的代码计算通用数组格式存贮的数据的矩阵乘积.

计算两个矩阵的积

```
double a[] = { 1, 2, 3, 4,
               5, 6, 7, 8,
               9, 10, 11, 12 };

double b[] = { 1, 5, 9,
               2, 6, 10,
               3, 7, 11,
               4, 8, 12 };

double c[9];
CvMat Ma, Mb, Mc ;

cvInitMatHeader( &Ma, 3, 4, CV_64FC1, a );
cvInitMatHeader( &Mb, 4, 3, CV_64FC1, b );
cvInitMatHeader( &Mc, 3, 3, CV_64FC1, c );

cvMatMulAdd( &Ma, &Mb, 0, &Mc );
// c 数组存贮 a(3x4) 和 b(4x3) 矩阵的积
```

[编辑]

Mat

初始化矩阵的头(轻磅变量)

```
CvMat cvMat( int rows, int cols, int type, void* data=NULL );
```

rows
: 矩阵行数

cols
: 列数.

type
: 元素类型(见CreateMat).

data
: 可选的分配给矩阵头的数据指针 .

函数 cvMat 是个一快速内连函数，替代函数 cvInitMatHeader. 也就是说它相当于:

```
CvMat mat;
cvInitMatHeader( &mat, rows, cols, type, data, CV_AUTOSTEP );
```

[编辑]

CloneMat

创建矩阵拷贝

```
CvMat* cvCloneMat( const CvMat* mat );
```

mat
: 输入矩阵.

函数 cvCloneMat 创建输入矩阵的一个拷贝并且返回 该矩阵的指针.

[编辑]

CreateMatND

创建多维密集数组

```
CvMatND* cvCreateMatND( int dims, const int* sizes, int type );
```

dims
数组维数. 但不许超过 CV_MAX_DIM (默认=32，但这个默认值可能在编译时被改变)的定义
sizes
数组的维大小.
type
数组元素类型. 与 CvMat相同

函数cvCreateMatND 分配头给多维密集数组并且分配下面的数据，返回指向被创建数组的指针 . 是下列的缩减形式:

```
CvMatND* mat = cvCreateMatNDHeader( dims, sizes, type );
cvCreateData( mat );
```

矩阵按行存贮. 所有的行以4个字节排列。.

[编辑]

CreateMatNDHeader

创建新的数组头

```
CvMatND* cvCreateMatNDHeader( int dims, const int* sizes, int type );
```

dims
数组维数.
sizes
维大小.
type
数组元素类型. 与 CvMat相同

函数cvCreateMatND 分配头给多维密集数组。数组数据可以用 cvCreateData 进一步的被分配或利用cvSetData由用户明确指定.

[编辑]

ReleaseMatND

删除多维数组

```
void cvReleaseMatND( CvMatND** mat );
```

mat
指向数组的双指针.

函数 cvReleaseMatND 缩减数组参考计数并释放数组头:

```
if( *mat )
    cvDecRefData( *mat );
cvFree( (void**)mat );
```

[编辑]

InitMatNDHeader

初始化多维数组头

```
CvMatND* cvInitMatNDHeader( CvMatND* mat, int dims, const int* sizes, int type, void* data=NULL
);
```

mat
指向要被出初始化的数组头指针.

dims
数组维数.
sizes
维大小.
type
数组元素类型. 与 CvMat相同
data
可选的分配给矩阵头的数据指针.

函数 cvInitMatNDHeader 初始化 用户指派的CvMatND 结构.

[编辑]

CloneMatND

创建多维数组的完整拷贝

```
CvMatND* cvCloneMatND( const CvMatND* mat );
```

mat
输入数组

函数 cvCloneMatND 创建输入数组的拷贝并返回指针.

[编辑]

DecRefData

缩减数组数据的引用计数

```
void cvDecRefData( CvArr* arr );
```

arr
数组头.

如果引用计数指针非NULL,函数 cvDecRefData 缩减CvMat 或CvMatND 数据的引用计数，如果计数到0就删除数据。在当前的版本中只有当数据是用cvCreateData 分配的引用计数才会是非NULL。在其他的情况下比如:

- 使用cvSetData指派外部数据给矩阵头;
- 代表部分大的矩阵或图像的矩阵头;
- 是从图像头或N维矩阵头转换过来的矩阵头,

在这些情况下引用计数被设置成NULL因此不会被缩减。 无论数据是否被删除,数据指针和引用计数指针都将被这个函数清空。

[编辑]

IncRefData

增加数组数据的引用计数

```
int cvIncRefData( CvArr* arr );
```

arr
数组头.

函数 cvIncRefData 增加 CvMat 或 CvMatND 数据引用计数，如果引用计数非空返回新的计数值 否则返回0。

[编辑]

CreateData

分配数组数据

```
void cvCreateData( CvArr* arr );
```

arr
　　数组头.

函数 cvCreateData 分配图像，矩阵或多维数组数据. 对于矩阵类型使用OpenCV的分配函数，对于 IplImage类型如果CV_TURN_ON_IPL_COMPATIBILITY没有被调用也是可以使用这种方法的反之使用 IPL 函数分配数据

[编辑]

ReleaseData

释放数组数据

```
void cvReleaseData( CvArr* arr );
```

arr
　　数组头

函数cvReleaseData 释放数组数据. 对于 CvMat 或 CvMatND 结构只需调用 cvDecRefData(), 也就是说这个函数不能删除外部数据。见 cvCreateData.

[编辑]

SetData

指派用户数据给数组头

```
void cvSetData( CvArr* arr, void* data, int step );
```

arr
　　数组头.
data
　　用户数据.
step
　　整行字节长.

函数cvSetData 指派用户数据给数组头. 头应该已经使用 cvCreate*Header, cvInit*Header 或 cvMat (对于矩阵)初始化过.

[编辑]

GetRawData

返回数组的底层信息

```
void cvGetRawData( const CvArr* arr, uchar** data,
                   int* step=NULL, CvSize* roi_size=NULL );
```

arr
　　数组头.
data
　　输出指针，指针指向整个图像的结构或ROI
step
　　输出行字节长

roi_size
输出ROI尺寸

函数 cvGetRawData 添充给输出变量数组的底层信息。所有的输出参数是可选的， 因此这些指针可设为NULL。如果数组是设置了ROI的 IplImage 结构， ROI参数被返回。

注意：输出指针指向数组头的对应的内存，不能释放。建议用memcpy。

接下来的例子展示怎样利用这个函数去访问数组元素。

使用 GetRawData 计算单通道浮点数组的元素绝对值。

```
float* data;
int step;

CvSize size;
int x, y;

cvGetRawData( array, (uchar**)&data, &step, &size );
step /= sizeof(data[0]);

for( y = 0; y < size.height; y++, data += step )
    for( x = 0; x < size.width; x++ )
        data[x] = (float)fabs(data[x]);
```

[编辑]

GetMat

从不确定数组返回矩阵头

```
CvMat* cvGetMat( const CvArr* arr, CvMat* header, int* coi=NULL, int allowND=0 );
```

arr
输入数组.
header
指向 CvMat结构的指针，作为临时缓存 .
coi
可选的输出参数，用于输出COI.
allowND
如果非0，函数就接收多维密集数组 (CvMatND*)并且返回 2D (如果 CvMatND 是二维的）或 1D 矩阵(当 CvMatND 是一维或多于二维). 数组必须是连续的.

函数 cvGetMat从输入的数组生成矩阵头，输入的数组可以是 - CvMat结构, IplImage结构 或多维密集数组 CvMatND* (后者只有当 allowND != 0时才可以使用) . 如果是矩阵函数只是返回指向矩阵的指针.如果是 IplImage* 或 CvMatND* 函数用当前图像的ROI初始化头结构并且返回指向这个临时结构的指针。因为CvMat不支持COI，所以他们的返回结果是不同的.

这个函数提供了一个简单的方法，用同一代码处理 IplImage 和 CvMat二种数据类型。这个函数的反向转换可以用 cvGetImage将 CvMat 转换成 IplImage .

输入的数组必须有已分配好的底层数据或附加的数据，否则该函数将调用失败 如果输入的数组是IplImage 格式，使用平面式数据编排并设置了COI，函数返回的指针指向被选定的平面并设置COI=0.利用OPENCV函数对于多通道平面编排图像可以处理每个平面。

[编辑]

GetImage

从不确定数组返回图像头

```
IplImage* cvGetImage( const CvArr* arr, IplImage* image header );
```

arr
　　输入数组.
image_header
　　指向IplImage结构的指针，该结构存贮在一个临时缓存 .

函数 cvGetImage 从输出数组获得图头，该数组可以是矩阵- CvMat*, 或图像 - IplImage*。 如果是图像的话函数只是返回输入参数的指针，如果是 CvMat* 的话函数用输入参数矩阵初始化图像头。因此如果我们把 IplImage 转换成 CvMat 然后再转换 CvMat 回 IplImage,如果ROI被设置过了我们可能会获得不同的头，这样一些计算图像跨度的IPL函数就会失败。

[编辑]

CreateSparseMat

创建稀疏数组

```
CvSparseMat* cvCreateSparseMat( int dims, const int* sizes, int type );
```

dims
　　数组维数。相对于密集型矩阵，稀疏数组的维数是不受限制的（最多可达 2^{16}）。
sizes
　　数组的维大小。
type
　　数组元素类型，见 CvMat。

函数 cvCreateSparseMat 分配多维稀疏数组。刚初始化的数组不含元素，因此cvGet*D 或 cvGetReal*D函数对所有索引都返回0。

[编辑]

ReleaseSparseMat

删除稀疏数组

```
void cvReleaseSparseMat( CvSparseMat** mat );
```

mat
　　双指针指向数组。

函数 cvReleaseSparseMat释放稀疏数组并清空数组指针

[编辑]

CloneSparseMat

创建稀疏数组的拷贝

```
CvSparseMat* cvCloneSparseMat( const CvSparseMat* mat );
```

mat
　　输入数组。

函数 cvCloneSparseMat 创建输入数组的拷贝并返回指向这个拷贝的指针。

[编辑]

获取元素和数组子集

[编辑]

GetSubRect

返回输入的图像或矩阵的矩形数组子集的矩阵头

```
CvMat* cvGetSubRect( const CvArr* arr, CvMat* submat, CvRect rect );
```

arr
　　输入数组。
submat
　　指向矩形数组子集矩阵头的指针。
rect
　　以0坐标为基准的ROI。

函数 cvGetSubRect 根据指定的数组矩形返回矩阵头，换句话说，函数允许像处理一个独立数组一样处理输入数组的一个指定子矩形。函数在处理时要考虑进输入数组的ROI，因此数组的ROI是实际上被提取的。

[编辑]

GetRow, GetRows

返回数组的一行或在一定跨度内的行

```
CvMat* cvGetRow( const CvArr* arr, CvMat* submat, int row );
CvMat* cvGetRows( const CvArr* arr, CvMat* submat, int start_row, int end_row, int delta_row=1 );
```

arr
　　输入数组。
submat
　　指向返回的子数组头的指针。
row
　　被选定行的索引下标，索引下标从0开始。
start_row
　　跨度的开始行（包括此行）索引下标，索引下标从0开始。
end_row
　　跨度的结束行（不包括此行）索引下标，索引下标从0开始。
delta_row
　　在跨度内的索引下标跨步，从开始行到结束行每隔delta_row行提取一行。

函数GetRow 和 GetRows 返回输入数组中指定的一行或在一定跨度内的行对应的数组头。 注意GetRow 实际上是 以下cvGetRows调用的简写:

```
cvGetRow( arr, submat, row ) ~ cvGetRows( arr, submat, row, row + 1, 1 );
```

[编辑]

GetCol, GetCols

返回数组的一列或一定跨度内的列

```
CvMat* cvGetCol( const CvArr* arr, CvMat* submat, int col );
CvMat* cvGetCols( const CvArr* arr, CvMat* submat, int start_col, int end_col );
```

arr
　　输入数组。
submat
　　指向结果子数组头的指针。
col
　　被选定列的索引下标，索引下标从0开始。

start_col
　　跨度的开始列（包括该列）索引下标，索引下标从0开始。
end_col
　　跨度的结束列（不包括该列）索引下标，索引下标从0开始。

函数 GetCol 和 GetCols 根据指定的列/列跨度返回对应的数组头。注意GetCol 实际上是以下 cvGetCols调用的简写形式:

```
cvGetCol( arr, submat, col ); // ~ cvGetCols( arr, submat, col, col + 1 );
```

[编辑]

GetDiag

返回一个数组对角线

```
CvMat* cvGetDiag( const CvArr* arr, CvMat* submat, int diag=0 );
```

arr
　　输入数组.
submat
　　指向结果子集的头指针.
diag
　　数组对角线。0是主对角线，-1是主对角线上面对角线，1是主对角线下对角线，以此类推。

函数 cvGetDiag 根据指定的diag参数返回数组的对角线头。

[编辑]

GetSize

返回矩阵或图像ROI的大小

```
CvSize cvGetSize( const CvArr* arr );
```

arr
　　数组头。

函数 cvGetSize 返回图像或矩阵的行数和列数，如果是图像就返回ROI的大小。

[编辑]

InitSparseMatIterator

初始化稀疏数组元素迭代器

```
CvSparseNode* cvInitSparseMatIterator( const CvSparseMat* mat,
                                       CvSparseMatIterator* mat_iterator );
```

mat
　　输入的数组.
mat_iterator
　　被初始化的迭代器.

函数 cvInitSparseMatIterator 初始化稀疏数组元素的迭代器并且返回指向第一个元素的指针，如果数组为空则返回NULL。

[编辑]

GetNextSparseNode

初始化稀疏数组元素迭代器

```
CvSparseNode* cvGetNextSparseNode( CvSparseMatIterator* mat_iterator );
```

mat_iterator
稀疏数组的迭代器

函数cvGetNextSparseNode 移动迭代器到下一个稀疏矩阵元素并返回指向他的指针。在当前的版本不存在任何元素的特殊顺序，因为元素是按HASH表存贮的下面的列子描述怎样在稀疏矩阵上迭代：

利用cvInitSparseMatIterator 和cvGetNextSparseNode 计算浮点稀疏数组的和。

```
double sum;
int i, dims = cvGetDims( array );
CvSparseMatIterator mat_iterator;
CvSparseNode* node = cvInitSparseMatIterator( array, &mat_iterator );

for( ; node != 0; node = cvGetNextSparseNode( &mat_iterator ))
{
    int* idx = CV_NODE_IDX( array, node ); /* get pointer to the element indices */
    float val = *(float*)CV_NODE_VAL( array, node ); /* get value of the element
                                                        (assume that the type is CV_32FC1) */
    printf( "(" );
    for( i = 0; i < dims; i++ )
        printf( "%4d%s", idx[i], i < dims - 1 ? "," : "): " );
    printf( "%g\n", val );

sum += val;


printf( "\nTotal sum = %g\n", sum );
```


GetElemType

返回数组元素类型

```
int cvGetElemType( const CvArr* arr );
```

arr
输入数组.

函数 GetElemType 返回数组元素类型就像在cvCreateMat 中讨论的一样:

CV_8UC1 ... CV_64FC4

GetDims, GetDimSize

返回数组维数和他们的大小或者特殊维的大小

```
int cvGetDims( const CvArr* arr, int* sizes=NULL );
int cvGetDimSize( const CvArr* arr, int index );
```

arr
输入数组.
sizes
可选的输出数组维尺寸向量，对于2D数组第一位是数组行数（高），第二位是数组列数（宽）
index
以0为基准的维索引下标（对于矩阵0意味着行数，1意味着列数，对于图象0意味着高，1意味着宽。

函数 cvGetDims 返回维数和他们的大小。如果是 IplImage 或 CvMat 总是返回2，不管图像/矩阵行数。函数 cvGetDimSize 返回特定的维大小（每维的元素数）。例如，接下来的代码使用二种方法计算数组元素总数。

```
// via cvGetDims()
int sizes[CV_MAX_DIM];
int i, total = 1;
int dims = cvGetDims( arr, size );
for( i = 0; i < dims; i++ )
    total *= sizes[i];

// via cvGetDims() and cvGetDimSize()
int i, total = 1;
int dims = cvGetDims( arr );
for( i = 0; i < dims; i++ )
    total *= cvGetDimsSize( arr, i );
```

[编辑]

Ptr*D

返回指向特殊数组元素的指针

```
uchar* cvPtr1D( const CvArr* arr, int idx0, int* type=NULL );
uchar* cvPtr2D( const CvArr* arr, int idx0, int idx1, int* type=NULL );
uchar* cvPtr3D( const CvArr* arr, int idx0, int idx1, int idx2, int* type=NULL );
uchar* cvPtrND( const CvArr* arr, int* idx, int* type=NULL, int create_node=1, unsigned*
precalc_hashval=NULL );
```

arr
: 输入数组.

idx0
: 元素下标的第一个以0为基准的成员

idx1
: 元素下标的第二个以0为基准的成员

idx2
: 元素下标的第三个以0为基准的成员

idx
: 数组元素下标

type
: 可选的，矩阵元素类型输出参数

create_node
: 可选的，为稀疏矩阵输入的参数。如果这个参数非零就意味着被需要的元素如果不存在就会被创建。

precalc_hashval
: 可选的，为稀疏矩阵设置的输入参数。如果这个指针非NULL，函数不会重新计算节点的HASH值，而是从指定位置获取。这种方法有利于提高智能组合数据的操作（TODO: 提供了一个例子)

函数cvPtr*D 返回指向特殊数组元素的指针。数组维数应该与转递给函数物下标数相匹配，除了 cvPtr1D 函数，它可以被用于顺序存取的1D，2D或nD密集数组

函数也可以用于稀疏数组，并且如果被需要的节点不存在函数可以创建这个节点并设置为0

就像其它获取数组元素的函数 (cvGet[Real]*D, cvSet[Real]*D)如果元素的下标超出了范围就会产生错误

[编辑]

Get*D

返回特殊的数组元素

```
CvScalar cvGet1D( const CvArr* arr, int idx0 );
CvScalar cvGet2D( const CvArr* arr, int idx0, int idx1 );
CvScalar cvGet3D( const CvArr* arr, int idx0, int idx1, int idx2 );
CvScalar cvGetND( const CvArr* arr, int* idx );
```

arr
: 输入数组.

idx0
元素下标第一个以0为基准的成员
idx1
元素下标第二个以0为基准的成员
idx2
元素下标第三个以0为基准的成员
idx
元素下标数组

函数cvGet*D 返回指定的数组元素。对于稀疏数组如果需要的节点不存在函数返回0 （不会创建新的节点）

[编辑]

GetReal*D

返回单通道数组的指定元素

```
double cvGetReal1D( const CvArr* arr, int idx0 );
double cvGetReal2D( const CvArr* arr, int idx0, int idx1 );
double cvGetReal3D( const CvArr* arr, int idx0, int idx1, int idx2 );
double cvGetRealND( const CvArr* arr, int* idx );
```

arr
输入数组，必须是单通道.
idx0
元素下标的第一个成员，以0为基准
idx1
元素下标的第二个成员，以0为基准
idx2
元素下标的第三个成员，以0为基准
idx
元素下标数组

函数cvGetReal*D 返回单通道数组的指定元素，如果数组是多通道的，就会产生运行时错误，而 cvGet*D 函数可以安全的被用于单通道和多通道数组，但他们运行时会有点慢

如果指定的点不存在对于稀疏数组点会返回0（不会创建新的节点）。

[编辑]

mGet

返回单通道浮点矩阵指定元素

```
double cvmGet( const CvMat* mat, int row, int col );
```

mat
输入矩阵.
row
行下标，以0为基点.
col
列下标，以0为基点

函数 cvmGet 是 cvGetReal2D对于单通道浮点矩阵的快速替代函数，函数运行比较快速因为它是内连函数 ，这个函数对于数组类型、数组元素类型的检查作的很少，并且仅在调式模式下检查数的行和列范围。

[编辑]

Set*D

修改指定的数组

```
void cvSet1D( CvArr* arr, int idx0, CvScalar value );
void cvSet2D( CvArr* arr, int idx0, int idx1, CvScalar value );
void cvSet3D( CvArr* arr, int idx0, int idx1, int idx2, CvScalar value );
void cvSetND( CvArr* arr, int* idx, CvScalar value );
```

arr
: 输入数组

idx0
: 元素下标的第一个成员，以0为基点

idx1
: 元素下标的第二个成员，以0为基点

idx2
: 元素下标的第三个成员，以0为基点

idx
: 元素下标数组

value
: 指派的值

函数 cvSet*D 指定新的值给指定的数组元素。对于稀疏矩阵如果指定节点不存在函数创建新的节点

[编辑]

SetReal*D

修改指定数组元素值

```
void cvSetReal1D( CvArr* arr, int idx0, double value );
void cvSetReal2D( CvArr* arr, int idx0, int idx1, double value );
void cvSetReal3D( CvArr* arr, int idx0, int idx1, int idx2, double value );
void cvSetRealND( CvArr* arr, int* idx, double value );
```

arr
: 输入数组.

idx0
: 元素下标的第一个成员，以0为基点

idx1
: 元素下标的第二个成员，以0为基点

idx2
: 元素下标的第三个成员，以0为基点

idx
: 元素下标数组

value
: 指派的值

函数 cvSetReal*D 分配新的值给单通道数组的指定元素，如果数组是多通道就会产生运行时错误。然而cvSet*D 可以安全的被用于多通道和单通道数组，只是稍微有点慢。

对于稀疏数组如果指定的节点不存在函数会创建该节点。

[编辑]

mSet

为单通道浮点矩阵的指定元素赋值。

```
void cvmSet( CvMat* mat, int row, int col, double value );
```

mat

矩阵.

row
行下标,以0为基点.

col
列下标,以0为基点.

value
矩阵元素的新值

函数cvmSet 是cvSetReal2D 快速替代,对于单通道浮点矩阵因为这个函数是内连的所以比较快,函数对于数组类型、数组元素类型的检查作的很少，并且仅在调式模式下检查数的行和列范围。

[编辑]

ClearND

清除指定数组元素

```
void cvClearND( CvArr* arr, int* idx );
```

arr
输入数组.

idx
数组元素下标

函数cvClearND 清除指定密集型数组的元素(置0)或删除稀疏数组的元素 ,如果元素不存在函数不作任何事

[编辑]

拷贝和添加

[编辑]

Copy

拷贝一个数组给另一个数组

```
void cvCopy( const CvArr* src, CvArr* dst, const CvArr* mask=NULL );
```

src
输入数组。

dst
输出数组。

mask
操作掩码是8比特单通道的数组，它指定了输出数组中被改变的元素。

函数cvCopy从输入数组中复制选定的成分到输出数组:

如果mask(I)!=0,则dst(I)=src(I)。

如果输入输出数组中的一个是IplImage类型的话，其ROI和COI将被使用。输入输出数组必须是同样的类型、维数和大小。函数也可以用来复制散列数组（这种情况下不支持mask）。

[编辑]

Set

设置数组所有元素为指定值

```
void cvSet( CvArr* arr, CvScalar value, const CvArr* mask=NULL );
```

arr
　　输出数组。
value
　　填充值。
mask
　　操作掩码是8比特单通道的数组，它指定了输出数组中被改变的元素。

函数 cvSet 拷贝数量值到输出数组的每一个被除数选定的元素:

　　arr(I)=value if mask(I)!=0

如果数组 arr 是 IplImage 类型, 那么就会使用ROI,但COI不能设置。

[编辑]

SetZero

清空数组

```
void cvSetZero( CvArr* arr );
#define cvZero cvSetZero
```

arr
　　要被清空数组.

函数 cvSetZero 清空数组. 对于密集型号数组(CvMat, CvMatND or IplImage) cvZero(array) 就相当于 cvSet(array,cvScalarAll(0),0), 对于稀疏数组所有的元素都将被删除.

[编辑]

SetIdentity

初始化带尺度的单位矩阵

```
void cvSetIdentity( CvArr* mat, CvScalar value=cvRealScalar(1) );
```

mat
　　待初始化的矩阵 (不一定是方阵)。
value
　　赋值给对角线元素的值。

函数 cvSetIdentity 初始化带尺度的单位矩阵:

　　arr(i,j)=value 如果 i=j,
　　否则为 0

[编辑]

Range

用指定范围的数来填充矩阵.

```
void cvRange( CvArr* mat, double start, double end );
```

mat
　　即将被初始化的矩阵,必须是指向单通道的32位(整型或浮点型)的矩阵的指针.
start
　　指定范围的最小边界
end

指定范围的最大边界

该函数按以下方式初始化矩阵:

arr(i,j)=(end-start)*(i*cols(arr)+j)/(cols(arr)*rows(arr))

例如:以下的代码将按相应的整型数初始化一维向量:

CvMat* A = cvCreateMat(1, 10, CV_32S); cvRange(A, 0, A->cols); //A将被初始化为[0,1,2,3,4,5,6,7,8,9]

[编辑]

变换和置换

[编辑]

Reshape

不拷贝数据修改矩阵/图像形状

```
CvMat* cvReshape( const CvArr* arr, CvMat* header, int new_cn, int new_rows=0 );
```

arr
　　输入的数组.
header
　　被添充的矩阵头
new_cn
　　新的通道数.new_cn = 0 意味着不修改通道数
new_rows
　　新的行数. 如果new_rows = 0保持原行数不修改否则根据 new_cn 值修改输出数组

函数 cvReshape 初始化 CvMat 头header 以便于让头指向修改后的形状（但数据保持原样）-也就是说修改通道数,修改行数或者两者者改变.

例如, 接下来的代码创建一个图像缓存、两个图像头，第一个是 320x240x3 图像第二个是 960x240x1 图像:

```
IplImage* color_img = cvCreateImage( cvSize(320,240), IPL_DEPTH_8U, 3 );
CvMat gray_mat_hdr;
IplImage gray_img_hdr, *gray_img;
cvReshape( color_img, &gray_mat_hdr, 1 );
gray_img = cvGetImage( &gray_mat_hdr, &gray_img_hdr );
```

下一个例子转换3x3 矩阵成单向量 1x9

```
CvMat* mat = cvCreateMat( 3, 3, CV_32F );
CvMat row_header, *row;
row = cvReshape( mat, &row_header, 0, 1 );
```

[编辑]

ReshapeMatND

修改多维数组形状，拷贝/不拷贝数据

```
 CvArr* cvReshapeMatND( const CvArr* arr,
                        int sizeof_header, CvArr* header,
                        int new_cn, int new_dims, int* new_sizes );

#define cvReshapeND( arr, header, new_cn, new_dims, new_sizes )   \
        cvReshapeMatND( (arr), sizeof(*(header)), (header), \
                        (new_cn), (new_dims), (new_sizes))
```

arr

输入数组

sizeof_header
输出头的大小，对于IplImage, CvMat 和 CvMatND 各种结构输出的头均是不同的.

header
被添充的输出头.

new_cn
新的通道数，如果new_cn = 0 则通道数保持原样

new_dims
新的维数. 如果new_dims = 0 则维数保持原样。

new_sizes
新的维大小.只有当 new_dims=1值被使用，因为要保持数组的总数一致，因此如果 new_dims = 1, new_sizes 是不被使用的

函数cvReshapeMatND 是 cvReshape 的高级版本，它可以处理多维数组（能够处理通用的图像和矩阵）并且修改维数，下面的是使用cvReshapeMatND重写 cvReshape的二个例子：

```
IplImage* color_img = cvCreateImage( cvSize(320,240), IPL_DEPTH_8U, 3 );
IplImage gray_img_hdr, *gray_img;
gray_img = (IplImage*)cvReshapeND( color_img, &gray_img_hdr, 1, 0, 0 );

...

/*second example is modified to convert 2x2x2 array to 8x1 vector */
int size[] = { 2, 2, 2 };
CvMatND* mat = cvCreateMatND( 3, size, CV_32F );
CvMat row_header, *row;
row = cvReshapeND( mat, &row_header, 0, 1, 0 );
```


Repeat

用原数组管道式添充输出数组

```
void cvRepeat( const CvArr* src, CvArr* dst );
```

src
输入数组, 图像或矩阵。

dst
输出数组，图像或矩阵

函数cvRepeat 使用被管道化的原数组添充输出数组:

```
dst(i,j)=src(i mod rows(src), j mod cols(src))
```

因此 ，输出数组可能小于也可能大于输入数组.

Flip

垂直，水平或即垂直又水平翻转二维数组

```
void  cvFlip( const CvArr* src, CvArr* dst=NULL, int flip_mode=0);
#define cvMirror cvFlip
```

src
原数组.

dst
目标责任制数组. 如果 dst = NULL 翻转是在内部替换.

flip_mode
　　指定怎样去翻转数组。

　　flip_mode = 0 沿X-轴翻转, flip_mode > 0 (如 1) 沿Y-轴翻转， flip_mode < 0 (如 -1) 沿X-轴和Y-轴翻转.见下面的公式

函数cvFlip 以三种方式之一翻转数组 (行和列下标是以0为基点的):

```
dst(i,j)=src(rows(src)-i-1,j) if flip_mode = 0
dst(i,j)=src(i,cols(src1)-j-1) if flip_mode > 0
dst(i,j)=src(rows(src)-i-1,cols(src)-j-1) if flip_mode < 0
```

函数主要使用在:

- 垂直翻转图像(flip_mode = 0)用于 顶-左和底-左图像结构的转换, 主要用于WIN32系统下的视频操作处理.
- 水平图像转换，使用连续的水平转换和绝对值差检查垂直轴对称（flip_mode > 0)
- 水平和垂直同时转换，用于连续的水平转换和绝对真理值差检查中心对称s(flip_mode < 0)
- 翻转1维指针数组的顺序(flip_mode > 0)

[编辑]

Split

分割多通道数组成几个单通道数组或者从数组中提取一个通道

```
void cvSplit( const CvArr* src, CvArr* dst0, CvArr* dst1,
              CvArr* dst2, CvArr* dst3 );
#define cvCvtPixToPlane cvSplit
```

src
　　原数组.
dst0...dst3
　　目标通道

函数 cvSplit 分割多通道数组成分离的单通道数组d。可获得两种操作模式 . 如果原数组有N通道且前N输出数组非NULL, 所有的通道都会被从原数组中提取，如果前N个通道只有一个通道非NULL函数只提取该指定通道，否则会产生一个错误，余下的通道（超过前N个通道的以上 的）必须被设置成NULL，对于设置了COI的IplImage 结使用cvCopy 也可以从图像中提取单通道。

[编辑]

Merge

从几个单通道数组组合成多通道数组或插入一个单通道数组

```
void cvMerge( const CvArr* src0, const CvArr* src1,
              const CvArr* src2, const CvArr* src3, CvArr* dst );
#define cvCvtPlaneToPix cvMerge
```

src0... src3
　　输入的通道.
dst
　　输出数组.

函数cvMerge 是前一个函数的反向操作。如果输出数组有N个通道并且前N个输入通道非NULL，就拷贝所有通道到输出数组，如果在前N个通道中只有一个单通道非NULL ，只拷贝这个通道到输出数组，否则 就会产生错误。除前N通道以外的余下的通道必须置NULL。对于设置了COI的 IplImage结构使用 cvCopy也可以实现向图像中插入一个通道 。

[编辑]

MixChannels

拷贝输入数组的若干个通道到输出数组的某些通道上面.

```
void cvMixChannels( const CvArr** src, int src_count,
                    CvArr** dst, int dst_count,
                    const int* from_to, int pair_count );
```

src
　　输入数组
src_count
　　输入数组的个数
dst
　　输出数组
dst_count
　　输出数组的个数
from_to
　　对数的阵列
pair_count
　　from_to里面的对数的个数,或者说被拷贝的位面的个数.

[编辑]

RandShuffle

随机交换数组的元素

```
void cvRandShuffle( CvArr* mat, CvRNG* rng, double iter_factor=1.);
```

mat
　　输入的矩阵,用来被随机处理.
rng
　　随机数产生器用来随机交换数组元素.如果为NULL,一个当前的随机数发生器将被创建与使用.
iter_factor
　　相关的参数,用来刻划交换操作的密度.请看下面的说明.

这个函数在每个反复的操作中交换随机选择的矩阵里面的元素(在多通道的数组里面每个元素可能包括若干个部分),反复的次数(也就是交换的对数)等于 round(iter_factor*rows(mat)*cols(mat)), 因此如果iter_factor=0,没有交换产生,如果等于1意味着随机交换了rows(mat)*cols(mat)对数.

[编辑]

算术，逻辑和比较

[编辑]

LUT

利用查找表转换数组

```
void cvLUT( const CvArr* src, CvArr* dst, const CvArr* lut );
```

src
　　元素为8位的原数组。
dst
　　与原数组有相同通道数的输出数组，深度不确定
lut

有256个元素的查找表;必须要与原输出数组有相同像深 。

函数cvLUT 使用查找表中的值添充输出数组. 坐标入口来自于原数组，也就是说函数处理每个元素按如下方式:

dst(I)=lut[src(I)+DELTA]

这里当src的深度是CV_8U时DELTA=0 ,src的深度是CV_8S时 DELTA=128

[编辑]

ConvertScale

使用线性变换转换数组

```
void cvConvertScale( const CvArr* src, CvArr* dst, double scale=1, double shift=0 );
#define cvCvtScale cvConvertScale
#define cvScale  cvConvertScale
#define cvConvert( src, dst )  cvConvertScale( (src), (dst), 1, 0 )
```

src
输入数组.
dst
输出数组
scale
比例因子.
shift
该加数被加到输入数组元素按比例缩放后得到的元素上

函数 cvConvertScale 有多个不同的目的因此就有多个同义函数（如上面的#define所示）。 该函数首先对输入数组的元素进行比例缩放，然后将shift加到比例缩放后得到的各元素上，即： dst(I)=src(I)*scale + (shift,shift,...)，最后可选的类型转换将结果拷贝到输出数组。

多通道的数组对各个通道是独立处理的。

类型转换主要用舍入和溢出截断来完成。也就是如果缩放+转换后的结果值不能用输出数组元素类型值精确表达，就设置成在输出数组数据轴上最接近该数的值。

如果 scale=1, shift=0 就不会进行比例缩放. 这是一个特殊的优化，相当于该函数的同义函数名：cvConvert 。如果原来数组和输出数组的类型相同，这是另一种特殊情形，可以被用于比例缩放和平移矩阵或图像，此时相当于该函数的同义函数名：cvScale。

[编辑]

ConvertScaleAbs

使用线性变换转换输入数组元素成8位无符号整型

```
void cvConvertScaleAbs( const CvArr* src, CvArr* dst, double scale=1, double shift=0 );
#define cvCvtScaleAbs cvConvertScaleAbs
```

src
原数组
dst
输出数组 (深度为 8u).
scale
比例因子.
shift
原数组元素按比例缩放后添加的值。

函数 cvConvertScaleAbs 与前一函数是相同的，但它是存贮变换结果的绝对值：

```
dst(I)=abs(src(I)*scale + (shift,shift,...))
```

函数只支持目标数数组的深度为 8u (8-bit 无符号)，对于别的类型函数仿效于cvConvertScale 和 cvAbs 函数的联合

[编辑]

Add

计算两个数组中每个元素的和

```
void cvAdd( const CvArr* src1, const CvArr* src2, CvArr* dst, const CvArr* mask=NULL );
```

src1
　　第一个原数组
src2
　　第二个原数组
dst
　　输出数组
mask
　　操作的覆盖面, 8-bit单通道数组; 只有覆盖面指定的输出数组被修改。

函数 cvAdd 加一个数组到别一个数组中：

```
dst(I)=src1(I)+src2(I) if mask(I)!=0
```

除覆盖面外所有的数组必须有相同的类型相同的大小（或ROI尺寸）。

[编辑]

AddS

计算数量和数组的和

```
void cvAddS( const CvArr* src, CvScalar value, CvArr* dst, const CvArr* mask=NULL );
```

src
　　原数组.
value
　　被加入数量
dst
　　输出数组
mask
　　操作的覆盖面（8-bit单通道数组）；只有覆盖面指定的输出数组被修改

函数 cvAddS 用数量值与原数组src1的每个元素想加并存贮结果到

```
dst(I)=src(I)+value if mask(I)!=0
```

除覆盖面外所有数组都必须有相同的类型，相同的大小（或ROI大小）

[编辑]

AddWeighted

计算两数组的加权值的和

```
void  cvAddWeighted( const CvArr* src1, double alpha,
                     const CvArr* src2, double beta,
                     double gamma, CvArr* dst );
```

src1
　　第一个原数组.
alpha
　　第一个数组元素的权值
src2
　　第二个原数组
beta
　　第二个数组元素的权值
dst
　　输出数组
gamma
　　添加的常数项。

函数 cvAddWeighted 计算两数组的加权值的和:

```
dst(I)=src1(I)*alpha+src2(I)*beta+gamma
```

所有的数组必须的相同的类型相同的大小（或ROI大小）

[编辑]

Sub

计算两个数组每个元素的差

```
void cvSub( const CvArr* src1, const CvArr* src2, CvArr* dst, const CvArr* mask=NULL );
```

src1
　　第一个原数组
src2
　　第二个原数组.
dst
　　输出数组.
mask
　　操作覆盖面（ 8-bit 单通道数组）；只有覆盖面指定的输出数组被修改

函数cvSub 从一个数组减去别一个数组:

```
dst(I)=src1(I)-src2(I) if mask(I)!=0
```

除覆盖面外所有数组都必须有相同的类型，相同的大小（或ROI大小）

[编辑]

SubS

计算数组和数量之间的差

```
void cvSubS( const CvArr* src, CvScalar value, CvArr* dst, const CvArr* mask=NULL );
```

src
　　原数组.
value
　　被减的数量.
dst
　　.

输出数组

mask
操作覆盖面（ 8-bit 单通道数组）；只有覆盖面指定的输出数组被修改

函数 cvSubS 从原数组的每个元素中减去一个数量：

```
dst(I)=src(I)-value if mask(I)!=0
```

除覆盖面外所有数组都必须有相同的类型，相同的大小（或ROI大小）。

[编辑]

SubRS

计算数量和数组之间的差

```
void cvSubRS( const CvArr* src, CvScalar value, CvArr* dst, const CvArr* mask=NULL );
```

src
第一个原数组。

value
被减的数量

dst
输出数组

mask
操作覆盖面（ 8-bit 单通道数组）；只有覆盖面指定的输出数组被修改

函数 cvSubRS 从一个数量减去原数组的每个元素：

```
dst(I)=value-src(I) if mask(I)!=0
```

除覆盖面外所有数组都必须有相同的类型，相同的大小（或ROI大小）。

[编辑]

Mul

计算两个数组中每个元素的积

```
void cvMul( const CvArr* src1, const CvArr* src2, CvArr* dst, double scale=1 );
```

src1
第一个原数组.

src2
第二个原数组.

dst
输出数组.

scale
设置的比例因子

函数 cvMul 计算两个数组中每个元素的积：

```
dst(I)=scale•src1(I)•src2(I)
```

所有的数组必须有相同的类型和相同的大小（或ROI大小）

[编辑]

Div

两个数组每个元素相除

```
void cvDiv( const CvArr* src1, const CvArr* src2, CvArr* dst, double scale=1 );
```

src1
　　第一个原数组。如该指针为NULL，假高该数组的所有元素都为1
src2
　　第二个原数组。
dst
　　输出数组
scale
　　设置的比例因子

函数 cvDiv 用一个数组除以另一个数组:

```
dst(I)=scale•src1(I)/src2(I), if src1!=NULL
dst(I)=scale/src2(I),:  if src1=NULL
```

所有的数组必须有相同的类型和相同的大小（或ROI大小）

[编辑]

And

计算两个数组的每个元素的按位与

```
void cvAnd( const CvArr* src1, const CvArr* src2, CvArr* dst, const CvArr* mask=NULL );
```

src1
　　第一个原数组
src2
　　第二个原数组.
dst
　　输出数组
mask
　　操作覆盖面（ 8-bit 单通道数组）；只有覆盖面指定的输出数组被修改

函数 cvAnd 计算两个数组的每个元素的按位逻辑与:

```
dst(I)=src1(I)&src2(I) if mask(I)!=0
```

对浮点数组按位表示操作是很有利的。除覆盖面，所有数组都必须有相同的类型，相同的大小（或ROI大小）。

[编辑]

AndS

计算数组每个元素与数量之间的按位与

```
void cvAndS( const CvArr* src, CvScalar value, CvArr* dst, const CvArr* mask=NULL );
```

src
　　原数组.
value
　　操作中用到的数量
dst
　　输出数组
mask

操作覆盖面（ 8-bit 单通道数组）；只有覆盖面指定的输出数组被修改

函数 AndS 计算数组中每个元素与数量之量的按位与：

```
dst(I)=src(I)&value if mask(I)!=0
```

在实际操作之前首先把数量类型转换成与数组相同的类型。对浮点数组按位表示操作是很有利的。除覆盖面，所有数组都必须有相同的类型，相同的大小（或ROI大小）。

接下来的例子描述怎样计算浮点数组元素的绝对值，通过清除最前面的符号位：

```
float a[] = { -1, 2, -3, 4, -5, 6, -7, 8, -9 };
CvMat A = cvMat( 3, 3, CV_32F, &a );
int i, abs_mask = 0x7fffffff;
cvAndS( &A, cvRealScalar(*(float*)&abs_mask), &A, 0 );
for( i = 0; i < 9; i++ )
    printf("%.1f ", a[i] );
```

代码结果是：

```
1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0
```

[编辑]

Or

计算两个数组每个元素的按位或

```
void cvOr( const CvArr* src1, const CvArr* src2, CvArr* dst, const CvArr* mask=NULL );
```

src1
　　第一个原数组
src2
　　第二个原数组
dst
　　输出数组.
mask
　　操作覆盖面（ 8-bit 单通道数组）；只有覆盖面指定的输出数组被修改

函数 cvOr 计算两个数组每个元素的按位或：

```
dst(I)=src1(I)|src2(I)
```

对浮点数组按位表示操作是很有利的。除覆盖面，所有数组都必须有相同的类型，相同的大小（或ROI大小）。

[编辑]

OrS

计算数组中每个元素与数量之间的按位或

```
void cvOrS( const CvArr* src, CvScalar value, CvArr* dst, const CvArr* mask=NULL );
```

src1
　　原数组
value
　　操作中用到的数量
dst
　　目数组.
mask

操作覆盖面（ 8-bit 单通道数组）；只有覆盖面指定的输出数组被修改

函数 OrS 计算数组中每个元素和数量之间的按位或：

```
dst(I)=src(I)|value if mask(I)!=0
```

在实际操作之前首先把数量类型转换成与数组相同的类型。对浮点数组按位表示操作是很有利的。除覆盖面，所有数组都必须有相同的类型，相同的大小（或ROI大小）。

[编辑]

Xor

计算两个数组中的每个元素的按位异或

```
void cvXor( const CvArr* src1, const CvArr* src2, CvArr* dst, const CvArr* mask=NULL );
```

src1
　第一个原数组
src2
　第二个原数组.
dst
　输出数组
mask
　操作覆盖面（ 8-bit 单通道数组）；只有覆盖面指定的输出数组被修改

函数 cvXor 计算两个数组元素的按位异或：

$$dst(I)=src1(I) \oplus src2(I) \text{ if } mask(I)!=0$$

对浮点数组按位表示操作是很有利的。除覆盖面，所有数组都必须有相同的类型，相同的大小（或ROI大小）。

[编辑]

XorS

计算数组元素与数量之间的按位异或

```
void cvXorS( const CvArr* src, CvScalar value, CvArr* dst, const CvArr* mask=NULL );
```

src
　原数组
value
　操作中用到的数量
dst
　输出数组.
mask
　操作覆盖面（ 8-bit 单通道数组）；只有覆盖面指定的输出数组被修改。

函数 XorS 计算数组元素与数量之间的按位异或：

$$dst(I)=src(I) \oplus value \text{ if } mask(I)!=0$$

在实际操作之前首先把数量类型转换成与数组相同的类型。对浮点数组按位表示操作是很有利的。除覆盖面，所有数组都必须有相同的类型，相同的大小（或ROI大小）。

下面例子描述怎样对共轭复数向量转换，通过转换前面的符号位：

```
float a[] = { 1, 0, 0, 1, -1, 0, 0, -1 }; /* 1, j, -1, -j */
CvMat A = cvMat( 4, 1, CV_32FC2, &a );
int i, neg_mask = 0x80000000;
cvXorS( &A, cvScalar( 0, *(float*)&neg_mask, 0, 0 ), &A, 0 );
for( i = 0; i < 4; i++ )
    printf("(%.1f, %.1f) ", a[i*2], a[i*2+1] );
```

The code should print:

```
(1.0,0.0) (0.0,-1.0) (-1.0,0.0) (0.0,1.0)
```

[编辑]

Not

计算数组元素的按位取反

```
void cvNot( const CvArr* src, CvArr* dst );
```

src1
　　原数组
dst
　　输出数组

函数不取反每个数组元素的每一位

```
dst(I)=~src(I)
```

[编辑]

Cmp

比较两个数组元素P

```
void cvCmp( const CvArr* src1, const CvArr* src2, CvArr* dst, int cmp_op );
```

src1
　　第一个原数组
src2
　　第二个原数组，这两个数组必须都是单通道数据。
dst
　　输出数组必须是 8u 或 8s 类型.
cmp_op
　　该标识指定要检查的元素之间的关系：

　　CV_CMP_EQ - src1(I) "等于" src2(I)
　　CV_CMP_GT - src1(I) "大于" src2(I)
　　CV_CMP_GE - src1(I) "大于等于" src2(I)
　　CV_CMP_LT - src1(I) "小于" src2(I)
　　CV_CMP_LE - src1(I) "小于等于" src2(I)
　　CV_CMP_NE - src1(I) "不等于" src2(I)

函数 cvCmp 比较两个数组的对应元素并且添充输出数组：

```
dst(I)=src1(I) op src2(I),
```

这里 op 是 '=', '>', '>=', '<', '<=' or '!='.

如果元素之间的关系为真则设置dst(I)为 0xff (也就是所有的位都为 '1') 否则为0。除了输出数组所有数组必须是相同的类型相同的大小（或ROI大小）。

[编辑]

CmpS

比较数组的每个元素与数量的关系

```
void cvCmpS( const CvArr* src, double value, CvArr* dst, int cmp_op );
```

src
　　原数,输入数组必须是单通道数据。
value
　　用与数组元素比较的数量值
dst
　　输出数组必须是 8u 或 8s 类型.
cmp_op
　　该标识指定要检查的元素之间的关系:

　　CV_CMP_EQ - src1(I) "等于" value
　　CV_CMP_GT - src1(I) "大于" value
　　CV_CMP_GE - src1(I) "大于等于" value
　　CV_CMP_LT - src1(I) "小于" value
　　CV_CMP_LE - src1(I) "小于等于" value
　　CV_CMP_NE - src1(I) "不等于" value

函数 cvCmpS 比较数组元素与数量并且添充目标覆盖面数组:

```
dst(I)=src(I) op scalar,
```

这里 op 是 '=', '>', '>=', '<', '<=' or '!='.

如果元素之间的关系为真则设置dst(I)为 0xff (也就是所有的位都为 '1') 否则为0。所有的数组必须有相同的大小（或ROI大小）

[编辑]

InRange

检查数组元素是否在两个数组之间

```
void cvInRange( const CvArr* src, const CvArr* lower, const CvArr* upper, CvArr* dst );
```

src
　　第一个原数组
lower
　　包括进的下边界数组
upper
　　不包括进的上边界线数组
dst
　　输出数组必须是 8u 或 8s 类型.

函数 cvInRange 对输入的数组作范围检查,对于单通道数组:

```
dst(I)=lower(I)0 <= src(I)0 < upper(I)0
```

对二通道数组:

```
dst(I)=lower(I)0 <= src(I)0 < upper(I)0 &&
       lower(I)1 <= src(I)1 < upper(I)1
```

以此类推

如果 src(I) 在范围内dst(I)被设置为 0xff (每一位都是 '1')否则置0 。 除了输出数组所有数组必须是相同的类型相同的大小（或ROI大小）。

[编辑]

InRangeS

检查数组元素是否在两个数量之间

```
void cvInRangeS( const CvArr* src, CvScalar lower, CvScalar upper, CvArr* dst );
```

src
　　第一个原数组
lower
　　包括进的下边界.
upper
　　不包括进的上边界
dst
　　输出数组必须是 8u 或 8s 类型.

函数 cvInRangeS 检查输入数组元素范围： 对于单通道数组：

```
dst(I)=lower0 <= src(I)0 < upper0
```

对于双通道数组以此类推:

```
dst(I)=lower0 <= src(I)0 < upper0 &&
       lower1 <= src(I)1 < upper1
```

如果 src(I) 在范围内dst(I)被设置为 0xff (每一位都是 '1')否则置0 。所有的数组必须有相同的大小（或ROI大小）

[编辑]

Max

查找两个数组中每个元素的较大值

```
void cvMax( const CvArr* src1, const CvArr* src2, CvArr* dst );
```

src1
　　第一个原数组
src2
　　第二个原数组
dst
　　输出数组

函数 cvMax 计算两个数组中每个元素的较大值：

```
dst(I)=max(src1(I), src2(I))
```

所有的数组必须的一个单通道，相同的数据类型和相同的大小（或ROI大小）

[编辑]

MaxS

查找数组元素与数量之间的较大值

```
void cvMaxS( const CvArr* src, double value, CvArr* dst );
```

src
　　第一个原数组.
value
　　数量值.
dst
　　输出数组

函数 cvMaxS 计算数组元素和数量之间的较大值:

```
dst(I)=max(src(I), value)
```

所有数组必须有一个单一通道，相同的数据类型相同的大小（或ROI大小）

[编辑]

Min

查找两个数组元素之间 的较小值

```
void cvMin( const CvArr* src1, const CvArr* src2, CvArr* dst );
```

src1
　　第一个原数组
src2
　　第二个原数组.
dst
　　输出数组.

函数cvMin计算两个数组元素的较小值

```
dst(I)=min(src1(I),src2(I))
```

所有数组必须有一个单一通道，相同的数据类型相同的大小（或ROI大小）

[编辑]

MinS

查找数组元素和数量之间的较小值

```
void cvMinS( const CvArr* src, double value, CvArr* dst );
```

src
　　第一个原数组
value
　　数量值.
dst
　　输出数组..

函数 cvMinS 计算数组元素和数量之量的较小值:

```
dst(I)=min(src(I), value)
```

所有数组必须有一个单一通道，相同的数据类型相同的大小（或ROI大小）

[编辑]

AbsDiff

计算两个数组差的绝对值

```
void cvAbsDiff( const CvArr* src1, const CvArr* src2, CvArr* dst );
```

src1
　　第一个原数组
src2
　　第二个原数组
dst
　　输出数组

函数 cvAbsDiff 计算两个数组差的绝对值

```
dst(I)c = abs(src1(I)c - src2(I)c).
```

所有数组必须有相同的数据类型相同的大小（或ROI大小）

[编辑]

AbsDiffS

计算数组元素与数量之间差的绝对值

```
void cvAbsDiffS( const CvArr* src, CvArr* dst, CvScalar value );
#define cvAbs(src, dst) cvAbsDiffS(src, dst, cvScalarAll(0))
```

src
　　原数组.
dst
　　输出数组
value
　　数量.

函数 cvAbsDiffS 计算数组元素与数量之间差的绝对值

```
dst(I)c = abs(src(I)c - valuec).
```

所有数组必须有相同的数据类型相同的大小（或ROI大小）

[编辑]

统计

[编辑]

CountNonZero

计算非零数组元素个数

```
int cvCountNonZero( const CvArr* arr );
```

arr
　　数组, 必须是单通道数组或者设置COI（感兴趣通道）的多通道图像。

函数 cvCountNonZero 返回arr中非零元素的数目:

```
result = sumI arr(I)!=0
```

当IplImage 支持ROI和COI。

[编辑]

Sum

计算数组元素的和

```
CvScalar cvSum( const CvArr* arr );
```

arr
　　数组.

函数 cvSum 独立地为每一个通道计算数组元素的和 S：

```
Sc = sumI arr(I)c
```

如果数组是IplImage类型和设置了COI, 该函数只处理选定的通道并将和存储到第一个标量成员 (S0)。 常见论坛讨论贴 cvSum的结果分析

[编辑]

Avg

计算数组元素的平均值

```
CvScalar cvAvg( const CvArr* arr, const CvArr* mask=NULL );
```

arr
　　数组.
mask
　　可选操作掩模

函数 cvAvg 独立地为每一个通道计算数组元素的平均值 M：

$$N = \sum_i (\mathrm{mask}(i) \neq 0)$$

$$M_c = \frac{1}{N} \sum_{i, \mathrm{mask}(i) \neq 0} \mathrm{arr}(i)_c$$

如果数组是 IplImage 类型和设置 COI，该函数只处理选定的通道并将和存储到第一个标量成员 (S0)。

[编辑]

AvgSdv

计算数组元素的平均值和标准差

```
void cvAvgSdv( const CvArr* arr, CvScalar* mean, CvScalar* std_dev, const CvArr* mask=NULL );
```

arr
　　数组
mean
　　指向平均值的指针， 如果不需要的话可以为空（ NULL）。
std_dev
　　指向标准差的指针。
mask
　　可选操作掩模。

函数 cvAvgSdv 独立地为每一个通道计算数组元素的平均值和标准差:

$$N = \sum_i (mask(i) \neq 0)$$

$$\text{mean}_c = \frac{1}{N} \sum_{i,\text{mask}(i)\neq 0} \text{arr}(i)_c$$

$$\text{std-dev}_c = \sqrt{\frac{1}{N} \sum_{i,\text{mask}(i)\neq 0} (\text{arr}(i)_c - \text{mean}_c)^2}$$

如果数组是 IplImage 类型和 设置了COI ， 该函数只处理选定的通道并将平均值和标准差存储到第一个输出标量成员 (mean$_0$ 和 std-dev$_0$)。

[编辑]

MinMaxLoc

查找数组和子数组的全局最小值和最大值

```
void cvMinMaxLoc( const CvArr* arr, double* min_val, double* max_val,
                  CvPoint* min_loc=NULL, CvPoint* max_loc=NULL, const CvArr* mask=NULL );
```

arr
　　输入数组， 单通道或者设置了 COI 的多通道。
min_val
　　指向返回的最小值的指针。
max_val
　　指向返回的最大值的指针。
min_loc
　　指向返回的最小值的位置指针。
max_loc
　　指向返回的最大值的位置指针。
mask
　　选择一个子数组的操作掩模。

函数 MinMaxLoc 查找元素中的最小值和最大值以及他们的位置。函数在整个数组、或选定的ROI区域(对IplImage)或当MASK不为NULL时指定的数组区域中，搜 索极值 。如果数组不止一个通道，它就必须是设置了 COI 的 IplImage 类型。 如果是多维数组 min_loc->x 和 max_loc->x 将包含极值的原始位置信息 (线性的)。

[编辑]

Norm

计算数组的绝对范数， 绝对差分范数或者相对差分范数

```
double cvNorm( const CvArr* arr1, const CvArr* arr2=NULL, int norm_type=CV_L2, const CvArr*
mask=NULL );
```

arr1
　　第一输入图像
arr2
　　第二输入图像 ， 如果为空（NULL），计算 arr1 的绝对范数，否则计算 arr1-arr2 的绝对范数或者相对范数。
normType
　　范数类型，参见“讨论”。

mask
可选操作掩模。

如果 arr2 为空（NULL），函数 cvNorm 计算 arr1 的绝对范数：

norm = ||arr1||C = maxI abs(arr1(I)), 如果 normType = CV_C
norm = ||arr1||L1 = sumI abs(arr1(I)), 如果 normType = CV_L1
norm = ||arr1||L2 = sqrt(sumI arr1(I)2), 如果 normType = CV_L2

如果 arr2 不为空（NULL）， 该函数计算绝对差分范数或者相对差分范数：

norm = ||arr1-arr2||C = maxI abs(arr1(I)-arr2(I)), 如果 normType = CV_C
norm = ||arr1-arr2||L1 = sumI abs(arr1(I)-arr2(I)), 如果 normType = CV_L1
norm = ||arr1-arr2||L2 = sqrt(sumI (arr1(I)-arr2(I))2), 如果 normType = CV_L2

或者

norm = ||arr1-arr2||C/||arr2||C, 如果 normType = CV_RELATIVE_C
norm = ||arr1-arr2||L1/||arr2||L1, 如果 normType = CV_RELATIVE_L1
norm = ||arr1-arr2||L2/||arr2||L2, 如果 normType = CV_RELATIVE_L2

函数 Norm 返回计算所得的范数。多通道数组被视为单通道处理，因此，所有通道的结果是结合在一起的。

Reduce

简化一个矩阵成为一个向量

```
cvReduce( const CvArr* src, CvArr* dst, int dim, int op=CV_REDUCE_SUM);
```

src
输入矩阵
dst
输出的通过处理输入矩阵的所有行/列而得到的单行/列向量
dim
矩阵被简化后的维数索引.0意味着矩阵被处理成一行,1意味着矩阵被处理成为一列,-1时维数将根据输出向量的大小自动选择.
op
简化操作的方式,可以有以下几种取值:
CV_REDUCE_SUM-输出是矩阵的所有行/列的和.
CV_REDUCE_AVG-输出是矩阵的所有行/列的平均向量.
CV_REDUCE_MAX-输出是矩阵的所有行/列的最大值.
CV_REDUCE_MIN-输出是矩阵的所有行/列的最小值.

这个函数通过把矩阵的每行/列当作一维向量并对其做某种特殊的操作将一个矩阵简化成为一个向量.例如,这个函数可以用于计算一个光栅图象的水平或者 垂直投影.在取值为CV_REDUCE_AVG与CV_REDUCE_SUM的情况下输出可能有很大的位深度用于维持准确性,这两种方式也适合于处理多通 道数组.

注意，对于CV_REDUCE_SUM和CV_REDUCE_AVG方式来说，输入和输出的位数定义有如下关系

输入：CV_8U 输出：CV_32S CV_32F

输入：CV_16U 输出：CV_32F CV_64F

输入：CV_16S 输出：CV_32F CV_64F

输入：CV_32F 输出： CV_32F CV_64F

输入：CV_64F 输出： CV_64F

而对于CV_REDUCE_MAX和CV_REDUCE_MIN方式来说，输入和输出的位数必须一致

[编辑]

线性代数

[编辑]

DotProduct

用欧几里得准则计算两个数组的点积

```
double cvDotProduct( const CvArr* src1, const CvArr* src2 );
```

src1
　　第一输入数组。
src2
　　第二输入数组。

函数 cvDotProduct 计算并返回两个数组的欧几里得点积。

```
src1•src2 = sumI(src1(I)*src2(I))
```

如果是多通道数组，所有通道的结果是累加在一起的。特别地， cvDotProduct(a,a),将返回 ||a||2 ，这里 a 是一个复向量。 该函数可以处理多通道数组,逐行或逐层等等。

[编辑]

Normalize

根据某种范数或者数值范围归一化数组.

```
void cvNormalize( const CvArr* src, CvArr* dst,
                  double a=1, double b=0, int norm_type=CV_L2,
                  const CvArr* mask=NULL );
```

src
　　输入数组
dst
　　输出数组,支持原地运算
a
　　输出数组的最小/最大值或者输出数组的范数
b
　　输出数组的最大/最小值
norm_type
　　归一化的类型,可以有以下的取值:
　　CV_C - 归一化数组的C-范数(绝对值的最大值)
　　CV_L1 - 归一化数组的L1-范数(绝对值的和)
　　CV_L2 - 归一化数组的(欧几里德)L2-范数
　　CV_MINMAX - 数组的数值被平移或缩放到一个指定的范围
mask
　　操作掩膜,用于指示函数是否仅仅对指定的元素进行操作

该函数归一化输入数组使它的范数或者数值范围在一定的范围内

当norm_type==CV_MINMAX:

dst(i,j)=(src(i,j)-min(src))*(b'-a')/(max(src)-min(src)) + a', if mask(i,j)!=0

dst(i,j)=src(i,j) otherwise

其中b'=MAX(a,b), a'=MIN(a,b);

当norm_type!=CV_MINMAX:

dst(i,j)=src(i,j)*a/cvNorm(src,0,norm_type,mask), if mask(i,j)!=0

dst(i,j)=src(i,j) otherwise

下面是一个简单的例子: float v[3] = { 1, 2, 3 };

CvMat V = cvMat(1, 3, CV_32F, v);

// make vector v unit-length;

// equivalent to

// for(int i=0;i<3;i++) v[i]/=sqrt(v[0]*v[0]+v[1]*v[1]+v[2]*v[2]);

cvNormalize(&V, &V);

[编辑]

CrossProduct

计算两个三维向量的叉积

```
void cvCrossProduct( const CvArr* src1, const CvArr* src2, CvArr* dst );
```

src1
　　第一输入向量。
src2
　　第二输入向量。
dst
　　输出向量

函数 cvCrossProduct 计算两个三维向量的差积:

```
dst = src1 × src2, (dst1 = src12src23 - src13src22 , dst2 = src13src21 - src11src23 , dst3 =
src11src22 - src12src21).
```

[编辑]

ScaleAdd

计算一个数组缩放后与另一个数组的和

```
void cvScaleAdd( const CvArr* src1, CvScalar scale, const CvArr* src2, CvArr* dst );
#define cvMulAddS cvScaleAdd
```

src1
　　第一输入数组
scale

第一输入数组的缩放因子

src2
第二输入数组

dst
输出数组

函数 cvScaleAdd 计算一个数组缩放后与另一个数组的和:

```
dst(I)=src1(I)*scale + src2(I)
```

所有的数组参数必须有相同的类型和大小。

[编辑]

GEMM

通用矩阵乘法

```
void  cvGEMM( const CvArr* src1, const CvArr* src2, double alpha,
              const CvArr* src3, double beta, CvArr* dst, int tABC=0 );
#define cvMatMulAdd( src1, src2, src3, dst ) cvGEMM( src1, src2, 1, src3, 1, dst, 0 )
#define cvMatMul( src1, src2, dst ) cvMatMulAdd( src1, src2, 0, dst )
```

src1
第一输入数组

src2
第二输入数组

src3
第三输入数组 (偏移量)，如果没有偏移量，可以为空（ NULL） 。

dst
输出数组

tABC
T操作标志，可以是 0 或者下面列举的值的组合:

CV_GEMM_A_T - 转置 src1
CV_GEMM_B_T - 转置 src2
CV_GEMM_C_T - 转置 src3

例如, CV_GEMM_A_T+CV_GEMM_C_T 对应

alpha*src1T*src2 + beta*src3T

函数 cvGEMM 执行通用矩阵乘法:

```
dst = alpha*op(src1)*op(src2) + beta*op(src3), 这里 op(X) 是 X 或者 XT
```

所有的矩阵应该有相同的数据类型和协调的矩阵大小。支持实数浮点矩阵或者复数浮点矩阵。

[编辑]

Transform

对数组每一个元素执行矩阵变换

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

src
输入数组

dst

输出数组

transmat

变换矩阵

shiftvec

可选偏移向量

函数 cvTransform 对数组 src 每一个元素执行矩阵变换并将结果存储到 dst:

dst(I)=transmat*src(I) + shiftvec

或者

dst(I)k=sumj(transmat(k,j)*src(I)j) + shiftvec(k)

N-通道数组 src 的每一个元素都被视为一个N元向量，使用一个 M×N 的变换矩阵 transmat 和偏移向量 shiftvec 把它变换到一个 M-通道的数组 dst 的一个元素中。 这里可以选择将偏移向量 shiftvec 嵌入到 transmat 中。这样的话 transmat 应该是 M×N+1 的矩阵，并且最右边的一列被看作是偏移向量 。

输入数组和输出数组应该有相同的位深（depth）和同样的大小或者 ROI 大小。 transmat 和 shiftvec 应该是实数浮点矩阵。

该函数可以用来进行 ND 点集的几何变换，任意的线性颜色空间变换，通道转换等。

[编辑]

PerspectiveTransform

向量数组的透视变换

```
void cvPerspectiveTransform( const CvArr* src, CvArr* dst, const CvMat* mat );
```

src

输入的三通道浮点数组

dst

输出三通道浮点数组

mat

4 × 4 变换矩阵

函数 cvPerspectiveTransform 用下面的方式变换 src 的每一个元素 (通过将其视为二维或者三维的向量):

(x, y, z) -> (x'/w, y'/w, z'/w)

或者

(x, y) -> (x'/w, y'/w),

这里

(x', y', z', w') = mat*(x, y, z, 1)

或者

(x', y', w') = mat*(x, y, 1)

并且 w = w' 如果 w'!=0, 否则 w = inf

[编辑]

MulTransposed

计算数组和数组的转置的乘积

```
void cvMulTransposed( const CvArr* src, CvArr* dst, int order, const CvArr* delta=NULL );
```

src
　　输入矩阵
dst
　　目标矩阵
order
　　乘法顺序
delta
　　一个可选数组， 在乘法之前从 src 中减去该数组。

函数 cvMulTransposed 计算 src 和它的转置的乘积。

函数求值公式:

如果 order=0

　　dst=(src-delta)*(src-delta)T

否则

　　dst=(src-delta)T*(src-delta)

[编辑]

Trace

返回矩阵的迹

```
CvScalar cvTrace( const CvArr* mat );
```

mat
　　输入矩阵

函数 cvTrace 返回矩阵mat的对角线元素的和。

$$\mathrm{tr}(src) = \sum_i \mathrm{mat}(i,i)$$

[编辑]

Transpose

矩阵的转置

```
void cvTranspose( const CvArr* src, CvArr* dst );
#define cvT cvTranspose
```

src
　　输入矩阵
dst
　　目标矩阵

函数 cvTranspose 对矩阵 src 求转置:

dst(i,j)=src(j,i)

注意，假设是复数矩阵不会求得复数的共轭。共轭应该是独立的：查看的 cvXorS 例子代码。

[编辑]

Det

返回矩阵的行列式值

```
double cvDet( const CvArr* mat );
```

mat
输入矩阵

函数 cvDet 返回方阵 mat 的行列式值。对小矩阵直接计算，对大矩阵用 高斯(GAUSSIAN)消去法。对于对称正定(positive-determined)矩阵也可以用 SVD 函数来求，U=V=NULL ，然后用 w 的对角线元素的乘积来计算行列式。

[编辑]

Invert

查找矩阵的逆矩阵或伪逆矩阵

```
double cvInvert( const CvArr* src, CvArr* dst, int method=CV_LU );
#define cvInv cvInvert
```

src
输入矩阵
dst
目标矩阵
method
求逆方法:

CV_LU -最佳主元选取的高斯消除法
CV_SVD - 奇异值分解法 (SVD)
CV_SVD_SYM - 正定对称矩阵的 SVD 方法

函数 cvInvert 对矩阵 src 求逆并将结果存储到 dst。

如果是 LU 方法该函数返回 src 的行列式值 (src 必须是方阵)。 如果是 0, 矩阵不求逆， dst 用 0 填充。

如果 SVD 方法该函数返回 src 的条件数的倒数(最小奇异值和最大奇异值的比值) ，如果 src 全为 0 则返回0。如果 src 是奇异的， SVD 方法计算一个伪逆矩阵。

[编辑]

Solve

求解线性系统或者最小二乘法问题

```
int cvSolve( const CvArr* src1, const CvArr* src2, CvArr* dst, int method=CV_LU );
```

src1
输入矩阵
src2

线性系统的右部

dst
输出解答

method
解决方法(矩阵求逆)：

CV_LU - 最佳主元选取的高斯消除法
CV_SVD - 奇异值分解法 (SVD)
CV_SVD_SYM - 对正定对称矩阵的 SVD 方法

函数 cvSolve 解决线性系统或者最小二乘法问题 (后者用 SVD 方法可以解决):

$$\texttt{dst} = \arg\min_X |\texttt{src1} \cdot X - \texttt{src2}|$$

如果使用 CV_LU 方法。 如果 src1 是非奇异的，该函数则返回 1 ，否则返回 0 ，在后一种情况下 dst 是无效的。

[编辑]

SVD

对实数浮点矩阵进行奇异值分解

```
void cvSVD( CvArr* A, CvArr* W, CvArr* U=NULL, CvArr* V=NULL, int flags=0 );
```

A
M×N 的输入矩阵

W
结果奇异值矩阵 (M×N 或者 N×N) 或者 向量 (N×1).

U
可选的左部正交矩阵 (M×M or M×N). 如果 CV_SVD_U_T 被指定， 应该交换上面所说的行与列的数目。

V
可选右部正交矩阵(N×N)

flags
操作标志； 可以是 0 或者下面的值的组合:

- CV_SVD_MODIFY_A 通过操作可以修改矩阵 src1 。这样处理速度会比较快。
- CV_SVD_U_T 意味着会返回转置矩阵 U ，指定这个标志将加快处理速度。
- CV_SVD_V_T 意味着会返回转置矩阵 V ，指定这个标志将加快处理速度。

函数 cvSVD 将矩阵 A 分解成一个对角线矩阵和两个正交矩阵的乘积:

$$A = UWV^T$$

这里 W 是一个奇异值的对角线矩阵，它可以被编码成奇异值的一维向量，U 和 V 也是一样。 所有的奇异值都是非负的并按降序存储。（U 和 V 也相应的存储）。

SVD 算法在数值处理上已经很稳定，它的典型应用包括:

- 当 A 是一个方阵、对称阵和正矩阵时精确的求解特征值问题，例如，当 A 时一个协方差矩阵时。在这种情况下 W 将是一个特征值的的向量，并且 U=V是矩阵的特征向量(因此，当需要计算特征向量时 U 和 V 只需要计算其中一个就可以了) 。
- 精确的求解病态线性系统。
- 超定线性系统的最小二乘求解。上一个问题和这个问题都可以用指定 CV_SVD 的 cvSolve 方法。
- 精确计算矩阵的不同特征，如秩(非零奇异值的数目), 条件数(最大奇异值和最小奇异值的比例), 行列式值(行列式的绝对值等于奇异值的乘积).上述的所有这些值都不要求计算矩阵 U 和 V 。

[编辑]

SVBkSb

奇异值回代算法（back substitution）

```
void  cvSVBkSb( const CvArr* W, const CvArr* U, const CvArr* V,
                const CvArr* B, CvArr* X, int flags );
```

W
　　奇异值矩阵或者向量
U
　　左正交矩阵 (可能是转置的)
V
　　右正交矩阵 (可能是转置的)
B
　　原始矩阵 A 的伪逆的乘法矩阵。这个是可选参数。如果它被省略则假定它是一个适当大小的单位矩阵(因此 x 将是 A 的伪逆的重建).。
X
　　目标矩阵: 奇异值回代算法的结果
flags
　　操作标志, 和刚刚讨论的 cvSVD 的标志一样。

函数 cvSVBkSb 为被分解的矩阵 A 和矩阵 B 计算回代逆（back substitution） (参见 cvSVD 说明) :

　　X=V*W-1*UT*B

这里

　　W-1(i,i)=1/W(i,i) 如果 W(i,i) > epsilon•sumiW(i,i),
　　否则:0.

epsilon 是一个依赖于矩阵数据类型的的很小的数。该函数和 cvSVD 函数被用来执行 cvInvert 和 cvSolve, 用这些函数 (svd & bksb)的原因是初级函数（low-level） 函数可以避免高级函数 (inv & solve) 计算中内部分配的临时矩阵。

[编辑]

EigenVV

计算对称矩阵的特征值和特征向量

```
void cvEigenVV( CvArr* mat, CvArr* evects, CvArr* evals, double eps=0 );
```

mat
　　输入对称方阵。在处理过程中将被改变。
evects
　　特征向量输出矩阵， 连续按行存储
evals
　　特征值输出矩阵，按降序存储(当然特征值和特征向量的排序是同步的)。
eps
　　对角化的精确度 (典型地， DBL_EPSILON=≈10-15 就足够了)。

函数 cvEigenVV 计算矩阵 A 的特征值和特征向量:

```
mat*evects(i,:)' = evals(i)*evects(i,:)' (在 MATLAB 的记法)
```

矩阵 A 的数据将会被这个函数修改。

目前这个函数比函数 cvSVD 要慢，精确度要低， 如果已知 A 是正定的，(例如, 它是一个协方差矩阵), 它通常被交给函数 cvSVD 来计算其特征值和特征向量，尤其是在不需要计算特征向量的情况下

[编辑]

CalcCovarMatrix

计算向量集合的协方差矩阵

```
void cvCalcCovarMatrix( const CvArr** vects, int count, CvArr* cov_mat, CvArr* avg, int flags );
```

vects
: 输入向量。他们必须有同样的数据类型和大小。这个向量不一定非是一维的，他们也可以是二维（例如，图像）等等。

count
: 输入向量的数目

cov_mat
: 输出协方差矩阵，它是浮点型的方阵。

avg
: 输入或者输出数组 (依赖于标记“flags”) - 输入向量的平均向量。

flags
: 操作标志,下面值的组合:

CV_COVAR_SCRAMBLED - 输出协方差矩阵按下面计算:

$scale * [vects[0] - avg, vects[1] - avg, ...]^T * [vects[0] - avg, vects[1] - avg, ...]$, 即协方差矩阵是 count×count. 这样一个不寻常的矩阵用于一组大型向量的快速PCA方法(例如, 人脸识别的 EigenFaces 技术)。这个混杂（"scrambled"）矩阵的特征值将和真正的协方差矩阵的特征值匹配，真正的特征向量可以很容易的从混杂 （"scrambled"）协方差矩阵的特征向量中计算出来。

CV_COVAR_NORMAL - 输出协方差矩阵被计算成:

$scale * [vects[0] - avg, vects[1] - avg, ...] * [vects[0] - avg, vects[1] - avg, ...]^T$, 也就是说, cov_mat 将是一个和每一个输入向量的元素数目具有同样线性大小的通常协方差矩阵。CV_COVAR_SCRAMBLED 和 CV_COVAR_NORMAL 只能同时指定其中一个。

CV_COVAR_USE_AVG - 如果这个标志被指定， 该函数将不会从输入向量中计算 avg ，而是用过去的 avg 向量，如果 avg 已经以某种方式计算出来了这样做是很有用的。或者如果协方差矩阵是部分计算出来的 - 倘若这样, avg 不是输入向量的子集的平均值，而是整个集合的平均向量。
CV_COVAR_SCALE - 如果这个标志被指定，协方差矩阵被缩放了。 the covariation matrix is scaled.在 "normal" 模式下缩放比例是 1./count, 在 "scrambled" 模式下缩放比例是每一个输入向量的元素总和的倒数。 缺省地(如果没有指定标志) 协方差矩阵不被缩放 (scale=1)。

函数 cvCalcCovarMatrix 计算输入向量的协方差矩阵和平均向量。该函数 可以被运用到主成分分析中（PCA），以及马氏距离（Mahalanobis distance）比较向量中等等。

[编辑]

Mahalanobis

计算两个向量之间的马氏距离（Mahalanobis distance）

```
double cvMahalanobis( const CvArr* vec1, const CvArr* vec2, CvArr* mat );
```

vec1

第一个一维输入向量

vec2
第二个一维输入向量

mat
协方差矩阵的逆矩阵

函数 cvMahalanobis 计算两个向量之间的加权距离，其返回结果是:

$$d(vec1, vec2) = \sqrt{\sum_{i,j}\{mat(i,j) * (vec1(i) - vec2(i)) * (vec1(j) - vec2(j))\}}$$

协方差矩阵可以用函数cvCalcCovarMatrix 计算出来，逆矩阵可以用函数 cvInvert 计算出来 (CV_SVD 方法是一个比较好的选择, 因为矩阵可能是奇异的).

[编辑]

CalcPCA

对一个向量集做PCA变换

```
void cvCalcPCA( const CvArr* data, CvArr* avg,
                CvArr* eigenvalues, CvArr* eigenvectors, int flags );
```

data
输入数据,每个向量是单行向量(CV_PCA_DATA_AS_ROW)或者单列向量(CV_PCA_DATA_AS_COL).

avg
平均向量,在函数内部计算或者由调用者提供

eigenvalues
输出的协方差矩阵的特征值

eigenvectors
输出的协方差矩阵的特征向量(也就是主分量),每个向量一行

flags
操作标志,可以是以下几种方式的组合:
CV_PCA_DATA_AS_ROW - 向量以行的方式存放(也就是说任何一个向量都是连续存放的)
CV_PCA_DATA_AS_COL - 向量以列的方式存放(也就是说某一个向量成分的数值是连续存放的)
(上面两种标志是互相排斥的)
CV_PCA_USE_AVG - 使用预先计算好的平均值

该函数对某个向量集做PCA变换.它首先利用cvCalcCovarMatrix计算协方差矩阵然后计算协方差矩阵的特征值与特征向量.输出的特征值/特征向量的个数小于或者等于MIN(rows(data),cols(data)).

[编辑]

ProjectPCA

把向量向某个子空间投影

```
void cvProjectPCA( const CvArr* data, const CvArr* avg,
                   const CvArr* eigenvectors, CvArr* result )
```

data
输入数据,每个向量可以是单行或者单列

avg
平均向量.要么它是单行向量那么意味着输入数据以行数据的形式存放,要么就是单列向量,那么就意味着那么输入向量就是以列的方式存放.

eigenvectors
特征向量(主分量),每个向量一行.

result
输出的分解系数矩阵,矩阵的行数必须与输入向量的个数相等,矩阵的列数必须小于特征向量的行数.

该函数将输入向量向一个正交系(eigenvectors)投影.在计算点乘之前,输入向量要减去平均向量:

result(i,:)=(data(i,:)-avg)*eigenvectors' // for CV_PCA_DATA_AS_ROW layout.

[编辑]

BackProjectPCA

根据投影系数重构原来的向量

```
void cvBackProjectPCA( const CvArr* proj, const CvArr* avg,
                       const CvArr* eigenvects, CvArr* result );
```

proj
输入数据,与cvProjectPCA里面的格式一致
avg
平均向量.如果它是单行向量,那么意味着输出向量是以行的方式存放.否则就是单列向量,那么输出向量就是以列的方式存放.
eigenvectors
特征向量(主分量),每个向量一行.
result
输出的重构出来的矩阵

该函数根据投影系数重构原来的向量:

result(i,:)=proj(i,:)*eigenvectors + avg // for CV_PCA_DATA_AS_ROW layout.

[编辑]

数学函数

[编辑]

Round, Floor, Ceil

转换浮点数为整数

```
int cvRound( double value );
int cvFloor( double value );
int cvCeil( double value );
```

value
输入浮点值

函数 cvRound, cvFloor, cvCeil 用一种舍入方法将输入浮点数转换成整数。 cvRound 返回和参数最接近的整数值。 cvFloor 返回不大于参数的最大整数值。cvCeil 返回不小于参数的最小整数值。在某些体系结构中该函数工作起来比标准 C 操作起来还要快。如果参数的绝对值大于 2^{31} ，结果是不可预料的。对特殊值 (±Inf, NaN) 未进行处理。

[编辑]

Sqrt

计算平方根

```
float cvSqrt( float value );
```

value
输入浮点值

函数 cvSqrt 计算输入值的平方根。如果输入的是复数， 结果将不可预料。

[编辑]

InvSqrt

计算平方根的倒数

```
float cvInvSqrt( float value );
```

value
输入浮点值

函数 cvInvSqrt 计算输入值的平方根的倒数，大多数情况下它比 1./sqrt(value) 要快。 如果输入的是 0 或者复数，结果将不可预料。特别值 (±Inf, NaN) 是不可控制的。

[编辑]

Cbrt

计算立方根

```
float cvCbrt( float value );
```

value
输入浮点值

函数 cvCbrt 计算输入值的立方根，大多数情况下它比 pow(value,1./3) 要快。 另外, 负数也是可操作的。特别值 (±Inf, NaN) 是不可控制的。

[编辑]

FastArctan

计算二维向量的角度

```
float cvFastArctan( float y, float x );
```

x
二维向量的 x 坐标
y
二维向量的 y 坐标

函数 cvFastArctan 计算二维向量的全范围角度角度， 变化范围是 0°到 360°。 精确度为 ~0.1° 。

[编辑]

IsNaN

判断输入是否是一个数字

```
int cvIsNaN( double value );
```

value
输入浮点值

函数 cvIsNaN 发现输入是一个数字则返回 1 (IEEE754 标准)，否则返回 0 。

[编辑]

IsInf

判断输入是否是无穷大

```
int cvIsInf( double value );
```

value
　　输入浮点值

函数 cvIsInf 如果输入是 ±Infinity (IEEE754 标准)则返回 1 ,否则返回 0 .

[编辑]

CartToPolar

计算二维向量的长度和/或者角度

```
void cvCartToPolar( const CvArr* x, const CvArr* y, CvArr* magnitude,
                    CvArr* angle=NULL, int angle_in_degrees=0 );
```

x
　　x 坐标数组
y
　　y 坐标数组
magnitude
　　存储向量长度输出数组， 如果不是必要的它可以为空（NULL）
angle
　　存储角度输出数组, 如果不是必要的它可以为空（NULL）。它可以被标准化为弧度 (0..2π) 或者度数(0..360°)

所有的数组只支持浮点类型的运算,也即x,y,magnitude,angle必须是浮点类型的数组。

angle_in_degrees
　　指示角度是用弧度或者度数表示的标志，缺省模式为弧度

函数 cvCartToPolar 计算二维向量(x(I),y(I))的长度，角度，或者两者同时计算:

$$magnitude(I) = sqrt(x(I)^2 + y(I)^2),$$

$$angle(I) = atan(y(I) / x(I))$$

角度的精确度 ≈0.1°. (0,0) 点的角度被设置为 0.

（建议：英文文档虽然是写成atan(y(I)/x(I))，但是建议和C中的表达方式统一。atan不能识别在那个象限，只能返回0-180°，atan2(x,y)才能返回0-360°的值）

[编辑]

PolarToCart

计算极坐标形式的二维向量对应的直角坐标

```
void cvPolarToCart( const CvArr* magnitude, const CvArr* angle,
                    CvArr* x, CvArr* y, int angle_in_degrees=0 );
```

magnitude
　　长度数组.如果为空（NULL），长度被假定为全是 1's.
angle

角度数组,弧度或者角度表示.

x
输出 x 坐标数组, 如果不需要，可以为空（NULL）.

y
输出 y 坐标数组, 如果不需要，可以为空（NULL）.

angle_in_degrees
指示角度是用弧度或者度数表示的标志，缺省模式为弧度

函数 cvPolarToCart 计算每个向量magnitude(I)*exp(angle(I)*j), j=sqrt(-1)的 x 坐标，y 坐标或者两者都计算:

```
x(I)=magnitude(I)*cos(angle(I)),
y(I)=magnitude(I)*sin(angle(I))
```

[编辑]

Pow

对数组内每个元素求幂

```
void cvPow( const CvArr* src, CvArr* dst, double power );
```

src
输入数组

dst
输出数组, 应该和输入数组有相同的类型

power
幂指数

函数 cvPow 计算输入数组的每个元素的 p 次幂:

```
dst(I)=src(I)^p, 如果p是整数
否则dst(I)=abs(src(I))^p
```

也就是说，对于非整型的幂指数使用输入数组元素的绝对值进行计算。然而，使用一些额外的操作，负值也可以得到正确的结果，象下面的例子，计算数组元素的立方根:

```
CvSize size = cvGetSize(src);
CvMat* mask = cvCreateMat( size.height, size.width, CV_8UC1 );
cvCmpS( src, 0, mask, CV_CMP_LT ); /* 查找负数 */
cvPow( src, dst, 1./3 );
cvSubRS( dst, cvScalarAll(0), dst, mask ); /* 输入的负值的结果求反 */
cvReleaseMat( &mask );
```

对于一些幂值, 例如整数值, 0.5 和 -0.5, 优化算法被使用。

[编辑]

Exp

计算数组元素的指数幂

```
void cvExp( const CvArr* src, CvArr* dst );
```

src
输入数组

dst
输出数组, 它应该是 double 型的或者和输入数组有相同的类型

函数 cvExp 计算输入数组的每个元素的 e 次幂:

```
dst(I)=exp(src(I))
```

最大相对误差为 ≈7e-6. 通常, 该函数转换无法输出的值为 0 输出。

[编辑]

Log

计算每个数组元素的绝对值的自然对数

```
void cvLog( const CvArr* src, CvArr* dst );
```

src
　　输入数组。
dst
　　输出数组，它应该是 double 型的或者和输入数组有相同的类型。

函数 cvLog 计算输入数组每个元素的绝对值的自然对数:

```
dst(I)=log(abs(src(I))), src(I)!=0
dst(I)=C,  src(I)=0
```

这里 C 是一个大负数 (≈-700 现在的实现中)。

[编辑]

SolveCubic

求解曲线函数的实根

```
void cvSolveCubic( const CvArr* coeffs, CvArr* roots );
```

coeffs
　　等式系数, 一个三到四个元素的数组.
roots
　　输出的矩阵等式的实根。它应该具有三个元素.

函数 cvSolveCubic 求解曲线函数的实根:

　　coeffs[0]*x^3 + coeffs[1]*x^2 + coeffs[2]*x + coeffs[3] = 0
　　(如果coeffs是四元素的矢量)

或者

　　x^3 + coeffs[0]*x^2 + coeffs[1]*x + coeffs[2] = 0
　　(如果coeffs是三元素的矢量)

函数返回求解得到的实根数目. 实根被存储在矩阵root中, 如果只有一个实根则用0来替代相关值.

[编辑]

随机数生成

[编辑]

RNG

初始化随机数生成器状态

```
CvRNG cvRNG( int64 seed=-1 );
```

seed

64-bit 的值用来初始化一个随机序列

函数 cvRNG 初始化随机数生成器并返回其状态。指向这个状态的指针可以传递给函数 cvRandInt, cvRandReal 和 cvRandArr . 在通常的实现中使用一个 multiply-with-carry generator 。

[编辑]

RandArr

用随机数填充数组并更新 RNG 状态

```
void cvRandArr( CvRNG* rng, CvArr* arr, int dist_type, CvScalar param1, CvScalar param2 );
```

rng
被 cvRNG 初始化的 RNG 状态.

arr
输出数组

dist_type
分布类型:

CV_RAND_UNI - 均匀分布
CV_RAND_NORMAL - 正态分布 或者 高斯分布

param1
分布的第一个参数。如果是均匀分布它是随机数范围的闭下边界。如果是正态分布它是随机数的平均值。

param2
分布的第二个参数。如果是均匀分布它是随机数范围的开上边界。如果是正态分布它是随机数的标准差。

函数 cvRandArr 用均匀分布的或者正态分布的随机数填充输出数组。在下面的例子中该函数被用来添加一些正态分布的浮点数到二维数组的随机位置。

```
/* let's noisy_screen be the floating-point 2d array that is to be "crapped" */
CvRNG rng_state = cvRNG(0xffffffff);
int i, pointCount = 1000;
/* allocate the array of coordinates of points */
CvMat* locations = cvCreateMat( pointCount, 1, CV_32SC2 );
/* arr of random point values */
CvMat* values = cvCreateMat( pointCount, 1, CV_32FC1 );
CvSize size = cvGetSize( noisy_screen );

cvRandInit( &rng_state,
            0, 1, /* 现在使用虚参数以后再调整 */
            0xffffffff /*这里使用一个确定的种子 */,
            CV_RAND_UNI /* 指定为均匀分布类型 */ );

/* 初始化 locations */
cvRandArr( &rng_state, locations, CV_RAND_UNI, cvScalar(0,0,0,0),
cvScalar(size.width,size.height,0,0) );

/* modify RNG to make it produce normally distributed values */
rng_state.disttype = CV_RAND_NORMAL;
cvRandSetRange( &rng_state,
                30 /* deviation */,
                100 /* average point brightness */,
                -1 /* initialize all the dimensions */ );
/* generate values */
cvRandArr( &rng_state, values, CV_RAND_NORMAL,
           cvRealScalar(100), // average intensity
           cvRealScalar(30) // deviation of the intensity
          );

/* set the points */
for( i = 0; i < pointCount; i++ )
{
    CvPoint pt = *(CvPoint*)cvPtr1D( locations, i, 0 );
```

```
    float value = *(float*)cvPtr1D( values, i, 0 );
    *((float*)cvPtr2D( noisy_screen, pt.y, pt.x, 0 )) += value;
}

/* not to forget to release the temporary arrays */
cvReleaseMat( &locations );
cvReleaseMat( &values );

/* RNG state does not need to be deallocated */
```

[编辑]

RandInt

返回 32-bit 无符号整型并更新 RNG

```
unsigned cvRandInt( CvRNG* rng );
```

rng
: 被 cvRNG 初始化的 RNG 状态，被 RandSetRange (虽然, 后面这个函数对我们正讨论的函数的结果没有什么影响)随意地设置。

函数 cvRandInt 返回均匀分布的随机 32-bit 无符号整型值并更新 RNG 状态。它和 C 运行库里面的 rand() 函数十分相似，但是它产生的总是一个 32-bit 数而 rand() 返回一个 0 到 RAND_MAX （它是 2**16 或者 2**32, 依赖于操作平台）之间的数。

该函数用来产生一个标量随机数，例如点， patch sizes, table indices 等,用模操作可以产生一个确定边界的整数，人和其他特定的边界缩放到 0.. 1可以产生一个浮点数。下面是用 cvRandInt 重写的前一个函数讨论的例子:

```
/* the input and the task is the same as in the previous sample. */
CvRNG rng_state = cvRNG(0xffffffff);
int i, pointCount = 1000;
/* ... - no arrays are allocated here */
CvSize size = cvGetSize( noisy_screen );
/* make a buffer for normally distributed numbers to reduce call overhead */
#define bufferSize 16
float normalValueBuffer[bufferSize];
CvMat normalValueMat = cvMat( bufferSize, 1, CV_32F, normalValueBuffer );
int valuesLeft = 0;

for( i = 0; i < pointCount; i++ )
{
    CvPoint pt;
    /* generate random point */
    pt.x = cvRandInt( &rng_state ) % size.width;
    pt.y = cvRandInt( &rng_state ) % size.height;

    if( valuesLeft <= 0 )
    {
        /* fulfill the buffer with normally distributed numbers if the buffer is empty */
        cvRandArr( &rng_state, &normalValueMat, CV_RAND_NORMAL, cvRealScalar(100),
cvRealScalar(30) );
        valuesLeft = bufferSize;
    }
    ((float*)cvPtr2D( noisy_screen, pt.y, pt.x, 0 ) = normalValueBuffer[--valuesLeft];
}

/* there is no need to deallocate normalValueMat because we have
both the matrix header and the data on stack. It is a common and efficient
practice of working with small, fixed-size matrices */
```

[编辑]

RandReal

返回浮点型随机数并更新 RNG

```
double cvRandReal( CvRNG* rng );
```

rng
被 cvRNG 初始化的 RNG 状态

函数 cvRandReal 返回均匀分布的随机浮点数，范围在 0..1 之间 (不包括 1)。

[编辑]

离散变换

[编辑]

DFT

执行一维或者二维浮点数组的离散傅立叶正变换或者离散傅立叶逆变换

```
#define CV_DXT_FORWARD  0
#define CV_DXT_INVERSE  1
#define CV_DXT_SCALE:2
#define CV_DXT_ROWS: 4
#define CV_DXT_INV_SCALE (CV_DXT_SCALE|CV_DXT_INVERSE)
#define CV_DXT_INVERSE_SCALE CV_DXT_INV_SCALE
void cvDFT( const CvArr* src, CvArr* dst, int flags, int nonzero_rows=0);
```

src
输入数组, 实数或者复数.
dst
输出数组，和输入数组有相同的类型和大小。
flags
变换标志, 下面的值的组合:

CV_DXT_FORWARD - 正向 1D 或者 2D 变换. 结果不被缩放.
CV_DXT_INVERSE - 逆向 1D 或者 2D 变换. 结果不被缩放.当然 CV_DXT_FORWARD 和 CV_DXT_INVERSE 是互斥的.
CV_DXT_SCALE - 对结果进行缩放: 用数组元素除以它. 通常, 它和 CV_DXT_INVERSE 组合在一起,可以使用缩写 CV_DXT_INV_SCALE.
CV_DXT_ROWS - 输入矩阵的每个独立的行进行整型或者逆向变换。这个标志允许用户同时变换多个向量，减少开销(它往往比处理它自己要快好几倍), 进行 3D 和高维的变换等等。

nonzero_rows
输入矩阵中非0行的个数（在2维的Forward变换中），或者是输出矩阵中感兴趣的行（在反向的2维变换中）。如果这个值是负数，0，或者大于总行数的一个值，它将会被忽略。这个参数可以用来加速2维DFT/IDFT的速度。见下面的例子。

函数 cvDFT 执行一维或者二维浮点数组的离散傅立叶正变换或者离散傅立叶逆变换:

N 元一维向量的正向傅立叶变换:

y = F(N)•x, 这里 F(N)jk=exp(-i•2Pi•j•k/N), i=sqrt(-1)

$$y[k] = \sum_{n=0}^{N-1} x[n] e^{-i\frac{2\pi}{N}nk} k = 0, 1, ...N-1$$

N 元一维向量的逆向傅立叶变换:

x'= (F(N))-1•y = conj(F(N))•y
x = (1/N)•x

M×N 元二维向量的正向傅立叶变换:

Y = F(M)•X•F(N)

M×N 元二维向量的逆向傅立叶变换:

X'= conj(F(M))•Y•conj(F(N))
X = (1/(M•N))•X'

假设时实数数据 (单通道) ,从 IPL 借鉴过来的压缩格式被用来表现一个正向傅立叶变换的结果或者逆向傅立叶变换的输入:

```
Re Y0,0:  Re Y0,1:Im Y0,1:Re Y0,2: Im Y0,2  ...  Re Y0,N/2-1   Im Y0,N/2-1  Re Y0,N/2
Re Y1,0:  Re Y1,1:Im Y1,1:Re Y1,2: Im Y1,2  ...  Re Y1,N/2-1   Im Y1,N/2-1  Re Y1,N/2
Im Y1,0:  Re Y2,1:Im Y2,1:Re Y2,2: Im Y2,2  ...  Re Y2,N/2-1   Im Y2,N/2-1  Im Y2,N/2
.....................................................................................
Re YM/2-1,0   Re YM-3,1   Im YM-3,1  Re YM-3,2   Im YM-3,2 ...  Re YM-3,N/2-1  Im YM-3,N/2-1 Re
YM-3,N/2
Im YM/2-1,0   Re YM-2,1   Im YM-2,1  Re YM-2,2   Im YM-2,2 ...  Re YM-2,N/2-1  Im YM-2,N/2-1 Im
YM-2,N/2
Re YM/2,0:Re YM-1,1   Im YM-1,1  Re YM-1,2   Im YM-1,2  ... Re YM-1,N/2-1  Im YM-1,N/2-1 Im YM-
1,N/2
```

注意:如果 N 时偶数最后一列存在（is present）, 如果 M 时偶数最后一行（is present）.

如果是一维实数的变换结果就像上面矩阵的第一行的形式。 利用DFT求解二维卷积

```
   CvMat* A = cvCreateMat( M1, N1, CV_32F );
   CvMat* B = cvCreateMat( M2, N2, A->type );

   // it is also possible to have only abs(M2-M1)+1×abs(N2-N1)+1
   // part of the full convolution result
   CvMat* conv = cvCreateMat( A->rows + B->rows - 1, A->cols + B->cols - 1, A->type );

   // initialize A and B
   ...

   int dft_M = cvGetOptimalDFTSize( A->rows + B->rows - 1 );
   int dft_N = cvGetOptimalDFTSize( A->cols + B->cols - 1 );

   CvMat* dft_A = cvCreateMat( dft_M, dft_N, A->type );
   CvMat* dft_B = cvCreateMat( dft_M, dft_N, B->type );
   CvMat tmp;

   // copy A to dft_A and pad dft_A with zeros
   cvGetSubRect( dft_A, &tmp, cvRect(0,0,A->cols,A->rows));
   cvCopy( A, &tmp );
   cvGetSubRect( dft_A, &tmp, cvRect(A->cols,0,dft_A->cols - A->cols,A->rows));
   cvZero( &tmp );
   // no need to pad bottom part of dft_A with zeros because of
   // use nonzero_rows parameter in cvDFT() call below

   cvDFT( dft_A, dft_A, CV_DXT_FORWARD, A->rows );

   // repeat the same with the second array
   cvGetSubRect( dft_B, &tmp, cvRect(0,0,B->cols,B->rows));
   cvCopy( B, &tmp );
   cvGetSubRect( dft_B, &tmp, cvRect(B->cols,0,dft_B->cols - B->cols,B->rows));
   cvZero( &tmp );
   // no need to pad bottom part of dft_B with zeros because of
   // use nonzero_rows parameter in cvDFT() call below

   cvDFT( dft_B, dft_B, CV_DXT_FORWBRD, B->rows );

   cvMulSpectrums( dft_A, dft_B, dft_A, 0 /* or CV_DXT_MUL_CONJ to get correlation
:::::::::::: rather than convolution */ );

   cvDFT( dft_A, dft_A, CV_DXT_INV_SCALE, conv->rows ); // calculate only the top part
   cvGetSubRect( dft_A, &tmp, cvRect(0,0,conv->cols,conv->rows) );

   cvCopy( &tmp, conv );
```

[编辑]

GetOptimalDFTSize

对于给定的矢量尺寸返回最优DFT尺寸

```
int cvGetOptimalDFTSize( int size0 );
```

size0
　　矢量长度.

函数 cvGetOptimalDFTSize 返回最小值 N that is greater to equal to size0, such that DFT of a vector of size N can be computed fast. In the current implementation N=2p×3q×5r for some p, q, r.

The function returns a negative number if size0 is too large (very close to INT_MAX)

MulSpectrums

两个傅立叶频谱的每个元素的乘法（Performs per-element multiplication of two Fourier spectrums）

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

src1
　　第一输入数组
src2
　　第二输入数组
dst
　　输出数组，和输入数组有相同的类型和大小。
flags
　　下面列举的值的组合:

　　CV_DXT_ROWS - 把数组的每一行视为一个单独的频谱 (参见 cvDFT 的参数讨论).
　　CV_DXT_MUL_CONJ - 在做乘法之前取第二个输入数组的共轭.

函数 cvMulSpectrums 执行两个 CCS-packed 或者实数或复数傅立叶变换的结果复数矩阵的每个元素的乘法。（performs per-element multiplication of the two CCS-packed or complex matrices that are results of real or complex Fourier transform.）

该函数和 cvDFT 可以用来快速计算两个数组的卷积.

DCT

执行一维或者二维浮点数组的离散余弦变换或者离散反余弦变换

```
#define CV_DXT_FORWARD  0
#define CV_DXT_INVERSE  1
#define CV_DXT_ROWS: 4
void cvDCT( const CvArr* src, CvArr* dst, int flags );
```

src
　　输入数组, 1D 或者 2D 实数数组.
dst
　　输出数组，和输入数组有相同的类型和大小。
flags
　　变换标志符, 下面值的组合:

　　CV_DXT_FORWARD - 1D 或者 2D 余弦变换.

CV_DXT_INVERSE - 1D or 2D 反余弦变换.
CV_DXT_ROWS - 对输入矩阵的每个独立的行进行余弦或者反余弦变换. 这个标志允许用户同时进行多个向量的变换，可以用来减少开销（它往往比处理它自己要快好几倍），以及 3D 和高维变换等等。

函数 cvDCT 执行一维或者二维浮点数组的离散余弦变换或者离散反余弦变换:

N 元一维向量的余弦变换:

y = C(N)•x, 这里 C(N)jk=sqrt((j==0?1:2)/N)•cos(Pi•(2k+1)•j/N)

$$y[m] = \sum_{k=0}^{N-1} x_k \cos\left[\frac{\pi}{N} m \left(k + \frac{1}{2}\right)\right]$$

N 元一维向量的反余弦变换:

x = (C(N))-1•y = (C(N))T•y

M×N 元二维向量的余弦变换:

Y = (C(M))•X•(C(N))T

M×N 元二维向量的反余弦变换:

X = (C(M))T•Y•C(N)

Cxcore动态结构

Wikipedia，自由的百科全书

目录

[隐藏]

[编辑]

内存存储（**memory storage)**

[编辑]

CvMemStorage

Growing memory storage

```
typedef struct CvMemStorage
{
    struct CvMemBlock* bottom;/* first allocated block */
    struct CvMemBlock* top; /* the current memory block - top of the stack */
    struct CvMemStorage* parent; /* borrows new blocks from */
    int block_size; /* block size */
```

```
    int free_space; /* free space in the top block (in bytes) */
} CvMemStorage;
```

内存存储器是一个可用来存储诸如序列，轮廓，图形,子划分等动态增长数据结构的底层结构。它是由一系列以同等大小的内存块构成，呈列表型 ---bottom 域指的是列首，top 域指的是当前指向的块但未必是列尾.在bottom和top之间所有的块(包括bottom, 不包括top）被完全占据了空间；在 top和列尾之间所有的块（包括块尾，不包括top)则是空的；而top块本身则被占据了部分空间 -- free_space 指的是top块剩余的空字节数。

新分配的内存缓冲区（或显式的通过 cvMemStorageAlloc 函数分配，或隐式的通过 cvSeqPush, cvGraphAddEdge等高级函数分配）总是起始于当前块（即top块)的剩余那部分，如果剩余那部分能满足要求（够分配的大小）。分配 后，free_space 就减少了新分配的那部分内存大小，外加一些用来保存适当列型的附加大小。当top块的剩余空间无法满足被分配的块（缓冲区）大小时，top块的下一个存储 块被置为当前块（新的top块） -- free_space 被置为先前分配的整个块的大小。

如果已经不存在空的存储块（即：top块已是列尾），则必须再分配一个新的块（或从parent那继承,见 cvCreateChildMemStorage)并将该块加到列尾上去。于是，存储器（memory storage)就如同栈（Stack)那样, bottom指向栈底，(top, free_space)对指向栈顶。栈顶可通过 cvSaveMemStoragePos保存，通过 cvRestoreMemStoragePos 恢复指向， 通过 cvClearStorage 重置。

CvMemBlock

内存存储块结构

```
typedef struct CvMemBlock
{
        struct CvMemBlock* prev;
        struct CvMemBlock* next;
} CvMemBlock;
```

CvMemBlock 代表一个单独的内存存储块结构。 内存存储块中的实际数据存储在 header块 之后(即：存在一个头指针 head 指向的块 header ，该块不存储数据）,于是，内存块的第 i 个字节可以通过表达式 ((char*)(mem_block_ptr+1))[i] 获得。然而，通常没必要直接去获得存储结构的域。

CvMemStoragePos

内存存储块地址

```
typedef struct CvMemStoragePos
{
    CvMemBlock* top;
    int free_space;
} CvMemStoragePos;
```

该结构（如以下所说）保存栈顶的地址，栈顶可以通过 cvSaveMemStoragePos 保存，也可以通过 cvRestoreMemStoragePos 恢复。

CreateMemStorage

创建内存块

```
CvMemStorage* cvCreateMemStorage( int block_size=0 );
```

block_size
存储块的大小以字节表示。如果大小是 0 byte, 则将该块设置成默认值 -- 当前默认大小为64k.

函数 cvCreateMemStorage 创建一内存块并返回指向块首的指针。起初，存储块是空的。头部（即：header)的所有域值都为 0，除了 block_size 外.

[编辑]

CreateChildMemStorage

创建子内存块

```
CvMemStorage* cvCreateChildMemStorage( CvMemStorage* parent );
```

parent
　　父内存块

函数 cvCreateChildMemStorage 创建一类似于普通内存块的子内存块，除了内存分配/释放机制不同外。当一个子存储块需要一个新的块加入时，它就试图从parent 那得到这样一个块。如果 parent 中 还未被占据空间的那些块中的第一个块是可获得的，就获取第一个块（依此类推），再将该块从 parent 那里去除。如果不存在这样的块，则 parent 要么分配一个，要么从它自己 parent (即：parent 的 parent) 那借个过来。换句话说，完全有可能形成一个链或更为复杂的结构，其中的内存存储块互为 child/ parent 关系（父子关系）。当子存储结构被释放或清除，它就把所有的块还给各自的 parent. 在其他方面，子存储结构同普通存储结构一样。

子存储结构在下列情况中是非常有用的。想象一下，如果用户需要处理存储在某个块中的动态数据，再将处理的结果存放在该块中。在使用了最简单 的方法处理后，临时数据作为输入和输出数据被存放在了同一个存储块中，于是该存储块看上去就类似下面处理后的样子： Dynamic data processing without using child storage.

结果，在存储块中，出现了垃圾（临时数据）。然而，如果在开始处理数据前就先建立一个子存储块，将临时数据写入子存储块中并在最后释放子存 储块，那么最终在 源/目的存储块 （source / destination storage) 中就不会出现垃圾, 于是该存储块看上去应该是如下形式：Dynamic data processing using a child storage.

[编辑]

ReleaseMemStorage

释放内存块

```
void cvReleaseMemStorage( CvMemStorage** storage );
```

storage
　　指向被释放了的存储块的指针

函数 cvReleaseMemStorage 释放所有的存储（内存）块 或者 将它们返回给各自的 parent（如果需要的话）。 接下来再释放 header块（即：释放头指针 head 指向的块 = free(head)）并清除指向该块的指针（即：head = NULL)。在释放作为 parent 的块之前，先清除各自的 child 块。

[编辑]

ClearMemStorage

清空内存存储块

```
void cvClearMemStorage( CvMemStorage* storage );
```

storage
　　存储存储块

函数 cvClearMemStorage 将存储块的 top 置到存储块的头部（注：清空存储块中的存储内容）。该函数并不释放内存（仅清空内存）。假使该内存块有一个父内存块（即：存在一内存块与其有父子关系），则函数就将所有的块返回给其 parent.

[编辑]

MemStorageAlloc

在存储块中分配以内存缓冲区

```
void* cvMemStorageAlloc( CvMemStorage* storage, size_t size );
```

storage
　　内存块.

size
　　缓冲区的大小.

函数 cvMemStorageAlloc 在存储块中分配一内存缓冲区。该缓冲区的大小不能超过内存块的大小，否则就会导致运行时错误。缓冲区的地址被调整为CV_STRUCT_ALIGN 字节 （当前为 sizeof(double)).

[编辑]

MemStorageAllocString

在存储块中分配一文本字符串

```
typedef struct CvString
{
    int len;
    char* ptr;
}
CvString;

CvString cvMemStorageAllocString( CvMemStorage* storage, const char* ptr, int len=-1 );
```

storage
　　存储块

ptr
　　字符串

len
　　字符串的长度（不计算‘\0’）。如果参数为负数，函数就计算该字符串的长度。

函数 cvMemStorageAlloString 在存储块中创建了一字符串的拷贝。它返回一结构，该结构包含字符串的长度（该长度或通过用户传递，或通过计算得到）和指向被拷贝了的字符串的指针。

[编辑]

SaveMemStoragePos

保存内存块的位置（地址）

```
void cvSaveMemStoragePos( const CvMemStorage* storage, CvMemStoragePos* pos );
```

storage
　　内存块.

pos
　　内存块顶部位置。

函数 cvSaveMemStoragePos 将存储块的当前位置保存到参数 pos 中。 函数 cvRestoreMemStoragePos 可进一步获取该位置（地址）。

[编辑]

RestoreMemStoragePos

恢复内存存储块的位置

```
void cvRestoreMemStoragePos( CvMemStorage* storage, CvMemStoragePos* pos );
```

storage
　　内存块.

pos
　　新的存储块的位置

函数 cvRestoreMemStoragePos 通过参数 pos 恢复内存块的位置。该函数和函数 cvClearMemStorage 是释放被占用内存块的唯一方法。注意：没有什么方法可去释放存储块中被占用的部分内存。

[编辑]

序列

[编辑]

CvSeq

可动态增长元素序列(OpenCV_1.0已发生改变，详见cxtypes.h) Growable sequence of elements

```
#define CV_SEQUENCE_FIELDS() \
    int flags; /* micsellaneous flags */ \
    int header_size; /* size of sequence header */ \
    struct CvSeq* h_prev; /* previous sequence */ \
    struct CvSeq* h_next; /* next sequence */ \
    struct CvSeq* v_prev; /* 2nd previous sequence */ \
    struct CvSeq* v_next; /* 2nd next sequence */ \
    int total; /* total number of elements */ \
    int elem_size;/* size of sequence element in bytes */ \
    char* block_max;/* maximal bound of the last block */ \
    char* ptr; /* current write pointer */ \
    int delta_elems; /* how many elements allocated when the sequence grows (sequence
granularity) */ \
    CvMemStorage* storage; /* where the seq is stored */ \
    CvSeqBlock* free_blocks; /* free blocks list */ \
    CvSeqBlock* first; /* pointer to the first sequence block */


typedef struct CvSeq
{
    CV_SEQUENCE_FIELDS()
} CvSeq;
```

结构CvSeq是所有OpenCV动态数据结构的基础。 在1.0版本中,将前六个成员剥离出来定义成一个宏. 通过不同寻常的宏定义简化了带有附加参数的结构 CvSeq 的扩展。为了扩展 CvSeq, 用户可以定义一新的数据结构或在通过宏CV_SEQUENCE_FIELDS()所包括的 CvSeq 的域后在放入用户自定义的域。

有两种类型的序列 -- 稠密序列和稀疏序列。稠密序列都派生自 CvSeq, 它们用来代表可扩展的一维数组 -- 向量，栈，队列，双端队列。数据间不存在空隙（即：连续存放）-- 如果元素从序列中间被删除或插入新的元素到序列中（不是两端），那么此元素后边的相关元素会被移动。稀疏序列都派生自 CvSet,后面会有详细的讨论。它们都是由节点所组成的序列，每一个节点要么被占用空间要么是空，由 flag 标志指定。这些序列作为无序的数据结构而被使用，如点集，图，哈希表等。

域 header_size(结构的大小) 含有序列头部节点的实际大小，此大小大于或等于 sizeof(CvSeq).当这个宏用在序列中时,应该等于 sizeof(CvSeq),若这个宏用在其他结构中,如CvContour,结构的大小应该大于sizeof(CvSeq); 域 h_prev, h_next, v_prev, v_next 可用来创建不同序列的层次结构。域 h_prev, h_next 指向同一层次结构前一个和后一个序列，而域 v_prev, v_next指向在垂直方向上的前一个和后一个序列，即：父亲和子孙。

域 first 指向第一个序列快，块结构在后面描述。

域 total 包含稠密序列的总元素数和稀疏序列被分配的节点数。

域 flags 的高16位描述（包含）特定的动态结构类型（CV_SEQ_MAGIC_VAL 表示稠密序列,CV_SET_MAGIC_VAL 表示稀疏序列），同时包含形形色色的信息。

低 CV_SEQ_ELTYPE_BITS 位包含元素类型的 ID(标示符）。大多数处理函数并不会用到元素类型，而会用到存放在 elem_size 中的元素大小 。如果序列中包含 CvMat 中的数据，那么元素的类型就与 CvMat 中的类型相匹配， 如：CV_32SC2 可以被使用为由二维空间中的点序列， CV_32FC1用为由浮点数组成的序列等。通过宏 CV_SEQ_ELTYPE(seq_header_ptr) 来获取序列中元素的类型。处理数字序列的函数判断： elem.size 等同于序列元素的大小。除了与 CvMat 相兼容的类型外，还有几个在头 cvtypes.h 中定义的额外的类型。

```
Standard Types of Sequence Elements

    #define CV_SEQ_ELTYPE_POINT          CV_32SC2  /* (x,y) */
    #define CV_SEQ_ELTYPE_CODE           CV_8UC1   /* freeman code: 0..7 */
    #define CV_SEQ_ELTYPE_GENERIC        0 /* unspecified type of sequence elements */
    #define CV_SEQ_ELTYPE_PTR            CV_USRTYPE1 /* =6 */
    #define CV_SEQ_ELTYPE_PPOINT         CV_SEQ_ELTYPE_PTR  /* &elem: pointer to element of other
sequence */
    #define CV_SEQ_ELTYPE_INDEX          CV_32SC1  /* #elem: index of element of some other
sequence */
    #define CV_SEQ_ELTYPE_GRAPH_EDGE     CV_SEQ_ELTYPE_GENERIC  /* &next_o, &next_d, &vtx_o,
&vtx_d */
    #define CV_SEQ_ELTYPE_GRAPH_VERTEX   CV_SEQ_ELTYPE_GENERIC  /* first_edge, &(x,y) */
    #define CV_SEQ_ELTYPE_TRIAN_ATR      CV_SEQ_ELTYPE_GENERIC  /* vertex of the binary tree   */
    #define CV_SEQ_ELTYPE_CONNECTED_COMP CV_SEQ_ELTYPE_GENERIC  /* connected component  */
    #define CV_SEQ_ELTYPE_POINT3D        CV_32FC3  /* (x,y,z)  */
```

后面的 CV_SEQ_KIND_BITS 字节表示序列的类型:

```
Standard Kinds of Sequences

    /* generic (unspecified) kind of sequence */
    #define CV_SEQ_KIND_GENERIC     (0 << CV_SEQ_ELTYPE_BITS)

    /* dense sequence suntypes */
    #define CV_SEQ_KIND_CURVE       (1 << CV_SEQ_ELTYPE_BITS)
    #define CV_SEQ_KIND_BIN_TREE    (2 << CV_SEQ_ELTYPE_BITS)

    /* sparse sequence (or set) subtypes */
    #define CV_SEQ_KIND_GRAPH       (3 << CV_SEQ_ELTYPE_BITS)
    #define CV_SEQ_KIND_SUBDIV2D    (4 << CV_SEQ_ELTYPE_BITS)
```


CvSeqBlock

连续序列块

```
typedef struct CvSeqBlock
{
    struct CvSeqBlock* prev; /* previous sequence block */
    struct CvSeqBlock* next; /* next sequence block */
    int start_index; /* index of the first element in the block +
    sequence->first->start_index */
    int count; /* number of elements in the block */
    char* data; /* pointer to the first element of the block */
} CvSeqBlock;
```

序列块构成一个双向的循环列表，因此指针 prev 和 next 永远不会为 null, 而总是指向序列中的前一个和后一个序列块。也就是说：最后一个序列块的 next 指向的就是序列中的第一个块，而第一个块的 prev 指向最后一个块。域 start_index 和 count 有助于跟踪序列中块的位置。 例如，一个含10个元素的序列被分成了3块，每一块的大小分别为3， 5， 2，第一块的参数 start_index 为 2， 那么该序列的 (start_index, count) 相应为（2，3），（5，5），（10，2）。第一个块的参数 start_index 通常为 0，除非一些元素已被插入到序列中。

[编辑]

CvSlice

序列分割

```
typedef struct CvSlice
{
    int start_index;
    int end_index;
} CvSlice;

inline CvSlice cvSlice( int start, int end );
#define CV_WHOLE_SEQ_END_INDEX 0x3fffffff
#define CV_WHOLE_SEQ  cvSlice(0, CV_WHOLE_SEQ_END_INDEX)

/* calculates the sequence slice length */
int cvSliceLength( CvSlice slice, const CvSeq* seq );
```

有关序列的一些操作函数将 CvSlice 作为输入参数，默认情况下该参数通常被设置成整个序列（CV_WHOLE_SEQ)。start_index 和 end_index 任何一个都可以是负数或 超过序列长度，start_index 是闭界，end_index 是开界。如果两者相等，那么分割被认为是空分割（即：不包含任何元素）。由于序列被看作是循环结构， 所以分割可以选择序列中靠后的几个元素，靠前的参数反而跟着它们，如 cvSlice（-2，3）。函数用下列方法来规范分割参数：首先， 调用 cvSliceLength 来决定分割的长度，然后， start_index 被使用类似于 cvGetSeqElem 的参数来规范（例如：负数也被允许）。实际的分割操作起始于规范化了的 start_index，中止于 start_index + cvSliceLength()。（再次假设序列是循环结构）

如果函数并不接受分割参数，但你还是想要处理序列的一部分，那么可以使用函数 cvSeqSlice 获取子序列。

[编辑]

CreateSeq

创建一序列

```
CvSeq* cvCreateSeq( int seq_flags, int header_size,
                    int elem_size, CvMemStorage* storage );
```

seq_flags
: 序列的符号标志。如果序列不会被传递给任何使用特定序列的函数，那么将它设为 0， 否则从预定义的序列类型中选择一合适的类型。

header_size
: 序列头部的大小；必须大于或等于 sizeof(CvSeq). 如果制定了类型或它的扩展名，则此类型必须适合基类的头部大小。

elem_size
: 元素的大小，以字节计。这个大小必须与序列类型相一致。例如，对于一个点的序列，元素类型 CV_SEQ_ELTYPE_POINT 应当被指定， 参数elem_size 必须等同于 sizeof(CvPoint).

函数 cvCreateSeq 创建一序列并且返回指向该序列的指针。函数在存储块中分配序列的头部作为一个连续躯体，并且设置结构的 flags域, elem_size域, header_size域 和 storage域 的值为被传递过来的值，设置 delta_elems 为默认值（可通过函数 cvSetSeqBlockSize 重新对其赋值），清空其他的头 部域，包括前sizeof(CvSeq) 个字节的空间。

[编辑]

SetSeqBlockSize

设置序列块的大小

```
void cvSetSeqBlockSize( CvSeq* seq, int delta_elems );
```

seq
　　序列

delta_elems
　　满足元素所需的块的大小

函数 cvSetSeqBlockSize 会对内存分配的粒度产生影响。 当序列缓冲区中空间消耗完时，函数为 delta_elems 个序列元素分配空间。如果新分配的空间与 之前分配的空间相邻的话，这两个块就合并，否则，就创建了一个新的序列快。因此，参数值越大，序列中出现碎片的可能性就越小，不过内存中更多的空间将被浪 费。当序列被创建后，参数 delta_elems 大小将被设置为 默认大小（1K).之后， 就可随时调用该函数，并影响内存分配。 函数可以修改被传递过来的参数值，以满足内存块的大小限制 。

[编辑]

SeqPush

添加元素到序列的尾部

```
char* cvSeqPush( CvSeq* seq, void* element=NULL );
```

seq
　　块
element
　　添加的元素

函数 cvSeqPush 在序列块的尾部添加一元素并返回指向该元素得指针。如果输入参数为 null, 函数就仅仅分配一空间，留给下一个元素使用。下列代码说明如何使用该函数去创建一空间。

```
The following code demonstrates how to create a new sequence using this function:

CvMemStorage* storage = cvCreateMemStorage(0);
CvSeq* seq = cvCreateSeq( CV_32SC1, /* sequence of integer elements */
                          sizeof(CvSeq), /* header size - no extra fields */
                          sizeof(int), /* element size */
                          storage /* the container storage */ );
int i;
for( i = 0; i < 100; i++ )
{
    int* added = (int*)cvSeqPush( seq, &i );
    printf( "%d is added\n", *added );
}

...
/* release memory storage in the end */
cvReleaseMemStorage( &storage );
```

函数 cvSeqPush 的时间复杂度为 O(1). 如果需要分配并使用的空间比较大，则存在一分配较快的函数（见：cvStartWriteSeq 和相关函数）

[编辑]

SeqPop

删除序列尾部元素

```
void cvSeqPop( CvSeq* seq, void* element=NULL );
```

seq
　　序列
element
　　可选参数。如果该指针不为空，就拷贝被删元素到指针指向的位置

函数 cvSeqPop 从序列中删除一元素。如果序列已经为空，就报告一错误。函数时间复杂度为 O(1).

[编辑]

SeqPushFront

在序列头部添加元素

```
char* cvSeqPushFront( CvSeq* seq, void* element=NULL );
```

seq
　　序列
element
　　添加的元素

函数 cvSeqPushFront 类似于 cvSeqPush， 不过是在序列头部添加元素。时间复杂度为O(1).

[编辑]

SeqPopFront

删除序列的头部元素

```
void cvSeqPopFront( CvSeq* seq, void* element=NULL );
```

seq
　　序列
element
　　可选参数。如果该指针不为空，就拷贝被珊元素到指针指向的位置。

函数 cvSeqPopFront 删除序列的头部元素。如果序列已经为空，就报告一错误。函数时间复杂度为 O(1).

[编辑]

SeqPushMulti

添加多个元素到序列尾部或头部。

```
void cvSeqPushMulti( CvSeq* seq, void* elements, int count, int in_front=0 );
```

seq
　　序列
elements
　　待添加的元素
count
　　添加的元素个数
in_front
　　标示在头部还是尾部添加元素

　　CV_BACK (= 0) -- 在序列尾部添加元素。
　　CV_FRONT(!= 0) -- 在序列头部添加元素。

函数 cvSeqPushMulti 在序列头部或尾部添加多个元素。 元素按输入数组中的顺序被添加到序列中，不过它们可以添加到不同的序列中

[编辑]

SeqPopMulti

删除多个序列头部或尾部的元素

```
void cvSeqPopMulti( CvSeq* seq, void* elements, int count, int in_front=0 );
```

seq
　　序列
elements
　　待删除的元素
count
　　删除的元素个数
in_front
　　标示在头部还是尾部删除元素

　　CV_BACK (= 0) -- 删除序列尾部元素。
　　CV_FRONT(!= 0) -- 删除序列头部元素。

函数 cvSeqPopMulti 删除多个序列头部或尾部的元素。 如果待删除的元素个数超过了序列中的元素总数，则函数删除尽可能多的元素 。

[编辑]

SeqInsert

在序列中添加元素

```
char* cvSeqInsert( CvSeq* seq, int before_index, void* element=NULL );
```

seq
　　序列
before_index
　　元素插入的位置（索引）。如果插入的位置在 0（允许的参数最小值）前，则该函数等同于函数 cvSeqPushFront. 如果是在 seq_total（允许的参数最大值）后，则该函数等同于 cvSeqPush.
element
　　待插入的元素

函数 cvSeqInsert 移动 从被插入的位置到序列尾部元素所在的位置的所有元素，如果 指针 element 不位 null, 则拷贝 element 中的元素到指定位置。函数返回指向被插入元素的指针。

[编辑]

SeqRemove

从序列中删除指定的元素。

```
void cvSeqRemove( CvSeq* seq, int index );
```

seq
　　目标序列
index
　　被删除元素的索引或位置。

函数 cvSeqRemove 删除seq中指定索引（位置）的元素。如果这个索引超出序列的元素个数，会报告出错。企图从空序列中删除元素，函数也将报告错误。函数通过移动序列中的元素来删除被索引的元素。

[编辑]

ClearSeq

清空序列

```
void cvClearSeq( CvSeq* seq );
```

seq
Sequence.
seq
序列

函数 cvClearSeq 删除序列中的所有元素。函数不会将内存返回到存储器中，当新的元素添加到序列中时，可重新使用该内存。函数时间复杂度为 O(1).

[编辑]

GetSeqElem

返回索引所指定的元素指针

```
char* cvGetSeqElem( const CvSeq* seq, int index );
#define CV_GET_SEQ_ELEM( TYPE, seq, index )  (TYPE*)cvGetSeqElem( (CvSeq*)(seq), (index) )
```

seq
序列
index
索引

函数 cvGetSeqElem 查找序列中索引所指定的元素，并返回指向该元素的指针。如果元素不存在，则返回 0。函数支持负数，即： -1 代表 序列的最后一个元素， -2 代表最后第二个元素，等。如果序列只包含一个块，或者所需的元素在第一个块中，那么应当使用宏, CV_GET_SEQ_ELEM(elemType, seq, index)宏中的参数 elemType 是序列中元素的类型（如：CvPoint), 参数 seq 表示序列， 参数 index 代表所需元素的索引。 该宏首先核查所需的元素是否属于第一个块，如果是，则返回该元素，否则，该宏就调用主函数 GetSeqElem. 如果索引为负数的话，则总是调用函数 cvGetSeqElem。函数的时间复杂度为 O(1), 假设块的大小要比元素的数量要小。

[编辑]

SeqElemIdx

返回序列中元素的索引

```
int cvSeqElemIdx( const CvSeq* seq, const void* element, CvSeqBlock** block=NULL );
```

seq
序列
element
指向序列中元素的指针
block
可选参数， 如果不为空(NULL),则存放包含该元素的块的地址

函数 cvSeqElemIdx 返回元素的索引，如果该元素不存在这个序列中，则返回一负数。

[编辑]

cvSeqToArray

拷贝序列中的元素到一个连续的内存块中

```
void* cvCvtSeqToArray( const CvSeq* seq, void* elements, CvSlice slice=CV_WHOLE_SEQ );
```

seq
序列
elemenets
指向目的（存放拷贝元素的）数组的指针，指针指向的空间必须足够大。

slice
　　拷贝到序列中的序列部分。

函数 cvCvtSeqToArray 拷贝整个序列或部分序列到指定的缓冲区中，并返回指向该缓冲区的指针.

[编辑]

MakeSeqHeaderForArray

构建序列

```
CvSeq* cvMakeSeqHeaderForArray( int seq_type, int header_size, int elem_size,
                                void* elements, int total,
                                CvSeq* seq, CvSeqBlock* block );
```

seq_type
　　序列的类型
header_size
　　序列的头部大小。大小必须大于等于数组的大小。
elem_size
　　元素的大小
elements
　　形成该序列的元素
total
　　序列中元素的总数。参数值必须等于数据的大小
seq
　　指向被使用作为序列头部的局部变量
block
　　指向局部变量的指针

函数 cvMakeSeqHeaderForArray 初始化序列的头部。序列块由用户分配（例如：在栈上）。该函数不拷贝数据。创建的序列只包含一个块，和一个 NULL指针，因此可以读取指针，但试图将元素添加到序列中则多数会引发错误。

[编辑]

SeqSlice

为各个序列碎片建立头

```
CvSeq* cvSeqSlice( const CvSeq* seq, CvSlice slice,
                   CvMemStorage* storage=NULL, int copy_data=0 );
```

seq
　　序列
slice
　　部分序列块
storage
　　存放新的序列和拷贝数据（如果需要）的目的存储空间。如果为NULL, 则函数使用包含该输入数据的存储空间。
copy_data
　　标示是否要拷贝元素， 如果 copy_data != 0, 则需要拷贝；如果 copy_data == 0, 则不需拷贝。

函数 cvSeqSlice 创建一序列，该序列表示输入序列中特定的一部分 （slice),。新序列要么与原序列共享元素要么拥有自己的一份拷贝。因此，如果 有人需要去 处理 该部分序列，但函数却没有 slice 参数， 则使用该函数去获取该序列。.

[编辑]

CloneSeq

创建序列的一份拷贝

```
CvSeq* cvCloneSeq( const CvSeq* seq, CvMemStorage* storage=NULL );
```

seq
　　序列
storage
　　存放新序列的 header部分和拷贝数据（如果需要）的目的存储块。如果为 NULL, 则函数使用包含输入序列的存储块 。

函数 cvCloneSeq 创建输入序列的一份完全拷贝。调用函数 cvCloneSeq (seq, storage) 等同于调用 cvSeqSlice(seq, CV_WHOLE_SEQ, storage, 1).

SeqRemoveSlice

删除序列的 slice部分

```
void cvSeqRemoveSlice( CvSeq* seq, CvSlice slice );
```

seq
　　序列
slice
　　序列中被移动的那部分

函数 cvSeqRemoveSlice 删除序列中的 slice 部分

SeqInsertSlice

在序列中插入一数组

```
void cvSeqInsertSlice( CvSeq* seq, int before_index, const CvArr* from_arr );
```

seq
　　序列
slice
　　序列中被移动的那部分
from_arr
　　从中获取元素的数组

函数 cvSeqInsertSlice 在指定位置插入 来自数组from_arr中 所有元素。数组 from_arr 可以是一个 矩阵也可以是另外一个 序列。

SeqInvert

将序列中的元素进行逆序操作

```
void cvSeqInvert( CvSeq* seq );
```

seq
　　序列

函数 cvSeqInvert 对序列进行逆序操作 -- 即：使第一个元素成为最后一个，最后一个元素为第一个。

[编辑]

SeqSort

使用特定的比较函数对序列中的元素进行排序

```
/* a < b ? -1 : a > b ? 1 : 0 */
typedef int (CV_CDECL* CvCmpFunc)(const void* a, const void* b, void* userdata);

void cvSeqSort( CvSeq* seq, CvCmpFunc func, void* userdata=NULL );
```

seq
: 待排序的序列

func
: 比较函数，按照元素间的大小关系返回负数，零，正数（见：上面的声明和下面的例子） --相关函数为 C 运行时库中的 qsort, 后者（qsort)不使用参数userdata.

userdata
: 传递给比较函数的用户参数；有些情况下，可避免全局变量的使用

函数 cvSeqSort 使用特定的标准对序列进行排序。下面是一个 使用该函数的实例

```
/* Sort 2d points in top-to-bottom left-to-right order */
static int cmp_func( const void* _a, const void* _b, void* userdata )
{
    CvPoint* a = (CvPoint*)_a;
    CvPoint* b = (CvPoint*)_b;
    int y_diff = a->y - b->y;
    int x_diff = a->x - b->x;
    return y_diff ? y_diff : x_diff;
}

...

CvMemStorage* storage = cvCreateMemStorage(0);
CvSeq* seq = cvCreateSeq( CV_32SC2, sizeof(CvSeq), sizeof(CvPoint), storage );
int i;

for( i = 0; i < 10; i++ )
{
    CvPoint pt;
    pt.x = rand() % 1000;
    pt.y = rand() % 1000;
    cvSeqPush( seq, &pt );
}

cvSeqSort( seq, cmp_func, 0 /* userdata is not used here */ );

/* print out the sorted sequence */
for( i = 0; i < seq->total; i++ )
{
    CvPoint* pt = (CvPoint*)cvSeqElem( seq, i );
    printf( "(%d,%d)\n", pt->x, pt->y );
}

cvReleaseMemStorage( &storage );
```

[编辑]

SeqSearch

查询序列中的元素

```
/* a < b ? -1 : a > b ? 1 : 0 */
typedef int (CV_CDECL* CvCmpFunc)(const void* a, const void* b, void* userdata);

char* cvSeqSearch( CvSeq* seq, const void* elem, CvCmpFunc func,
                   int is_sorted, int* elem_idx, void* userdata=NULL );
```

seq
　　序列
elem
　　待查询的元素
func
　　比较函数，按照元素间的大小关系返回负数，零，正数（见：cvSeqSort）
is_sorted
　　标示序列是否已经排序
elem_idx
　　输出参数；（已查找到）元素的索引值
user_data
　　传递到比较函数的用户参数；在某些情况下，有助于避免使用全局变量。

函数 cvSeqSearch 查找序列中的元素。如果序列已被排序，则使用二分查找（时间复杂度为 O(log(N))否则使用简单线性查找。若查找的元素不存在，函数返回 NULL 指针，而索引值设置为序列中的元素数（如果使用的是线性查找）或 满足表达式 seq(i) > elem 的最小的 i.

[编辑]

StartAppendToSeq

将数据写入序列中，并初始化该过程

```
void cvStartAppendToSeq( CvSeq* seq, CvSeqWriter* writer );
```

seq
　　指向序列的指针
writer
　　writer 的状态； 由该函数初始化

函数 cvStartAppendToSeq 初始化将数据写入序列这个过程。通过宏 CV_WRITE_SEQ_ELEM(written_elem, writer)，写入的元素被添加到序列尾部。注意：在写入期间，序列的其他操作可能会产生的错误的结果，甚至破怀该序列（见 cvFlushSeqWriter 的相关描述，有助于避免这些错误）

[编辑]

StartWriteSeq

创建新序列，并初始化写入部分（writer)

```
void cvStartWriteSeq( int seq_flags, int header_size, int elem_size,
                      CvMemStorage* storage, CvSeqWriter* writer );
```

seq_flags
　　标示被创建的序列。如果序列还未传递给任何处理特定序列类型的函数，则序列值等于0， 否则，必须从之前定义的序列类型中选择一个合适的类型。
header_size
　　头部的大小。 参数值不小于 sizeof(CvSeq). 如果定义了某一类型，则该类型不许符合基类的条件。
elem_size
　　元素的大小（以字节计）；必须与序列类型相一致。例如：如果创建了包含指针的序列（元素类型为 CV_SEQ_ELTYPE_POINT), 那么elem_size 必须等同于 sizeof(CvPoint).
storage
　　序列的（在内存）位置
writer
　　写入部分 writer 的状态； 由该函数初始化

函数 cvStartWriteSeq 是 函数 cvCreateSeq 和函数 cvStartAppendToSeq 的组合。 指向被创建的序列的指针存放在 writer->seq 中, 通过函数cvEndWriteSeq 返回（因当在最后调用）

[编辑]

EndWriteSeq

完成写入操作

```
CvSeq* cvEndWriteSeq( CvSeqWriter* writer );
```

writer
　　写入部分 writer 的状态

函数 cvEndWriteSeq 完成写入操作并返回指向被写入元素的序列的地址。同时，函数会截取最后那个不完整的序列块，将块的剩余部分返回到内存中之后，序列就可以被安全的读和写。

[编辑]

FlushSeqWriter

根据写入状态，刷新序列头部

```
void cvFlushSeqWriter( CvSeqWriter* writer );
```

writer
　　写入部分的状态

函数 cvFlushSeqWriter 用来使用户在写入过程中每当需要时读取序列元素，比如说，核查制定的条件。函数更新序列的头部，从而使读取序列中的数据成为可能。不过，写入并没有被关 闭，为的是随时都可以将数据写入序列。在有些算法中，经常需要刷新，考虑使用 cvSeqPush 代替该函数。

[编辑]

StartReadSeq

初始化序列中的读取过程

```
void cvStartReadSeq( const CvSeq* seq, CvSeqReader* reader, int reverse=0 );
```

seq
　　序列
reader
　　读取部分的状态； 由该函数初始化
reverse
　　决定遍历序列的方向。如果 reverse 为 0 ，则读取顺序被定位从序列头部元素开始，否则从尾部开始读取

函数 cvStartReadSeq 初始化读取部分的状态。毕竟，顺序读取可通过调用宏 CV_READ_SEQ_ELEM(read_elem, reader)，逆序读取可通过调用宏CV_REV_READ_SEQ_ELEM(read_elem, reader)。这两个宏都将序列元素读进read_elem中， 并将指针移到下一个元素。下面代码显示了如何去使用reader 和 writer.

```
CvMemStorage* storage = cvCreateMemStorage(0);
CvSeq* seq = cvCreateSeq( CV_32SC1, sizeof(CvSeq), sizeof(int), storage );
CvSeqWriter writer;
CvSeqReader reader;
int i;

cvStartAppendToSeq( seq, &writer );
for( i = 0; i < 10; i++ )
{
    int val = rand()%100;
    CV_WRITE_SEQ_ELEM( val, writer );
    printf("%d is written\n", val );
}
```

```
cvEndWriteSeq( &writer );

cvStartReadSeq( seq, &reader, 0 );
for( i = 0; i < seq->total; i++ )
{
    int val;
#if 1
    CV_READ_SEQ_ELEM( val, reader );
    printf("%d is read\n", val );
#else /* alternative way, that is prefferable if sequence elements are large,
         or their size/type is unknown at compile time */
    printf("%d is read\n", *(int*)reader.ptr );
    CV_NEXT_SEQ_ELEM( seq->elem_size, reader );
#endif
}
...

cvReleaseStorage( &storage );
```

[编辑]

GetSeqReaderPos

返回当前的读取器的位置

```
int cvGetSeqReaderPos( CvSeqReader* reader );
```

reader
: 读取器的状态.

函数 cvGetSeqReaderPos 返回当前的 reader 位置 （在 0 到 reader->seq->total - 1 中）

[编辑]

SetSeqReaderPos

移动读取器到指定的位置。

```
void cvSetSeqReaderPos( CvSeqReader* reader, int index, int is_relative=0 );
```

reader
: reader 的状态

index
: 目的位置。如果使用了 positioning mode, 则实际位置为 index % reader->seq->total.

is_relative
: 如果不位 0, 那么索引（index) 就相对于当前的位置

函数 cvSetSeqReaderPos 将 read 的位置移动到绝对位置，或相对于当前的位置（相对位置）

[编辑]

集合

[编辑]

CvSet

Collection of nodes

```
typedef struct CvSetElem
{
    int flags; /* it is negative if the node is free and zero or positive otherwise */
    struct CvSetElem* next_free; /* if the node is free, the field is a
                                    pointer to next free node */
}
CvSetElem;
```

```
#define CV_SET_FIELDS()    \
    CV_SEQUENCE_FIELDS()   /* inherits from CvSeq */ \
    struct CvSetElem* free_elems; /* list of free nodes */

typedef struct CvSet
{
    CV_SET_FIELDS()
} CvSet;
```

在 OpenCV 的稀疏数据结构中， CvSet 是一基本结构。

从上面的声明中可知：CvSet 继承自 CvSeq， 并在此基础上增加了个 free_elems 域，该域是空节点组成的列表。集合中的每一个节点，无论空否，都是线性表中的一个元素。尽管对于稠密的表中的元素没有限制，集合（派生的结构）元素必须起 始于整数域，并与结构 CvSetElem 相吻合，因为这两个域对于（由空节点组成）集合的组织是必要的。如果节点为空，flags 为负，next_free 指向下一个空节点。如果节点已被占据空间，flags 为正， flags 包含节点索引值（使用表达式 set_elem->flags & CV_SET_ELEM_IDX_MASKH 获取), flags 的剩余内容由用户决定。宏 CV_IS_SET_ELEM(set_elem.ptr)用来识别特定的节点是否为空。

起初,集合 set 同表 list 都为空。当需要一个来自集合中的新节点时，就从表 list 中去获取，然后表进行了更新。如果表 list 碰巧为空，于是就分配一内存块，块中的所有节点与表 list 相连。结果，集合的 total 域被设置为空节点和非空节点的和。当非空节点别释放后，就将它加到空节点列表中。最先被释放的节点也就是最先被占用空间的节点

在 OpenCV 中， CvSet 用来代表图形（CvGraph), 稀疏多维数组（CvSparseMat), 平面子划分（planner subdivisions)等

[编辑]

CreateSet

创建空的数据集

```
CvSet* cvCreateSet( int set_flags, int header_size,
                    int elem_size, CvMemStorage* storage );
```

set_flags
: 集合的类型

header_size
: 头节点的大小；应该等于 sizeof(CvSet)

elem_size
: 元素的大小； 不能小8

storage
: 相关容器

函数 CvCreateSet 创建一具有特定头部节点大小和元素类型的空集。并返回指向该集合的指针。

[编辑]

SetAdd

占用集合中的一个节点

```
int cvSetAdd( CvSet* set_header, CvSetElem* elem=NULL, CvSetElem** inserted_elem=NULL );
```

set_header
: 集合

elem
: 可选的输入参数，被插入的元素。如果不为 NULL, 函数就将数据拷贝到新分配的节点。（拷贝后，清空第一个域的 MSB)

函数 cvSetAdd 分配一新的节点，将输入数据拷贝给它（可选），并且返回指向该节点的指针和节点的索引值。索引值可通过节点的flags域的低位中获得。函数的时间复杂度为 O(1), 不过，存在着一个函数可快速的分配内存。（见 cvSetNew)

[编辑]

SetRemove

从点集中删除元素

```
void cvSetRemove( CvSet* set_header, int index );
```

set_header
　　集合
index
　　被删元素的索引值

函数 cvSetRemove 从点集中删除一具有特定索引值的元素。如果指定位置的节点为空，函数将什么都不做。函数的时间复杂度为 O(1), 不过,存在一函数可更快速的完成该操作，该函数就是 cvSetRemoveByPtr

[编辑]

SetNew

添加元素到点集中

```
CvSetElem* cvSetNew( CvSet* set_header );
```

set_header
　　集合

函数 cvSetNew 是 cvSetAdd 的变体，内联函数。它占用一新节点，并返回指向该节点的指针而不是索引。

[编辑]

SetRemoveByPtr

删除指针指向的集合元素

```
void cvSetRemoveByPtr( CvSet* set_header, void* elem );
```

set_header
　　集合
elem
　　被删除的元素

函数 cvSetRemoveByPtr 是一内联函数，是函数 cvSetRemove 轻微变化而来的。该函数并不会检查节点是否为空 -- 用户负责这一检查。

[编辑]

GetSetElem

通过索引值查找相应的集合元素

```
CvSetElem* cvGetSetElem( const CvSet* set_header, int index );
```

set_header
　　集合
index

索引值

函数 cvGetSetElem 通过索引值查找相应的元素。函数返回指向该元素的指针，如果索引值无效或相应的节点为空，则返回 0。 若函数使用 cvGetSeqElem 去查找节点，则函数支持负的索引值。

[编辑]

ClearSet

清空点集

```
void cvClearSet( CvSet* set_header );
```

set_header
待清空的点集

函数 cvClearSet 删除集合中的所有元素。时间复杂度为 O(1).

[编辑]

图

[编辑]

CvGraph

有向权图和无向权图

```
#define CV_GRAPH_VERTEX_FIELDS()    \
    int flags; /* vertex flags */   \
    struct CvGraphEdge* first; /* the first incident edge */

typedef struct CvGraphVtx
{
    CV_GRAPH_VERTEX_FIELDS()
}
CvGraphVtx;

#define CV_GRAPH_EDGE_FIELDS()      \
    int flags; /* edge flags */     \
    float weight; /* edge weight */ \
    struct CvGraphEdge* next[2]; /* the next edges in the incidence lists for staring (0) */ \
                                 /* and ending (1) vertices */ \
    struct CvGraphVtx* vtx[2]; /* the starting (0) and ending (1) vertices */

typedef struct CvGraphEdge
{
    CV_GRAPH_EDGE_FIELDS()
}
CvGraphEdge;

#define  CV_GRAPH_FIELDS()                  \
    CV_SET_FIELDS() /* set of vertices */   \
    CvSet* edges;   /* set of edges */

typedef struct CvGraph
{
    CV_GRAPH_FIELDS()
}
CvGraph;
```

在 OpenCV 图形结构中，CvGraph 是一基本结构。

图形结构继承自 CvSet -- 该部分描绘了普通图的属性和图的顶点，也包含了一个点集作为其成员 -- 该点集描述了图的边缘。利用宏（可以简化结构扩展和定制）使用与其它OpenCV可扩展结构一样的方法和技巧，同样的方法和技巧，我们声明了定点，边和头 部结构。虽然顶点结构和边结构无法从CvSetElem 显式地继承时，但

它们满足点集元素的两个条件（在开始是有一个整数域和满足 CvSetElem 结构）。 flags 域用来标记顶点和边是否已被占用或者处于其他目的，如：遍历图时（见：cvStartScanGraph 等），因此最好不要去直接使用它们。图代表的就是边的集合。存在有向和无向的区别。对于后者（无向图），在连接顶点 A 到 顶点 B 的边同连接顶点 B 到 顶点 A的边是没什么区别的，在某一时刻，只可能存在一个，即：要么是<A, B>要么是<B, A>.

[编辑]

CreateGraph

创建一个空树

```
CvGraph* cvCreateGraph( int graph_flags, int header_size, int vtx_size,
                        int edge_size, CvMemStorage* storage );
```

graph_flags
: 被创建的图的类型。通常，无向图为 CV_SEQ_KIND_GRAPH,有向图为 CV_SEQ_KIND_GRAPH | CV_GRAPH_FLAG_ORIENTED.

header_size
: 头部大小；可能小于 sizeof(CvGraph)

vtx_size
: 顶点大小；常规的定点结构必须来自 CvGraphVtx (使用宏 CV_GRAPH_VERTEX_FIELDS())

edge_size
: 边的大小；常规的边结构必须来自 CvGraphEdge (使用宏 CV_GRAPH_EDGE_FIELDS())

storage
: 图的容器

函数 cvCreateGraph 创建一空图并且返回指向该图的指针。

[编辑]

GraphAddVtx

插入一顶点到图中

```
int cvGraphAddVtx( CvGraph* graph, const CvGraphVtx* vtx=NULL,
                   CvGraphVtx** inserted_vtx=NULL );
```

graph
: 图

vtx
: 可选输入参数，用来初始化新加入的顶点（仅大小超过 sizeof(CvGraphVtx) 的用户自定义的域才会被拷贝）

inserted_vertex
: 可选的输出参数。如果不为 NULL, 则传回新加入顶点的地址

函数 cvGraphAddVtx 将一顶点加入到图中，并返回定点的索引

[编辑]

GraphRemoveVtx

通过索引从图中删除一顶点

```
int cvGraphRemoveVtx( CvGraph* graph, int index );
```

graph
: 图

vtx_idx

被珊顶点的索引

函数 cvGraphRemoveAddVtx 从图中删除一顶点，连同删除含有此顶点的边。如果输入的顶点不属于该图的话，将报告删除出错（不存在而无法删除）。返回值为被删除的边数，如果顶点不属于该图的话，返回 -1。

[编辑]

GraphRemoveVtxByPtr

通过指针从图中删除一顶点

```
int cvGraphRemoveVtxByPtr( CvGraph* graph, CvGraphVtx* vtx );
```

graph
图
vtx
指向被删除的边的指针

函数 cvGraphRemoveVtxByPtr 从图中删除一顶点，连同删除含有此顶点的边。如果输入的顶点不属于该图的话，将报告删除出错（不存在而无法删除）。返回值为被删除的边数，如果顶点不属于该图的话，返回 -1。

[编辑]

GetGraphVtx

通过索引值查找图的相应顶点

```
CvGraphVtx* cvGetGraphVtx( CvGraph* graph, int vtx_idx );
```

graph
图
vtx_idx
定点的索引值

函数 cvGetGraphVtx 通过索引值查找对应的顶点，并返回指向该顶点的指针，如果不存在则返回 NULL.

[编辑]

GraphVtxIdx

返回定点相应的索引值

```
int cvGraphVtxIdx( CvGraph* graph, CvGraphVtx* vtx );
```

graph
图
vtx
指向顶点的指针

函数 cvGraphVtxIdx 返回与顶点相应的索引值

[编辑]

GraphAddEdge

通过索引值在图中加入一条边

```
int cvGraphAddEdge( CvGraph* graph, int start_idx, int end_idx,
                    const CvGraphEdge* edge=NULL, CvGraphEdge** inserted_edge=NULL );
```

graph
图
start_idx
边的起始顶点的索引值
end_idx
边的尾部顶点的索引值（对于无向图，参数的次序无关紧要，即：start_idx 和 end_idx 可互为起始顶点和尾部顶点）
edge
可选的输入参数，初始化边的数据
inserted_edge
可选的输出参数，包含被插入的边的地址。

函数 cvGraphAddEdge 连接两特定的顶点。如果该边成功地加入到图中，返回 1； 如果连接两顶点的边已经存在，返回 0； 如果顶点没被发现（不存在）或者起始顶点和尾部顶点是同一个定点，或其他特殊情况，返回 -1。 如果是后者（即：返回值为负），函数默认的报告一个错误。

[编辑]

GraphAddEdgeByPtr

通过指针在图中加入一条边

```
int cvGraphAddEdgeByPtr( CvGraph* graph, CvGraphVtx* start_vtx, CvGraphVtx* end_vtx,
                         const CvGraphEdge* edge=NULL, CvGraphEdge** inserted_edge=NULL );
```

graph
图
start_vtx
指向起始顶点的指针
end_vtx
指向尾部顶点的指针。对于无向图来说，顶点参数的次序无关紧要。
edge
可选的输入参数，初始化边的数据
inserted_edge
可选的输出参数，包含被插入的边的地址。

函数 cvGraphAddEdge 连接两特定的顶点。如果该边成功地加入到图中，返回 1； 如果连接两顶点的边已经存在，返回 0； 如果顶点没被发现（不存在）或者起始顶点和尾部顶点是同一个定点，或其他特殊情况，返回 -1。 如果是后者（即：返回值为负），函数默认的报告一个错误

[编辑]

GraphRemoveEdge

通过索引值从图中删除顶点

```
void cvGraphRemoveEdge( CvGraph* graph, int start_idx, int end_idx );
```

graph
图
start_idx
起始顶点的索引值
end_idx
尾部顶点的索引值。对于无向图来说，顶点参数的次序无关紧要。

函数 cvGraphRemoveEdge 删除连接两特定顶点的边。若两顶点并没有相连接（即：不存在由这两个顶点连接的边），函数什么都不做。

[编辑]

GraphRemoveEdgeByPtr

通过指针从图中删除边

```
void cvGraphRemoveEdgeByPtr( CvGraph* graph, CvGraphVtx* start_vtx, CvGraphVtx* end_vtx );
```

graph
　　图
start_vtx
　　指向起始顶点的指针
end_vtx
　　指向尾部顶点的指针。对于无向图来说，顶点参数的次序无关紧要。

函数 cvGraphRemoveEdgeByPtr 删除连接两特定顶点的边。若两顶点并没有相连接（即：不存在由这两个顶点连接的边），函数什么都不做。

[编辑]

FindGraphEdge

通过索引值在图中查找相应的边

```
CvGraphEdge* cvFindGraphEdge( const CvGraph* graph, int start_idx, int end_idx );
#define cvGraphFindEdge cvFindGraphEdge
```

graph
　　图
start_idx
　　起始顶点的索引值
end_idx
　　尾部顶点的索引值。对于无向图来说，顶点参数的次序无关紧要

函数 cvFindGraphEdge 查找与两特定顶点相对应的边，并返回指向该边的指针。如果该边不存在，返回 NULL.

[编辑]

FindGraphEdgeByPtr

通过指针在图中查找相应的边

```
CvGraphEdge* cvFindGraphEdgeByPtr( const CvGraph* graph, const CvGraphVtx* start_vtx,
                                   const CvGraphVtx* end_vtx );
#define cvGraphFindEdgeByPtr cvFindGraphEdgeByPtr
```

graph
　　图
start_vtx
　　指向起始顶点的指针
end_vtx
　　指向尾部顶点的指针。对于无向图来说，顶点参数的次序无关紧要。

函数 cvFindGraphEdgeByPtr 查找与两特定顶点相对应的边，并返回指向该边的指针。如果该边不存在，返回 NULL

[编辑]

GraphEdgeIdx

返回与该边相应的索引值

```
int cvGraphEdgeIdx( CvGraph* graph, CvGraphEdge* edge );
```

graph
　　图
edge
　　指向该边的指针

函数 cvGraphEdgeIdx 返回与边对应的索引值。

[编辑]

GraphVtxDegree

（通过索引值）统计与顶点相关联的边数

```
int cvGraphVtxDegree( const CvGraph* graph, int vtx_idx );
```

graph
　　图
vtx_idx
　　顶点对应的索引值

函数 cvGraphVtxDegree 返回与特定顶点相关联的边数，包括以该顶点为起始顶点的和尾部顶点的。统计边数，可以适用下列代码：

```
CvGraphEdge* edge = vertex->first; int count = 0;
while( edge )
{
    edge = CV_NEXT_GRAPH_EDGE( edge, vertex );
    count++;
}
```

宏 CV_NEXT_GRAPH_EDGE(edge, vertex) 返回依附于该顶点的下一条边。

[编辑]

GraphVtxDegreeByPtr

（通过指针）统计与顶点相关联的边数

```
int cvGraphVtxDegreeByPtr( const CvGraph* graph, const CvGraphVtx* vtx );
```

graph
　　图
vtx
　　顶点对应的指针

函数 cvGraphVtxDegreeByPtr 返回与特定顶点相关联的边数，包括以该顶点为起始顶点的和尾部顶点的

[编辑]

ClearGraph

删除图

```
void cvClearGraph( CvGraph* graph );
```

graph
　　图

函数 cvClearGraph 删除该图的所有顶点和边。时间复杂度为 O(1).

[编辑]

CloneGraph

克隆图

```
CvGraph* cvCloneGraph( const CvGraph* graph, CvMemStorage* storage );
```

graph
　　待拷贝的图
storage
　　容器，存放拷贝

函数 cvCloneGraph 创建图的完全拷贝。如果顶点和边含有指向外部变量的指针，那么图和它的拷贝共享这些指针。在新的图中，顶点和边可能存在不同，因为函数重新分割了顶点和边的点集。

[编辑]

CvGraphScanner

图的遍历

```
typedef struct CvGraphScanner
{
    CvGraphVtx* vtx;        /* current graph vertex (or current edge origin) */
    CvGraphVtx* dst;        /* current graph edge destination vertex */
    CvGraphEdge* edge;      /* current edge */

    CvGraph* graph;         /* the graph */
    CvSeq*   stack;         /* the graph vertex stack */
    int      index;         /* the lower bound of certainly visited vertices */
    int      mask;          /* event mask */
}
CvGraphScanner;
```

结构 cvGraphScanner 深度遍历整个图。 函数的相关讨论如下（看：StartScanGraph)

[编辑]

StartScanGraph

创建一结构，用来对图进行深度遍历

```
CvGraphScanner*  cvCreateGraphScanner( CvGraph* graph, CvGraphVtx* vtx=NULL,
                                       int mask=CV_GRAPH_ALL_ITEMS );
```

graph
　　图
vtx
　　开始遍历的（起始）顶点。如果为 NULL, 便利就从第一个顶点开始（指：顶点序列中，具有最小索引值的顶点）
mask
　　事件掩码（event mask)代表用户感兴趣的事件（此时 函数 cvNextGraphItem 将控制返回给用户）。这个只可能是 CV_GRAPH_ALL_ITEMS (如果用户对所有的事件都感兴趣的话）或者是下列标志的组合:

　　CV_GRAPH_VERTEXT -- 在第一次被访问的顶点处停下
　　CV_GRAPH_TREE_EDGE -- 在 tree edge 处停下(tree edge 指连接最后被访问的顶点与接下来被访问的顶点的边)
　　CV_GRAPH_BACK_EDGE -- 在 back edge 处停下（back edge 指连接最后被访问的顶点与其在搜索树中祖先的边)

CV_GRAPH_FORWARD_EDGE -- 在 forward edge 处停下 （forward edge 指连接最后被访问的顶点与其在搜索树中后裔的边）
CV_GRAPH_CROSS_EDGE -- 在 cross edge 处停下（cross edge 指连接不同搜索树中或同一搜索树中不同分支的边.只有在有向图中， 才存在着一 概念）
CV_GRAPH_ANY_EDGE -- 在 any edge 处停下（any edge 指 任何边，包括 tree edge, back edge, forward edge, cross edge)
CV_GRAPH_NEW_TREE -- 在每一个新的搜索树开始处停下。首先遍历从起始顶点开始可以访问到的顶点和边，然后查找搜索图中访问不到的顶点或边并恢复遍历。在开始遍历一颗新的树时 （包括第一次调用 cvNextGraphItem 时的树），产生 CV_GRAPH_NEW_TREE 事件。

函数 cvCreateGraphScanner 创建一结构用来深度遍历搜索树。函数 cvNextGraphItem 要使用该初始化了的结构 -- 层层遍历的过程。

[编辑]

NextGraphItem

逐层遍历整个图

```
int cvNextGraphItem( CvGraphScanner* scanner );
```

scanner
图的遍历状态。被此函数更新。

函数 cvNextGraphItem 遍历整个图，直到用户感兴趣的事件发生（即：调用 cvCreateGraphScanner 时，mask 对应的事件）或遍历结束。在前面一种情况下，函数返回 参数mask 相应的事件，当再次调用函数时，恢复遍历）。在后一种情况下，返回 CV_GRAPH_OVER(-1)。当 mask 相应的事件为 CV_GRAPH_BACKTRACKING 或 CV_GRAPH_NEW_TEEE 时， 当前正在被访问的顶点被存放在 scanner->vtx 中。如果事件与 边edge 相关，那么 edge 本身被存放在 scanner->edge, 该边的起始顶点存放在 scanner->vtx 中， 尾部节点存放在 scanner->dst 中。

[编辑]

ReleaseGraphScanner

完成图地遍历过程

```
void cvReleaseGraphScanner( CvGraphScanner** scanner );
```

scanner
指向遍历器的指针.

函数 cvGraphScanner 完成图的遍历过程，并释放遍历器的状态。

[编辑]

树

[编辑]

CV_TREE_NODE_FIELDS

用于树结点类型声明的（助手）宏

```
#define CV_TREE_NODE_FIELDS(node_type)                          \
    int       flags;         /* micsellaneous flags */          \
    int       header_size;   /* size of sequence header */      \
    struct    node_type* h_prev; /* previous sequence */        \
    struct    node_type* h_next; /* next sequence */            \
    struct    node_type* v_prev; /* 2nd previous sequence */    \
```

```
    struct    node_type* v_next; /* 2nd next sequence */
```

宏 CV_TREE_NODE_FIELDS() 用来声明一层次性结构，例如 CvSeq -- 所有动态结构的基本类型。如果树的节点是由该宏所声明的，那么就可以使用（该部分的）以下函数对树进行相关操作。

[编辑]

CvTreeNodeIterator

打开现存的存储结构或者创建新的文件存储结构

```
typedef struct CvTreeNodeIterator
{
    const void* node;
    int level;
    int max_level;
}
CvTreeNodeIterator;
```

结构 CvTreeNodeIterator 用来对树进行遍历。该树的节点是由宏 CV_TREE_NODE_FIELDS 声明。

[编辑]

InitTreeNodeIterator

用来初始化树结点的迭代器

```
void cvInitTreeNodeIterator( CvTreeNodeIterator* tree_iterator,
                             const void* first, int max_level );
```

tree_iterator
: 初始化了的迭代器

first
: （开始）遍历的第一个节点

max_level
: 限制对树进行遍历的最高层（即：第 max_level 层）(假设第一个节点所在的层为第一层）。例如：1 指的是遍历第一个节点所在层，2 指的是遍历第一层和第二层

函数 cvInitTreeNodeIterator 用来初始化树的迭代器。

[编辑]

NextTreeNode

返回当前节点，并将迭代器 iterator 移向当前节点的下一个节点

```
void* cvNextTreeNode( CvTreeNodeIterator* tree_iterator );
```

tree_iterator
: 初始化了的迭代器

函数 cvNextTreeNode 返回当前节点并且更新迭代器（iterator) -- 并将 iterator 移向（当前节点）下一个节点。换句话说，函数的行为类似于表达式 *p++ (通常的 C 指针 或 C++ 集合迭代器）。 如果没有更多的节点（即：当前节点为最后的节点），则函数返回值为 NULL.

[编辑]

PrevTreeNode

返回当前节点，并将迭代器 iterator 移向当前节点的前一个节点

```
void* cvPrevTreeNode( CvTreeNodeIterator* tree_iterator );
```

tree_iterator
　　初始化了的迭代器

函数 cvPrevTreeNode 返回当前节点并且更新迭代器（iterator) -- 并将 iterator 移向（当前节点的）前一个节点。换句话说，函数的行为类似于表达式 *p-- (通常的 C 指针 或 C++ 集合迭代器）。 如果没有更多的节点（即：当前节点为头节点），则函数返回值为 NULL.

[编辑]

TreeToNodeSeq

将所有的节点指针（即：指向树结点的指针）收集到线性表 sequence 中

```
CvSeq* cvTreeToNodeSeq( const void* first, int header_size, CvMemStorage* storage );
```

first
　　初始树结点
header_size
　　线性表的表头大小，大小通常为 sizeof(CvSeq)

函数 cvTreeToNodeSeq 将树的节点指针挨个的存放到线性表中。存放的顺序以深度为先。

[编辑]

InsertNodeIntoTree

将新的节点插入到树中

```
void cvInsertNodeIntoTree( void* node, void* parent, void* frame );
```

node
　　待插入的节点
parent
　　树中的父节点（即：含有子节点的节点）
frame
　　顶部节点。如果 节点parent 等同于 节点frame， 则将节点的域 v_prev 设为 NULL 而不是 parent.

函数 cvInsertNodeIntoTree 将另一个节点插入到树中。函数不分配任何内存，仅仅修改树节点的连接关系。

[编辑]

RemoveNodeFromTree

从树中删除节点

```
void cvRemoveNodeFromTree( void* node, void* frame );
```

node
　　待删除的节点。
frame
　　顶部节点。如果 node->v.prev = NULL 且 node->h.prev = NULL, 则将 frame->v.next 设为 node->h.next

函数 cvRemoveNodeFromTree 从树中删除节点。它不会释放任何内存，仅仅修改树中节点的连接关系

Cxcore绘图函数

Wikipedia，自由的百科全书

绘图函数作用于任何象素深度的矩阵/图像. Antialiasing技术只能在8位图像上实现.所有的函数包括彩色图像的色彩参数（色彩参数是指rgb它是由宏CV_RGB或cvScalar函数构成。）和灰度图像的亮度。

如果一幅绘制图形部分或全部位于图像之外，那么对它先做裁剪。 对于彩色图像正常的色彩通道是B(蓝),G(绿),R(红)..。如果需要其它的色彩，可以通过cvScalar中的特殊色彩通道构造色彩，或者在绘制图像之前或之后 使用 cvCvtColor或者cvTransform来转换。

目录

[隐藏]

[编辑]

曲线与形状

[编辑]

CV_RGB

创建一个色彩值.

```
#define CV_RGB( r, g, b )  cvScalar( (b), (g), (r) )
```

[编辑]

Line

绘制连接两个点的线段

```
void cvLine( CvArr* img, CvPoint pt1, CvPoint pt2, CvScalar color,
```

```
            int thickness=1, int line_type=8, int shift=0 );
```

img
　　图像。
pt1
　　线段的第一个端点。
pt2
　　线段的第二个端点。
color
　　线段的颜色。
thickness
　　线段的粗细程度。
line_type
　　线段的类型。

　　8 (or 0) - 8-connected line（8邻接)连接 线。
　　4 - 4-connected line(4邻接)连接线。
　　CV_AA - antialiased 线条。

shift
　　坐标点的小数点位数。

函数cvLine 在图像中的点1和点2之间画一条线段。线段被图像或感兴趣的矩形(ROI rectangle)所裁剪。对于具有整数坐标的non-antialiasing 线条，使用8-连接或者4-连接Bresenham 算法。画粗线条时结尾是圆形的。画antialiased 线条使用高斯滤波。要指定线段颜色，用户可以使用使用宏CV_RGB(r, g, b)。

[编辑]

Rectangle

绘制简单、指定粗细或者带填充的 矩形

```
void cvRectangle( CvArr* img, CvPoint pt1, CvPoint pt2, CvScalar color,
                  int thickness=1, int line_type=8, int shift=0 );
```

img
　　图像.
pt1
　　矩形的一个顶点。
pt2
　　矩形对角线上的另一个顶点
color
　　线条颜色 (RGB) 或亮度（灰度图像 ）(grayscale image）。
thickness
　　组成矩形的线条的粗细程度。取负值时（如 CV_FILLED）函数绘制填充了色彩的矩形。
line_type
　　线条的类型。见cvLine的描述
shift
　　坐标点的小数点位数。

函数 cvRectangle 通过对角线上的两个顶点绘制矩形。

[编辑]

Circle

绘制圆形。

```
void cvCircle( CvArr* img, CvPoint center, int radius, CvScalar color,
               int thickness=1, int line_type=8, int shift=0 );
```

img
　　图像。
center
　　圆心坐标。
radius
　　圆形的半径。
color
　　线条的颜色。
thickness
　　如果是正数，表示组成圆的线条的粗细程度。否则，表示圆是否被填充。
line_type
　　线条的类型。见 cvLine 的描述
shift
　　圆心坐标点和半径值的小数点位数。

函数cvCircle绘制或填充一个给定圆心和半径的圆。圆被感兴趣矩形所裁剪。 若指定圆的颜色，可以使用宏 CV_RGB (r, g, b)。

[编辑]

Ellipse

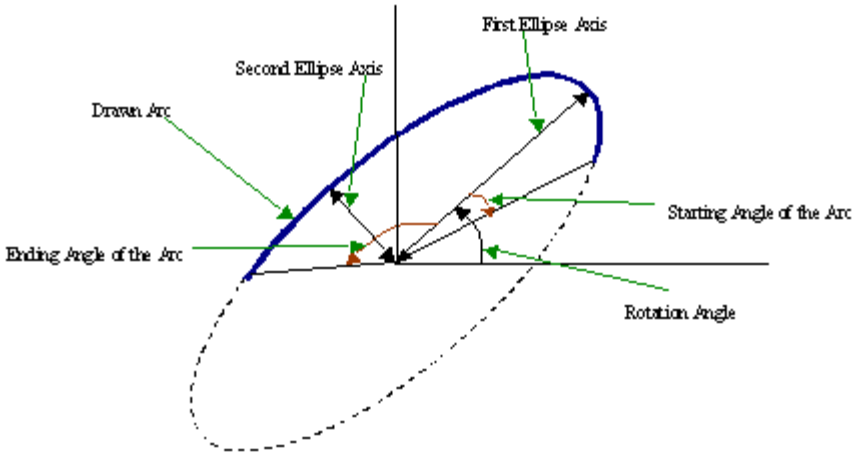
绘制椭圆圆弧和椭圆扇形。

```
void cvEllipse( CvArr* img, CvPoint center, CvSize axes, double angle,
                double start_angle, double end_angle, CvScalar color,
                int thickness=1, int line_type=8, int shift=0 );
```

img
　　图像。
center
　　椭圆圆心坐标。
axes
　　轴的长度。
angle
　　偏转的角度。
start_angle
　　圆弧起始角的角度。.
end_angle
　　圆弧终结角的角度。
color
　　线条的颜色。
thickness
　　线条的粗细程度。
line_type
　　线条的类型,见CVLINE的描述。
shift
　　圆心坐标点和数轴的精度。

函数cvEllipse用来绘制或者填充一个简单的椭圆弧或椭圆扇形。圆弧被ROI矩形所忽略。反走样弧线和粗弧线使用线性分段近似值。所有的角都是以角度的形式给定的。下面的图片将解释这些参数的含义。

Parameters of Elliptic Arc



[编辑]

EllipseBox

使用一种简单的方式来绘制椭圆圆弧和椭圆扇形。

```
void cvEllipseBox( CvArr* img, CvBox2D box, CvScalar color,
                   int thickness=1, int line_type=8, int shift=0 );
```

img
　　图像。
box
　　绘制椭圆圆弧所需要的外界矩形.
thickness
　　分界线线条的粗细程度。
line_type
　　分界线线条的类型,见CVLINE的描述。
shift
　　椭圆框顶点坐标的精度。

The function cvEllipseBox draws a simple or thick ellipse outline, or fills an ellipse. The functions provides a convenient way to draw an ellipse approximating some shape; that is what cvCamShift and cvFitEllipse do. The ellipse drawn is clipped by ROI rectangle. A piecewise-linear approximation is used for antialiased arcs and thick arcs.

[编辑]

FillPoly

填充多边形内部

```
void cvFillPoly( CvArr* img, CvPoint** pts, int* npts, int contours,
                 CvScalar color, int line_type=8, int shift=0 );
```

img
　　图像。
pts
　　指向多边形的数组指针。
npts
　　多边形的顶点个数的数组。
contours
　　组成填充区域的线段的数量。

color
多边形的颜色。
line_type
组成多边形的线条的类型。
shift
顶点坐标的小数点位数。

函数cvFillPoly用于一个单独被多边形轮廓所限定的区域内进行填充。函数可以填充复杂的区域,例如，有漏洞的区域和有交叉点的区域等等。

[编辑]

FillConvexPoly

填充凸多边形

```
void cvFillConvexPoly( CvArr* img, CvPoint* pts, int npts,
                       CvScalar color, int line_type=8, int shift=0 );
```

img
图像。
pts
指向单个多边形的指针数组。
npts
多边形的顶点个数。
color
多边形的颜色。
line_type
组成多边形的线条的类型。参见cvLine
shift
顶点坐标的小数点位数。

函数cvFillConvexPoly填充凸多边形内部。这个函数比函数cvFillPoly 更快。它除了可以填充凸多边形区域还可以填充任何的单调多边形。例如：一个被水平线（扫描线）至多两次截断的多边形。

[编辑]

PolyLine

绘制简单线段或折线。

```
void cvPolyLine( CvArr* img, CvPoint** pts, int* npts, int contours, int is_closed,
                 CvScalar color, int thickness=1, int line_type=8, int shift=0 );
```

img
图像。
pts
折线的顶点指针数组。
npts
折线的定点个数数组。也可以认为是pts指针数组的大小
contours
折线的线段数量。
is_closed
指出多边形是否封闭。如果封闭，函数将起始点和结束点连线。
color
折线的颜色。
thickness
线条的粗细程度。

line_type
线段的类型。参见cvLine。
shift
顶点的小数点位数。

函数cvPolyLine 绘制一个简单直线或折线。

[编辑]

文本

[编辑]

InitFont

初始化字体结构体。

```
void cvInitFont( CvFont* font, int font_face, double hscale,
                 double vscale, double shear=0,
                 int thickness=1, int line_type=8 );
```

font
被初始化的字体结构体。
font_face
字体名称标识符。只是Hershey 字体集(http://sources.isc.org/utils/misc/hershey-font.txt)的一个子集得到支持。

CV_FONT_HERSHEY_SIMPLEX - 正常大小无衬线字体。
CV_FONT_HERSHEY_PLAIN - 小号无衬线字体。
CV_FONT_HERSHEY_DUPLEX - 正常大小无衬线字体。(比CV_FONT_HERSHEY_SIMPLEX更复杂)
CV_FONT_HERSHEY_COMPLEX - 正常大小有衬线字体。
CV_FONT_HERSHEY_TRIPLEX - 正常大小有衬线字体 (比CV_FONT_HERSHEY_COMPLEX更复杂)
CV_FONT_HERSHEY_COMPLEX_SMALL - CV_FONT_HERSHEY_COMPLEX 的小译本。
CV_FONT_HERSHEY_SCRIPT_SIMPLEX - 手写风格字体。
CV_FONT_HERSHEY_SCRIPT_COMPLEX - 比CV_FONT_HERSHEY_SCRIPT_SIMPLEX更复杂。

这个参数能够由一个值和可选择的CV_FONT_ITALIC字体标记合成，就是斜体字。
hscale
字体宽度。如果等于1.0f，字符的宽度是最初的字体宽度。如果等于0.5f，字符的宽度是最初的字体宽度的一半。
vscale
字体高度。如果等于1.0f，字符的高度是最初的字体高度。如果等于0.5f，字符的高度是最初的字体高度的一半。
shear
字体的斜度。当值为0时 ，字符不倾斜；当值为1.0f时，字体倾斜≈45度，等等。厚度让字母着重显示。函数cvLine用于绘制字母。
thickness
字体笔划的粗细程度。
line_type
字体笔划的类型，参见cvLine。

函数cvInitFont初始化字体结构体，字体结构体可以被传递到文字显示函数中。

[编辑]

PutText

在图像中显示文本字符串。

```
void cvPutText( CvArr* img, const char* text, CvPoint org, const CvFont* font, CvScalar color );
```

img
　　输入图像。
text
　　要显示的字符串。
org
　　第一个字符左下角的坐标。
font
　　字体结构体。
color
　　文本的字体颜色。

函数cvPutText将具有指定字体的和指定颜色的文本加载到图像中。加载到图像中的文本被感兴趣的矩形框(ROI rectangle)剪切。不属于指定字体库的字符用矩形字符替代显示。

[编辑]

GetTextSize

获得字符串的宽度和高度。

```
void cvGetTextSize( const char* text_string, const CvFont* font, CvSize* text_size, int* baseline );
```

font
　　字体结构体
text_string
　　输入字符串。
text_size
　　合成字符串的字符的大小。文本的高度不包括基线以下的部分。
baseline
　　相对于文字最底部点的基线的Y坐标。

函数cvGetTextSize是用于在指定字体时计算字符串的绑定区域(binding rectangle)。

[编辑]

点集和轮廓

[编辑]

DrawContours

在图像中绘制外部和内部的轮廓。

```
void cvDrawContours( CvArr *img, CvSeq* contour,
                     CvScalar external_color, CvScalar hole_color,
                     int max_level, int thickness=1,
                     int line_type=8, CvPoint offset=cvPoint(0,0) );
```

img
　　用以绘制轮廓的图像。和其他绘图函数一样，边界图像被感兴趣区域（ROI）所剪切。
contour
　　指针指向第一个轮廓。
external_color
　　外层轮廓的颜色。
hole_color
　　内层轮廓的颜色。

max_level

绘制轮廓的最大等级。如果等级为0，绘制单独的轮廓。如果为1，绘制轮廓及在其后的相同的级别下轮廓。如果值为2，所有的轮廓。如果等级为2，绘制 所有同级轮廓及所有低一级轮廓，诸此种种。如果值为负数，函数不绘制同级轮廓，但会升序绘制直到级别为abs(max_level)-1的子轮廓。

thickness
: 绘制轮廓时所使用的线条的粗细度。如果值为负(e.g. =CV_FILLED),绘制内层轮廓。

line_type
: 线条的类型。参考cvLine.

offset
: 按照给出的偏移量移动每一个轮廓点坐标.当轮廓是从某些感兴趣区域(ROI)中提取的然后需要在运算中考虑ROI偏移量时，将会用到这个参数。

当thickness>=0,函数cvDrawContours在图像中绘制轮廓,或者当thickness<0时，填充轮廓所限制的区域。

```
#include "cv.h"
#include "highgui.h"

int main( int argc, char** argv )
{
    IplImage* src;
    // 第一条命令行参数确定了图像的文件名。
    if( argc == 2 && (src=cvLoadImage(argv[1], 0))!= 0)
    {
        IplImage* dst = cvCreateImage( cvGetSize(src), 8, 3 );
        CvMemStorage* storage = cvCreateMemStorage(0);
        CvSeq* contour = 0;

        cvThreshold( src, src, 1, 255, CV_THRESH_BINARY );
        cvNamedWindow( "Source", 1 );
        cvShowImage( "Source", src );

        cvFindContours( src, storage, &contour, sizeof(CvContour), CV_RETR_CCOMP,
CV_CHAIN_APPROX_SIMPLE );
        cvZero( dst );

        for( ; contour != 0; contour = contour->h_next )
        {
            CvScalar color = CV_RGB( rand()&255, rand()&255, rand()&255 );
            /* 用1替代 CV_FILLED  所指示的轮廓外形 */
            cvDrawContours( dst, contour, color, color, -1, CV_FILLED, 8 );
        }

        cvNamedWindow( "Components", 1 );
        cvShowImage( "Components", dst );
        cvWaitKey(0);
    }
}
```

在样本中用1替代 CV_FILLED 以指示的得到外形。

（注意：在cvFindContours中参数为CV_CHAIN_CODE时，cvDrawContours用CV_FILLED时不会画出任何图形）

[编辑]

InitLineIterator

初始化直线迭代器

```
int cvInitLineIterator( const CvArr* image, CvPoint pt1, CvPoint pt2,
                        CvLineIterator* line_iterator, int connectivity=8,
                        int left_to_right=0 );
```

img
: 用以获取直线的图像。

pt1

线段的第一个端点。

pt2
线段的第二个端点。

line_iterator
指向直线迭代状态结构体的指针。

connectivity
直线的邻接方式，4邻接或者8邻接。

left_to_right
标志值，指出扫描直线是从pt1和pt2外面最左边的点扫描到最右边的点(left_to_right≠0)，还是按照指定的顺序，从pt1到pt2(left_to_right=0)。

函数cvInitLineIterator初始化直线迭代器并返回两个端点间点的数目。两个端点都必须在图像内部。在迭代器初始化以后，所有的在连 接两个终点的栅栏线上的点，可以通过访问CV_NEXT_LINE_POINT点的方式获得。在线上的这些点使用4-邻接或者8-邻接的 Bresenham算法计算得到。

例：使用直线迭代来计算沿着彩色线上的点的像素值。

```
CvScalar sum_line_pixels( IplImage* image, CvPoint pt1, CvPoint pt2 )
    {
        CvLineIterator iterator;
        int blue_sum = 0, green_sum = 0, red_sum = 0;
        int count = cvInitLineIterator( image, pt1, pt2, &iterator, 8, 0 );

        for( int i = 0; i < count; i++ ){
            blue_sum += iterator.ptr[0];
            green_sum += iterator.ptr[1];
            red_sum += iterator.ptr[2];
            CV_NEXT_LINE_POINT(iterator);

            /* print the pixel coordinates: demonstrates how to calculate the coordinates */
            {
            int offset, x, y;
            /* assume that ROI is not set, otherwise need to take it into account. */
            offset = iterator.ptr - (uchar*)(image->imageData);
            y = offset/image->widthStep;
            x = (offset - y*image->widthStep)/(3*sizeof(uchar) /* size of pixel */);
            printf("(%d,%d)\n", x, y );
            }
        }
        return cvScalar( blue_sum, green_sum, red_sum );
    }
```

[编辑]

ClipLine

剪切图像矩形区域内部的直线。

```
int cvClipLine( CvSize img_size, CvPoint* pt1, CvPoint* pt2 );
```

img_size
图像的大小。

pt1
线段的第一个端点，会被函数修改。

pt2
线段的第二个端点，会被函数修改。

函数cvClipLine计算线段完全在图像中的一部分。如果线段完全在图像中，返回0，否则返回1。

[编辑]

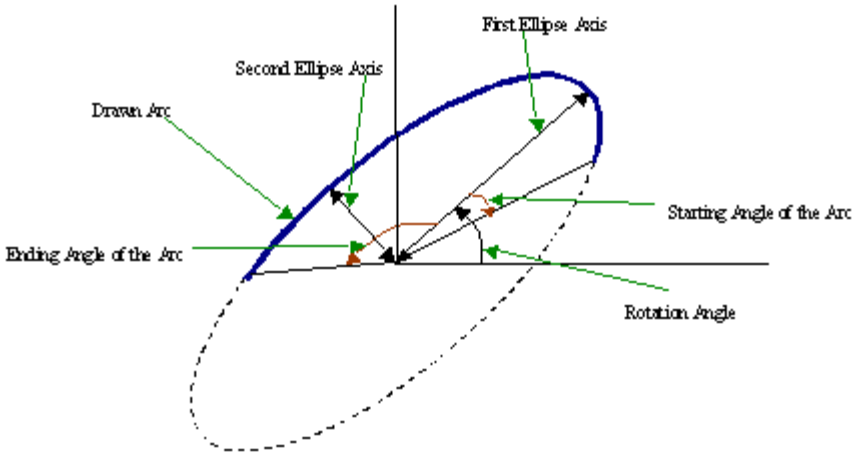
Ellipse2Poly

用折线逼近椭圆弧

```
int cvEllipse2Poly( CvPoint center, CvSize axes,
                    int angle, int arc_start,
                    int arc_end, CvPoint* pts, int delta );
```

center
　　弧线的中心。
axes
　　弧线的Half-sizes。参见下图。
angle
　　椭圆的旋转角度(Rotation angle),参见下图。
start_angle
　　椭圆的Starting angle，参见下图。
end_angle
　　椭圆的Ending angle，参见下图。
pts
　　坐标点矩阵数组，由本函数填充。
delta
　　与下一条折线定点的夹角，近似精度。故，得到的点数最大为ceil((end_angle - start_angle)/delta) + 1。

函数cvEllipse2Poly计算给定的椭圆弧的逼近折线的顶点，被cvEllipse使用。



Cxcore数据保存和运行时类型信息

Wikipedia，自由的百科全书

目录

[隐藏]

[编辑]

文件存储

[编辑]

CvFileStorage

文件存储结构

```
typedef struct CvFileStorage
{
    ...         // hidden fields
} CvFileStorage;
```

构造函数 CvFileStorage 是将磁盘上存储的文件关联起来的“黑匣子”。在下列函数描述中利用CvFileStorage 作为输入，允许存储或载入各种格式数据组成的层次集合，这些数据由标量值（scalar ）,或者CXCore 对象(例如矩阵,序列，图表) 和用户自定义对象。

CXCore 能将数据读入或写入 XML (http://www.w3c.org/XML) or YAML (http://www.yaml.org) 格式. 下面这个例子是利用CXCore函数将3×3单位浮点矩阵存入XML 和 YAML文档。

XML:

```
<?xml version="1.0">
<opencv_storage>
<A type_id="opencv-matrix">
  <rows>3</rows>
  <cols>3</cols>
  <dt>f</dt>
  <data>1. 0. 0. 0. 1. 0. 0. 0. 1.</data>
</A>
</opencv_storage>
```

YAML:

```
%YAML:1.0
A: !!opencv-matrix
  rows: 3
  cols: 3
  dt: f
  data: [ 1., 0., 0., 0., 1., 0., 0., 0., 1.]
```

从例子中可以看到, XML是用嵌套标签来表现层次，而 YAML用缩排来表现（类似于Python语言） 。

相同的 CXCore 函数也能够在这两种格式下读写数据，特殊的格式决定了文件的扩展名， .xml 是 XML 的扩展名， .yml 或 .yaml 是 YAML的扩展名。

[编辑]

CvFileNode

文件存储器节点

```
/* 文件节点类型 */
#define CV_NODE_NONE        0
#define CV_NODE_INT         1
#define CV_NODE_INTEGER     CV_NODE_INT
#define CV_NODE_REAL        2
#define CV_NODE_FLOAT       CV_NODE_REAL
#define CV_NODE_STR         3
#define CV_NODE_STRING      CV_NODE_STR
#define CV_NODE_REF         4 /* not used */
#define CV_NODE_SEQ         5
#define CV_NODE_MAP         6
#define CV_NODE_TYPE_MASK   7
```

```
/* 可选标记 */
#define CV_NODE_USER        16
#define CV_NODE_EMPTY       32
#define CV_NODE_NAMED       64

#define CV_NODE_TYPE(tag)  ((tag) & CV_NODE_TYPE_MASK)

#define CV_NODE_IS_INT(tag)         (CV_NODE_TYPE(tag) == CV_NODE_INT)
#define CV_NODE_IS_REAL(tag)        (CV_NODE_TYPE(tag) == CV_NODE_REAL)
#define CV_NODE_IS_STRING(tag)      (CV_NODE_TYPE(tag) == CV_NODE_STRING)
#define CV_NODE_IS_SEQ(tag)         (CV_NODE_TYPE(tag) == CV_NODE_SEQ)
#define CV_NODE_IS_MAP(tag)         (CV_NODE_TYPE(tag) == CV_NODE_MAP)
#define CV_NODE_IS_COLLECTION(tag) (CV_NODE_TYPE(tag) >= CV_NODE_SEQ)
#define CV_NODE_IS_FLOW(tag)        (((tag) & CV_NODE_FLOW) != 0)
#define CV_NODE_IS_EMPTY(tag)       (((tag) & CV_NODE_EMPTY) != 0)
#define CV_NODE_IS_USER(tag)        (((tag) & CV_NODE_USER) != 0)
#define CV_NODE_HAS_NAME(tag)       (((tag) & CV_NODE_NAMED) != 0)

#define CV_NODE_SEQ_SIMPLE 256
#define CV_NODE_SEQ_IS_SIMPLE(seq) (((seq)->flags & CV_NODE_SEQ_SIMPLE) != 0)

typedef struct CvString
{
    int len;
    char* ptr;
}
CvString;

/*所有已读存储在文件元素的关键字被存储在hash表中，这样可以加速查找操作 */
typedef struct CvStringHashNode
{
    unsigned hashval;
    CvString str;
    struct CvStringHashNode* next;
}
CvStringHashNode;

/* 文件存储器的基本元素是-标量或集合*/
typedef struct CvFileNode
{
    int tag;
    struct CvTypeInfo* info; /* 类型信息（只能用于用户自定义对象，对于其它对象它为0） */
    union
    {
        double f; /* 浮点数*/
        int i;    /* 整形数 */
        CvString str; /* 字符文本 */
        CvSeq* seq; /* 序列 (文件节点的有序集合) */
        struct CvMap* map; /*图表 (指定的文件节点的集合 ) */
    } data;
}
CvFileNode;
```

这个构造函数只是用于重新找到文件存储器上的数据(例如 ，从文件中下载数据）。 当数据已经写入文件时,按顺序写入，只用最小的缓冲完成，此时没有数据存放在文件存储器。

相反，当从文件中读数据时，所有文件在内存中像树一样被解析和描绘。树的每一个节点被CvFileNode表现出来。文件节点N的类型能够 通过CV_NODE_TYPE(N->tag) 被重新找到。一些节点（叶结点）作为变量：字符串文本，整数，浮点数。其它的文件节点是集合文件节点,有两个类型集合：序列和图表 (我们这里使用 YAML 符号,无论用哪种方法，对于XML符号流也是同样有效）。序列(不要与CvSeq混淆） 是由有序的非指定文件节点（注：没有关键字）构成的，图表是由无序的指定文件节点(注：有关键字)构成的。因而 ,序列的数据是通过索引（cvGetSepElem)来存取，图形的数据是通过名字（cvGetFileNodeByName)来存取 下表描述不同类型的节点：

Type	CV_NODE_TYPE(node->tag)	Value
Integer	CV_NODE_INT	node->data.i
Floating-point	CV_NODE_REAL	node->data.f
Text string	CV_NODE_STR	node->data.str.ptr
Sequence	CV_NODE_SEQ	node->data.seq

Map	CV_NODE_MAP	node->data.map*

这里不需要直接存取图表内容（顺便说一下CvMap 是一个隐藏的构造函数）。图形中的数据可以用cvGetFileNodeByName函数得到,函数返回指向图表文件节点的指针。

一个用户对象是一个标准的类型实例，例如CvMat, CvSeq等,或者任何一个已注册的类型使用cvRegisterTypeInfo。这样的对象最初在文件中表现为一种层级关系，(像表现XML 和 YAM示例文件一样)。在文件存储器打开并分析之后。当用户调用cvRead或cvReadByName函数时 那么对象将请求被解析 (按照原来的存储方式）。

[编辑]

CvAttrList

属性列表

```
typedef struct CvAttrList
{
    const char** attr; /* NULL-指向数组对(attribute_name,attribute_value) 的空指针 */
    struct CvAttrList* next; /* 指向下一个属性块的指针 */
}
CvAttrList;

/* 初始化构造函数CvAttrList */
inline CvAttrList cvAttrList( const char** attr=NULL, CvAttrList* next=NULL );

/* 返回值为属性值，找不到适合的属性则返回值为0(NULL)*/
const char* cvAttrValue( const CvAttrList* attr, const char* attr_name );
```

在当前版本的属性列表用来传递额外的参数，在使用cvWrite写入自定义数据对象时。除了对象类型说明(type_id 属性)以外，它不支持 XML 在标签内的属性（注：例如<A name="test"></A>不支持）。

[编辑]

OpenFileStorage

打开文件存储器读/写数据。

```
CvFileStorage* cvOpenFileStorage( const char* filename, CvMemStorage* memstorage, int flags );
```

filename
: 内存中的相关文件的文件名。

memstorage
: 内存中通常存储临时数据和动态结构，例如 CvSeq 和 CvGraph。如果memstorage 为空,将建立和使用一个暂存器。

flags
: 读/写选择器。

    CV_STORAGE_READ - 内存处于读状态。
    CV_STORAGE_WRITE - 内存处于写状态。

函数cvOpenFileStorage打开文件存储器读写数据，之后建立文件或继续使用现有的文件 。文件扩展名决定读文件的类型： .xml 是 XML的扩展名, .yml 或 .yaml 是 YAML的扩展名。该函数的返回指针指向CvFileStorage结构。

[编辑]

ReleaseFileStorage

释放文件存储单元

```
void  cvReleaseFileStorage( CvFileStorage** fs );
```

fs
双指针指向被关闭的文件存储器。

函数cvReleaseFileStorage 关闭一个相关的文件存储器并释放所有的临时内存。只有在内存的I/O操作完成后才能关闭文件存储器。

[编辑]

写数据

[编辑]

StartWriteStruct

向文件存储器中写数据

```
void  cvStartWriteStruct( CvFileStorage* fs, const char* name,
                          int struct_flags, const char* type_name=NULL,
                          CvAttrList attributes=cvAttrList());
```

fs
初始化文件存储器。
name
被写入的数据结构的名称。在存储器被读取时可以通过名称访问数据结构。
struct_flags
有下列两个值:

CV_NODE_SEQ - 被写入的数据结构为序列结构。这样的数据没有名称。
CV_NODE_MAP - 被写入的数据结构为图表结构。这样的数据含有名称。

这两个标志符必须被指定一个

CV_NODE_FLOW - 这个可选择标识符只能作用于YAML流。被写入的数据结构被看做一个数据流（不是数据块），它更加紧凑，当结构或数组里的数据是标量时，推荐用这个标志。

type_name
可选参数 - 对象类型名称。如果是XML用打开标识符type_id 属性写入。如果是YAML 用冒号后面的数据结构名写入，:: 基本上它是伴随用户对象出现的。当读存储器时，编码类型名通常决定对象类型（见Cvtypeinfo和cvfindtypeinfo） 。
attributes
这个参数当前版本没有使用。

函数 cvStartWriteStruct 开始写复合的数据结构（数据集合）包括序列或图表, 在结构体中所有的字段（可以是标量和新的结构）被写入后， 需要调用 cvEndWriteStruct . 该函数能够合并一些对象或写入一些用户对象（参考 CvTypeInfo ）。

[编辑]

EndWriteStruct

结束数据结构的写操作

```
void  cvEndWriteStruct( CvFileStorage* fs );
```

fs
初始化文件存储器。

函数cvEndWriteStruct 结束普通的写数据操作。

[编辑]

WriteInt

写入一个整型值

```
void  cvWriteInt( CvFileStorage* fs, const char* name, int value );
```

fs
　　初始的文件存储器。
name
　　写入值的名称 。如果母结构是一个序列，把name的值置为NULL。
value
　　写入的整型值。

函数 cvWriteInt 将一个单独的整型值（有符号的或无符号的）写入文件存储器。

[编辑]

WriteReal

写入一个浮点数

```
void  cvWriteReal( CvFileStorage* fs, const char* name, double value );
```

fs
　　文件存储器。
name
　　写入值的名称 。如果父结构是一个序列，则name的值应为NULL。
value
　　写入的浮点数。

函数 cvWriteReal 将一个单精度浮点数（有符号的或无符号的）写入文件存储器。 一些特殊的值以特殊的编码表示:

```
    NaN     表示不是数字
    +.Inf   表示正无穷
    -.Inf   表示负无穷
```

下面的实例展示 怎样使用底层写函数存储自定义数据结构。

```
void write_termcriteria( CvFileStorage* fs, const char* struct_name,
                         CvTermCriteria* termcrit )
{
    cvStartWriteStruct( fs, struct_name, CV_NODE_MAP, NULL, cvAttrList(0,0));
    cvWriteComment( fs, "termination criteria", 1 );
    if( termcrit->type & CV_TERMCRIT_ITER )
        cvWriteInteger( fs, "max_iterations", termcrit->max_iter );
    if( termcrit->type & CV_TERMCRIT_EPS )
        cvWriteReal( fs, "accuracy", termcrit->epsilon );
    cvEndWriteStruct( fs );
}
```

[编辑]

WriteString

写入文本字符串

void cvWriteString(CvFileStorage* fs, const char* name,

```
                    const char* str, int quote=0 );
```

fs
　　文件存储器。
name
　　写入字符串的名称 。如果父结构是一个序列，name的值应为NULL。
str
　　写入的文本字符串。
quote
　　如果不为0，不管是否需要引号，字符串都将被写入引号 。如果标识符为0。只有在需要的情况下写入引号（例如字符串的首位是数字或者空格时候就需要两边加上引号）。

函数 cvWriteString将文本字符串写入文件存储器。

[编辑]

WriteComment

写入注释

```
void  cvWriteComment( CvFileStorage* fs, const char* comment, int eol_comment );
```

fs
　　文件存储器。
comment
　　写入的注释,注释可以是单行的或者多行的。
eol_comment
　　如果不为0，函数将注释加到当前行的后面。 如果为0，并且是多行注释或者当前行放不下，那么注释将从新的一行开始 。

函数 cvWriteComment将注释写入文件存储器。读内存时注释将被跳过，它只能被用于调试和查看描述。

[编辑]

StartNextStream

打开下一个数据流

```
void  cvStartNextStream( CvFileStorage* fs );
```

fs
　　初始化文件存储器。

函数 cvStartNextStream 从文件存储器中打开下一个数据流。 YAML 和 XML 都支持多数据流。这对连接多个文件和恢复写入的程序很有用。

[编辑]

Write

写入用户对象

```
void  cvWrite( CvFileStorage* fs, const char* name,
               const void* ptr, CvAttrList attributes=cvAttrList() );
```

fs
　　文件存储器。
name
　　写入对象的名称 。如果父结构是一个序列，name的值为NULL。
ptr
　　定义指针指向对象。

attributes
　　定义对象的属性，每种类型都有特别的指定（见讨论）。

函数 cvWrite将对象写入文件存储器 。首先，使用cvTypeOf 查找恰当的类型信息。其次写入指定的方法类型信息。

属性被用于定制写入程序。下面的属性支持标准类型 (所有的*dt 属性在cvWriteRawData中都有相同的格式）：

　　CvSeq

- header_dt -序列首位用户区的描述，它紧跟在CvSeq或CvChain（如果是自由序列）或CvContour（如果是轮廓或点序列）之后。
- dt - 序列元素的描述 。
- recursive - 如果属性被引用并且不等于“0”或“false"，则所有的序列树（轮廓）都被存储（注：递归存储）。：

　　CvGraph

- header_dt - 图表头用户区的描述，它紧跟在 CvGraph之后。
- vertex_dt - 图表顶点用户区的描述。
- edge_dt - 图表边用户区的描述（ 注意权重值总是被写入，所以不需要明确的说明）。

下面的代码的含义是建立YAML文件用来描述CvFileStorage ：

```
#include "cxcore.h"

int main( int argc, char** argv )
{
    CvMat* mat = cvCreateMat( 3, 3, CV_32F );
    CvFileStorage* fs = cvOpenFileStorage( "example.yml", 0, CV_STORAGE_WRITE );

    cvSetIdentity( mat );
    cvWrite( fs, "A", mat, cvAttrList(0,0) );

    cvReleaseFileStorage( &fs );
    cvReleaseMat( &mat );
    return 0;
}
```

[编辑]

WriteRawData

写入基本数据数组

```
void  cvWriteRawData( CvFileStorage* fs, const void* src,
                      int len, const char* dt );
```

fs
　　文件存储器。
src
　　指针指向输入数组。
len
　　写入数组的长度。
dt
　　下面是每一个数组元素说明的格式： ([count]{'u'|'c'|'w'|'s'|'i'|'f'|'d'})..., 这些特性与C语言的类型相似：

- 'u' - 8位无符号数。
- 'c' - 8位符号数。
- 'w' - 16位无符号数。
- 's' - 16

位符号数。
- 'i' - 32位符号数。
- 'f' - 单精度浮点数。
- 'd' - 双精度浮点数。
- 'r' - 指针。输入的带符号的低32位整数。 这个类型常被用来存储结构体之间的链接。

count 是可选的，是当前类型的计数器 。 例如, dt='2if' 是指任意的一个数组元素的结构是：2个字节整形数，后面跟一个单精度浮点数。上面的说明与'iif', '2i1f' 等相同。另外一个例子：dt='u'是指一个由类型组成的数组， dt='2d'是指由两个双精度浮点数构成的数组。

函数 cvWriteRawData 将数组写入文件存储器，数组由单独的数值构成。这个函数也可以用循环调用 cvWriteInt 和 cvWriteReal 替换，但是一个单独的函数更加有效。需要说明的是，那是因为元素没有名字， 把它们写入序列（无名字）比写入图表（有名字关联）速度会快。

[编辑]

WriteFileNode

将文件节点写入另一个文件存储器

```
void cvWriteFileNode( CvFileStorage* fs, const char* new_node_name,
                      const CvFileNode* node, int embed );
```

fs
: 文件存储器

new_file_node
: 在目的文件存储器中设置新的文件节点名。保持现有的文件节点名，使用cvGetFileNodeName(节点).

node
: 被写入的节点。

embed
: 如果被写入的节点是个集合并且embed不为0，不建立额外的层次结构。 否则所有的节点元素被写入新建的文件节点上 。不过需要确定一点的是，图表元素只被写入图表，序列元素只被写入序列

函数 cvWriteFileNode将一个文件节点的拷贝写入文件存储器 可能应用范围是： 将几个文件存储器合而为一。在XML 和YAML 之间变换格式等。

[编辑]

读取数据

从文件存储器中得到数据有两种步骤：首先查找文件节点包括哪些被请求的数据；然后利用手动或者使用自定义read 方法取得数据。

[编辑]

GetRootFileNode

从文件存储器中得到一个高层节点

```
CvFileNode* cvGetRootFileNode( const CvFileStorage* fs, int stream_index=0 );
```

fs
: 初始化文件存储器 。

stream_index
: 从零开始计数的基索引 。参考 cvStartNextStream. 在通常情况下，文件中只有一个流，但是可以拥有多个。

函数 cvGetRootFileNode 返回一个高层文件节点。 高层节点没有名称，它们和流相对应，接连存入文件存储

器。如果超出索引范围， 函数返回NULL指针， 所以要得到所有高层节点需要反复调用函数stream_index=0,1,...,直到返回NULL指针。这个函数是在文件存储器中递归寻找的基础方 法。

[编辑]

GetFileNodeByName

在图表或者文件存储器中查找节点

```
CvFileNode* cvGetFileNodeByName( const CvFileStorage* fs,
                                 const CvFileNode* map,
                                 const char* name );
```

fs
　　初始化文件存储器。
map
　　设置父图表。如果为NULL，函数 在所有的高层节点（流）中检索， 从第一个开始。
name
　　设置文件节点名。

函数 cvGetFileNodeByName 文件节点通过name 查找文件节点 该节点在图表中被查找，或者如果指针为NULL，那么在内存中的高层文件节点中查找。 在图表中或者在序列调用cvGetSeqElem中使用到这个函数，这样可能遍历整个文件存储器。 为了加速确定某个关键字的多重查询（例如 结构数组 ），可以在cvGetHashedKey 和cvGetFileNode之中用到一个。

[编辑]

GetHashedKey

返回一个指向已有名称的唯一指针

```
CvStringHashNode* cvGetHashedKey( CvFileStorage* fs, const char* name,
                                  int len=-1, int create_missing=0 );
```

fs
　　初始化文件存储器。
name
　　设置文字节点名。
len
　　名称的长度（已知），如果值为-1 长度会被计算出来 。
create_missing
　　标识符说明 ，是否应该将一个缺省节点的值加入哈希表。

函数 cvGetHashedKey返回指向每一个特殊文件节点名的唯一指针。这个指针可以传递给cvGetFileNode函数。它比cvGetFileNodeByName快， 因为比较指针相对比较字符串快些。

观察下面例子： 用二维图来表示一个点集，例：

```
%YAML:1.0
points:
  - { x: 10, y: 10 }
  - { x: 20, y: 20 }
  - { x: 30, y: 30 }
  # ...
```

因而,它使用哈希指针“x”和“y"加速对点的解析。

例：从一个文件存储器中读取一组的结构

```
#include "cxcore.h"
```

```
int main( int argc, char** argv )
{
    CvFileStorage* fs = cvOpenFileStorage( "points.yaml", 0, CV_STORAGE_READ );
    CvStringHashNode* x_key = cvGetHashedKey( fs, "x", -1, 1 );
    CvStringHashNode* y_key = cvGetHashedKey( fs, "y", -1, 1 );
    CvFileNode* points = cvGetFileNodeByName( fs, 0, "points" );

    if( CV_NODE_IS_SEQ(points->tag) )
    {
        CvSeq* seq = points->data.seq;
        int i, total = seq->total;
        CvSeqReader reader;
        cvStartReadSeq( seq, &reader, 0 );
        for( i = 0; i < total; i++ )
        {
            CvFileNode* pt = (CvFileNode*)reader.ptr;
#if 1 /* 快变量 */
            CvFileNode* xnode = cvGetFileNode( fs, pt, x_key, 0 );
            CvFileNode* ynode = cvGetFileNode( fs, pt, y_key, 0 );
            assert( xnode && CV_NODE_IS_INT(xnode->tag) &&
                    ynode && CV_NODE_IS_INT(ynode->tag));
            int x = xnode->data.i; // or x = cvReadInt( xnode, 0 );
            int y = ynode->data.i; // or y = cvReadInt( ynode, 0 );
#elif 1 /* 慢变量: 不使用x值与y值  */
            CvFileNode* xnode = cvGetFileNodeByName( fs, pt, "x" );
            CvFileNode* ynode = cvGetFileNodeByName( fs, pt, "y" );
            assert( xnode && CV_NODE_IS_INT(xnode->tag) &&
                    ynode && CV_NODE_IS_INT(ynode->tag));
            int x = xnode->data.i; // or x = cvReadInt( xnode, 0 );
            int y = ynode->data.i; // or y = cvReadInt( ynode, 0 );
#else /* 最慢的可以轻松使用的变量  */
            int x = cvReadIntByName( fs, pt, "x", 0 /* default value */ );
            int y = cvReadIntByName( fs, pt, "y", 0 /* default value */ );
#endif
            CV_NEXT_SEQ_ELEM( seq->elem_size, reader );
            printf("%d: (%d, %d)\n", i, x, y );
        }
    }
    cvReleaseFileStorage( &fs );
    return 0;
}
```

请注意,无论使用那一种方法访问图表 ，都比使用序列慢，例如上面的例子，如果把数据作为整数对放在在单一数字序列中，效率会更高 。

[编辑]

GetFileNode

在图表或者文件存储器中查找节点

```
CvFileNode* cvGetFileNode( CvFileStorage* fs, CvFileNode* map,
                           const CvStringHashNode* key, int create_missing=0 );
```

fs
: 初始化文件存储器 。

map
: 设置母图表。如果为NULL，函数 在所有的高层节点（流）中检索，如果图表与值都为 NULLs,函数返回到根节点-图表包含高层节点

key
: 指向节点名的特殊节点，从cvGetHashedKey中得到 。

create_missing
: 标识符说明 ，是否应该将一个缺省节点加入图表。

函数 cvGetFileNode 查找一个文件节点。函数能够插入一个新的节点，当它不在图表中时

[编辑]

GetFileNodeName

返回文件节点名

```
const char* cvGetFileNodeName( const CvFileNode* node );
```

node
　　初始化文件节点 。

函数 cvGetFileNodeName 返回文件节点名或返回NULL(如果文件节点没有名称或者node为NULL 。

[编辑]

ReadInt

从文件节点中得到整形值

```
int cvReadInt( const CvFileNode* node, int default_value=0 );
```

node
　　初始化文件节点
default_value
　　如果node为NULL，返回一个值。

函数 cvReadInt 从文件节点中返回整数。如果文件节点为NULL， default_value 被返回 。另外如果文件节点有类型 CV_NODE_INT, 则 node->data.i 被返回 。如果文件节点有类型 CV_NODE_REAL, 则 node->data.f 被修改成整数后返回。 其他的情况是则结果不确定。

[编辑]

ReadIntByName

查找文件节点返回它的值

```
int cvReadIntByName( const CvFileStorage* fs, const CvFileNode* map,
                     const char* name, int default_value=0 );
```

fs
　　初始化文件存储器 。
map
　　设置父图表。如果为NULL，函数 在所有的高层节点（流）中检索。
name
　　设置节点名。
default_value
　　如果文件节点为NULL，返回一个值。

函数 cvReadIntByName是 cvGetFileNodeByName 和 cvReadInt的简单重叠.

[编辑]

ReadReal

从文件节点中得到浮点形值

```
double cvReadReal( const CvFileNode* node, double default_value=0. );
```

node
　　初始化文件节点 。
default_value
　　如果node为NULL，返回一个值。

函数cvReadReal 从文件节点中返回浮点形值。如果文件节点为NULL， default_value 被返回（这样就不用检

查cvGetFileNode 返回的指针是否为空了） 。另外如果文件节点有类型 CV_NODE_REAL , 则node->data.f 被返回 。如果文件节点有类型 CV_NODE_INT , 则 node->data.i 被修改成浮点数后返回。 另外一种情况是，结果不确定。 .

[编辑]

ReadRealByName

查找文件节点返回它的浮点形值

```
double  cvReadRealByName( const CvFileStorage* fs, const CvFileNode* map,
                          const char* name, double default_value=0. );
```

fs
　　初始化文件存储器 。
map
　　设置父图表。如果为NULL，函数 在所有的高层节点（流）中检索。
name
　　设置节点名。
default_value
　　如果node为NULL，返回一个值。

函数 cvReadRealByName 是 cvGetFileNodeByName 和cvReadReal 的简单重叠。

[编辑]

ReadString

从文件节点中得到字符串文本

```
const char* cvReadString( const CvFileNode* node, const char* default_value=NULL );
```

node
　　初始化文件节点。
default_value
　　如果node为NULL，返回一个值。

函数cvReadString 从文件节点中返回字符串文本。如果文件节点为NULL， default_value 被返回 。另外如果文件节点有类型CV_NODE_STR, 则data.str.ptr 被返回 。 另外一种情况是，结果不确定。

[编辑]

ReadStringByName

查找文件节点返回它的字符串文本

```
const char* cvReadStringByName( const CvFileStorage* fs, const CvFileNode* map,
                                const char* name, const char* default_value=NULL );
```

fs
　　初始化文件存储器 。
map
　　设置母图表。如果为NULL，函数 在所有的高层节点（流）中检索。
name
　　设置节点名。
default_value
　　如果文件节点为NULL，返回一个值。

函数 cvReadStringByName是 cvGetFileNodeByName 和cvReadString 的简单重叠。

[编辑]

Read

解释对象并返回指向它的指针

```
void* cvRead( CvFileStorage* fs, CvFileNode* node,
              CvAttrList* attributes=NULL );
```

fs
　　初始化文件存储器。
node
　　设置对象根节点。
attributes

不被使用的参数.

函数 cvRead 解释用户对象 (在文件存储器子树中建立新的对象）并返回。对象被解释 ，必须按原有的支持读方法的类型 (参考 CvTypeInfo).用类型名决定对象，并在文件中被解释 。如果对象是动态结构，它将在内存中创建传递给cvOpenFileStorage或者使NULL指针被建立在临时性内存中。当 cvReleaseFileStorage 被调用时释放内存。 如果对象不是动态结构 ,将在堆中被建立，释放它的内存需要用专用函数或通用函数cvRelease。

[编辑]

ReadByName

查找对象并解释

```
void* cvReadByName( CvFileStorage* fs, const CvFileNode* map,
                    const char* name, CvAttrList* attributes=NULL );
```

fs
　　初始化文件存储器。
map
　　设置父节点。如果它为NULL，函数从高层节点中查找。
name
　　设置节点名称。
attributes
　　不被使用的参数.

函数 cvReadByName 是由cvGetFileNodeByName 和 cvRead叠合的。 .

[编辑]

ReadRawData

读重数

```
void cvReadRawData( const CvFileStorage* fs, const CvFileNode* src,
                    void* dst, const char* dt );
```

fs
　　初始化文件存储器 。
src
　　设置文件节点（有序的）来读数。
dst
　　设置指向目的数组的指针。
dt
　　数组元素的说明。格式参考 cvWriteRawData。

函数 cvReadRawData从有序的文件节点中读取标量元素。

[编辑]

StartReadRawData

初始化文件节点读取器

```
void cvStartReadRawData( const CvFileStorage* fs, const CvFileNode* src,
                         CvSeqReader* reader );
```

fs
　　初始化文件存储器。
src
　　设置文件节点(序列）来读数。
reader
　　设置顺序读取指针。

函数 cvStartReadRawData 初始化序列读取器从文件节点中读取数据。初始化的首部通过传给cvReadRawDataSlice使用 。

[编辑]

ReadRawDataSlice

初始化文件节点序列

```
void cvReadRawDataSlice( const CvFileStorage* fs, CvSeqReader* reader,
                         int count, void* dst, const char* dt );
```

fs
　　文件存储器。
reader
　　设置序列读取器．用 cvStartReadRawData.初始化。
count
　　被读取元素的数量。
dst
　　指向目的数组的指针。
dt
　　数组元素的说明。格式参考 cvWriteRawData。

函数 cvReadRawDataSlice 从文件节点读一个或多个元素，组成一个序列用于指定数组。读入元素的总数由其他数组的元素总和乘以每个数组元素数目。例如 如果 dt='2if', 函数将读是total*3数量的序列元素。对于任何数组，可以使用cvSetSeqReaderPos自由定位，跳过某些位置或者重复读取。

[编辑]

运行时类型信息和通用函数

[编辑]

CvTypeInfo

类型信息

```
typedef int (CV_CDECL *CvIsInstanceFunc)( const void* struct_ptr );
typedef void (CV_CDECL *CvReleaseFunc)( void** struct_dblptr );
typedef void* (CV_CDECL *CvReadFunc)( CvFileStorage* storage, CvFileNode* node );
typedef void (CV_CDECL *CvWriteFunc)( CvFileStorage* storage,
                                      const char* name,
                                      const void* struct_ptr,
```

```
                                        CvAttrList attributes );
typedef void* (CV_CDECL *CvCloneFunc)( const void* struct_ptr );

typedef struct CvTypeInfo
{
    int flags; /* 不常用 */
    int header_size; /*  (CvTypeInfo)的大小 sizeof(CvTypeInfo) */
    struct CvTypeInfo* prev; /* 在列表中已定义过的类型  */
    struct CvTypeInfo* next; /* 在列表中下一个已定义过的类型  */
    const char* type_name; /*定义类型名，并写入文件存储器 */

    /* methods */
    CvIsInstanceFunc is_instance; /* 选择被传递的对象是否属于的类型  */
    CvReleaseFunc release; /* 释放对象的内存空间  */
    CvReadFunc read; /* 从文件存储器中读对象 */
    CvWriteFunc write; /* 将对象写入文件存储器 */
    CvCloneFunc clone; /* 复制一个对象 */
}
CvTypeInfo;
```

结构 CvTypeInfo包含的信息包括标准的或用户自定义的类型。类型的实例可能有也可能没有包含指向相应的CvTypeInfo结构的指针。在已有的对象 中查找类型的方法是使用cvTypeOf函数。在从文件存储器中读对象的时候已有的类型信息可以通过类型名使用cvFindType来查找 。 用户可以通过cvRegisterType定义一个新的类型 ，并将类型信息结构加到文件列表的开始端， 它可以从标准类型中建立专门的类型，重载基本的方法。

[编辑]

RegisterType

定义新类型

```
void cvRegisterType( const CvTypeInfo* info );
```

info
　　类型信息结构。

函数 cvRegisterType 定义一个新类型,可以通过信息来描述它。这个函数在内存创建了一个copy,所以在用完以后，应该删除它。

[编辑]

UnregisterType

删除定义的类型

```
void cvUnregisterType( const char* type_name );
```

type_name
　　被删除的类型的名称。

函数 cvUnregisterType通过指定的名称删除已定义的类型。 如果不知道类型名,可以用cvTypeOf或者连续扫描类型列表，从cvFirstType开始,然后调用 cvUnregisterType(info->type_name)。

[编辑]

FirstType

返回类型列表的首位。

```
CvTypeInfo* cvFirstType( void );
```

函数 cvFirstType 返回类型列表中的第一个类型。可以利用CvTypeInfo 的prev next来实现遍历。

[编辑]

FindType

通过类型名查找类型

```
CvTypeInfo* cvFindType( const char* type_name );
```

type_name
　　类型名

函数 cvFindType通过类型名查找指定的类型。如果找不到返回值为NULL。

[编辑]

TypeOf

返回对象的类型

```
CvTypeInfo* cvTypeOf( const void* struct_ptr );
```

struct_ptr
　　定义对象指针。

函数 cvTypeOf 查找指定对象的类型。它反复扫描类型列表，调用每一个类型信息结构中的函数和方法与对象做比较，直到它们中的一个的返回值不为0或者所有的类型都被访问。

[编辑]

Release

删除对象

```
void cvRelease( void** struct_ptr );
```

struct_ptr
　　定义指向对象的双指针。

函数 cvRelease 查找指定对象的类型，然后调用release。

[编辑]

Clone

克隆一个对象

```
void* cvClone( const void* struct_ptr );
```

struct_ptr
　　定义被克隆的对象

函数 cvClone 查找指定对象的类型，然后调用 clone。

[编辑]

Save

存储对象到文件中

```
void cvSave( const char* filename, const void* struct_ptr,
             const char* name=NULL,
             const char* comment=NULL,
             CvAttrList attributes=cvAttrList());
```

filename
　　初始化文件名。
struct_ptr
　　指定要存储的对象。
name
　　可选择的对象名 。如果为 NULL, 对象名将从filename中列出。
comment
　　可选注释。加在文件的开始处。
attributes
　　可选属性。 传递给cvWrite。

函数 cvSave存储对象到文件。它给cvWrite提供一个简单的接口。

[编辑]

Load

从文件中打开对象。

```
void* cvLoad( const char* filename, CvMemStorage* memstorage=NULL,
              const char* name=NULL, const char** real_name=NULL );
```

filename
　　初始化文件名
memstorage
　　动态结构的内存，例如CvSeq或CvGraph。不能作用于矩阵或图像。:
name
　　可选对象名。如果为 NULL,内存中的第一个高层对象被打开。
real_name
　　可选输出参数 。它包括已打开的对象的名称 (如果 name=NULL时有效）。

函数 cvLoad 从文件中打开对象。它给cvRead提供一个简单的接口.对象被打开之后，文件存储器被关闭，所有的临时缓冲区被删除。因而，为了能打开一个动态结构，如序列，轮廓或图像，你应该为该函数传递一个有效的目标存储器。

Cxcore其它混合函数

Wikipedia，自由的百科全书

[编辑]

CheckArr

检查输入数组的每一个元素是否是合法值

```
int  cvCheckArr( const CvArr* arr, int flags=0,
                 double min_val=0, double max_val=0);
#define cvCheckArray cvCheckArr
```

arr
　　待检查数组
flags
　　操作标志, 0 或者下面值的组合:

　　CV_CHECK_RANGE - 如果设置这个标志, 函数检查数组的每一个值是否在范围 [minVal,maxVal) 以内；否则，它只检查每一个元素是否是 NaN 或者 ±Inf。
　　CV_CHECK_QUIET - 设置这个标志后, 如果一个元素是非法值的或者越界时，函数不会产生错误。

min_val
　　有效值范围的闭下边界。只有当 CV_CHECK_RANGE 被设置的时候它才有作用。
max_val
　　有效值范围的开上边界。只有当 CV_CHECK_RANGE 被设置的时候它才有作用。

函数 cvCheckArr 检查每一个数组元素是否是 NaN 或者 ±Inf。如果 CV_CHECK_RANGE 被设定, 它将检查每一个元素是否大于等于 minVal 并且小于maxVal。如果检查成功函数返回非零值，例如，所有元素都是合法的并且在范围内，如果检查失败则返回 0 。 在后一种情况下如果 CV_CHECK_QUIET 标志没有被设定，函数将报出运行错误。

[编辑]

KMeans2

按照给定的类别数目对样本集合进行聚类

```
void cvKMeans2( const CvArr* samples, int cluster_count,
                CvArr* labels, CvTermCriteria termcrit );
```

samples
　　输入样本的浮点矩阵，每个样本一行。
cluster_count
　　所给定的聚类数目
labels
　　输出整数向量：每个样本对应的类别标识
termcrit
　　指定聚类的最大迭代次数和／或精度（两次迭代引起的聚类中心的移动距离）

函数 cvKMeans2 执行 k-means 算法 搜索 cluster_count 个类别的中心并对样本进行分类，输出 labels(i) 为样本 i 的类别标识。

例子. 用 k-means 对高斯分布的随机样本进行聚类

```
#include "cxcore.h"
```

```
#include "highgui.h"

int main( int argc, char** argv )
{
    #define MAX_CLUSTERS 5
    CvScalar color_tab[MAX_CLUSTERS];
    IplImage* img = cvCreateImage( cvSize( 500, 500 ), 8, 3 );
    CvRNG rng = cvRNG(0xffffffff);

    color_tab[0] = CV_RGB(255,0,0);
    color_tab[1] = CV_RGB(0,255,0);
    color_tab[2] = CV_RGB(100,100,255);
    color_tab[3] = CV_RGB(255,0,255);
    color_tab[4] = CV_RGB(255,255,0);

    cvNamedWindow( "clusters", 1 );

    for(;;)
    {
        int k, cluster_count = cvRandInt(&rng)%MAX_CLUSTERS + 1;
        int i, sample_count = cvRandInt(&rng)%1000 + 1;
        CvMat* points = cvCreateMat( sample_count, 1, CV_32FC2 );
        CvMat* clusters = cvCreateMat( sample_count, 1, CV_32SC1 );

        /* generate random sample from multigaussian distribution */
        for( k = 0; k < cluster_count; k++ )
        {
            CvPoint center;
            CvMat point_chunk;
            center.x = cvRandInt(&rng)%img->width;
            center.y = cvRandInt(&rng)%img->height;
            cvGetRows( points, &point_chunk, k*sample_count/cluster_count,
                       k == cluster_count - 1 ? sample_count : (k+1)*sample_count/cluster_count
);
            cvRandArr( &rng, &point_chunk, CV_RAND_NORMAL,
                       cvScalar(center.x,center.y,0,0),
                       cvScalar(img->width/6, img->height/6,0,0) );
        }

        /* shuffle samples */
        for( i = 0; i < sample_count/2; i++ )
        {
            CvPoint2D32f* pt1 = (CvPoint2D32f*)points->data.fl + cvRandInt(&rng)%sample_count;
            CvPoint2D32f* pt2 = (CvPoint2D32f*)points->data.fl + cvRandInt(&rng)%sample_count;
            CvPoint2D32f temp;
            CV_SWAP( *pt1, *pt2, temp );
        }

        cvKMeans2( points, cluster_count, clusters,
                   cvTermCriteria( CV_TERMCRIT_EPS+CV_TERMCRIT_ITER, 10, 1.0 ));

        cvZero( img );

        for( i = 0; i < sample_count; i++ )
        {
            CvPoint2D32f pt = ((CvPoint2D32f*)points->data.fl)[i];
            int cluster_idx = clusters->data.i[i];
            cvCircle( img, cvPointFrom32f(pt), 2, color_tab[cluster_idx], CV_FILLED );
        }

        cvReleaseMat( &points );
        cvReleaseMat( &clusters );

        cvShowImage( "clusters", img );

        int key = cvWaitKey(0);
        if( key == 27 ) // 'ESC'
            break;
    }
}
```

[编辑]

SeqPartition

拆分序列为等效的类

```
typedef int (CV_CDECL* CvCmpFunc)(const void* a, const void* b, void* userdata);
int cvSeqPartition( const CvSeq* seq, CvMemStorage* storage, CvSeq** labels,
```

```
                    CvCmpFunc is_equal, void* userdata );
```

seq
: 划分序列

storage
: 存储序列的等效类的存储器，如果为空, 函数用 seq->storage 存储输出标签。

labels
: 输出参数。指向序列指针的指针，这个序列存储以0为开始的输入序列元素的标签。

is_equal
: 比较函数指针。如果两个特殊元素是来自同一个类，那这个比较函数返回非零值，否则返回 0 。划分算法用比较函数的传递闭包得到等价类。

userdata
: 直接传递给 is_equal 函数的指针。

函数 cvSeqPartition 执行二次方程算法为拆分集合为一个或者更多的等效类。 函数返回等效类的数目。

例子：拆分二维点集。

```
#include "cxcore.h"
#include "highgui.h"
#include <stdio.h>

CvSeq* point_seq = 0;
IplImage* canvas = 0;
CvScalar* colors = 0;
int pos = 10;

int is_equal( const void* _a, const void* _b, void* userdata )
{
    CvPoint a = *(const CvPoint*)_a;
    CvPoint b = *(const CvPoint*)_b;
    double threshold = *(double*)userdata;
    return (double)(a.x - b.x)*(a.x - b.x) + (double)(a.y - b.y)*(a.y - b.y) <= threshold;
}

void on_track( int pos )
{
    CvSeq* labels = 0;
    double threshold = pos*pos;
    int i, class_count = cvSeqPartition( point_seq, 0, &labels, is_equal, &threshold );
    printf("%4d classes\n", class_count );
    cvZero( canvas );

    for( i = 0; i < labels->total; i++ )
    {
        CvPoint pt = *(CvPoint*)cvGetSeqElem( point_seq, i );
        CvScalar color = colors[*(int*)cvGetSeqElem( labels, i )];
        cvCircle( canvas, pt, 1, color, -1 );
    }

    cvShowImage( "points", canvas );
}

int main( int argc, char** argv )
{
    CvMemStorage* storage = cvCreateMemStorage(0);
    point_seq = cvCreateSeq( CV_32SC2, sizeof(CvSeq), sizeof(CvPoint), storage );
    CvRNG rng = cvRNG(0xffffffff);

    int width = 500, height = 500;
    int i, count = 1000;
    canvas = cvCreateImage( cvSize(width,height), 8, 3 );

    colors = (CvScalar*)cvAlloc( count*sizeof(colors[0]) );
    for( i = 0; i < count; i++ )
    {
        CvPoint pt;
        int icolor;
        pt.x = cvRandInt( &rng ) % width;
        pt.y = cvRandInt( &rng ) % height;
        cvSeqPush( point_seq, &pt );
        icolor = cvRandInt( &rng ) | 0x00404040;
        colors[i] = CV_RGB(icolor & 255, (icolor >> 8)&255, (icolor >> 16)&255);
```

```
    }

    cvNamedWindow( "points", 1 );
    cvCreateTrackbar( "threshold", "points", &pos, 50, on_track );
    on_track(pos);
    cvWaitKey(0);
    return 0;
}
```

Cxcore错误处理和系统函数

Wikipedia，自由的百科全书

目录

[隐藏]

[编辑]

错误处理

在 OpenCV 中错误处理和 IPL (Image Processing Library)很相似。如果调用函数出现错误将并不直接返回错误代码，而是用CV_ERROR 宏调用 cvError 函数报错，按次序地，用 cvSetErrStatus 函数设置错误状态，然后调用标准的或者用户自定义的错误处理器(它可以显示一个消息对话框，导出错误日志等等, 参考函数 cvRedirectError, cvNulDevReport, cvStdErrReport, cvGuiBoxReport)。每个程序的线程都有一个全局变量，它包含了错误状态(一个整数值)。这个状态可以被 cvGetErrStatus 函数检索到。

有三个错误处理模式(参考 cvSetErrMode 和 cvGetErrMode):

Leaf
: 错误处理器被调用以后程序被终止。这是缺省值。它在调试中是很有用的，当错误发生的时候立即产生错误信息。然而对于产生式系统，后面两种模式提供的更多控制能力可能会更有用。

Parent
: 错误处理器被调用以后程序不会被终止。栈被清空 (它用 C++ 异常处理机制完成写/输出--w/o)。当调用 CxCore 的函数 cvGetErrStatus 起作用以后用户可以检查错误代码。

Silent
: 和 Parent 模式相似, 但是没有错误处理器被调用。

事实上, Leaf 和 Parent 模式的语义被错误处理器执行，上面的描述对 cvNulDevReport, cvStdErrReport. cvGuiBoxReport 的行为有一些细微的差别，一些自定义的错误处理器可能语义上会有很大的不同。 错误处理宏

报错，检查错误等的宏

```
/* special macros for enclosing processing statements within a function and separating
   them from prologue (resource initialization) and epilogue (guaranteed resource release) */
#define __BEGIN__       {
#define __END__         goto exit; exit: ; }
/* proceeds to "resource release" stage */
#define EXIT            goto exit

/* Declares locally 函数 name for CV_ERROR() use */
#define CV_FUNCNAME( Name )  \
    static char cvFuncName[] = Name

/* Raises an error within the current context */
#define CV_ERROR( Code, Msg )                                                \
{                                                                            \
     cvError( (Code), cvFuncName, Msg, __FILE__, __LINE__ );                 \
     EXIT;                                                                   \
}

/* Checks status after calling CXCORE function */
#define CV_CHECK()                                                           \
{                                                                            \
    if( cvGetErrStatus() < 0 )                                          \
        CV_ERROR( CV_StsBackTrace, "Inner function failed." );          \
}

/* Provies shorthand for CXCORE function call and CV_CHECK() */
#define CV_CALL( Statement )                                                 \
{                                                                            \
    Statement;                                                               \
    CV_CHECK();                                                              \
}

/* Checks some condition in both debug and release configurations */
#define CV_ASSERT( Condition )                                                  \
{                                                                               \
    if( !(Condition) )                                                          \
        CV_ERROR( CV_StsInternal, "Assertion: " #Condition " failed" ); \
}

/* these macros are similar to their CV_... counterparts, but they
   do not need exit label nor cvFuncName to be defined */
#define OPENCV_ERROR(status,func_name,err_msg) ...
#define OPENCV_ERRCHK(func_name,err_msg) ...
#define OPENCV_ASSERT(condition,func_name,err_msg) ...
#define OPENCV_CALL(statement) ...
```

取代上面的讨论, 这里有典型的 CXCORE 函数和这些函数使用的样例。 错误处理宏的使用

```
#include "cxcore.h"
#include <stdio.h>

void cvResizeDCT( CvMat* input_array, CvMat* output_array )
{
    CvMat* temp_array = 0; // declare pointer that should be released anyway.

    CV_FUNCNAME( "cvResizeDCT" ); // declare cvFuncName

    __BEGIN__; // start processing. There may be some declarations just after this macro,
               // but they couldn't be accessed from the epilogue.

    if( !CV_IS_MAT(input_array) || !CV_IS_MAT(output_array) )
        // use CV_ERROR() to raise an error
        CV_ERROR( CV_StsBadArg, "input_array or output_array are not valid matrices" );

    // some restrictions that are going to be removed later, may be checked with CV_ASSERT()
    CV_ASSERT( input_array->rows == 1 && output_array->rows == 1 );

    // use CV_CALL for safe function call
    CV_CALL( temp_array = cvCreateMat( input_array->rows, MAX(input_array->cols,output_array-
>cols),
                                       input_array->type ));

    if( output_array->cols > input_array->cols )
        CV_CALL( cvZero( temp_array ));

    temp array->cols = input array->cols;
```

```
    CV_CALL( cvDCT( input_array, temp_array, CV_DXT_FORWARD ));
    temp_array->cols = output_array->cols;
    CV_CALL( cvDCT( temp_array, output_array, CV_DXT_INVERSE ));
    CV_CALL( cvScale( output_array, output_array, 1./sqrt((double)input_array->cols*output_array-
>cols), 0 ));

    __END__; // finish processing. Epilogue follows after the macro.

    // release temp_array. If temp_array has not been allocated before an error occured,
cvReleaseMat
    // takes care of it and does nothing in this case.
    cvReleaseMat( &temp_array );
}


int main( int argc, char** argv )
{
    CvMat* src = cvCreateMat( 1, 512, CV_32F );
#if 1 /* no errors */
    CvMat* dst = cvCreateMat( 1, 256, CV_32F );
#else
    CvMat* dst = 0; /* test error processing mechanism */
#endif
    cvSet( src, cvRealScalar(1.), 0 );
#if 0 /* change 0 to 1 to suppress error handler invocation */
    cvSetErrMode( CV_ErrModeSilent );
#endif
    cvResizeDCT( src, dst ); // if some error occurs, the message box will popup, or a message
will be
                             // written to log, or some user-defined processing will be done
    if( cvGetErrStatus() < 0 )
        printf("Some error occured" );
    else
        printf("Everything is OK" );
    return 0;
}
```

[编辑]

GetErrStatus

返回当前错误状态

```
int cvGetErrStatus( void );
```

函数 cvGetErrStatus 返回当前错误状态 - 这个状态是被最近调用的 cvSetErrStatus 设置的。 注意, 在 Leaf 模式下错误一旦发生程序立即被终止 , 因此对于总是需要调用函数仍然获得控制的应用 , 可以调用 cvSetErrMode 函数将错误模式设置为 Parent 或 Silent 。

[编辑]

SetErrStatus

设置错误状态

```
void cvSetErrStatus( int status );
```

status
    错误状态

函数 cvSetErrStatus 设置错误状态为指定的值。 大多数情况下, 该函数 被用来重设错误状态(设置为 CV_StsOk) 以从错误中恢复。在其他情况下调用 cvError 或 CV_ERROR 更自然一些。

[编辑]

GetErrMode

返回当前错误模式

```
int cvGetErrMode( void );
```

函数 cvGetErrMode 返回当前错误模式 - 这个值是被最近一次 cvSetErrMode 函数调用所设定的。

[编辑]

SetErrMode

设置当前错误模式

```
#define CV_ErrModeLeaf    0
#define CV_ErrModeParent  1
#define CV_ErrModeSilent  2
int cvSetErrMode( int mode );
```

mode
: 错误模式

函数 cvSetErrMode 设置指定的错误模式。关于不同的错误模式的讨论参考本节开始.

[编辑]

Error

产生一个错误

```
int cvError( int status, const char* func_name,
             const char* err_msg, const char* file_name, int line );
```

status
: 错误状态

func_name
: 产生错误的函数名

err_msg
: 关于错误的额外诊断信息

file_name
: 产生错误的文件名

line
: 产生错误的行号

函数 cvError 设置错误状态为指定的值(通过 cvSetErrStatus) ，如果错误模式不是 Silent, 调用错误处理器。

[编辑]

ErrorStr

返回错误状态编码的原文描述

```
const char* cvErrorStr( int status );
```

status
: 错误状态

函数 cvErrorStr 返回指定错误状态编码的原文描述。如果是未知的错误状态该函数返回空 （NULL）指针。

[编辑]

RedirectError

设置一个新的错误处理器

```
typedef int (CV_CDECL *CvErrorCallback)( int status, const char* func_name,
                    const char* err_msg, const char* file_name, int line, void* userdata  );
```

```
CvErrorCallback cvRedirectError( CvErrorCallback error_handler,
                                 void* userdata=NULL, void** prev_userdata=NULL );
```

error_handler
: 新的错误处理器

userdata
: 传给错误处理器的任意透明指针

prev_userdata
: 指向前面分配给用户数据的指针的指针

函数 cvRedirectError 在标准错误处理器或者有确定借口的自定义错误处理器中选择一个新的错误处理器 。错误处理器和 cvError 函数有相同的参数。 如果错误处理器返回非零的值, 程序终止, 否则, 程序继续运行。错误处理器通过 cvGetErrMode 检查当前错误模式而作出决定。

[编辑]

cvNulDevReport cvStdErrReport cvGuiBoxReport

提供标准错误操作

```
int cvNulDevReport( int status, const char* func_name,
                    const char* err_msg, const char* file_name,
                    int line, void* userdata );

int cvStdErrReport( int status, const char* func_name,
                    const char* err_msg, const char* file_name,
                    int line, void* userdata );

int cvGuiBoxReport( int status, const char* func_name,
                    const char* err_msg, const char* file_name,
                    int line, void* userdata );
```

status
: 错误状态

func_name
: 产生错误的函数名

err_msg
: 关于错误的额外诊断信息

file_name
: 产生错误的文件名

line
: 产生错误的行号

userdata
: 指向用户数据的指针，被标准错误操作忽略。

函数 cvNullDevReport, cvStdErrReport, cvGuiBoxReport 提供标准错误操作。cvGuiBoxReport 是 Win32 系统缺省的错误处理器， cvStdErrReport - 其他系统. cvGuiBoxReport 弹出错误描述的消息框并提供几个选择。 下面是一个消息框的例子，如果和例子中的错误描述相同，它和上面的例子代码可能是兼容的。

错误消息对话框

如果错误处理器是 cvStdErrReport, 上面的消息将被打印到标准错误输出，程序将要终止和继续依赖于当前错误模式。

错误消息打印到标准错误输出 (在 Leaf 模式)

```
OpenCV ERROR: Bad argument (input_array or output_array are not valid matrices)
        in function cvResizeDCT, D:\User\VP\Projects\avl_proba\a.cpp(75)
Terminating the application...
```

[编辑]

系统函数

[编辑]

Alloc

分配内存缓冲区

```
void* cvAlloc( size_t size );
```

size
以字节为单位的缓冲区大小

函数 cvAlloc 分配字节缓冲区大小并返回分配的缓冲区的指针。如果错误处理函数产生了一个错误报告 则返回一个空（NULL）指针。 缺省地 cvAlloc 调用 icvAlloc 而 icvAlloc 调用 malloc ，然而用 cvSetMemoryManager 调用用户自定义的内存分配和释放函数也是可能的。

[编辑]

Free

释放内存缓冲区

```
void cvFree( void** ptr );
```

buffer
指向被释放的缓冲区的双重指针

函数 cvFree 释放被 cvAlloc 分配的缓冲区。在退出的时候它清除缓冲区指针，这就是为什么要使用双重指针的原因 。 如果 *buffer 已经是空（NULL），函数什么也不做。

[编辑]

GetTickCount

Returns number of tics

```
int64 cvGetTickCount( void );
```

函数 cvGetTickCount 返回从依赖于平台的事件(从启动开始 CPU 的ticks 数目, 从1970年开始的微秒数目等等)开始的 tics 的数目 。 该函数对于精确测量函数/用户代码的执行时间是很有用的。要转化 tics 的数目为时间单位，使用函数 cvGetTickFrequency 。

[编辑]

GetTickFrequency

返回每个微秒的 tics 的数目

```
double cvGetTickFrequency( void );
```

函数 cvGetTickFrequency 返回每个微秒的 tics 的数目。 因此， cvGetTickCount() 和 cvGetTickFrequency() 将给出从依赖于平台的事件开始的 tics 的数目 。

[编辑]

RegisterModule

Registers another module 注册另外的模块

```
typedef struct CvPluginFuncInfo
{
    void** func_addr;
    void* default_func_addr;
    const char* func_names;
    int search_modules;
    int loaded_from;
}
CvPluginFuncInfo;

typedef struct CvModuleInfo
{
    struct CvModuleInfo* next;
    const char* name;
    const char* version;
    CvPluginFuncInfo* func_tab;
}
CvModuleInfo;

int cvRegisterModule( const CvModuleInfo* module_info );
```

module_info
    模块信息

函数 cvRegisterModule 添加模块到已注册模块列表中。模块被注册后，用 cvGetModuleInfo 函数可以检索到它的信息。注册模块可以通过 CXCORE的 支持利用优化插件 (IPP, MKL, ...)。 CXCORE ， CV (computer vision), CVAUX (auxilary computer vision) 和 HIGHGUI (visualization & image/video acquisition) 自身就是模块的例子。 通常注册后共享库就被载入。参考 cxcore/src/cxswitcher.cpp and cv/src/cvswitcher.cpp 获取细节信息, 怎样注册的参考 cxcore/src/cxswitcher.cpp ， cxcore/src/_cxipp.h 显示了 IPP 和 MKL 是怎样连接到模块的。

[编辑]

GetModuleInfo

检索注册模块和插件的信息

```
void  cvGetModuleInfo( const char* module_name,
                       const char** version,
                       const char** loaded_addon_plugins );
```

module_name

模块名, 或者 NULL ，则代表所有的模块

version
输出参数，模块的信息，包括版本信息

loaded_addon_plugins
优化插件的名字和版本列表，这里 CXCORE 可以被找到和载入

函数 cvGetModuleInfo 返回一个或者所有注册模块的信息。返回信息被存储到库当中，因此，用户不用释放或者修改返回的文本字符。

[编辑]

UseOptimized

在优化/不优化两个模式之间切换

```
int cvUseOptimized( int on_off );
```

on_off
优化(<>0) 或者 不优化 (0).

函数 cvUseOptimized 在两个模式之间切换,这里只有纯 C 才从 cxcore, OpenCV 等执行。如果可用 IPP 和 MKL 函数也可使用。 当 cvUseOptimized(0) 被调用, 所有的优化库都不被载入。该函数在调试模式下是很有用的, IPP&MKL 不工作, 在线跨速比较等。它返回载入的优化函数的数目。注意，缺省地优化插件是被载入的，因此在程序开始调用 cvUseOptimized(1) 是没有必要的(事实上, 它只会增加启动时间)

[编辑]

SetMemoryManager

分配自定义/缺省内存管理函数

```
typedef void* (CV_CDECL *CvAllocFunc)(size_t size, void* userdata);
typedef int (CV_CDECL *CvFreeFunc)(void* pptr, void* userdata);

void cvSetMemoryManager( CvAllocFunc alloc_func=NULL,
                         CvFreeFunc free_func=NULL,
                         void* userdata=NULL );
```

alloc_func
分配函数; 除了 userdata 可能用来确定上下文关系外，接口和 malloc 相似

free_func
释放函数; 接口和 free 相似

userdata
透明的传给自定义函数的用户数据

函数 cvSetMemoryManager 设置将被 cvAlloc，cvFree 和高级函数 (例如. cvCreateImage) 调用的用户自定义内存管理函数(代替 malloc 和 free)。 注意, 当用 cvAlloc 分配数据的时候该函数被调用。 当然, 为了避免无限递归调用， 它不允许从自定义分配/释放函数调用 cvAlloc 和 cvFree 。

如果 alloc_func 和 free_func 指针是 NULL, 恢复缺省的内存管理函数。

[编辑]

SetIPLAllocators

切换图像 IPL 函数的分配/释放

```
typedef IplImage* (CV_STDCALL* Cv_iplCreateImageHeader)
                            (int,int,int,char*,char*,int,int,int,int,int,
                            IplROI*,IplImage*,void*,IplTileInfo*);
typedef void (CV_STDCALL* Cv_iplAllocateImageData)(IplImage*,int,int);
```

```
typedef void (CV_STDCALL* Cv_iplDeallocate)(IplImage*,int);
typedef IplROI* (CV_STDCALL* Cv_iplCreateROI)(int,int,int,int,int);
typedef IplImage* (CV_STDCALL* Cv_iplCloneImage)(const IplImage*);

void cvSetIPLAllocators( Cv_iplCreateImageHeader create_header,
                         Cv_iplAllocateImageData allocate_data,
                         Cv_iplDeallocate deallocate,
                         Cv_iplCreateROI create_roi,
                         Cv_iplCloneImage clone_image );

#define CV_TURN_ON_IPL_COMPATIBILITY()                                      \
    cvSetIPLAllocators( iplCreateImageHeader, iplAllocateImage,         \
                        iplDeallocate, iplCreateROI, iplCloneImage )
```

create_header
: 指向 iplCreateImageHeader 的指针

allocate_data
: 指向 iplAllocateImage 的指针

deallocate
: 指向 iplDeallocate 的指针

create_roi
: 指向 iplCreateROI 的指针

clone_image
: 指向 iplCloneImage 的指针

函数 cvSetIPLAllocators 使用 CXCORE 来进行图像 IPL 函数的 分配/释放 操作。 为了方便, 这里提供了环绕宏 CV_TURN_ON_IPL_COMPATIBILITY。 当 IPL 和 CXCORE/OpenCV 同时使用以及调用 iplCreateImageHeader 等情况该函数很有用。如果 IPL 仅仅是被调用来进行数据处理，该函数就必要了，因为所有的分配/释放都由 CXCORE 来完成, 或者所有的分配/释放都由 IPL 和一些 OpenCV 函数来处理数据。

机器学习中文参考手册

Wikipedia，自由的百科全书

本文翻译尚未完成，请您将英文部分翻译为中文。

目录

[隐藏]

[编辑]

简介：通用类和函数

机器学习库（MLL）是一些用于分类、回归和数据聚类的类和函数。

大部分分类和回归算法是用C++类来实现。尽管这些算法有一些不同的特性（像处理missing measurements的能力，或者categorical input variables等），这些类之间有一些相同之处。这些相同之处在类 CvStatModel 中被定义，其他 ML 类都是从这个类中继承。

[编辑]

CvStatModel

ML库中的统计模型基类。

```
class CvStatModel
{
public:
    /* CvStatModel(); */
    /* CvStatModel( const CvMat* train_data ... ); */

    virtual ~CvStatModel();

    virtual void clear()=0;

    /* virtual bool train( const CvMat* train_data, [int tflag,] ..., const CvMat* responses,
...,
    [const CvMat* var_idx,] ..., [const CvMat* sample_idx,] ...
    [const CvMat* var_type,] ..., [const CvMat* missing_mask,] <misc_training_alg_params> ... )=0;
     */

    /* virtual float predict( const CvMat* sample ... ) const=0; */

    virtual void save( const char* filename, const char* name=0 )=0;
    virtual void load( const char* filename, const char* name=0 )=0;

    virtual void write( CvFileStorage* storage, const char* name )=0;
    virtual void read( CvFileStorage* storage, CvFileNode* node )=0;
};
```

在上面的声明中，一些函数被注释掉。实际上，一些函数没有一个单一的API（缺省的构造函数除外），然而，在本节后面描述的语法和定义方面有一些相似之处，好像他们是基类的一部分一样。

注意：opencv 1.0 版本对 CvStatModel 类做了修改，类的声明如下。

```
class CV_EXPORTS CvStatModel
{
```

```
public:
    CvStatModel();
    virtual ~CvStatModel();

    virtual void clear();

    virtual void save( const char* filename, const char* name=0 );
    virtual void load( const char* filename, const char* name=0 );

    virtual void write( CvFileStorage* storage, const char* name );
    virtual void read( CvFileStorage* storage, CvFileNode* node );

protected:
    const char* default_model_name;
};
```

[编辑]

CvStatModel::CvStatModel

缺省构造函数

```
CvStatModel::CvStatModel();
```

ML中的每个统计模型都有一个无参数构造函数。这个构造函数在"两步法"构造时非常有用，先调用这个缺省构造函数,紧接着调用 tranin() 或者load() 函数.（This constructor is useful for 2-stage model construction, when the default constructor is followed by train() or load().）

[编辑]

CvStatModel::CvStatModel(...)

训练构造函数

```
CvStatModel::CvStatModel( const CvMat* train_data ... ); */
```

大多数 ML 类都提供一个单步创建+训练的构造函数。此构造函数等价于缺省构造函数，加上一个紧接的 train() 方法调用，所传入的参数即为调用的参数。

[编辑]

CvStatModel::~CvStatModel

虚拟析构函数（Virtual destructor）

```
CvStatModel::~CvStatModel();
```

基类析构被声明为虚方法，因此你可以安全地写出下面的代码:

```
CvStatModel* model;
if( use_svm )
    model = new CvSVM(... /* SVM params */);
else
    model = new CvDTree(... /* Decision tree params */);
...
delete model;
```

一般，每个继承类的析构器不用做任何操作，但是如果调用了重载的clear()方法，将释放全部内存资源。

[编辑]

CvStatModel::clear

释放内存，重置模型状态

```
void CvStatModel::clear();
```

clear方法和析构函数发生的行为相似，比如：clear方法释放类成员所占用的内存空间。然而，和析构函数不同的是，clear方法不析构对象 自身，也即调用clear方法后，对象本身在将来仍然可以使用。一般情况下，析构器、load方法、read方法、派生类train成员调用clear方 法释放内存空间，甚至是用户也可以进行明确的调用。

[编辑]

CvStatModel::save

将模型保存到文件

```
void CvStatModel::save( const char* filename, const char* name=0 );
```

save方法将整个模型状态以指定名称或默认名称（取决于特定的类）保存到指定的XML或YAML文件中。该方法使用的是cxcore中的数据保存功能。

[编辑]

CvStatModel::load

从文件中装载模型

```
void CvStatModel::load( const char* filename, const char* name=0 );
```

load方法从指定的XML或YAML文件中装载指定名称（或默认的与模型相关名称）的整个模型状态。之前的模型状态将被clear()清零。

请注意，这个方法是虚的，因此任何模型都可以用这个虚方法来加载。然而，不像OpenCV中的C类型可以用通用函数cvLoad()来加载，这里模型类型无论如何都要是已知的，因为一个空模型作为恰当类的一种，必须被预先建构。这个限制将会在未来的ML版本中移除。

[编辑]

CvStatModel::write

将模型写入文件存储

```
void CvStatModel::write( CvFileStorage* storage, const char* name );
```

write方法将整个模型状态用指定的或默认的名称（取决于特定的类）写到文件存储中去。这个方法被save()调用。

[编辑]

CvStatModel::read

从文件存储中读出模型

```
void CvStatMode::read( CvFileStorage* storage, CvFileNode* node );
```

read方法从文件存储中的指定节点中读取整个模型状态。这个节点必须由用户来定位，如使用cvGetFileNodeByName()函数。这个方法被load()调用。

之前的模型状态被clear()清零。

[编辑]

CvStatModel::train

训练模型

```
bool CvStatMode::train( const CvMat* train_data, [int tflag,] ..., const CvMat* responses, ...,
    [const CvMat* var_idx,] ..., [const CvMat* sample_idx,] ...
    [const CvMat* var_type,] ..., [const CvMat* missing_mask,] <misc_training_alg_params> ... );
```

这个函数利用输入的特征向量和对应的响应值(responses)来训练统计模型。特征向量和其对应的响应值都是用矩阵来表示。缺省情况下，特征向 量都以行向量被保存在train_data中，也就是所有的特征向量元素都是连续存储。不过，一些算法可以处理转置表示，即特征向量用列向量来表示，所有 特征向量的相同位置的元素连续存储。如果两种排布方式都支持，这个函数的参数tflag可以使用下面的取值：

tflag=CV_ROW_SAMPLE
　　表示特征向量以行向量存储；
tflag=CV_COL_SAMPLE
　　表示特征向量以列向量存储；

训练数据必须是32fC1（32位的浮点数，单通道）格式 响应值通常是以向量方式存储（一个行，或者一个列向量），存储格式为32sC1（仅在分类问题中）或者32fC1格式，每个输入特征向量对应一个值（虽然一些算法，比如某几种神经网络，响应值为向量）。

对于分类问题，响应值是离散的类别标签；对于回归问题，响应值是被估计函数的输出值。一些算法只能处理分类问题，一些只能处理回归问题，另 一些算法这两类问题都能处理。In the latter case the type of output variable is either passed as separate parameter, or as a last element of var_type vector:
CV_VAR_CATEGORICAL means that the output values are discrete class labels,
CV_VAR_ORDERED(=CV_VAR_NUMERICAL) means that the output values are ordered, i.e. 2 different values can be compared as numbers, and this is a regression problem The types of input variables can be also specified using var_type. Most algorithms can handle only ordered input variables.

ML中的很多模型也可以仅仅使用选择特征的子集，或者（并且）使用选择样本的子集来训练。为了让用户易于使用，train函数通常包含 var_idx和sample_idx参数。var_idx指定感兴趣的特征，sample_idx指定感兴趣的样本。这两个向量可以是整数 (32sC1)向量，例如以0为开始的索引，或者8位(8uC1)的使用的特征或者样本的掩码。用户也可以传入NULL指针，用来表示训练中使用所有变量 ／样本。

除此之外，一些算法支持数据缺失情况，也就是某个训练样本的某个特征值未知（例如，他们忘记了在周一测量病人A的温度）。参数missing_mask，一个8位的同train_data同样大小的矩阵掩码，用来指示缺失的数据（掩码中的非零值）。

通常来说，在执行训练操作前，可以用clear()函数清除掉早先训练的模型状态。然而，一些函数可以选择用新的数据更新模型，而不是将模型重置，一切从头再来。

[编辑]

CvStatModel::predict

预测样本的response

```
float CvStatMode::predict( const CvMat* sample[, <prediction_params>] ) const;
```

这个函数用来预测一个新样本的响应值(response)。在分类问题中，这个函数返回类别编号；在回归问题中，返回函数值。输入的样本必须与传给 train_data的训练样本同样大小。如果训练中使用了var_idx参数，一定记住在predict函数中使用跟训练特征一致的特征。

后缀const是说预测不会影响模型的内部状态，所以这个函数可以很安全地从不同的线程调用。

[编辑]

Normal Bayes 分类器

这个简单的分类器模型是建立在每一个类别的特征向量服从正态分布的基础上的（尽管，不必是独立的），因此，整个分布函数被假设为一个高斯分布，每一类别一组系数。当给定了训练数据，算法将会估计每一个类别的向量均值和方差矩阵，然后根据这些进行预测。 [Fukunaga90] K. Fukunaga. Introduction to Statistical Pattern Recognition. second ed., New York: Academic Press, 1990.

注：OpenCV 1.0rc1(0.9.9)版本的贝叶斯分类器有个小bug， 训练数据时候会提示错误

```
OpenCV ERROR: Formats of input arguments do not match ()
        in function cvSVD, cxsvd.cpp(1243)
```

修改方法为将文件 ml/src/mlnbayes.cpp 中的193行：

```
CV_CALL( cov = cvCreateMat( _var_count, _var_count, CV_32FC1 ));
```

改为

```
CV_CALL( cov = cvCreateMat( _var_count, _var_count, CV_64FC1 ));
```

此问题在OpenCV 1.0.0中已经得到修正。

[编辑]

CvNormalBayesClassifier

对正态分布的数据的贝叶斯分类器

```
class CvNormalBayesClassifier : public CvStatModel
{
public:
    CvNormalBayesClassifier();
    virtual ~CvNormalBayesClassifier();

    CvNormalBayesClassifier( const CvMat* _train_data, const CvMat* _responses,
        const CvMat* _var_idx=0, const CvMat* _sample_idx=0 );

    virtual bool train( const CvMat* _train_data, const CvMat* _responses,
        const CvMat* _var_idx = 0, const CvMat* _sample_idx=0, bool update=false );

    virtual float predict( const CvMat* _samples, CvMat* results=0 ) const;
    virtual void clear();

    virtual void save( const char* filename, const char* name=0 );
    virtual void load( const char* filename, const char* name=0 );

    virtual void write( CvFileStorage* storage, const char* name );
    virtual void read( CvFileStorage* storage, CvFileNode* node );
protected:
    ...
};
```

[编辑]

CvNormalBayesClassifier::train

训练这个模型

```
bool CvNormalBayesClassifier::train( const CvMat* _train_data, const CvMat* _responses,
                const CvMat* _var_idx = 0, const CvMat* _sample_idx=0, bool update=false );
```

这个函数训练正态贝叶斯分类器。并且遵循通常训练“函数”的以下一些限制：只支持CV_ROW_SAMPLE类型的数据，输入的变量全部应该是有序 的，输出的变量是一个分类结果。（例如，_responses中的元素必须是整数，因此向量的类型有可能是32fC1类型的），不支持missing, measurements。

另外，有一个update标志，标志着模型是否使用新数据升级。 In addition, there is update flag that identifies, whether the model should be trained from scratch (update=false) or be updated using the new training data (update=true).

[编辑]

CvNormalBayesClassifier::predict

对未知的样本或或本集进行预测

```
float CvNormalBayesClassifier::predict( const CvMat* samples, CvMat* results=0 ) const;
```

这个函数估计输入向量的最有可能的类别。输入向量（一个或多个）被储存在矩阵的每一行中。对于多个输入向量，则输出会是一个向量结果。对于单一的输入，函数本身的返回值就是预测结果。

[编辑]

K近邻算法

这个算法首先贮藏所有的训练样本，然后通过分析（包括选举，计算加权和等方式）一个新样本周围K个最近邻以给出该样本的相应值。这种方法有时候被称作“基于样本的学习”，即为了预测，我们对于给定的输入搜索最近的已知其相应的特征向量。

[编辑]

CvKNearest

K近邻类

```
class CvKNearest : public CvStatModel //继承自ML库中的统计模型基类
{
public:

    CvKNearest();
    virtual ~CvKNearest();  //虚函数定义

    CvKNearest( const CvMat* _train_data, const CvMat* _responses,
                const CvMat* _sample_idx=0, bool _is_regression=false, int max_k=32 );

    virtual bool train( const CvMat* _train_data, const CvMat* _responses,
                        const CvMat* _sample_idx=0, bool is_regression=false,
                        int _max_k=32, bool _update_base=false );

    virtual float find_nearest( const CvMat* _samples, int k, CvMat* results,
        const float** neighbors=0, CvMat* neighbor_responses=0, CvMat* dist=0 ) const;

    virtual void clear();
    int get_max_k() const;
    int get_var_count() const;
    int get_sample_count() const;
    bool is_regression() const;

protected:
    ...
};
```

[编辑]

CvKNearest::train

训练KNN模型

```
bool CvKNearest::train( const CvMat* _train_data, const CvMat* _responses,
```

```
        const CvMat* _sample_idx=0, bool is_regression=false,
        int _max_k=32, bool _update_base=false );
```

这个类的方法训练K近邻模型。它遵循一个一般训练方法约定的限制：只支持CV_ROW_SAMPLE数据格式，输入向量必须都是有序的，而输出可以 是 无序的(当is_regression=false)，可以是有序的（is_regression=true）。并且变量子集和省略度量是不被支持的。

参数_max_k 指定了最大邻居的个数，它将被传给方法find_nearest。 参数 _update_base 指定模型是由原来的数据训练(_update_base=false)，还是被新训练数据更新后再训练(_update_base=true）。在后一种情况下_max_k 不能大于原值, 否则它会被忽略.

[编辑]

CvKNearest::find_nearest

寻找输入向量的最近邻

```
float CvKNearest::find_nearest( const CvMat* _samples, int k, CvMat* results=0,
        const float** neighbors=0, CvMat* neighbor_responses=0, CvMat* dist=0 ) const;
```

对每个输入向量（表示为matrix_sample的每一行），该方法找到k（k≤get_max_k() ）个最近邻。在回归中，预测结果将是指定向量的近邻的响应的均值。在分类中，类别将由投票决定。

对传统分类和回归预测来说，该方法可以有选择的返回近邻向量本身的指针(neighbors, array of k*_samples->rows pointers)，它们相对应的输出值(neighbor_responses, a vector of k*_samples->rows elements) ，和输入向量与近邻之间的距离(dist, also a vector of k*_samples->rows elements)。

对每个输入向量来说，近邻将按照它们到该向量的距离排序。

对单个输入向量，所有的输出矩阵是可选的，而且预测值将由该方法返回。In case of a single input vector all the output matrices are optional and the predicted value is returned by the method.

[编辑]

例程：使用**kNN**进行**2**维样本集的分类，样本集的分布为混合高斯分布

```
#include "ml.h"
#include "highgui.h"

int main( int argc, char** argv )
{
    const int K = 10;
    int i, j, k, accuracy;
    float response;
    int train_sample_count = 100;
    CvRNG rng_state = cvRNG(-1);
    CvMat* trainData = cvCreateMat( train_sample_count, 2, CV_32FC1 );
    CvMat* trainClasses = cvCreateMat( train_sample_count, 1, CV_32FC1 );
    IplImage* img = cvCreateImage( cvSize( 500, 500 ), 8, 3 );
    float _sample[2];
    CvMat sample = cvMat( 1, 2, CV_32FC1, _sample );
    cvZero( img );

    CvMat trainData1, trainData2, trainClasses1, trainClasses2;

    // form the training samples
    cvGetRows( trainData, &trainData1, 0, train_sample_count/2 );
    cvRandArr( &rng_state, &trainData1, CV_RAND_NORMAL, cvScalar(200,200), cvScalar(50,50) );

    cvGetRows( trainData, &trainData2, train_sample_count/2, train_sample_count );
    cvRandArr( &rng_state, &trainData2, CV_RAND_NORMAL, cvScalar(300,300), cvScalar(50,50) );

    cvGetRows( trainClasses, &trainClasses1, 0, train_sample_count/2 );
    cvSet( &trainClasses1, cvScalar(1) );

    cvGetRows( trainClasses, &trainClasses2, train_sample_count/2, train_sample_count );
    cvSet( &trainClasses2, cvScalar(2) );
```

```
    // learn classifier
    CvKNearest knn( trainData, trainClasses, 0, false, K );
    CvMat* nearests = cvCreateMat( 1, K, CV_32FC1);

    for( i = 0; i < img->height; i++ )
    {
        for( j = 0; j < img->width; j++ )
        {
            sample.data.fl[0] = (float)j;
            sample.data.fl[1] = (float)i;

            // estimates the response and get the neighbors' labels
            response = knn.find_nearest(&sample,K,0,0,nearests,0);

            // compute the number of neighbors representing the majority
            for( k = 0, accuracy = 0; k < K; k++ )
            {
                if( nearests->data.fl[k] == response)
                    accuracy++;
            }
            // highlight the pixel depending on the accuracy (or confidence)
            cvSet2D( img, i, j, response == 1 ?
                (accuracy > 5 ? CV_RGB(180,0,0) : CV_RGB(180,120,0)) :
                (accuracy > 5 ? CV_RGB(0,180,0) : CV_RGB(120,120,0)) );
        }
    }

    // display the original training samples
    for( i = 0; i < train_sample_count/2; i++ )
    {
        CvPoint pt;
        pt.x = cvRound(trainData1.data.fl[i*2]);
        pt.y = cvRound(trainData1.data.fl[i*2+1]);
        cvCircle( img, pt, 2, CV_RGB(255,0,0), CV_FILLED );
        pt.x = cvRound(trainData2.data.fl[i*2]);
        pt.y = cvRound(trainData2.data.fl[i*2+1]);
        cvCircle( img, pt, 2, CV_RGB(0,255,0), CV_FILLED );
    }

    cvNamedWindow( "classifier result", 1 );
    cvShowImage( "classifier result", img );
    cvWaitKey(0);

    cvReleaseMat( &trainClasses );
    cvReleaseMat( &trainData );
    return 0;
}
```

[编辑]

支持向量机部分

支持向量机（SVM），起初由vapnik提出时，是作为寻求最优（在一定程度上）二分类器的一种技术。后来它又被拓展到回归和聚类应用。SVM是 一种基于核函数的方法，它通过某些核函数把特征向量映射到高维空间，然后建立一个线性判别函数（或者说是一个高维空间中的能够区分训练数据的最优超平面， 参考异或那个经典例子）。假如SVM没有明确定义核函数，高维空间中任意两点距离就需要定义。

解是最优的在某种意义上是两类中距离分割面最近的特征向量和分割面的距离最大化。离分割面最近的特征向量被称为"支持向量"，意即其它向量不影响分割面（决策函数）。

有很多关于SVM的参考文献，这是两篇较好的入门文献。

【Burges98】 C. Burges. "A tutorial on support vector machines for pattern recognition", Knowledge Discovery and Data Mining 2(2), 1998. (available online at [1]).

LIBSVM - A Library for Support Vector Machines. By Chih-Chung Chang and Chih-Jen Lin ([2])

[编辑]

CvSVM

支持矢量机

```
class CvSVM : public CvStatModel //继承自基类CvStatModel
{
public:
    // SVM type
    enum { C_SVC=100, NU_SVC=101, ONE_CLASS=102, EPS_SVR=103, NU_SVR=104 };//SVC是SVM分类
器，SVR是SVM回归

    // SVM kernel type
    enum { LINEAR=0, POLY=1, RBF=2, SIGMOID=3 }; //提供四种核函数，分别是线性，多项式，径向基，sigmoid型函
数。

    CvSVM();
    virtual ~CvSVM();

    CvSVM( const CvMat* _train_data, const CvMat* _responses,
           const CvMat* _var_idx=0, const CvMat* _sample_idx=0,
           CvSVMParams _params=CvSVMParams() );

    virtual bool train( const CvMat* _train_data, const CvMat* _responses,
                        const CvMat* _var_idx=0, const CvMat* _sample_idx=0,
                        CvSVMParams _params=CvSVMParams() );

    virtual float predict( const CvMat* _sample ) const;
    virtual int get_support_vector_count() const;
    virtual const float* get_support_vector(int i) const;
    virtual void clear();

    virtual void save( const char* filename, const char* name=0 );
    virtual void load( const char* filename, const char* name=0 );

    virtual void write( CvFileStorage* storage, const char* name );
    virtual void read( CvFileStorage* storage, CvFileNode* node );
    int get_var_count() const { return var_idx ? var_idx->cols : var_all; }

protected:
    ...
};
```

[编辑]

CvSVMParams

SVM训练参数struct

```
struct CvSVMParams
{
    CvSVMParams();
    CvSVMParams( int _svm_type, int _kernel_type,
                 double _degree, double _gamma, double _coef0,
                 double _C, double _nu, double _p,
                 CvMat* _class_weights, CvTermCriteria _term_crit );

    int          svm_type;
    int          kernel_type;
    double       degree; // for poly
    double       gamma;  // for poly/rbf/sigmoid
    double       coef0;  // for poly/sigmoid

    double       C;  // for CV_SVM_C_SVC, CV_SVM_EPS_SVR and CV_SVM_NU_SVR
    double       nu; // for CV_SVM_NU_SVC, CV_SVM_ONE_CLASS, and CV_SVM_NU_SVR
    double       p; // for CV_SVM_EPS_SVR
    CvMat*       class_weights; // for CV_SVM_C_SVC
    CvTermCriteria term_crit; // termination criteria
};
```

svm_type，SVM的类型：

CvSVM::C_SVC - n(n>=2)分类器，允许用异常值惩罚因子C进行不完全分类。
CvSVM::NU_SVC - n类似然不完全分类的分类器。参数nu取代了c，其值在区间【0，1】中，nu越大，决策边界越平滑。
CvSVM::ONE_CLASS - 单分类器，所有的训练数据提取自同一个类里，然后SVM建立了一个分界线以分

割该类在特征空间中所占区域和其它类在特征空间中所占区域。
CvSVM::EPS_SVR - 回归。 训练集中的特征向量和拟合出来的超平面的距离需要小于p。异常值惩罚因子C被采用。
CvSVM::NU_SVR - 回归；nu 代替了p

kernel_type//核类型：

CvSVM::LINEAR - 没有任何向映射至高维空间，线性区分（或回归）在原始特征空间中被完成，这是最快的选择。 d(x,y) = x•y == (x,y)
CvSVM::POLY - 多项式核: d(x,y) = (gamma*(x•y)+coef0)degree
CvSVM::RBF - 径向基，对于大多数情况都是一个较好的选择：d(x,y) = exp(-gamma*|x-y|2)
CvSVM::SIGMOID - sigmoid函数被用作核函数: d(x,y) = tanh(gamma*(x•y)+coef0)

degree, gamma, coef0：都是核函数的参数，具体的参见上面的核函数的方程。

C, nu, p：在一般的SVM优化求解时的参数。

class_weights:可选权重，赋给指定的类别。一般乘以C以后去影响不同类别的错误分类惩罚项。权重越大，某一类别的误分类数据的惩罚项就越大。

term_crit:SVM的迭代训练过程的中止。（解决了部分受约束二次最优问题）

该结构需要初始化，并传递给CvSVM的训练函数。

[编辑]

CvSVM::train

训练SVM

```
bool CvSVM::train( const CvMat* _train_data, const CvMat* _responses,
                   const CvMat* _var_idx=0, const CvMat* _sample_idx=0,
                   CvSVMParams _params=CvSVMParams() );
```

The method trains SVM model. It follows the conventions of generic train "method" with the following limitations: only CV_ROW_SAMPLE data layout is supported, the input variables are all ordered, the output variables can be either categorical (_params.svm_type=CvSVM::C_SVC or _params.svm_type=CvSVM::NU_SVC) or ordered (_params.svm_type=CvSVM::EPS_SVR or _params.svm_type=CvSVM::NU_SVR) or not required at all (_params.svm_type=CvSVM::ONE_CLASS), missing measurements are not supported.

所有的参数都被集成在CvSVMParams这个结构中。

[编辑]

CvSVM::get_support_vector*

得到支持矢量和特殊矢量的数

```
int CvSVM::get_support_vector_count() const;
const float* CvSVM::get_support_vector(int i) const;
```

这个方法可以被用来得到支持矢量的集合。

[编辑]

补充:在WindowsXP+OpenCVRC1平台下整合OpenCV与libSVM

虽然从RC1版开始opencv开始增设ML类，提供对常见的分类器和回归算法的支持。但是尚存在一些问题，比如说例子少（官方许诺说很快会提供一 批新例子,见CVS版）。单说SVM这种算法，它自己提供了一套比较完备的函数，但是并不见得优于老牌的libsvm（它也应该参考过libsvm，至于 是否效率优于libsvm，我并没有作过测试，官方也没有什么说法，但是libsvm持续开源更新，是公认的现存的开源SVM库中最易上手，性能最好的库）。所以在你的程序里整合opencv和libSVM还是一种比较好的解决方案。在VC中整合有些小地方需要注意，这篇文档主要是提供把图象作为SVM 输入时程序遇到的这些小问题的解决方案。希望大家遇到问题时，多去读libSVM的源码，它本身也是开源的，C代码写得也很优秀，数据结构定义得也比较 好。

首先是SVM的训练，这部分我并没有整合到VC里，直接使用它提供的python程序，虽然网格搜索这种简易搜索也可以自己写，但是识别时 只需要训练生成的SVMmodel文件即可，所以可以和主程序分离开。至于python在windows下的使用，还是要设置一下的，首先要在系统环境变 量path里把python的路径设进去，libsvm画训练等高线图还需要gnuplot的支持，打开python源程序(grid.py)，把 gnuplot_exe设置成你机器里的安装路径，这样才能正确的运行程序。然后就是改步长和搜索范围，官方建议是先用大步长搜索，搜到最优值后再用小步 长在小范围内搜索（我觉得这有可能会陷入局部最优，不过近似出的结果还可以接受）。我用的python版本是2.4，gnuplot4.0。

[编辑]

常用**libSVM**资料链接

官方站点，有一些tutorial和测试数据

哈工大的机器学习论坛，非常好（外网似乎不能登录）

上交的一个研究生还写过libsvm2.6版的代码中文注释，源链接找不着了，大家自己搜搜吧，写得很好，上海交通大学模式分析与机器智能实验室。 http://www.pami.sjtu.edu.cn/people/gpliu/

[编辑]

决策树

本节所讨论的 ML 类(s)实现了 [Brieman84] 中描述的分类与回归树算法。

类CvDTree可以表示一个单独使用的简单决策树，也可以表示树集成分类器中的一个基础分类器（参见Boosting和Random Trees）

决策树是一个二叉树（即树的非叶节点仅有两个子节点）。当每个叶节点用类别标识（多个叶子可能有相同的标识）时，它可以表示分类树；当每个叶节点被分配了一个常量（所以回归函数是分段常量）时，决策树就成了回归树。

用决策树进行预测

预测算法从根结点开始，到达某个叶结点，然后得到输入特征向量的响应。在每一个非叶子结点，算法会根据变量值选择向左走或者向右走（比如选 择左子结点作为下一个观测结点），该变量的索引值储存在被观测结点中。这个变量可以是数值的或者类型的。如果变量是数值的，那么变量值就跟一个固定的阈值 （也储存在被观测结点中）来比较，如果该变量小于阈值，那么算法就往左走，否则就往右（比如说，如果重量小于1kg，那么算法流程就往左走，否则就往 右）。如果变量值是categorical的，那么这个离散变量值会被测试是否属于某个特定的子集（也储存在该结点中），这个子集取自于一个该变量可以取 到的有限集合。如果变量属于该子集，那么算法流程就往左走，否则就往右（比如，如果颜色是绿色的或者红色的，就往左，否则就往右）。也就是说，在每个结 点，使用一对实体对象(<variable_index>, <decision_rule (threshold/subset)>) 。这叫做分裂点（在变量#<variable_index>分裂）。如果到达了某一个叶子结点，赋给该结点的值就作为预测算法的输出值。

To reach a leaf node, and thus to obtain a response for the input feature vector, the prediction procedure

starts with the root node. From each non-leaf node the procedure goes to the left (i.e. selects the left child node as the next observed node), or to the right based on the value of a certain variable, which index is stored in the observed node. The variable can be either ordered or categorical. In the first case, the variable value is compared with the certain threshold (which is also stored in the node); if the value is less than the threshold, the procedure goes to the left, otherwise, to the right (for example, if the weight is less than 1 kilo, the procedure goes to the left, else to the right). And in the second case the discrete variable value is tested, whether it belongs to a certain subset of values (also stored in the node) from a limited set of values the variable could take; if yes, the procedure goes to the left, else - to the right (for example, if the color is green or red, go to the left, else to the right). That is, in each node, a pair of entities (<variable_index>, <decision_rule (threshold/subset)>) is used. This pair is called split (split on the variable #<variable_index>). Once a leaf node is reached, the value assigned to this node is used as the output of prediction procedure.

有时候，输入向量的某些特征缺失（比如说，在黑暗中很难去确定对象的颜色），而且预测算法可能在某一个结点上（在上面提到结点用色彩划分的 例子里）反复运算。决策树用替代分裂点（surrogate splits）来避免这样的情况发生。这就是说，除了最佳初始分裂点（the best "primary" split）以外，每一个树结点可能都要被一个或多个几乎有一样的结果的变量分裂。

Sometimes, certain features of the input vector are missed (for example, in the darkness it is difficult to determine the object color), and the prediction procedure may get stuck in the certain node (in the mentioned example if the node is split by color). To avoid such situations, decision trees use so-called surrogate splits. That is, in addition to the best "primary" split, every tree node may also be split on one or more other variables with nearly the same results.

Training Decision Trees

训练决策树

决策树是从根结点递归构造的。用所有的训练数据（特征向量和对应的响应）来在根结点处进行分裂。在每个结点处，优化准则（比如最优分裂）是 基于一些基本原则来确定的（比如ML中的“纯度purity”原则被用来进行分类，方差之和用来进行回归）。Then, if necessary, the surrogate splits are found that resemble at the most the results of the primary split on the training data; 所有的数据根据初始和替代分裂点来划分给左、右孩子结点（就像在预测算法里做的一样）。然后算法回归的继续分裂左右孩子结点。在以下情况下算法可能会在某 个结点停止（i.e. stop splitting the node further）：

- 树的深度达到了指定的最大值
- 在该结点训练样本的数目少于指定值，比如，没有统计意义上的集合来进一步进行结点分裂了。
- 在该结点所有的样本属于同一类（或者，如果是回归的话，变化已经非常小了）
- 跟随机选择相比，能选择到的最好的分裂已经基本没有什么有意义的改进了。

The tree is built recursively, starting from the root node. The whole training data (feature vectors and the responses) are used to split the root node. In each node the optimum decision rule (i.e. the best "primary" split) is found based on some criteria (in ML gini "purity" criteria is used for classification, and sum of squared errors is used for regression). Then, if necessary, the surrogate splits are found that resemble at the most the results of the primary split on the training data; all data are divided using the primary and the surrogate splits (just like it is done in the prediction procedure) between the left and the right child node. Then the procedure recursively splits both left and right nodes etc. At each node the recursive procedure may stop (i.e. stop splitting the node further) in one of the following cases:

- depth of the tree branch being constructed has reached the specified maximum value.
- number of training samples in the node is less than the specified threshold, i.e. it is not statistically representative set to split the node further.
- all the samples in the node belong to the same class (or, in case of regression, the variation is too small).
- the best split found does not give any noticeable improvement comparing to just a random choice.

当树建好之后，如果需要的话，可能需要用交叉验证来进行剪枝。这就是说，树的某些导致模型过拟合的分支将被剪掉。一般情况下，这个过程只用于单决策树上，因为tree ensembles通常会建立一些小的足够的树并且用他们自身的保护机制来防止过拟合。

When the tree is built, it may be pruned using cross-validation procedure, if need. That is, some branches of the tree that may lead to the model overfitting are cut off. Normally, this procedure is only applied to standalone decision trees, while tree ensembles usually build small enough trees and use their own protection schemes against overfitting.

“变量的重要性” **Variable importance** 决策树除了它的重要用途——预测以外，还可以用在多变量分析上。构造好的决策树算法的一个关键特性就是它可能被用在计算每个变量的重要性（相关决策力） 上。举个例子，在一个垃圾邮件过滤器中，用一个经常出现在邮件中的词汇集合来作为特征向量，那么变量的重要率就可以用来决定最“垃圾邮件指示词”，这样就 可以保证词汇集合的大小的合理性。

Besides the obvious use of decision trees - prediction, the tree can be also used for various data analysis. One of the key properties of the constructed decision tree algorithms is that it is possible to compute importance (relative decisive power) of each variable. For example, in a spam filter that uses a set of words occurred in the message as a feature vector, the variable importance rating can be used to determine the most "spam-indicating" words and thus help to keep the dictionary size reasonable.

每个变量的重要性的计算是在所有的在这个变量上的分裂进行的，不管是初始的还是替代的。这样的话，要准确计算变量重要性，即使没有缺失数据，替代分裂也必须包含在训练参数中。

Importance of each variable is computed over all the splits on this variable in the tree, primary and surrogate ones. Thus, to compute variable importance correctly, the surrogate splits must be enabled in the training parameters, even if there is no missing data.

[Brieman84] Breiman, L., Friedman, J. Olshen, R. and Stone, C. (1984), "Classification and Regression Trees", Wadsworth.

[编辑]

CvDTreeSplit

Decision tree node split

```
struct CvDTreeSplit
{
    int var_idx;
    int inversed;
    float quality;
    CvDTreeSplit* next;
    union
    {
        int subset[2];
        struct
        {
            float c;
            int split_point;
        }
        ord;
    };
};
```

var_idx
: 分裂中所用到的变量的索引

inversed
: 当等于1的时候，采用inverse split rule（比如，左分支和右分支在表达式下被交换）

quality
: 分裂值，正数。用来选择最佳初始分裂点，然后用来对替代分裂点进行选择和排序。构造好树之后，还用来计算向量重要性

next

指向结点分裂表中的下一个分裂点。

subset
二值集合，用在在类别向量的分裂上。规则如下：如果var_value在subset里，那么next_node<-left，否则next_node<-right。

c
用在数值变量的分裂上的阈值。规则如下：如果var_value<c，那么next_node<-left，否则next_node<-right。

split_point
用在训练算法内部。

var_idx
Index of the variable used in the split

inversed
When it equals to 1, the inverse split rule is used (i.e. left and right branches are exchanged in the expressions below)

quality
The split quality, a positive number. It is used to choose the best primary split, then to choose and sort the surrogate splits. After the tree is constructed, it is also used to compute variable importance.

next
Pointer to the next split in the node split list.

subset
Bit array indicating the value subset in case of split on a categorical variable. The rule is: if var_value in subset then next_node<-left else next_node<-right

c
The threshold value in case of split on an ordered variable. The rule is: if var_value < c then next_node<-left else next_node<-right

split_point
Used internally by the training algorithm.

CvDTreeNode

Decision tree node

```
struct CvDTreeNode
{
    int class_idx;
    int Tn;
    double value;

    CvDTreeNode* parent;
    CvDTreeNode* left;
    CvDTreeNode* right;

    CvDTreeSplit* split;

    int sample_count;
    int depth;
    ...
};
```

value
赋给结点的值。可以是一个类别标签，也可以是估计函数值。

class_idx
赋给结点的归一化的类别索引(从 0到class_count-1)，用在分类树和树集成中。

Tn
在有序排列的树中的树索引。用在剪枝过程中和之后。在整棵树中根结点有最大的Tn值，子结点的Tn值小于或等于父结点，Tn≤CvDTree::pruned_tree_idx 的结点在预测阶段不予考虑(对应的分枝也被剪掉)，即使它们还没有在物理上被从树上移除。

parent, left, right
指向父结点、左右孩子结点的指针。

split
　　指向第一个分裂点（初始分裂点）的指针。
sample_count
　　在训练阶段所用到的样本数目。用来解决复杂的问题——当初始分裂点所用到的变量缺失时，而 且其他替代分裂点所用到的变量也缺失时，如果left->sample_count>right->sample_count，那么样 本直接进入左边的子结点，否则往右。
depth
　　结点深度，根结点的深度是0，子结点的深度是父结点的深度加1。

CvDTreeNode的其他数据在训练阶段使用。

value
　　The value assigned to the tree node. It is either a class label, or the estimated function value.
class_idx
　　The assigned to the node normalized class index (to 0..class_count-1 range), it is used internally in classification trees and tree ensembles.
Tn
　　The tree index in a ordered sequence of trees. The indices are used during and after the pruning procedure. The root node has the maximum value Tn of the whole tree, child nodes have Tn less than or equal to the parent's Tn, and the nodes with Tn≤CvDTree::pruned_tree_idx are not taken into consideration at the prediction stage (the corresponding branches are considered as cut-off), even if they have not been physically deleted from the tree at the pruning stage.
parent, left, right
　　Pointers to the parent node, left and right child nodes.
split
　　Pointer to the first (primary) split.
sample_count
　　The number of samples that fall into the node at the training stage. It is used to resolve the difficult cases - when the variable for the primary split is missing, and all the variables for other surrogate splits are missing too, the sample is directed to the left if left->sample_count>right->sample_count and to the right otherwise.
depth
　　The node depth, the root node depth is 0, the child nodes depth is the parent's depth + 1.

Other numerous fields of CvDTreeNode are used internally at the training stage.

CvDTreeParams

Decision tree training parameters

```
struct CvDTreeParams
{
    int max_categories;
    int max_depth;
    int min_sample_count;
    int cv_folds;
    bool use_surrogates;
    bool use_1se_rule;
    bool truncate_pruned_tree;
    float regression_accuracy;
    const float* priors;

    CvDTreeParams() : max_categories(10), max_depth(INT_MAX), min_sample_count(10),
        cv_folds(10), use_surrogates(true), use_1se_rule(true),
        truncate_pruned_tree(true), regression_accuracy(0.01f), priors(0)
    {}

    CvDTreeParams( int _max_depth, int _min_sample_count,
                   float _regression_accuracy, bool _use_surrogates,
                   int _max_categories, int _cv_folds,
                   bool _use_1se_rule, bool _truncate_pruned_tree,
```

```
                    const float* _priors );
};
```

max_depth
: 该参数指定树的最大可能深度。当某结点的深度小于max_depth时，训练算法将继续试图分裂该结点。如果满足了其他的终止准则，那么实际深度可能比max_depth小（参见本部门的训练过程），而且可能会被剪枝。

min_sample_count
: 如果跟某结点相关的样本数量小于该参数值那么该结点就不被分裂。

regression_accuracy
: 另一个终止准则——只用于回归树。一旦估计结点值与训练样本的响应的差小于该参数值，该结点就不再分裂。

use_surrogates
: 如果该参数为真，替代结点就被建立。替代结点用在解决缺失数据和变量重要性估计。

max_categories
: 在训练过程试图进行分裂时，如果一个离散变量比max_categories大，最佳 精度子集的估计就需要花很长时间（比如算法是指数型的）。作为替代，许多决策树引擎（包括ML）试图通过把所有样本间聚成max_categories个 类来找到最优子分裂点（比如，把一些类别聚到一起）。

max_depth
: This parameter specifies the maximum possible depth of the tree. That is the training algorithms attempts to split a node while its depth is less than max_depth. The actual depth may be smaller if the other termination criteria are met (see the outline of the training procedure in the beginning of the section), and/or if the tree is pruned.

min_sample_count
: A node is not split if the number of samples directed to the node is less than the parameter value.

regression_accuracy
: Another stop criteria - only for regression trees. As soon as the estimated node value differs from the node training samples responses by less than the parameter value, the node is not split further.

use_surrogates
: If true, surrogate splits are built. Surrogate splits are needed to handle missing measurements and for variable importance estimation.

max_categories
: If a discrete variable, on which the training procedure tries to make a split, takes more than max_categories values, the precise best subset estimation may take a very long time (as the algorithm is exponential). Instead, many decision trees engines (including ML) try to find sub-optimal split in this case by clustering all the samples into max_categories clusters (i.e. some categories are merged together).

要注意到，这个技术只用于n（n>2）类分类问题上。在回归问题和2类分类问题上，不用聚类就可以有效地找到最优的分裂点，在这些情况下就不用这个参数。

cv_folds
: 如果该参数>1，就用cv_folds——叠一交叉验证来进行剪枝。

use_1se_rule
: 如果为真，就用剪枝算法减去树顶端的一些部分。这会让树变得紧致，而且对训练数据的噪声更有抵抗力一些，但是会牺牲一些精度。

truncate_pruned_tree
: 如果为真，要被剪掉的结点（with Tn≤CvDTree::pruned_tree_idx）将被从树上移除。除非它们被保留，而且CvDTree::pruned_tree_idx（比如设该值为-1）递减，那么仍有可能从最初的没有被剪枝的树中获得结果。

priors
: 类别先验概率的array，按类别标签值排序。该参数可以用在针对某个特定的类别而对决策树进行剪枝时。比如，如果用户希望探测到某个比较少见的异常变化，但训练可能包含了比异常多得多的正常情况，那么很可能分类结果就是认为每一个情况都是正常的。为了 避免这一点，先验就必须被指定，异常情况发生的概率需要人为的增加（增加到0.5甚至更高），这样误分类的异常情况的权重就会变大，树也能够得到适当的调 整。

Note that this technique is used only in N(>2)-class classification problems. In case of regression and 2-class classification the optimal split can be found efficiently without employing clustering, thus the parameter is not used in these cases.

cv_folds
: If this parameter is >1, the tree is pruned using cv_folds-fold cross validation.

use_1se_rule
: If true, the tree is truncated a bit more by the pruning procedure. That leads to compact, and more resistant to the training data noise, but a bit less accurate decision tree.

truncate_pruned_tree
: If true, the cut off nodes (with Tn≤CvDTree::pruned_tree_idx) are physically removed from the tree. Otherwise they are kept, and by decreasing CvDTree::pruned_tree_idx (e.g. setting it to -1) it is still possible to get the results from the original unpruned (or pruned less aggressively) tree.

priors
: The array of a priori class probabilities, sorted by the class label value. The parameter can be used to tune the decision tree preferences toward a certain class. For example, if users want to detect some rare anomaly occurrence, the training base will likely contain much more normal cases than anomalies, so a very good classification performance will be achieved just by considering every case as normal. To avoid this, the priors can be specified, where the anomaly probability is artificially increased (up to 0.5 or even greater), so the weight of the misclassified anomalies becomes much bigger, and the tree is adjusted properly.

关于内存管理：the field priors is a pointer to the array of floats。该列向量应该由用户指定分配，并且在CvDTreeParams这个结构传递给CvDTreeTrainData或者CvDTree构造函 数或者方法（就像做了一个列向量的拷贝）之后进行释放。

A note about memory management: the field priors is a pointer to the array of floats. The array should be allocated by user, and released just after the CvDTreeParams structure is passed to CvDTreeTrainData or CvDTree constructors/methods (as the methods make a copy of the array).

该结构包含了所有的决策树训练所需的参数。一个缺省的构造函数可以用缺省值来初始化所有的参数，构造一棵基本的分类树。任何参数都可以 overridden，或者结构可以用构造函数的高级变量来进行完整的初始化。
The structure contains all the decision tree training parameters. There is a default constructor that initializes all the parameters with the default values tuned for standalone classification tree. Any of the parameters can be overridden then, or the structure may be fully initialized using the advanced variant of the constructor.

CvDTreeTrainData

Decision tree training data and shared data for tree ensembles

```
struct CvDTreeTrainData
{
    CvDTreeTrainData();
    CvDTreeTrainData( const CvMat* _train_data, int _tflag,
                      const CvMat* _responses, const CvMat* _var_idx=0,
                      const CvMat* _sample_idx=0, const CvMat* _var_type=0,
                      const CvMat* _missing_mask=0,
                      const CvDTreeParams& _params=CvDTreeParams(),
                      bool _shared=false, bool _add_labels=false );
    virtual ~CvDTreeTrainData();

    virtual void set_data( const CvMat* _train_data, int _tflag,
                           const CvMat* _responses, const CvMat* _var_idx=0,
                           const CvMat* _sample_idx=0, const CvMat* _var_type=0,
                           const CvMat* _missing_mask=0,
                           const CvDTreeParams& _params=CvDTreeParams(),
                           bool _shared=false, bool _add_labels=false,
                           bool _update_data=false );

    virtual void get vectors( const CvMat*  subsample idx,
```

```
        float* values, uchar* missing, float* responses, bool get_class_idx=false );

    virtual CvDTreeNode* subsample_data( const CvMat* _subsample_idx );

    virtual void write_params( CvFileStorage* fs );
    virtual void read_params( CvFileStorage* fs, CvFileNode* node );

    // release all the data
    virtual void clear();

    int get_num_classes() const;
    int get_var_type(int vi) const;
    int get_work_var_count() const;

    virtual int* get_class_labels( CvDTreeNode* n );
    virtual float* get_ord_responses( CvDTreeNode* n );
    virtual int* get_labels( CvDTreeNode* n );
    virtual int* get_cat_var_data( CvDTreeNode* n, int vi );
    virtual CvPair32s32f* get_ord_var_data( CvDTreeNode* n, int vi );
    virtual int get_child_buf_idx( CvDTreeNode* n );

    ////////////////////////////////////

    virtual bool set_params( const CvDTreeParams& params );
    virtual CvDTreeNode* new_node( CvDTreeNode* parent, int count,
                                   int storage_idx, int offset );

    virtual CvDTreeSplit* new_split_ord( int vi, float cmp_val,
                int split_point, int inversed, float quality );
    virtual CvDTreeSplit* new_split_cat( int vi, float quality );
    virtual void free_node_data( CvDTreeNode* node );
    virtual void free_train_data();
    virtual void free_node( CvDTreeNode* node );

    int sample_count, var_all, var_count, max_c_count;
    int ord_var_count, cat_var_count;
    bool have_labels, have_priors;
    bool is_classifier;

    int buf_count, buf_size;
    bool shared;

    CvMat* cat_count;
    CvMat* cat_ofs;
    CvMat* cat_map;

    CvMat* counts;
    CvMat* buf;
    CvMat* direction;
    CvMat* split_buf;

    CvMat* var_idx;
    CvMat* var_type; // i-th element =
                     //   k<0  - ordered
                     //   k>=0 - categorical, see k-th element of cat_* arrays
    CvMat* priors;

    CvDTreeParams params;

    CvMemStorage* tree_storage;
    CvMemStorage* temp_storage;

    CvDTreeNode* data_root;

    CvSet* node_heap;
    CvSet* split_heap;
    CvSet* cv_heap;
    CvSet* nv_heap;

    CvRNG rng;
};
```

这个结构体通常有效的用在存储单树和决策树集成中。通常包含3类信息:

1. 训练参数，CvDTreeParams instance。
2. 训练数据，通常为了有效地找到最佳分裂点，需要被预处理。对树集成来说，被预处理后的数据要反复用在所有的树中。另外，训练数据特征为树集成中的所有的树所共享，并如下储存: 变量类型，类别

数，类别标签压缩图等。
3. Buffers，树结点、分裂点和其他树结构元素的内存储存

有两种使用该结构的方式。在一些简单的情况下（比如，单树，或者从ML中得到的黑盒子树集成，如随机树或者boosting），这就没有需要去注意 甚至了解这个结构——只要去构建需要的统计模型，训练并使用就行了。CvDTreeTrainData这个结构体可以在内部构建和使用。但是，对传统的树 算法或者其他一些复杂的情况来说，这个结构体就必须被精确的构建和使用，如下所示：

1. 结构体需要在set_data之后用缺省构造函数初始化（或者用完整的构造函数构建）。参数_shared设置为true。
2. 用这个数据训练一棵或者多棵树，具体见方法CvDTree::train。
3. 最后，该结构体只能在所有使用它的树被释放后释放。

This structure is mostly used internally for storing both standalone trees and tree ensembles efficiently. Basically, it contains 3 types of information

1. The training parameters, CvDTreeParams instance.
2. The training data, preprocessed in order to find the best splits more efficiently. For tree ensembles this preprocessed data is reused by all the trees. Additionally, the training data characteristics that are shared by all trees in the ensemble are stored here: variable types, the number of classes, class label compression map etc.
3. Buffers, memory storages for tree nodes, splits and other elements of the trees constructed.

There are 2 ways of using this structure. In simple cases (e.g. standalone tree, or ready-to-use "black box" tree ensemble from ML, like Random Trees or Boosting) there is no need to care or even to know about the structure - just construct the needed statistical model, train it and use it. The CvDTreeTrainData structure will be constructed and used internally. However, for custom tree algorithms, or another sophisticated cases, the structure may be constructed and used explicitly. The scheme is the following:

1. The structure is initialized using the default constructor, followed by set_data (or it is built using the full form of constructor). The parameter _shared must be set to true.
2. One or more trees are trained using this data, see the special form of the method CvDTree::train.
3. Finally, the structure can be released only after all the trees using it are released.

[编辑]

CvDTree

Decision tree

```
class CvDTree : public CvStatModel
{
public:
    CvDTree();
    virtual ~CvDTree();

    virtual bool train( const CvMat* _train_data, int _tflag,
                        const CvMat* _responses, const CvMat* _var_idx=0,
                        const CvMat* _sample_idx=0, const CvMat* _var_type=0,
                        const CvMat* _missing_mask=0,
                        CvDTreeParams params=CvDTreeParams() );

    virtual bool train( CvDTreeTrainData* _train_data, const CvMat* _subsample_idx );

    virtual CvDTreeNode* predict( const CvMat* _sample, const CvMat* _missing_data_mask=0,
                                  bool raw_mode=false ) const;
    virtual const CvMat* get_var_importance();
    virtual void clear();

    virtual void read( CvFileStorage* fs, CvFileNode* node );
    virtual void write( CvFileStorage* fs, const char* name );

    // special read & write methods for trees in the tree ensembles
    virtual void read( CvFileStorage* fs, CvFileNode* node,
```

```
                        CvDTreeTrainData* data );
    virtual void write( CvFileStorage* fs );

    const CvDTreeNode* get_root() const;
    int get_pruned_tree_idx() const;
    CvDTreeTrainData* get_data();

protected:

    virtual bool do_train( const CvMat* _subsample_idx );

    virtual void try_split_node( CvDTreeNode* n );
    virtual void split_node_data( CvDTreeNode* n );
    virtual CvDTreeSplit* find_best_split( CvDTreeNode* n );
    virtual CvDTreeSplit* find_split_ord_class( CvDTreeNode* n, int vi );
    virtual CvDTreeSplit* find_split_cat_class( CvDTreeNode* n, int vi );
    virtual CvDTreeSplit* find_split_ord_reg( CvDTreeNode* n, int vi );
    virtual CvDTreeSplit* find_split_cat_reg( CvDTreeNode* n, int vi );
    virtual CvDTreeSplit* find_surrogate_split_ord( CvDTreeNode* n, int vi );
    virtual CvDTreeSplit* find_surrogate_split_cat( CvDTreeNode* n, int vi );
    virtual double calc_node_dir( CvDTreeNode* node );
    virtual void complete_node_dir( CvDTreeNode* node );
    virtual void cluster_categories( const int* vectors, int vector_count,
        int var_count, int* sums, int k, int* cluster_labels );

    virtual void calc_node_value( CvDTreeNode* node );

    virtual void prune_cv();
    virtual double update_tree_rnc( int T, int fold );
    virtual int cut_tree( int T, int fold, double min_alpha );
    virtual void free_prune_data(bool cut_tree);
    virtual void free_tree();

    virtual void write_node( CvFileStorage* fs, CvDTreeNode* node );
    virtual void write_split( CvFileStorage* fs, CvDTreeSplit* split );
    virtual CvDTreeNode* read_node( CvFileStorage* fs, CvFileNode* node, CvDTreeNode* parent );
    virtual CvDTreeSplit* read_split( CvFileStorage* fs, CvFileNode* node );
    virtual void write_tree_nodes( CvFileStorage* fs );
    virtual void read_tree_nodes( CvFileStorage* fs, CvFileNode* node );

    CvDTreeNode* root;

    int pruned_tree_idx;
    CvMat* var_importance;

    CvDTreeTrainData* data;
};
```


CvDTree::train

Trains decision tree

```
bool CvDTree::train( const CvMat* _train_data, int _tflag,
                     const CvMat* _responses, const CvMat* _var_idx=0,
                     const CvMat* _sample_idx=0, const CvMat* _var_type=0,
                     const CvMat* _missing_mask=0,
                     CvDTreeParams params=CvDTreeParams() );

bool CvDTree::train( CvDTreeTrainData* _train_data, const CvMat* _subsample_idx );
```

在CvDTree中一共有两种训练方法。 第一种方法从基类CvStatModel::train而来，很完整。支持所有的数据方式（_tflag=CV_ROW_SAMPLE and _tflag=CV_COL_SAMPLE），同时还有样本和变量子集，缺失数据（missing measurements），输入和输出的变量类型的人工合并等。最后一个参数包含所有必须的训练参数，具体参见CvDTreeParams的描述。

第二种训练方法最多的是用在构建树集成。它采用了预构造的CvDTreeTrainData instance和训练集合的可选择子集。_subsample_idx的索引值与_sample_indx相关，并传递给 CvDTreeTrainData构造函数。比如，如果_sample_idx=[1，5，7，100]，那么_subsample_idx=[0，3] 就表示样本原始样本集合的[1，100]被使用。

There are 2 train methods in CvDTree.

The first method follows the generic CvStatModel::train conventions, it is the most complete form of it. Both data layouts (_tflag=CV_ROW_SAMPLE and _tflag=CV_COL_SAMPLE) are supported, as well as sample and variable subsets, missing measurements, arbitrary combinations of input and output variable types etc. The last parameter contains all the necessary training parameters, see CvDTreeParams description.

The second method train is mostly used for building tree ensembles. It takes the pre-constructed CvDTreeTrainData instance and the optional subset of training set. The indices in _subsample_idx are counted relatively to the _sample_idx, passed to CvDTreeTrainData constructor. For example, if _sample_idx=[1, 5, 7, 100], then _subsample_idx=[0,3] means that the samples [1, 100] of the original training set are used.

[编辑]

CvDTree::predict

返回输入向量对应的叶子结点。

Returns the leaf node of decision tree corresponding to the input vector

```
CvDTreeNode* CvDTree::predict( const CvMat* _sample, const CvMat* _missing_data_mask=0,
                               bool raw_mode=false ) const;
```

该方法用特征向量和可选的缺失数据的掩模作为输入，在决策树计算之后，用到达的叶子结点作为输出。预测结果，无论是类别标签还是预测函数值，都在 CvDTreeNode结构体中找到，比如，dtree->predict(sample,mask)->value。

最后一个参数通常设置为false，这样就表示正常的输入。如果为true，该方法就假定所有的离散输入变量值都已经归一化到 0<num_of_categoriesi>-1这个区间内。（因为决策树在内部都用归一化的表示）。这对用树集成进行快速预测非常有用。对 数值输入变量就不用该标记。

例子，对Mushroom构建树进行分类

见mushroom.cpp，演示了怎样构建和使用决策树。

The method takes the feature vector and the optional missing measurement mask on input, traverses the decision tree and returns the reached leaf node on output. The prediction result, either the class label or the estimated function value, may be retrieved as value field of the CvDTreeNode structure, for example: dtree->predict(sample,mask)->value

The last parameter is normally set to false that implies a regular input. If it is true, the method assumes that all the values of the discrete input variables have been already normalized to 0..<num_of_categoriesi>-1 ranges. (as the decision tree uses such normalized representation internally). It is useful for faster prediction with tree ensembles. For ordered input variables the flag is not used. Example. Building Tree for Classifying Mushrooms

See mushroom.cpp sample that demonstrates how to build and use the decision tree.

[编辑]

Boosting

一般的机器学习方法是监督学习方法：训练数据由输入和期望的输出组成，然后对非训练数据进行预测输出，也就是找出输入x与输出y之间的函数关系F: y = F(x)。根据输出的精确特性又可以分为分类和回归。

Boosting 是个非常强大的学习方法，它也是一个监督的分类学习方法。它组合许多“弱”分类器来产生一个强大的分类器组[HTF01]。一个弱分类器的性能只是比随机选择好一点，因此它可以被 设计的非常简单并且不会有

太大的计算花费。将很多弱分类器结合起来组成一个集成的类似于SVM或者神经网络的强分类器。

在设计boosting分类器的时候最常用的弱分类器是决策树。通常每个树只具有一个节点的这种最简单的决策树就足够了。

Boosting模型的学习时间里在N个训练样本{(xi,yi)}1N 其中 xi ∈ RK 并且 yi ∈ {−1, +1}。xi是一个K维向量。每一维对应你所要分类的问题中的一个特征。输出的两类为-1和+1。

几种不同的boosting如离散AdaBoost, 实数AdaBoost, LogitBoost和Gentle AdaBoost[FHT98]。它们有非常类似的总体结构。因此，我们只需要了解下面表格中最基础的两类：离散AdaBoost和实数Adaboost 算法。为每一个样本初始化使它们具有相同的权值(step 2)。然后一个弱分类器f(x)在具有权值的训练数据上进行训练。计算错误率和换算系数cm(step 3b). 被错分的样本的权重会增加。所有的权重进行归一化，并继续寻找若其他分类器M-1次。最后得到的分类器F(x)是这些独立的弱分类器的和的符号函数 (step 4)。

1. 给定N样本 (x_i,y_i) 其中 $x_i \in R^k, y_i \in -1, +1.$
2. 初始化权值 $w_i = 1/N, i = 1, \cdots, N.$
3. 重复 for m = 1,2,...,M:
    1. 根据每个训练数据的w_i计算 $f_m(x) \in -1, 1$。
    2. 计算 $err_m = E_w[1(y \neq f_m(x))], c_m = \log((1 - err_m)/err_m)$.
    3. 更新权值 $w_i \leftarrow w_i \exp[c_m 1(y_i \neq f_m(x_i))], i = 1, 2, \cdots, N$, 并归一化使 $\Sigma_i w_i = 1$.
4. 输出分类器$sign[\Sigma_{m=1}^{M} c_m f_m(x)]$.

两类问题的算法：训练(step 1~3)和估计(step 4)

注意：作为传统的boosting算法，上述的算法只可以解决两类问题。对于多类的分类问题可以使用AdaBoost.MH算法将其简化为两类分类的问题，但同时需要较大的训练数据，可以在[FHT98]中找到相应的说明。

为了减少计算的时间复杂度而不减少精度，可以使用influence trimming方法。随着训练算法的进行和集合中树的数量的增加和信任度的增加，大部分的训练数据被正确的分类，从而这些样本的权重不断的降低。具有较 低相关权重的样本对弱分类器的训练有较低的影响。因此这些样本会在训练分类器时排除在外而不对分类器造成较大影响。控制这个过程的参数是 weight_trim_rate。只有样本的每小部分的weight_trim_rate的总和较大时才会被用于弱分类器的训练。注意每个样本的系数在 每个循环中被重新计算。一些已经删除的样本可能会在训练更多的分类器时被再次使用。[FHT98].

[HTF01] Hastie, T., Tibshirani, R., Friedman, J. H. The Elements of Statistical Learning: Data Mining, Inference, and Prediction. Springer Series in Statistics. 2001.

[FHT98] Friedman, J. H., Hastie, T. and Tibshirani, R. Additive Logistic Regression: a Statistical View of Boosting. Technical Report, Dept. of Statistics, Stanford University, 1998.

[编辑]

CvBoostParams

Boosting 训练参数

```
struct CvBoostParams : public CvDTreeParams
{
    int boost_type;
    int weak_count;
    int split_criteria;
    double weight_trim_rate;
```

```
    CvBoostParams();
    CvBoostParams( int boost_type, int weak_count, double weight_trim_rate,
                   int max_depth, bool use_surrogates, const float* priors );
};
```

boost_type

Boosting type, 下面的一种

CvBoost::DISCRETE - 离散 AdaBoost
CvBoost::REAL - 实数 AdaBoost
CvBoost::LOGIT - LogitBoost
CvBoost::GENTLE - Gentle AdaBoost
Gentle AdaBoost 和实数 AdaBoost 是最经常的选择。

weak_count
建立的弱分类器的个数。

split_criteria
分裂准则, 用以为弱分类树选择最优的分裂系数。构造函数:
CvBoost::DEFAULT - 为特定的boosting算法选择默认系数;见下文。
CvBoost::GINI - 使用 Gini 索引。这是实数AdaBoost的默认方法; 也可以被用做离散AdaBoost。
CvBoost::MISCLASS - 使用误分类速率。这是离散AdaBoost的默认方法; 也可以被用做实数AdaBoost。
CvBoost::SQERR - 使用最小二乘准则. 这是LogitBoost和Gentle AdaBoost的默认选择和唯一选择。

weight_trim_rate
The weight trimming ratio, 在0到1之间，参考上面的讨论。 如果这个系数小于等于0或者大于1，那么这个trimming将不会被使用，所有的样本都会在每个循环中使用。其默认值为0.95

这个结构从CvDTreeParams继承，但不是所有的决策树参数都支持。特别的，cross-validation不被支持。

CvBoostTree

弱分类树

```
class CvBoostTree: public CvDTree
{
public:
    CvBoostTree();
    virtual ~CvBoostTree();

    virtual bool train( CvDTreeTrainData* _train_data,
                        const CvMat* subsample_idx, CvBoost* ensemble );
    virtual void scale( double s );
    virtual void read( CvFileStorage* fs, CvFileNode* node,
                       CvBoost* ensemble, CvDTreeTrainData* _data );
    virtual void clear();

protected:
    ...
    CvBoost* ensemble;
};
```

作为一个boost树分类器CvBoost的组成部分，这个弱分类器是从CvDTree派生得来的。通常，不需要直接使用弱分类器，虽然它们可以作为CvBoost::weak的元素序列通过CvBoost::get_weak_predictions来访问。

注意：在LogitBoost和Gentle AdaBoost的情况下，每一个若预测器是一个递归树，而不是分类树。甚至在离散AdaBoost和实数AdaBoost的情况 下，CvBoost::predict的返回值(CvDTreeNode::value)也不是输出的类别标号；一个负数代表为类别#0，一个正数代表类 别#1。而这些数只是权重。每一个独立的树的权重可以通过函数CvBoostTree::scale增加或减少。

CvBoost

Boost树分类器

```
class CvBoost : public CvStatModel
{
public:
    // Boosting type
    enum { DISCRETE=0, REAL=1, LOGIT=2, GENTLE=3 };

    // Splitting criteria
    enum { DEFAULT=0, GINI=1, MISCLASS=3, SQERR=4 };

    CvBoost();
    virtual ~CvBoost();

    CvBoost( const CvMat* _train_data, int _tflag,
             const CvMat* _responses, const CvMat* _var_idx=0,
             const CvMat* _sample_idx=0, const CvMat* _var_type=0,
             const CvMat* _missing_mask=0,
             CvBoostParams params=CvBoostParams() );

    virtual bool train( const CvMat* _train_data, int _tflag,
             const CvMat* _responses, const CvMat* _var_idx=0,
             const CvMat* _sample_idx=0, const CvMat* _var_type=0,
             const CvMat* _missing_mask=0,
             CvBoostParams params=CvBoostParams(),
             bool update=false );

    virtual float predict( const CvMat* _sample, const CvMat* _missing=0,
                           CvMat* weak_responses=0, CvSlice slice=CV_WHOLE_SEQ,
                           bool raw_mode=false ) const;

    virtual void prune( CvSlice slice );

    virtual void clear();

    virtual void write( CvFileStorage* storage, const char* name );
    virtual void read( CvFileStorage* storage, CvFileNode* node );

    CvSeq* get_weak_predictors();
    const CvBoostParams& get_params() const;
    ...

protected:
    virtual bool set_params( const CvBoostParams& _params );
    virtual void update_weights( CvBoostTree* tree );
    virtual void trim_weights();
    virtual void write_params( CvFileStorage* fs );
    virtual void read_params( CvFileStorage* fs, CvFileNode* node );

    CvDTreeTrainData* data;
    CvBoostParams params;
    CvSeq* weak;
    ...
};
```


CvBoost::train

训练boost树分类器

```
bool CvBoost::train( const CvMat* _train_data, int _tflag,
             const CvMat* _responses, const CvMat* _var_idx=0,
             const CvMat* _sample_idx=0, const CvMat* _var_type=0,
             const CvMat* _missing_mask=0,
             CvBoostParams params=CvBoostParams(),
             bool update=false );
```

这个分类函数使用的是最通常的模板，最后一个参数update表示分类器是否需要更新(例如，新的弱分类树是否被加入到已存在的总和里), 或者是分类器是否需要重新建立。返回必须是类别，boost树不能用于递归的建立，且必须是两类。 注：如_train_data的维数是data_number*featureDim,则_responses的维数为data_number*1,_var_type的维数为featureDim+1*1,也就是说_var_type的维数是特征维数+1。并且，训练样本的样本数有一定的限制，样本的个数最低为10个，否则会报错。

[编辑]

CvBoost::predict

预测对输入样本的响应

```
float CvBoost::predict( const CvMat* sample, const CvMat* missing=0,
                        CvMat* weak_responses=0, CvSlice slice=CV_WHOLE_SEQ,
                        bool raw_mode=false ) const;
```

sample
: 输入样本。

missing
: 输入的sample中可能存在一些缺少的测量（即一部分测量可能没有观测到），因此需要用一个掩模来指出这些部分。要处理缺少的测量，弱分类器必须包含替代的方法（参见CvDTreeParams::use_surrogates）。

weak_reponses
: 可选的输出参数，weak_reponses是一个浮点数向量，向量中每一个浮点数都对应一个独立弱分类器的响应。向量中元素的数量必须与弱分类器的数量相等。

slice
: 弱分类器响应中用于预测的连续子集。默认情况下使用所有的弱分类器。

raw_mode
: 与在CvDTree::predict中的意思相同。一般而言被设为false。

CvBoost::predict方法通过总体的树运行样本，返回输出的加权类标签。

[编辑]

CvBoost::prune

移除指定的弱分类器

```
void CvBoost::prune( CvSlice slice );
```

这个方法从结果中移除指定的弱分类器。请注意不要将这个方法与一个目前还不支持的移除单独决策树的方法混淆起来。

[编辑]

CvBoost::get_weak_predictors

返回弱分类树的结果

```
CvSeq* CvBoost::get_weak_predictors();
```

这个方法返回弱分类器的结果。每个元素的结果都是一个指向CvBoostTree类的指针（或者也可能指向它的一些派生）。

[编辑]

Random Trees

Random trees have been introduced by Leo Breiman and Adele Cutler: http://www.stat.berkeley.edu/users/breiman/RandomForests/. The algorithm can deal with both classification and regression problems. Random trees is a collection (ensemble) of tree predictors that is called forest further in this section (the term has been also introduced by L. Brieman). The classification works as following: the random trees classifier takes the input feature vector, classifies it with every tree in

the forest, and outputs the class label that has got the majority of "votes". In case of regression the classifier response is the average of responses over all the trees in the forest.

All the trees are trained with the same parameters, but on the different training sets, which are generated from the original training set using bootstrap procedure: for each training set we randomly select the same number of vectors as in the original set (=N). The vectors are chosen with replacement. That is, some vectors will occur more than once and some will be absent. At each node of each tree trained not all the variables are used to find the best split, rather than a random subset of them. The each node a new subset is generated, however its size is fixed for all the nodes and all the trees. It is a training parameter, set to sqrt(<number_of_variables>) by default. None of the tree built is pruned.

In random trees there is no need in any accuracy estimation procedures, such as cross-validation or bootstrap, or a separate test set to get an estimate of the training error. The error is estimated internally during the training. When the training set for the current tree is drawn by sampling with replacement, some vectors are left out (so-called oob (out-of-bag) data). The size of oob data is about N/3. The classification error is estimated by using this oob-data as following:

Get a prediction for each vector, which is oob relatively to the i-th tree, using the very i-th tree. After all the trees have been trained, for each vector that has ever been oob, find the class-"winner" for it (i.e. the class that has got the majority of votes in the trees, where the vector was oob) and compare it to the ground-truth response. Then the classification error estimate is computed as ratio of number of missclassified oob vectors to all the vectors in the original data. In the case of regression the oob-error is computed as the squared error for oob vectors difference divided by the total number of vectors.

References:

Machine Learning, Wald I, July 2002 Looking Inside the Black Box, Wald II, July 2002 Software for the Masses, Wald III, July 2002 And other articles from the web-site http://www.stat.berkeley.edu/users/breiman/RandomForests/cc_home.htm.

CvRTParams Training Parameters of Random Trees

struct CvRTParams : public CvDTreeParams {

```
    bool calc_var_importance;
    int nactive_vars;
    CvTermCriteria term_crit;

    CvRTParams() : CvDTreeParams( 5, 10, 0, false, 10, 0, false, false, 0 ),
        calc_var_importance(false), nactive_vars(0)
    {
        term_crit = cvTermCriteria( CV_TERMCRIT_ITER+CV_TERMCRIT_EPS, 50, 0.1 );
    }

    CvRTParams( int _max_depth, int _min_sample_count,
                float _regression_accuracy, bool _use_surrogates,
                int _max_categories, const float* _priors,
                bool _calc_var_importance,
                int _nactive_vars, int max_tree_count,
                float forest_accuracy, int termcrit_type );
```

};

calc_var_importance If it is set, then variable importance is computed by the training procedure. To retrieve the computed variable importance array, call the method CvRTrees::get_var_importance(). nactive_vars The number of variables that are randomly selected at each tree node and that are used to find the best split(s). term_crit Termination criteria for growing the forest: term_crit.max_iter is the maximum number of trees in the forest (see also max_tree_count parameter of the constructor, by default it is set to 50)

term_crit.epsilon is the sufficient accuracy (OOB error). The set of training parameters for the forest is the superset of the training parameters for a single tree. However, Random trees do not need all the functionality/features of decision trees, most noticeably, the trees are not pruned, so the cross-validation parameters are not used.

CvRTrees Random Trees

class CvRTrees : public CvStatModel { public:

```
   CvRTrees();
   virtual ~CvRTrees();
   virtual bool train( const CvMat* _train_data, int _tflag,
                       const CvMat* _responses, const CvMat* _var_idx=0,
                       const CvMat* _sample_idx=0, const CvMat* _var_type=0,
                       const CvMat* _missing_mask=0,
                       CvRTParams params=CvRTParams() );
   virtual float predict( const CvMat* sample, const CvMat* missing = 0 ) const;
   virtual void clear();

   virtual const CvMat* get_var_importance();
   virtual float get_proximity( const CvMat* sample_1, const CvMat* sample_2 ) const;

   virtual void read( CvFileStorage* fs, CvFileNode* node );
   virtual void write( CvFileStorage* fs, const char* name );

   CvMat* get_active_var_mask();
   CvRNG* get_rng();

   int get_tree_count() const;
   CvForestTree* get_tree(int i) const;
```

protected:

```
   bool grow_forest( const CvTermCriteria term_crit );

   // array of the trees of the forest
   CvForestTree** trees;
   CvDTreeTrainData* data;
   int ntrees;
   int nclasses;
   ...

};
```

CvRTrees::train Trains Random Trees model

bool CvRTrees::train(const CvMat* train_data, int tflag,

```
                      const CvMat* responses, const CvMat* comp_idx=0,
                      const CvMat* sample_idx=0, const CvMat* var_type=0,
                      const CvMat* missing_mask=0,
                      CvRTParams params=CvRTParams() );
```

The method CvRTrees::train is very similar to the first form of CvDTree::train() and follows the generic method CvStatModel::train conventions. All the specific to the algorithm training parameters are passed as CvRTParams instance. The estimate of the training error (oob-error) is stored in the protected class member oob_error.

CvRTrees::predict Predicts the output for the input sample

double CvRTrees::predict(const CvMat* sample, const CvMat* missing=0) const; The input parameters of the prediction method are the same as in CvDTree::predict, but the return value type is different. This method returns the cummulative result from all the trees in the forest (the class that receives the majority of voices, or the mean of the regression function estimates).

CvRTrees::get_var_importance Retrieves the variable importance array

const CvMat* CvRTrees::get_var_importance() const; The method returns the variable importance vector, computed at the training stage when CvRTParams::calc_var_importance is set. If the training flag is not set, then the NULL pointer is returned. This is unlike decision trees, where variable importance can be computed anytime after the training.

CvRTrees::get_proximity Retrieves proximitity measure between two training samples

float CvRTrees::get_proximity(const CvMat* sample_1, const CvMat* sample_2) const; The method returns proximity measure between any two samples (the ratio of the those trees in the ensemble, in which the samples fall into the same leaf node, to the total number of the trees).

Example. Prediction of mushroom edibility using random trees classifier

1. include <float.h>
2. include <stdio.h>
3. include <ctype.h>
4. include "ml.h"

int main(void) {

```
CvStatModel*    cls = NULL;
CvFileStorage*  storage = cvOpenFileStorage( "Mushroom.xml", NULL,CV_STORAGE_READ );
CvMat*          data = (CvMat*)cvReadByName(storage, NULL, "sample", 0 );
CvMat           train_data, test_data;
CvMat           response;
CvMat*          missed = NULL;
CvMat*          comp_idx = NULL;
CvMat*          sample_idx = NULL;
CvMat*          type_mask = NULL;
int             resp_col = 0;
int             i,j;
CvRTreesParams  params;
CvTreeClassifierTrainParams cart_params;
const int       ntrain_samples = 1000;
const int       ntest_samples  = 1000;
const int       nvars = 23;

if(data == NULL || data->cols != nvars)
{
    puts("Error in source data");
    return -1;
}

cvGetSubRect( data, &train_data, cvRect(0, 0, nvars, ntrain_samples) );
cvGetSubRect( data, &test_data, cvRect(0, ntrain_samples, nvars,
    ntrain_samples + ntest_samples) );

resp_col = 0;
cvGetCol( &train_data, &response, resp_col);
```

```
    /* create missed variable matrix */
    missed = cvCreateMat(train_data.rows, train_data.cols, CV_8UC1);
    for( i = 0; i < train_data.rows; i++ )
        for( j = 0; j < train_data.cols; j++ )
            CV_MAT_ELEM(*missed,uchar,i,j) = (uchar)(CV_MAT_ELEM(train_data,float,i,j) < 0);

    /* create comp_idx vector */
    comp_idx = cvCreateMat(1, train_data.cols-1, CV_32SC1);
    for( i = 0; i < train_data.cols; i++ )
    {
        if(i<resp_col)CV_MAT_ELEM(*comp_idx,int,0,i) = i;
        if(i>resp_col)CV_MAT_ELEM(*comp_idx,int,0,i-1) = i;
    }

    /* create sample_idx vector */
    sample_idx = cvCreateMat(1, train_data.rows, CV_32SC1);
    for( j = i = 0; i < train_data.rows; i++ )
    {
        if(CV_MAT_ELEM(response,float,i,0) < 0) continue;
        CV_MAT_ELEM(*sample_idx,int,0,j) = i;
        j++;
    }
    sample_idx->cols = j;

    /* create type mask */
    type_mask = cvCreateMat(1, train_data.cols+1, CV_8UC1);
    cvSet( type_mask, cvRealScalar(CV_VAR_CATEGORICAL), 0);

    // initialize training parameters
    cvSetDefaultParamTreeClassifier((CvStatModelParams*)&cart_params);
    cart_params.wrong_feature_as_unknown = 1;
    params.tree_params = &cart_params;
    params.term_crit.max_iter = 50;
    params.term_crit.epsilon = 0.1;
    params.term_crit.type = CV_TERMCRIT_ITER|CV_TERMCRIT_EPS;

    puts("Random forest results");
    cls = cvCreateRTreesClassifier( &train_data, CV_ROW_SAMPLE, &response,
        (CvStatModelParams*)& params, comp_idx, sample_idx, type_mask, missed );
    if( cls )
    {
        CvMat sample = cvMat( 1, nvars, CV_32FC1, test_data.data.fl );
        CvMat test_resp;
        int wrong = 0, total = 0;
        cvGetCol( &test_data, &test_resp, resp_col);
        for( i = 0; i < ntest_samples; i++, sample.data.fl += nvars )
        {
            if( CV_MAT_ELEM(test_resp,float,i,0) >= 0 )
            {
                float resp = cls->predict( cls, &sample, NULL );
                wrong += (fabs(resp-response.data.fl[i]) > 1e-3 ) ? 1 : 0;
                total++;
            }
        }
        printf( "Test set error = %.2f\n", wrong*100.f/(float)total );
    }
    else
       puts("Error forest creation");



    cvReleaseMat(&missed);
    cvReleaseMat(&sample_idx);
    cvReleaseMat(&comp_idx);
    cvReleaseMat(&type_mask);
    cvReleaseMat(&data);
    cvReleaseStatModel(&cls);
    cvReleaseFileStorage(&storage);
    return 0;

}
```


Expectation-Maximization

The EM (Expectation-Maximization) algorithm estimates the parameters of the multivariate probability density function in a form of the Gaussian mixture distribution with a specified number of mixtures.

Consider the set of the feature vectors {x1, x2,..., xN}: N vectors from d-dimensional Euclidean space drawn from a Gaussian mixture:

where m is the number of mixtures, pk is the normal distribution density with the mean ak and covariance matrix Sk, πk is the weight of k-th mixture. Given the number of mixtures m and the samples {xi, i=1..N} the algorithm finds the maximum-likelihood estimates (MLE) of the all the mixture parameters, i.e. ak, Sk and πk:

EM algorithm is an iterative procedure. Each iteration of it includes two steps. At the first step (Expectation-step, or E-step), we find a probability pi,k (denoted αi,k in the formula below) of sample #i to belong to mixture #k using the currently available mixture parameter estimates:

At the second step (Maximization-step, or M-step) the mixture parameter estimates are refined using the computed probabilities:

Alternatively, the algorithm may start with M-step when initial values for pi,k can be provided. Another alternative, when pi,k are unknown, is to use a simpler clustering algorithm to pre-cluster the input samples and thus obtain initial pi,k. Often (and in ML) k-means algorithm is used for that purpose. One of the main that EM algorithm should deal with is the large number of parameters to estimate. The majority of the parameters sits in covariation matrices, which are d×d elements each (where d is the feature space dimensionality). However, in many practical problems the covariation matrices are close to diagonal, or even to μk*I, where I is identity matrix and μk is mixture-dependent "scale" parameter. So a robust computation scheme could be to start with the harder constraints on the covariation matrices and then use the estimated parameters as an input for a less constrained optimization problem (often a diagonal covariation matrix is already a good enough approximation).

References:

[Bilmes98] J. A. Bilmes. A Gentle Tutorial of the EM Algorithm and its Application to Parameter Estimation for Gaussian Mixture and Hidden Markov Models. Technical Report TR-97-021, International Computer Science Institute and Computer Science Division, University of California at Berkeley, April 1998.

[编辑]

CvEMParams

EM算法估计混合高斯模型所需要的参数

Parameters of EM algorithm

struct CvEMParams {

```
CvEMParams() : nclusters(10), cov_mat_type(CvEM::COV_MAT_DIAGONAL),
    start_step(CvEM::START_AUTO_STEP), probs(0), weights(0), means(0), covs(0)
{
```

```
        term_crit=cvTermCriteria( CV_TERMCRIT_ITER+CV_TERMCRIT_EPS, 100, FLT_EPSILON );
    }

    CvEMParams( int _nclusters, int _cov_mat_type=1/*CvEM::COV_MAT_DIAGONAL*/,
                int _start_step=0/*CvEM::START_AUTO_STEP*/,
                CvTermCriteria _term_crit=cvTermCriteria(CV_TERMCRIT_ITER+CV_TERMCRIT_EPS, 100,
FLT_EPSILON),
                CvMat* _probs=0, CvMat* _weights=0, CvMat* _means=0, CvMat** _covs=0 ) :
                nclusters(_nclusters), cov_mat_type(_cov_mat_type), start_step(_start_step),
                probs(_probs), weights(_weights), means(_means), covs(_covs), term_crit(_term_crit)
    {}

    int nclusters;
    int cov_mat_type;
    int start_step;
    const CvMat* probs;
    const CvMat* weights;
    const CvMat* means;
    const CvMat** covs;
    CvTermCriteria term_crit;

};
```

[编辑]

nclusters

高斯成分的个数，必须事先指定。有些EM的算法可以在给定区间内，自动确定最优个数，不过本算法未实现此功能。

The number of mixtures. Some of EM implementation could determine the optimal number of mixtures within a specified value range, but that is not the case in ML yet.

[编辑]

cov_mat_type

协变矩阵的类型，只能为下面三种类型之一

The type of the mixture covariation matrices; should be one of the following:

对称矩阵 每一个混合的协变矩阵都是任意的对称正定矩阵，因此每个矩阵的自由参数为d2/2。通常来说，除非对参数有非常精确的估计，或者有足够大量的样本，否则不建议使用这个选项。

CvEM::COV_MAT_GENERIC - a covariation matrix of each mixture may be arbitrary symmetrical positively defined matrix, so the number of free parameters in each matrix is about d2/2. It is not recommended to use this option, unless there is pretty accurate initial estimation of the parameters and/or a huge number of training samples.

对角阵（假设数据各维之间是独立的，最常用的情况） 每一个混合的协变矩阵都是任意的正对角阵，也就是说，非对角位置的值必须是0，因此每一个矩阵的自由参数为d。这是一个最常用的选项，也能得到很好的结果。

CvEM::COV_MAT_DIAGONAL - a covariation matrix of each mixture may be arbitrary diagonal matrix with positive diagonal elements, that is, non-diagonal elements are forced to be 0's, so the number of free parameters is d for each matrix. This is most commonly used option yielding good estimation results.

球矩阵 每一个混合的协变矩阵都是全N阵，即μk*I，一次唯一的参数就是μk。这个选项通常用在一些特殊的场合，比如约束条件是相关的，或者作为某些优化算法的 第一步（比如，用PCA来预处理数据时）。这个预估计的结果会被再次传递到优化过程中，比如 cov_mat_type=CvEM::COV_MAT_DIAGONAL。

CvEM::COV_MAT_SPHERICAL - a covariation matrix of each mixture is a scaled identity matrix, μk*I, so the only parameter to be estimated is μk. The option may be used in special cases, when the constraint is

relevant, or as a first step in the optimization (e.g. in case when the data is preprocessed with PCA). The results of such preliminary estimation may be passed again to the optimization procedure, this time with cov_mat_type=CvEM::COV_MAT_DIAGONAL.

[编辑]

start_step

开始步骤， 3种选择

The initial step the algorithm starts from; should be one of the following:

CvEM::START_E_STEP - the algorithm starts with E-step. At least, the initial values of mean vectors, CvEMParams::means must be passed. Optionally, the user may also provide initial values for weights (CvEMParams::weights) and/or covariation matrices (CvEMParams::covs).

CvEM::START_M_STEP - the algorithm starts with M-step. The initial probabilities pi,k must be provided.

CvEM::START_AUTO_STEP - No values are required from the user, k-means algorithm is used to estimate initial mixtures parameters.

[编辑]

term_crit

E步和M步 迭代停止的准则。 EM算法会在一定的迭代次数之后（term_crit.num_iter)，或者当模型参数在两次迭代之间的变化小于预定值（term_crit.epsilon）时停止

Termination criteria of the procedure. EM algorithm stops either after a certain number of iterations (term_crit.num_iter), or when the parameters change too little (no more than term_crit.epsilon) from iteration to iteration.

[编辑]

probs

初始的后验概率 Initial probabilities pi,k; are used (and must be not NULL) only when start_step=CvEM::START_M_STEP.

[编辑]

weights

初始的各个成分的概率 Initial mixture weights πk; are used (if not NULL) only when start_step=CvEM::START_E_STEP.

[编辑]

covs

初始的协方差矩阵 Initial mixture covariation matrices Sk; are used (if not NULL) only when start_step=CvEM::START_E_STEP.

[编辑]

means

初始的均值 Initial mixture means ak; are used (and must be not NULL) only when start_step=CvEM::START_E_STEP.

The structure has 2 constructors, the default one represents a rough rule-of-thumb, with another one it is possible to override a variety of parameters, from a single number of mixtures (the only essential problem-dependent parameter), to the initial values for the mixture parameters.

CvEM EM model

class CV_EXPORTS CvEM : public CvStatModel { public:

```
// Type of covariation matrices
enum { COV_MAT_SPHERICAL=0, COV_MAT_DIAGONAL=1, COV_MAT_GENERIC=2 };

// The initial step
enum { START_E_STEP=1, START_M_STEP=2, START_AUTO_STEP=0 };

CvEM();
CvEM( const CvMat* samples, const CvMat* sample_idx=0,
      CvEMParams params=CvEMParams(), CvMat* labels=0 );
virtual ~CvEM();

virtual bool train( const CvMat* samples, const CvMat* sample_idx=0,
                    CvEMParams params=CvEMParams(), CvMat* labels=0 );

virtual float predict( const CvMat* sample, CvMat* probs ) const;
virtual void clear();

int get_nclusters() const { return params.nclusters; }
const CvMat* get_means() const { return means; }
const CvMat** get_covs() const { return covs; }
const CvMat* get_weights() const { return weights; }
const CvMat* get_probs() const { return probs; }
```

protected:

```
virtual void set_params( const CvEMParams& params,
                         const CvVectors& train_data );
virtual void init_em( const CvVectors& train_data );
virtual double run_em( const CvVectors& train_data );
virtual void init_auto( const CvVectors& samples );
virtual void kmeans( const CvVectors& train_data, int nclusters,
                     CvMat* labels, CvTermCriteria criteria,
                     const CvMat* means );
CvEMParams params;
double log_likelihood;

CvMat* means;
CvMat** covs;
CvMat* weights;
CvMat* probs;

CvMat* log_weight_div_det;
CvMat* inv_eigen_values;
CvMat** cov_rotate_mats;
```

};

CvEM::train Estimates Gaussian mixture parameters from the sample set

void CvEM::train(const CvMat* samples, const CvMat* sample_idx=0,

```
                CvEMParams params=CvEMParams(), CvMat* labels=0 );
```

Unlike many of ML models, EM is an unsupervised learning algorithm and it does not take responses (class labels or the function values) on input. Instead, it computes MLE of Gaussian mixture parameters from the input sample set, stores all the parameters inside the stucture: pi,k in probs, ak in means Sk in covs[k], πk in weights and optionally computes the output "class label" for each sample: labelsi=arg maxk(pi,k), i=1..N (i.e. indices of the most-probable mixture for each sample).

The trained model can be used further for prediction, just like any other classifier. The model trained is similar to the normal bayes classifier.

Example. Clustering random samples of multi-gaussian distribution using EM

1. include "ml.h"
2. include "highgui.h"

int main(int argc, char** argv) {

```
const int N = 4;
const int N1 = (int)sqrt((double)N);
const CvScalar colors[] = {{{0,0,255}},Template:0,255,0,Template:0,255,255,Template:255,255,0};
int i, j;
int nsamples = 100;
CvRNG rng_state = cvRNG(-1);
CvMat* samples = cvCreateMat( nsamples, 2, CV_32FC1 );
CvMat* labels = cvCreateMat( nsamples, 1, CV_32SC1 );
IplImage* img = cvCreateImage( cvSize( 500, 500 ), 8, 3 );
float _sample[2];
CvMat sample = cvMat( 1, 2, CV_32FC1, _sample );
CvEM em_model;
CvEMParams params;
CvMat samples_part;

cvReshape( samples, samples, 2, 0 );
for( i = 0; i < N; i++ )
{
    CvScalar mean, sigma;

    // form the training samples
    cvGetRows( samples, &samples_part, i*nsamples/N, (i+1)*nsamples/N );
    mean = cvScalar(((i%N1)+1.)*img->width/(N1+1), ((i/N1)+1.)*img->height/(N1+1));
    sigma = cvScalar(30,30);
    cvRandArr( &rng_state, &samples_part, CV_RAND_NORMAL, mean, sigma );
}
cvReshape( samples, samples, 1, 0 );

// initialize model's parameters
params.covs      = NULL;
params.means     = NULL;
params.weights   = NULL;
params.probs     = NULL;
params.nclusters = N;
params.cov_mat_type       = CvEM::COV_MAT_SPHERICAL;
params.start_step         = CvEM::START_AUTO_STEP;
params.term_crit.max_iter = 10;
params.term_crit.epsilon  = 0.1;
params.term_crit.type     = CV_TERMCRIT_ITER|CV_TERMCRIT_EPS;

// cluster the data
em_model.train( samples, 0, params, labels );
```

1. if 0

```
    // the piece of code shows how to repeatedly optimize the model
    // with less-constrained parameters (COV_MAT_DIAGONAL instead of COV_MAT_SPHERICAL)
    // when the output of the first stage is used as input for the second.
    CvEM em_model2;
    params.cov_mat_type = CvEM::COV_MAT_DIAGONAL;
    params.start_step = CvEM::START_E_STEP;
    params.means = em_model.get_means();
    params.covs = (const CvMat**)em_model.get_covs();
    params.weights = em_model.get_weights();

    em_model2.train( samples, 0, params, labels );
    // to use em_model2, replace em_model.predict() with em_model2.predict() below
```

1. endif

```
    // classify every image pixel
    cvZero( img );
    for( i = 0; i < img->height; i++ )
    {
        for( j = 0; j < img->width; j++ )
        {
            CvPoint pt = cvPoint(j, i);
            sample.data.fl[0] = (float)j;
            sample.data.fl[1] = (float)i;
            int response = cvRound(em_model.predict( &sample, NULL ));
            CvScalar c = colors[response];

            cvCircle( img, pt, 1, cvScalar(c.val[0]*0.75,c.val[1]*0.75,c.val[2]*0.75), CV_FILLED );
        }
    }

    //draw the clustered samples
    for( i = 0; i < nsamples; i++ )
    {
        CvPoint pt;
        pt.x = cvRound(samples->data.fl[i*2]);
        pt.y = cvRound(samples->data.fl[i*2+1]);
        cvCircle( img, pt, 1, colors[labels->data.i[i]], CV_FILLED );
    }

    cvNamedWindow( "EM-clustering result", 1 );
    cvShowImage( "EM-clustering result", img );
    cvWaitKey(0);

    cvReleaseMat( &samples );
    cvReleaseMat( &labels );
    return 0;

}
```


神经网络

ML implements feedforward artificial neural networks, more particularly, multi-layer perceptrons (MLP), the most commonly used type of neural networks. MLP consists of the input layer, output layer and one or more hidden layers. Each layer of MLP includes one or more neurons that are directionally linked with the neurons from the previous and the next layer. Here is an example of 3-layer perceptron with 3 inputs, 2 outputs and the hidden layer including 5 neurons:

ML实现了前馈(feedforward)人工神经元网络(ANN)，准确地说是最常用的神经元网络，multi-layerperceptrons (MLP)。MLP由输入层、输出层和一个或多个隐藏层构成。MLP的每一层包含了一个或多个神经元，它们从之前的层和之后的层的神经元有向的连接在一 起。下面是一个三层感知器(perceptron)的例子，其由3个输入、2个输出以及包含5个神经元的隐藏层构成：

All the neurons in MLP are similar. Each of them has several input links (i.e. it takes the output values from several neurons in the previous layer on input) and several output links (i.e. it passes the response to

several neurons in the next layer). The values retrieved from the previous layer are summed with certain weights, individual for each neuron, plus the bias term, and the sum is transformed using the activation function f that may be also different for different neurons. Here is the picture:

MLP中所有的神经元都是相似的。每一个有多个输入链路（比如，取前一层的多个神经元的输出作为自己的输入）和多个输出链路（比如传递响应到下一层的多个神经元）。

In other words, given the outputs {xj} of the layer n, the outputs {yi} of the layer n+1 are computed as:

```
ui=sumj(w(n+1)i,j*xj) + w(n+1)i,bias
yi=f(ui)
```

Different activation functions may be used, the ML implements 3 standard ones:

Identity function (CvANN_MLP::IDENTITY): f(x)=x Symmetrical sigmoid (CvANN_MLP::SIGMOID_SYM): f(x)=β*(1-e-αx)/(1+e-αx), the default choice for MLP; the standard sigmoid with β=1, α=1 is shown below: Gaussian function (CvANN_MLP::GAUSSIAN): f(x)=βe-αx*x, not completely supported by the moment. In ML all the neurons have the same activation functions, with the same free parameters (α, β) that are specified by user and are not altered by the training algorithms.

So the whole trained network works as following. It takes the feature vector on input, the vector size is equal to the size of the input layer, when the values are passed as input to the first hidden layer, the outputs of the hidden layer are computed using the weights and the activation functions and passed further downstream, until we compute the output layer.

So, in order to compute the network one need to know all the weights w(n+1)i,j. The weights are computed by the training algorithm. The algorithm takes a training set: multiple input vectors with the corresponding output vectors, and iteratively adjusts the weights to try to make the network give the desired response on the provided input vectors.

The larger the network size (the number of hidden layers and their sizes), the more is the potential network flexibility, and the error on the training set could be made arbitrarily small. But at the same time the learned network will also "learn" the noise present in the training set, so the error on the test set usually starts increasing after the network size reaches some limit. Besides, the larger networks are train much longer than the smaller ones, so it is reasonable to preprocess the data (using PCA or similar technique) and train a smaller network on only the essential features.

Another feature of the MLP's is their inability to handle categorical data as is, however there is a workaround. If a certain feature in the input or output (i.e. in case of n-class classifier for n>2) layer is categorical and can take M (>2) different values, it makes sense to represent it as binary tuple of M elements, where i-th element is 1 if and only if the feature is equal to the i-th value out of M possible. It will increase the size of the input/output layer, but will speedup the training algorithm convergence and at the same time enable "fuzzy" values of such variables, i.e. a tuple of probabilities instead of a fixed value.

ML implements 2 algorithms for training MLP's. The first is the classical random sequential backpropagation algorithm and the second (default one) is batch RPROP algorithm

References:

http://en.wikipedia.org/wiki/Backpropagation. Wikipedia article about the backpropagation algorithm. Y. LeCun, L. Bottou, G.B. Orr and K.-R. Muller, "Efficient backprop", in Neural Networks---Tricks of the Trade, Springer Lecture Notes in Computer Sciences 1524, pp.5-50, 1998. M. Riedmiller and H. Braun, "A Direct Adaptive Method for Faster Backpropagation Learning: The RPROP Algorithm", Proc. ICNN, San Fransisco (1993).

CvANN_MLP_TrainParams Parameters of MLP training algorithm

struct CvANN_MLP_TrainParams {

```
    CvANN_MLP_TrainParams();
    CvANN_MLP_TrainParams( CvTermCriteria term_crit, int train_method,
                           double param1, double param2=0 );
    ~CvANN_MLP_TrainParams();

    enum { BACKPROP=0, RPROP=1 };

    CvTermCriteria term_crit;
    int train_method;

    // backpropagation parameters
    double bp_dw_scale, bp_moment_scale;

    // rprop parameters
    double rp_dw0, rp_dw_plus, rp_dw_minus, rp_dw_min, rp_dw_max;
```

}; term_crit The termination criteria for the training algorithm. Identifies how many iteration is done by the algorithm (for sequential backpropagation algorithm the number is multiplied by the size of the training set) and how much the weights could change between the iterations to make the algorithm continue. train_method The training algoithm to use; can be one of CvANN_MLP_TrainParams::BACKPROP (sequential backpropagation algorithm) or CvANN_MLP_TrainParams::RPROP (RPROP algorithm, default value). bp_dw_scale (Backpropagation only): The coefficient to multiply the computed weight gradient by. The recommended value is about 0.1. The parameter can be set via param1 of the constructor. bp_moment_scale (Backpropagation only): The coefficient to multiply the difference between weights on the 2 previous iterations. Provides some inertia to smooth the random fluctuations of the weights. Can vary from 0 (the feature is disabled) to 1 and beyond. The value 0.1 or so is good enough. The parameter can be set via param2 of the constructor. rp_dw0 (RPROP only): Initial magnitude of the weight delta. The default value is 0.1. The parameter can be set via param1 of the constructor. rp_dw_plus (RPROP only): The increase factor for the weight delta. Must be >1, default value is 1.2 that should work well in most cases, according to the algorithm author. The parameter can only be changed explicitly by modifying the structure member. rp_dw_minus (RPROP only): The decrease factor for the weight delta. Must be <1, default value is 0.5 that should work well in most cases, according to the algorithm author. The parameter can only be changed explicitly by modifying the structure member. rp_dw_min (RPROP only): The minimum value of the weight delta. Must be >0, the default value is FLT_EPSILON. The parameter can be set via param2 of the constructor. rp_dw_max (RPROP only): The maximum value of the weight delta. Must be >1, the default value is 50. The parameter can only be changed explicitly by modifying the structure member. The structure has default constructor that initalizes parameters for RPROP algorithm. There is also more advanced constructor to customize the parameters and/or choose backpropagation algorithm. Finally, the individial parameters can be adjusted after the structure is created.

CvANN_MLP MLP model

class CvANN_MLP : public CvStatModel { public:

```
    CvANN_MLP();
    CvANN_MLP( const CvMat* _layer_sizes,
               int _activ_func=SIGMOID_SYM,
               double _f_param1=0, double _f_param2=0 );

    virtual ~CvANN_MLP();

    virtual void create( const CvMat* _layer_sizes,
                         int _activ_func=SIGMOID_SYM,
                         double _f_param1=0, double _f_param2=0 );

    virtual int train( const CvMat* _inputs, const CvMat* _outputs,
                       const CvMat* _sample_weights, const CvMat* _sample_idx=0,
                       CvANN_MLP_TrainParams _params = CvANN_MLP_TrainParams(),
                       int flags=0 );
    virtual float predict( const CvMat* _inputs,
                           CvMat* _outputs ) const;

    virtual void clear();
```

```
    // possible activation functions
    enum { IDENTITY = 0, SIGMOID_SYM = 1, GAUSSIAN = 2 };

    // available training flags
    enum { UPDATE_WEIGHTS = 1, NO_INPUT_SCALE = 2, NO_OUTPUT_SCALE = 4 };

    virtual void read( CvFileStorage* fs, CvFileNode* node );
    virtual void write( CvFileStorage* storage, const char* name );

    int get_layer_count() { return layer_sizes ? layer_sizes->cols : 0; }
    const CvMat* get_layer_sizes() { return layer_sizes; }
```

protected:

```
    virtual bool prepare_to_train( const CvMat* _inputs, const CvMat* _outputs,
            const CvMat* _sample_weights, const CvMat* _sample_idx,
            CvANN_MLP_TrainParams _params,
            CvVectors* _ivecs, CvVectors* _ovecs, double** _sw, int _flags );

    // sequential random backpropagation
    virtual int train_backprop( CvVectors _ivecs, CvVectors _ovecs, const double* _sw );

    // RPROP algorithm
    virtual int train_rprop( CvVectors _ivecs, CvVectors _ovecs, const double* _sw );

    virtual void calc_activ_func( CvMat* xf, const double* bias ) const;
    virtual void calc_activ_func_deriv( CvMat* xf, CvMat* deriv, const double* bias ) const;
    virtual void set_activ_func( int _activ_func=SIGMOID_SYM,
                                 double _f_param1=0, double _f_param2=0 );
    virtual void init_weights();
    virtual void scale_input( const CvMat* _src, CvMat* _dst ) const;
    virtual void scale_output( const CvMat* _src, CvMat* _dst ) const;
    virtual void calc_input_scale( const CvVectors* vecs, int flags );
    virtual void calc_output_scale( const CvVectors* vecs, int flags );

    virtual void write_params( CvFileStorage* fs );
    virtual void read_params( CvFileStorage* fs, CvFileNode* node );

    CvMat* layer_sizes;
    CvMat* wbuf;
    CvMat* sample_weights;
    double** weights;
    double f_param1, f_param2;
    double min_val, max_val, min_val1, max_val1;
    int activ_func;
    int max_count, max_buf_sz;
    CvANN_MLP_TrainParams params;
    CvRNG rng;
```

}; Unlike many other models in ML that are constructed and trained at once, in MLP these steps are separated. First, a network with the specified topology is created using the non-default constructor or the method create. All the weights are set to zero's. Then the network is trained using the set of input and output vectors. The training procedure can be repeated more than once, i.e. the weights can be adjusted based on the new training data.

CvANN_MLP::create Constructs the MLP with the specified topology

void CvANN_MLP::create(const CvMat* _layer_sizes,

```
                        int _activ_func=SIGMOID_SYM,
                        double _f_param1=0, double _f_param2=0 );
```

_layer_sizes The integer vector, specifying the number of neurons in each layer, including the input and the output ones. _activ_func Specifies the activation function for each neuron; one of CvANN_MLP::IDENTITY, CvANN_MLP::SIGMOID_SYM and CvANN_MLP::GAUSSIAN. _f_param1, _f_param2 Free parameters of the activation function, α and β, respectively. See the formulas in the introduction section.

The method creates MLP network with the specified topology and assigns the same activation function to all the neurons.

CvANN_MLP::train Trains/updates MLP

int CvANN_MLP::train(const CvMat* _inputs, const CvMat* _outputs,

```
const CvMat* _sample_weights, const CvMat* _sample_idx=0,
CvANN_MLP_TrainParams _params = CvANN_MLP_TrainParams(),
int flags=0 );
```

_inputs A floating-point matrix of input vectors, one vector per row. _outputs A floating-point matrix of the corresponding output vectors, one vector per row. _sample_weights (RPROP only) The optional floating-point vector of weights for each sample. Some samples may be more important than others for training, e.g. user may want to gain the weight of certain classes to find the right balance between hit-rate and false-alarm rate etc. _sample_idx The optional integer vector indicating the samples (i.e. rows of _inputs and _outputs) that are taken into account. _params The training params. See CvANN_MLP_TrainParams description. _flags The various algorithm parameters. May be a combination of the following: UPDATE_WEIGHTS = 1 - update the network weights, rather than compute them from scratch (in the latter case the weights are intialized using Nguyen-Widrow algorithm). NO_INPUT_SCALE - do not normalize the input vectors. If the flag is not set, the training algorithm normalizes each input feature independently, shifting its mean value to 0 and making the standard deviation =1. If the network is assumed to be updated frequently, the new training data should be much different from original one. In this case user should take care of proper normalization. NO_OUTPUT_SCALE - do not normalize the output vectors. If the flag is not set, the training algorithm normalizes each output features independently, by trasforming it to the certain range depending on the activation function used.

The method applies the specified training algorithm to compute/adjust the network weights. It returns the number of iterations done.

[编辑]

中文翻译者

- 于仕琪，中科院自动化所自由软件协会
- 张鲁闽，中科院自动化所自由软件协会
- 王国钦，中科院自动化所自由软件协会
- 姚 谦， 中科院遥感应用研究所

CvAux中文参考手册

Wikipedia,自由的百科全书

目录

[编辑]

立体匹配

[编辑]

FindStereoCorrespondence

计算一对校正好的图像的视差图

```
cvFindStereoCorrespondence(
                    const  CvArr* leftImage, const  CvArr* rightImage,
                    int     mode, CvArr*  depthImage,
                    int     maxDisparity,
                    double  param1, double  param2, double  param3,
                    double  param4, double  param5  );
```

leftImage:: 左图,必须为8位的灰度图
rightImage:: 右图,必须为8位的灰度图
mode:: 指定采用的算法(当前只支持 CV_DISPARITY_BIRCHFIELD)
depthImage:: 输出的视差图, 8位的灰度图
maxDisparity:: 指定最大的可能差异(视差).物体越近视差越大.

param1, param2, param3, param4, param5:: - 算法的参数 ,param1 为遮挡时的处罚值(constant occlusion penalty), param2 为匹配时的奖励值, param3 定义高可靠区域 (set of contiguous pixels whose reliability is at least param3), param4 定义比较可靠区域defines a moderately reliable region, param5 定义有些可靠的区域defines a slightly reliable region. 如果省略一些参数就会采用默认值.在Birchfield算法中param1 = 25, param2 = 5, param3 = 12, param4 = 15, param5 = 25 (这些数值来自书籍"Depth Discontinuities by Pixel-to-Pixel Stereo" Stanford University Technical Report STAN-CS-TR-96-1573, July 1996.)

函数

```
cvFindStereoCorrespondence
```

计算两个校正后的灰度图像的视差图

例子。计算一对图像的视差

```
/*-------------------------------------------------------------------------------------*/
IplImage* srcLeft = cvLoadImage("left.jpg",1);
IplImage* srcRight = cvLoadImage("right.jpg",1);
IplImage* leftImage = cvCreateImage(cvGetSize(srcLeft), IPL_DEPTH_8U, 1);
IplImage* rightImage = cvCreateImage(cvGetSize(srcRight), IPL_DEPTH_8U, 1);
IplImage* depthImage = cvCreateImage(cvGetSize(srcRight), IPL_DEPTH_8U, 1);

cvCvtColor(srcLeft, leftImage, CV_BGR2GRAY);
cvCvtColor(srcRight, rightImage, CV_BGR2GRAY);

cvFindStereoCorrespondence( leftImage, rightImage, CV_DISPARITY_BIRCHFIELD, depthImage, 50, 15, 3,
6, 8, 15 );
/*-------------------------------------------------------------------------------------*/
```

本例子使用的图片可在以下地址下载

```
http://opencvlibrary.sourceforge.net/pics/left.jpg
http://opencvlibrary.sourceforge.net/pics/right.jpg
```

[编辑]

View Morphing Functions

[编辑]

MakeScanlines

Calculates scanlines coordinates for two cameras by fundamental matrix

```
void cvMakeScanlines( const CvMatrix3* matrix, CvSize img size, int* scanlines1,
```

```
                    int* scanlines2, int* lengths1, int* lengths2, int* line_count );
```

matrix:: Fundamental matrix.imgSize:: Size of the image.scanlines1:: Pointer to the array of calculated scanlines of the first image.scanlines2:: Pointer to the array of calculated scanlines of the second image.lengths1:: Pointer to the array of calculated lengths (in pixels) of the first image scanlines.lengths2:: Pointer to the array of calculated lengths (in pixels) of the second image scanlines.line_count:: Pointer to the variable that stores the number of scanlines.

The function

`cvMakeScanlines`

finds coordinates of scanlines for two images. This function returns the number of scanlines. The function does nothing except calculating the number of scanlines if the pointers

`scanlines1`

or

`scanlines2`

are equal to zero.

PreWarpImage

Rectifies image

```
void cvPreWarpImage( int line_count, IplImage* img, uchar* dst,
                     int* dst_nums, int* scanlines );
```

line_count:: Number of scanlines for the image.img:: Image to prewarp.dst:: Data to store for the prewarp image.dst_nums:: Pointer to the array of lengths of scanlines.scanlines:: Pointer to the array of coordinates of scanlines.

The function

`cvPreWarpImage`

rectifies the image so that the scanlines in the rectified image are horizontal. The output buffer of size

`max(width,height)*line_count*3`

must be allocated before calling the function.

FindRuns

Retrieves scanlines from rectified image and breaks them down into runs

```
void cvFindRuns( int line_count, uchar* prewarp1, uchar* prewarp2,
                 int* line_lengths1, int* line_lengths2,
                 int* runs1, int* runs2,
                 int* num_runs1, int* num_runs2 );
```

line_count:: Number of the scanlines.prewarp1:: Prewarp data of the first image.prewarp2:: Prewarp data of the second image.line_lengths1:: Array of lengths of scanlines in the first image.line_lengths2:: Array of lengths of scanlines in the second image.runs1:: Array of runs in each scanline in the first image.runs2:: Array of runs in each scanline in the second image.num_runs1:: Array of numbers of runs in each scanline

in the first image.num_runs2:: Array of numbers of runs in each scanline in the second image.

The function

`cvFindRuns`

retrieves scanlines from the rectified image and breaks each scanline down into several runs, that is, series of pixels of almost the same brightness.

[编辑]

DynamicCorrespondMulti

Finds correspondence between two sets of runs of two warped images

```
void cvDynamicCorrespondMulti( int line_count, int* first, int* first_runs,
                               int* second, int* second_runs,
                               int* first_corr, int* second_corr );
```

line_count:: Number of scanlines.first:: Array of runs of the first image.first_runs:: Array of numbers of runs in each scanline of the first image.second:: Array of runs of the second image.second_runs:: Array of numbers of runs in each scanline of the second image.first_corr:: Pointer to the array of correspondence information found for the first runs.second_corr:: Pointer to the array of correspondence information found for the second runs.

The function

`cvDynamicCorrespondMulti`

finds correspondence between two sets of runs of two images. Memory must be allocated before calling this function. Memory size for one array of correspondence information is

`max( width,height )* numscanlines*3*sizeof ( int )`

.

[编辑]

MakeAlphaScanlines

Calculates coordinates of scanlines of image from virtual camera

```
void cvMakeAlphaScanlines( int* scanlines1, int* scanlines2,
                           int* scanlinesA, int* lengths,
                           int line_count, float alpha );
```

scanlines1:: Pointer to the array of the first scanlines.scanlines2:: Pointer to the array of the second scanlines.scanlinesA:: Pointer to the array of the scanlines found in the virtual image.lengths:: Pointer to the array of lengths of the scanlines found in the virtual image.line_count:: Number of scanlines.alpha:: Position of virtual camera

`(0.0 - 1.0)`

. The function

`cvMakeAlphaScanlines`

finds coordinates of scanlines for the virtual camera with the given camera position. Memory must be allocated before calling this function. Memory size for the array of correspondence runs is

`numscanlines*2*4*sizeof(int)`

. Memory size for the array of the scanline lengths is

```
numscanlines*2*4*sizeof(int)
```

.

[编辑]

MorphEpilinesMulti

Morphs two pre-warped images using information about stereo correspondence

```
void cvMorphEpilinesMulti( int line_count, uchar* first_pix, int* first_num,
                           uchar* second_pix, int* second_num,
                           uchar* dst_pix, int* dst_num,
                           float alpha, int* first, int* first_runs,
                           int* second, int* second_runs,
                           int* first_corr, int* second_corr );
```

line_count:: Number of scanlines in the prewarp image.first_pix:: Pointer to the first prewarp image.first_num:: Pointer to the array of numbers of points in each scanline in the first image.second_pix:: Pointer to the second prewarp image.second_num:: Pointer to the array of numbers of points in each scanline in the second image.dst_pix:: Pointer to the resulting morphed warped image.dst_num:: Pointer to the array of numbers of points in each line.alpha:: Virtual camera position

```
(0.0 - 1.0)
```

.first:: First sequence of runs.first_runs:: Pointer to the number of runs in each scanline in the first image.second:: Second sequence of runs.second_runs:: Pointer to the number of runs in each scanline in the second image.first_corr:: Pointer to the array of correspondence information found for the first runs.second_corr:: Pointer to the array of correspondence information found for the second runs. The function

```
cvMorphEpilinesMulti
```

morphs two pre-warped images using information about correspondence between the scanlines of two images.

[编辑]

PostWarpImage

Warps rectified morphed image back

```
void cvPostWarpImage( int line_count, uchar* src, int* src_nums,
                      IplImage* img, int* scanlines );
```

line_count:: Number of the scanlines.src:: Pointer to the prewarp image virtual image.src_nums:: Number of the scanlines in the image.img:: Resulting unwarp image.scanlines:: Pointer to the array of scanlines data.

The function

```
cvPostWarpImage
```

warps the resultant image from the virtual camera by storing its rows across the scanlines whose coordinates are calculated by cvMakeAlphaScanlines.

[编辑]

DeleteMoire

Deletes moire in given image

```
void cvDeleteMoire( IplImage* img );
```

img:: Image.

The function

`cvDeleteMoire`

deletes moire from the given image. The post-warped image may have black (un-covered) points because of possible holes between neighboring scanlines. The function deletes moire (black pixels) from the image by substituting neighboring pixels for black pixels. If all the scanlines are horizontal, the function may be omitted.

[编辑]

3D Tracking Functions

The section discusses functions for tracking objects in 3d space using a stereo camera. Besides C API, there is DirectShow 3dTracker filter and the wrapper application 3dTracker. Here you may find a description how to test the filter on sample data.

[编辑]

3dTrackerCalibrateCameras

Simultaneously determines position and orientation of multiple cameras

```
CvBool cv3dTrackerCalibrateCameras(int num_cameras,
            const Cv3dTrackerCameraIntrinsics camera_intrinsics[],
            CvSize checkerboard_size,
            IplImage *samples[],
            Cv3dTrackerCameraInfo camera_info[]);
```

num_cameras:: the number of cameras to calibrate. This is the size of each of the three array parameters.camera_intrinsics:: camera intrinsics for each camera, such as determined by CalibFilter.checkerboard_size:: the width and height (in number of squares) of the checkerboard.samples:: images from each camera, with a view of the checkerboard.camera_info:: filled in with the results of the camera calibration. This is passed into 3dTrackerLocateObjects to do tracking.

The function

`cv3dTrackerCalibrateCameras`

searches for a checkerboard of the specified size in each of the images. For each image in which it finds the checkerboard, it fills in the corresponding slot in

`camera_info`

with the position and orientation of the camera relative to the checkerboard and sets the

`valid`

flag. If it finds the checkerboard in all the images, it returns true; otherwise it returns false. This function does not change the members of the

`camera_info`

array that correspond to images in which the checkerboard was not found. This allows you to calibrate each

camera independently, instead of simultaneously. To accomplish this, do the following: 1. clear all the

`valid`

flags before calling this function the first time;

`1. call this function with each set of images;`

1. check all the

`valid`

flags after each call. When all the

`valid`

flags are set, calibration is complete. . Note that this method works well only if the checkerboard is rigidly mounted; if it is handheld, all the cameras should be calibrated simultanously to get an accurate result. To ensure that all cameras are calibrated simultaneously, ignore the

`valid`

flags and use the return value to decide when calibration is complete.

[编辑]

3dTrackerLocateObjects

Determines 3d location of tracked objects

```
int  cv3dTrackerLocateObjects(int num_cameras,
         int num_objects,
         const Cv3dTrackerCameraInfo camera_info[],
         const Cv3dTracker2dTrackedObject tracking_info[],
         Cv3dTrackerTrackedObject tracked_objects[]);
```

num_cameras:: the number of cameras.num_objects:: the maximum number of objects found by any camera. (Also the maximum number of objects returned in

`tracked_objects`

.)camera_info:: camera position and location information for each camera, as determined by 3dTrackerCalibrateCameras.tracking_info:: the 2d position of each object as seen by each camera. Although this is specified as a one-dimensional array, it is actually a two-dimensional array:

`const Cv3dTracker2dTrackedObject tracking_info[num_cameras][num_objects]`

. The

`id`

field of any unused slots must be -1. Ids need not be ordered or consecutive.tracked_objects:: filled in with the results. The function

`cv3dTrackerLocateObjects`

determines the 3d position of tracked objects based on the 2d tracking information from multiple cameras and the camera position and orientation information computed by 3dTrackerCalibrateCameras. It locates any objects with the same

`id`

that are tracked by more than one camera. It fills in the

`tracked_objects`

array and returns the number of objects located. The

`id`

fields of any unused slots in

`tracked_objects`

are set to -1.

[编辑]

Eigen Objects (PCA) Functions

The functions described in this section do PCA analysis and compression for a set of 8-bit images that may not fit into memory all together. If your data fits into memory and the vectors are not 8-bit (or you want a simpler interface), use cvCalcCovarMatrix, cvSVD and cvGEMM to do PCA

[编辑]

CalcCovarMatrixEx

Calculates covariance matrix for group of input objects

```
void cvCalcCovarMatrixEx( int object_count, void* input, int io_flags,
                          int iobuf_size, uchar* buffer, void* userdata,
                          IplImage* avg, float* covar_matrix );
```

object_count:: Number of source objects.input:: Pointer either to the array of

`IplImage`

input objects or to the read callback function according to the value of the parameter

`ioFlags`

.io_flags:: Input/output flags.iobuf_size:: Input/output buffer size.buffer:: Pointer to the input/output buffer.userdata:: Pointer to the structure that contains all necessary data for thecallback:: functions.avg:: Averaged object.covar_matrix:: Covariance matrix. An output parameter; must be allocated before the call. The function

`cvCalcCovarMatrixEx`

calculates a covariance matrix of the input objects group using previously calculated averaged object. Depending on

`ioFlags`

parameter it may be used either in direct access or callback mode. If

`ioFlags`

is not

`CV_EIGOBJ_NO_CALLBACK`

, buffer must be allocated before calling the function.

[编辑]

CalcEigenObjects

Calculates orthonormal eigen basis and averaged object for group of input objects

```
void cvCalcEigenObjects( int nObjects, void* input, void* output, int ioFlags,
                         int ioBufSize, void* userData, CvTermCriteria* calcLimit,
                         IplImage* avg, float* eigVals );
```

nObjects:: Number of source objects.

input:: Pointer either to the array of IplImage input objects or to the read callback function according to the value of the parameter

```
ioFlags output:: Pointer either to the array of eigen objects or to the write callback function
according to the value of the parameter ioFlags .ioFlags:: Input/output flags.ioBufSize::
Input/output buffer size in bytes. The size is zero, if unknown.userData:: Pointer to the
structure that contains all necessary data for the callback functions.calcLimit:: Criteria that
determine when to stop calculation of eigen objects.avg:: Averaged object.eigVals:: Pointer to
the eigenvalues array in the descending order; may be <pre>NULL
```

. The function

```
cvCalcEigenObjects
```

calculates orthonormal eigen basis and the averaged object for a group of the input objects. Depending on

```
ioFlags
```

parameter it may be used either in direct access or callback mode. Depending on the parameter

```
calcLimit
```

, calculations are finished either after first

```
calcLimit.max_iter
```

dominating eigen objects are retrieved or if the ratio of the current eigenvalue to the largest eigenvalue comes down to

```
calcLimit.epsilon
```

threshold. The value

```
calcLimit -> type
```

must be

```
CV_TERMCRIT_NUMB, CV_TERMCRIT_EPS
```

, or

```
CV_TERMCRIT_NUMB | CV_TERMCRIT_EPS
```

. The function returns the real values

```
calcLimit->max_iter
```

and

```
calcLimit->epsilon
```

.

The function also calculates the averaged object, which must be created previously. Calculated eigen objects are arranged according to the corresponding eigenvalues in the descending order.

. The parameter

```
eigVals
```

may be equal to

`NULL`

, if eigenvalues are not needed. The function

`cvCalcEigenObjects`

uses the function cvCalcCovarMatrixEx.

[编辑]

CalcDecompCoeff

Calculates decomposition coefficient of input object

```
double cvCalcDecompCoeff( IplImage* obj, IplImage* eigObj, IplImage* avg );
```

obj:: Input object.eigObj:: Eigen object.avg:: Averaged object.

The function

`cvCalcDecompCoeff`

calculates one decomposition coefficient of the input object using the previously calculated eigen object and the averaged object.

[编辑]

EigenDecomposite

Calculates all decomposition coefficients for input object

```
void cvEigenDecomposite( IplImage* obj, int eigenvec_count, void* eigInput,
                         int ioFlags, void* userData, IplImage* avg, float* coeffs );
```

obj:: Input object.eigenvec_count:: Number of eigen objects.eigInput:: Pointer either to the array of

`IplImage`

input objects or to the read callback function according to the value of the parameter

`ioFlags`

.ioFlags:: Input/output flags.userData:: Pointer to the structure that contains all necessary data for the callback functions.avg:: Averaged object.coeffs:: Calculated coefficients; an output parameter. The function

`cvEigenDecomposite`

calculates all decomposition coefficients for the input object using the previously calculated eigen objects basis and the averaged object. Depending on

`ioFlags`

parameter it may be used either in direct access or callback mode.

[编辑]

EigenProjection

Calculates object projection to the eigen sub-space

```
void cvEigenProjection( void* input_vecs, int eigenvec_count, int io_flags, void* userdata,
```

```
float* coeffs, IplImage* avg, IplImage* proj );
```

input_vec:: Pointer to either an array of

`IplImage`

input objects or to a callback function, depending on

`io_flags`

.eigenvec_count:: Number of eigenvectors.io_flags:: Input/output flags; see cvCalcEigenObjects.userdata:: Pointer to the structure that contains all necessary data for the callback functions.coeffs:: Previously calculated decomposition coefficients.avg:: Average vector, calculated by cvCalcEigenObjects.proj:: Projection to the eigen sub-space. The function

`cvEigenProjection`

calculates an object projection to the eigen sub-space or, in other words, restores an object using previously calculated eigen objects basis, averaged object, and decomposition coefficients of the restored object. Depending on

`io_flags`

parameter it may be used either in direct access or callback mode.

[编辑]

Embedded Hidden Markov Models Functions

In order to support embedded models the user must define structures to represent 1D HMM and 2D embedded HMM model.

[编辑]

CvHMM

Embedded HMM Structure

```
typedef struct _CvEHMM
{
    int level;
    int num_states;
    float* transP;
    float** obsProb;
    union
    {
        CvEHMMState* state;
        struct _CvEHMM* ehmm;
    } u;
} CvEHMM;
```

level:: Level of embedded HMM. If

`level ==0`

, HMM is most external. In 2D HMM there are two types of HMM: 1 external and several embedded. External HMM has

`level ==1`

, embedded HMMs have

`level ==0`

.num_states:: Number of states in 1D HMM.transP:: State-to-state transition probability, square matrix

```
(num_state×num_state )
```

.obsProb:: Observation probability matrix.state:: Array of HMM states. For the last-level HMM, that is, an HMM without embedded HMMs, HMM states are real.ehmm:: Array of embedded HMMs. If HMM is not last-level, then HMM states are not real and they are HMMs.

For representation of observations the following structure is defined:

CvImgObsInfo

Image Observation Structure

```
typedef struct CvImgObsInfo
{
    int obs_x;
    int obs_y;
    int obs_size;
    float** obs;
    int* state;
    int* mix;
} CvImgObsInfo;
```

obs_x:: Number of observations in the horizontal direction.obs_y:: Number of observations in the vertical direction.obs_size:: Length of every observation vector.obs:: Pointer to observation vectors stored consequently. Number of vectors is

```
obs_x*obs_y
```

.state:: Array of indices of states, assigned to every observation vector.mix:: Index of mixture component, corresponding to the observation vector within an assigned state.

obs_x 水平方向的观测向量,obs_垂直方向的观测和向量。obs_size:每个观测向量的长度。obs

指向储存观测向量数:大小是:obs_x*obs_y。state:状态列,赋值给第个观测各量。mix:各部分组成的索引。确保观测向量在一个指派值内。

Create2DHMM

Creates 2D embedded HMM

```
CvEHMM* cvCreate2DHMM( int* stateNumber, int* numMix, int obsSize );
```

stateNumber:: Array, the first element of the which specifies the number of superstates in the HMM. All subsequent elements specify the number of states in every embedded HMM, corresponding to each superstate. So, the length of the array is

```
stateNumber [0]+1
```

.numMix:: Array with numbers of Gaussian mixture components per each internal state. The number of elements in the array is equal to number of internal states in the HMM, that is, superstates are not counted here.obsSize:: Size of observation vectors to be used with created HMM. The function

```
cvCreate2DHMM
```

returns the created structure of the type CvEHMM with specified parameters.

[编辑]

Release2DHMM

Releases 2D embedded HMM

```
void cvRelease2DHMM(CvEHMM** hmm );
```

hmm:: Address of pointer to HMM to be released.

The function

```
cvRelease2DHMM
```

frees all the memory used by HMM and clears the pointer to HMM.

[编辑]

CreateObsInfo

Creates structure to store image observation vectors

```
CvImgObsInfo* cvCreateObsInfo( CvSize numObs, int obsSize );
```

numObs:: Numbers of observations in the horizontal and vertical directions. For the given image and scheme of extracting observations the parameter can be computed via the macro

```
CV_COUNT_OBS( roi, dctSize, delta, numObs )
```

, where

```
roi, dctSize, delta, numObs
```

are the pointers to structures of the type CvSize . The pointer

```
roi
```

means size of

```
roi
```

of image observed,

```
numObs
```

is the output parameter of the macro.obsSize:: Size of observation vectors to be stored in the structure. The function

```
cvCreateObsInfo
```

creates new structures to store image observation vectors. For definitions of the parameters

```
roi, dctSize
```

, and

```
delta
```

see the specification of The function

```
cvImgToObs_DCT
```

.

---

numObs:: 一个储存着水平分量与垂直分量的数组。

[编辑]

ReleaseObsInfo

Releases observation vectors structure

```
void cvReleaseObsInfo( CvImgObsInfo** obsInfo );
```

obsInfo:: Address of the pointer to the structure CvImgObsInfo .

The function

```
cvReleaseObsInfo
```

frees all memory used by observations and clears pointer to the structure CvImgObsInfo .

[编辑]

ImgToObs_DCT

Extracts observation vectors from image

```
void cvImgToObs_DCT( IplImage* image, float* obs, CvSize dctSize,
                     CvSize obsSize, CvSize delta );
```

image:: Input image.obs:: Pointer to consequently stored observation vectors.dctSize:: Size of image blocks for which DCT (Discrete Cosine Transform) coefficients are to be computed.obsSize:: Number of the lowest DCT coefficients in the horizontal and vertical directions to be put into the observation vector.delta:: Shift in pixels between two consecutive image blocks in the horizontal and vertical directions.

The function

```
cvImgToObs_DCT
```

extracts observation vectors, that is, DCT coefficients, from the image. The user must pass

```
obsInfo.obs
```

as the parameter

```
obs
```

to use this function with other HMM functions and use the structure

```
obsInfo
```

of the CvImgObsInfo type.

```
Calculating Observations for HMM
    CvImgObsInfo* obs_info;
        ...
        cvImgToObs_DCT( image,obs_info->obs, //!!!
        dctSize, obsSize, delta );
```

[编辑]

UniformImgSegm

Performs uniform segmentation of image observations by HMM states

```
void cvUniformImgSegm( CvImgObsInfo* obsInfo, CvEHMM* hmm );
```

obsInfo:: Observations structure.hmm:: HMM structure.

The function

`cvUniformImgSegm`

segments image observations by HMM states uniformly (see __Initial Segmentation__ for 2D Embedded HMM for 2D embedded HMM with 5 superstates and 3, 6, 6, 6, 3 internal states of every corresponding superstate).

Initial Segmentation for 2D Embedded HMM

[编辑]

InitMixSegm

Segments all observations within every internal state of HMM by state mixture components

```
void cvInitMixSegm( CvImgObsInfo** obsInfoArray, int numImg, CvEHMM* hmm );
```

obsInfoArray:: Array of pointers to the observation structures.numImg:: Length of above array.hmm:: HMM.

The function

`cvInitMixSegm`

takes a group of observations from several training images already segmented by states and splits a set of observation vectors within every internal HMM state into as many clusters as the number of mixture components in the state.

[编辑]

EstimateHMMStateParams

Estimates all parameters of every HMM state

```
void cvEstimateHMMStateParams( CvImgObsInfo** obsInfoArray, int numImg, CvEHMM* hmm );
```

obsInfoArray:: Array of pointers to the observation structures.numImg:: Length of the array.hmm:: HMM.

The function

`cvEstimateHMMStateParams`

computes all inner parameters of every HMM state, including Gaussian means, variances, etc.

[编辑]

EstimateTransProb

Computes transition probability matrices for embedded HMM

```
void cvEstimateTransProb( CvImgObsInfo** obsInfoArray, int numImg, CvEHMM* hmm );
```

obsInfoArray:: Array of pointers to the observation structures.numImg:: Length of the above array.hmm:: HMM.

The function

`cvEstimateTransProb`

uses current segmentation of image observations to compute transition probability matrices for all embedded and external HMMs.

EstimateObsProb

Computes probability of every observation of several images

```
void cvEstimateObsProb( CvImgObsInfo* obsInfo, CvEHMM* hmm );
```

obsInfo:: Observation structure.hmm:: HMM structure.

The function

`cvEstimateObsProb`

computes Gaussian probabilities of each observation to occur in each of the internal HMM states.

EViterbi

Executes Viterbi algorithm for embedded HMM

```
float cvEViterbi( CvImgObsInfo* obsInfo, CvEHMM* hmm );
```

obsInfo:: Observation structure.hmm:: HMM structure.

The function

`cvEViterbi`

executes Viterbi algorithm for embedded HMM. Viterbi algorithm evaluates the likelihood of the best match between the given image observations and the given HMM and performs segmentation of image observations by HMM states. The segmentation is done on the basis of the match found.

MixSegmL2

Segments observations from all training images by mixture components of newly assigned states

```
void cvMixSegmL2( CvImgObsInfo** obsInfoArray, int numImg, CvEHMM* hmm );
```

obsInfoArray:: Array of pointers to the observation structures.numImg:: Length of the array.hmm:: HMM.

The function

`cvMixSegmL2`

segments observations from all training images by mixture components of newly Viterbi algorithm-assigned states. The function uses Euclidean distance to group vectors around the existing mixtures centers.

CvvImage类参考手册

Wikipedia，自由的百科全书

目录

[隐藏]

[编辑]

CvvImage使用说明和注意事项

由于CvvImage是在 highgui.h 头文件中声明的，因此如果您的程序中需要使用，则必须在开头包含此头文件

```
#include <highgui.h>
```

CvvImage对应CImage宏:

```
#define CImage CvvImage
```

注意事项:

- 由于CImage太常见， 很容易造成冲突， 因此建议不要使用该宏(可以直接删去此宏定义)。
- 警告：参数中含有HDC（注：一种windows系统下定义的变量类型，用来描述设备描述表的句柄类型）类型的并不能保证移植到其他平台，例如Show/DrawToHDC等。
- 后文中的**DC**，即device context（设备环境），一般可以理解为windows操作系统为方便绘图而抽象的”绘图表面“，“往窗口上绘图”，有时也被说成是“往窗口DC上绘图”。

[编辑]

CvvImage::Create

```
bool CvvImage::Create(int w, int h, int bpp, int origin);
```

创建一个图像。 成功返回true， 失败返回false。

w
: 图像宽

h
　　图像高
bpp
　　每个像素的bit数， 值等于像素深度乘以通道数
origin
　　0 - 顶—左结构， 1 - 底—左结构 (Windows bitmaps 风格)

例:

```
// 创建一个400行600列的， IPL_DEPTH_8U类型的3通道图像， 顶—左结构

CvvImage img;
bool flag = img.Create(600, 400, IPL_DEPTH_8U*3, 0);
if(!flag)
    printf("创建图像失败!");
```

[编辑]

CvvImage::CopyOf

```
void  CvvImage::CopyOf(CvvImage& img, int desired_color);
void  CvvImage::CopyOf(IplImage* img, int desired_color);
```

从img复制图像到当前的对象中。

img
　　要复制的图像。
desired_color
　　为复制后图像的通道数， 复制后图像的像素深度为8bit。

例:

```
// 读一个图像， 然后复制为1个3通道的彩色图像

CvvImage img1, img2;

img1.Load("example.tiff");
img2.CopyOf(img1, 3);
```

[编辑]

CvvImage::Load

```
bool CvvImage::Load(const char* filename, int desired_color);
```

装载一个图像。

filename
　　图像文件名称。
desired_color
　　图像波段数， 和cvLoadImage类似。

[编辑]

CvvImage::LoadRect

```
bool CvvImage::LoadRect(const char* filename, int desired_color, CvRect rect);
```

从图像读出一个区域。

filename
　　图像名称。

desired_color
　　图像波段数， 和cvLoadImage类似。
rect
　　要读取的图像范围。

注
　　LoadRect是先用Load装载一个图像， 然后再将rect设置为图像的ROI区域， 然后复制图像得到，因此，如果一个图像很大(例如几百MB)， 即使想从中只读几个像素也是会失败的。

[编辑]

CvvImage::Save

```
bool  CvvImage::Save(const char* filename);
```

保存图像。 和cvSaveImage相似。

[编辑]

CvvImage::Show

```
void  CvvImage::Show(const char* window);
```

显示一个图像。 和cvShowImage相似。

[编辑]

CvvImage::Show

```
void  CvvImage::Show(HDC dc, int x, int y, int w, int h, int from_x, int from_y);
```

绘制图像的部分到DC。 图像没有缩放。此函数仅在Windows下有效。

dc
　　设备描述符。
x
　　局部图像显示在DC上,从DC上的第x列开始。
y
　　局部图像显示在DC上,从DC上的第y列开始。
(x,y)为局部图像显示在DC上的起始位置。
w
　　局部图像宽度。
h
　　局部图像高度。
from_x
　　从图像的第from_x列开始显示。
from_y
　　从图像的第from_y行开始显示。

例:

```
// 从图像10行20列开始， 显示400行600列图像到设备描述符的100行200列开始的位置

CvvImage img;
img.Load("example.tiff");

img.Show(hDC, 200, 100, 600, 400, 20, 10);
```

[编辑]

CvvImage::DrawToHDC

```
void  CImage::DrawToHDC(HDC hDCDst, RECT* pDstRect);
```

绘制图像的ROI区域到DC的pDstRect， 如果图像大小和pDstRect不一致， 图像会拉伸/压缩。此函数仅在Windows下有效。

hDCDst
　　设备描述符。
pDstRect
　　对应的设备描述符区域。

例:

```
CvvImage img;
img.Load("example.tiff");

CRect rect;
rect.left = 100;
rect.top = 200;
rect.right = rect.left + 600;
rect.bottom = rect.top + 400;

img.DrawToHDC(hDC, &rect);
```

[编辑]

CvvImage::Fill

void CvvImage::Fill(int color);

以color颜色填充图像。

[编辑]

CvvImage类定义

```
/* CvvImage class definition */
class CV_EXPORTS CvvImage
{
public:
    CvvImage();
    virtual ~CvvImage();

    /* Create image (BGR or grayscale) */
    virtual bool  Create( int width, int height, int bits_per_pixel, int image_origin = 0 );

    /* Load image from specified file */
    virtual bool  Load( const char* filename, int desired_color = 1 );

    /* Load rectangle from the file */
    virtual bool  LoadRect( const char* filename,
                            int desired_color, CvRect r );

#ifdef WIN32
    virtual bool  LoadRect( const char* filename,
                            int desired_color, RECT r )
    {
        return LoadRect( filename, desired_color,
                         cvRect( r.left, r.top, r.right - r.left, r.bottom - r.top ));
    }
#endif

    /* Save entire image to specified file. */
    virtual bool  Save( const char* filename );

    /* Get copy of input image ROI */
    virtual void  CopyOf( CvvImage& image, int desired_color = -1 );
    virtual void  CopyOf( IplImage* img, int desired_color = -1 );
```

```
    IplImage* GetImage() { return m_img; };
    virtual void  Destroy(void);

    /* width and height of ROI */
    int Width() { return !m_img ? 0 : !m_img->roi ? m_img->width : m_img->roi->width; };
    int Height() { return !m_img ? 0 : !m_img->roi ? m_img->height : m_img->roi->height;};
    int Bpp() { return m_img ? (m_img->depth & 255)*m_img->nChannels : 0; };

    virtual void  Fill( int color );

    /* draw to highgui window */
    virtual void  Show( const char* window );

#ifdef WIN32
    /* draw part of image to the specified DC */
    virtual void  Show( HDC dc, int x, int y, int width, int height,
                        int from_x = 0, int from_y = 0 );
    /* draw the current image ROI to the specified rectangle of the destination DC */
    virtual void  DrawToHDC( HDC hDCDst, RECT* pDstRect );
#endif

protected:

    IplImage*  m_img;
```

CvImage类参考手册

Wikipedia，自由的百科全书

CvImage使用前需要包含 cv.h 头文件

```
#include <cv.h>
```

目录

[隐藏]

[编辑]

CvImage::CvImage

```
bool CvImage::CvImage();
bool CvImage::CvImage(CvSize size, int depth, int channels);
bool CvImage::CvImage(IplImage* pIplImg);
bool CvImage::CvImage(const CvImage& cvImg);
bool CvImage::CvImage(const char* filename, const char* imgname=0, int color=-1);
bool CvImage::CvImage(CvFileStorage* fs, const char* mapname, const char* imgname);
```

```
bool CvImage::CvImage(CvFileStorage* fs, const char* seqname, int idx);
```

默认构造函数，创建一个图像。影象对应的数据在析构的时候自动被释放。

size
　　图像大小
depth
　　像素深度
channels
　　通道数

pIplImg
　　IplImage结构影象
cvImg
　　CvImage对象的引用

filename
　　暂无
imgname
　　暂无
color
　　暂无
fs
　　暂无
mapname
　　暂无
seqname
　　暂无
idx
　　暂无

[编辑]

CvImage::~CvImage

```
CvImage::~CvImage();
```

析构函数。

[编辑]

CvImage::clone

```
CvImage CvImage::clone();
```

生成当前影象的一个copy。

[编辑]

CvImage::create

```
void CvImage::create(CvSize size, int depth, int channels);
```

创建一个影象。

size

图像大小

depth

像素深度

channels

通道数

[编辑]

CvImage::release

```
void CvImage::release();
```

释放一个打开的影象。

[编辑]

CvImage::clear

```
void CvImage::clear();
```

释放一个打开的影象。和release功能相同。

[编辑]

CvImage::attach

```
void CvImage::attach(IplImage* img, bool use_refcount=true);
```

将一个影象和当前对象绑定。如果使用引用计数，则引用计数增加1。 当引用计数减少到0的时候释放img对应的内存。

img

图像数据

use_refcount

是否使用引用计数

[编辑]

CvImage::detach

```
void CvImage::detach();
```

取消当前对象绑定的数据。如果使用引用计数，则引用计数减少1。 当引用计数减少到0的时候释放img对应的内存。

[编辑]

CvImage::load

```
bool CvImage::load(const char* filename, const char* imgname=0, int color=-1);
```

暂无。

[编辑]

CvImage::read

```
bool CvImage::read(CvFileStorage* fs, const char* mapname, const char* imgname );
bool CvImage::read(CvFileStorage* fs, const char* seqname, int idx);
```

暂无。

[编辑]

CvImage::save

```
void CvImage::save(const char* filename, const char* imgname);
```

暂无。

[编辑]

CvImage::write

```
void CvImage::write(CvFileStorage* fs, const char* imgname);
```

暂无。

[编辑]

CvImage::show

```
void CvImage::show(const char* window_name);
```

在指定窗口中显示图像。

window_name
窗口的名字。

[编辑]

CvImage::is_valid

```
bool CvImage::is_valid();
```

当前对象是否已经帮定了有效的影象。是返回true，否返回false。

[编辑]

CvImage::width

```
int CvImage::width()const;
```

返回影象的宽度。没有影象的话返回0。

[编辑]

CvImage::height

```
int CvImage::height()const;
```

返回影象的高度。没有影象的话返回0。

[编辑]

CvImage::size

```
CvSize CvImage::size()const;
```

返回影象的尺寸。没有影象的话返回cvSize(0,0)。

[编辑]

CvImage::roi_size

```
CvSize CvImage::roi_size()const;
```

返回影象的ROI区域大小。没有影象的话返回cvSize(0,0)。

[编辑]

CvImage::roi

```
CvRect CvImage::roi()const;
```

返回影象的ROI区域。没有影象的话返回cvRect(0,0,0,0)。

[编辑]

CvImage::coi

```
int CvImage::coi()const;
```

返回影象的COI。没有影象的话返回0。

[编辑]

CvImage::set_roi

```
void CvImage::set_roi(CvRect roi);
```

设置影象的ROI。

roi
: ROI区域。

[编辑]

CvImage::reset_roi

```
void CvImage::reset_roi();
```

重置影象的ROI。

[编辑]

CvImage::set_coi

```
void CvImage::set_coi();
```

返回影象的COI。

[编辑]

CvImage::depth

```
int CvImage::depth();
```

返回像素的深度。没有影象的话返回0。

[编辑]

CvImage::channels

```
int CvImage::channels();
```

返回影象通道数。没有影象的话返回0。

[编辑]

CvImage::pix_size

```
int CvImage::pix_size();
```

返回影象像素的大小。值等于像素深度乘以通道数。

[编辑]

CvImage::data

```
uchar* CvImage::data();
const uchar* CvImage::data()const;
```

获取对应对应的影象数据地址。没有影象的话返回NULL。

[编辑]

CvImage::step

```
int CvImage::step()const;
```

返回IplImage::widthStep，没有影象的话返回0。

[编辑]

CvImage::origin

```
int CvImage::origin()const;
```

返回影象结构。0-顶—左结构，1-底—左结构 (Windows bitmaps 风格)。

[编辑]

CvImage::roi_row

```
uchar* CvImage::roi_row(int y);
const uchar* CvImage::roi_row(int y)const;
```

返回第y行数据。没有影象的话返回NULL。

y
　　影象行数。

[编辑]

运算符重载

```
operator const IplImage* ();
operator IplImage* ();
CvImage& operator=(const CvImage& img);
```

CvImage中的陷阱和BUG

Wikipedia，自由的百科全书

目录

[隐藏]

[编辑]

CvImage类的定义

```
class CV_EXPORTS CvImage
{
public:

        CvImage() : image(0), refcount(0) {}

        CvImage( CvSize size, int depth, int channels )
        {
                image = cvCreateImage( size, depth, channels );
                refcount = image ? new int(1) : 0;
        }

        CvImage( IplImage* img ) : image(img)
        {
                refcount = image ? new int(1) : 0;
        }

        ~CvImage()
        {
                if( refcount && !(--*refcount) )
                {
                        cvReleaseImage( &image );
                        delete refcount;
                }
        }

        void attach( IplImage* img, bool use_refcount=true )
        {
                if( refcount )
                {
                        if( --*refcount == 0 )
                                cvReleaseImage( &image );
                        delete refcount;
                }
                image = img;
                refcount = use_refcount && image ? new int(1) : 0;
        }

        void detach()
        {
                if( refcount )
```

```
                {
                        if( --*refcount == 0 )
                                cvReleaseImage( &image );
                        delete refcount;
                        refcount = 0;
                }
                image = 0;
        }


        CvImage& operator = (const CvImage& img)
        {
                if( img.refcount )
                        ++*img.refcount;
                if( refcount && !(--*refcount) )
                        cvReleaseImage( &image );
                image=img.image;
                refcount=img.refcount;
                return *this;
        }

protected:
        IplImage* image;        // 实际影象
        int* refcount;                  // 引用计数
};
```

CvImage类的相关代码在以下位置:

```
OpenCV\cxcore\include\cxcore.hpp
OpenCV\cxcore\src\cximage.cpp
```

这里给出的只是部分函数。

为了提高效率，CvImage采用的是引用计数。不过目前的CvImage实现中，引用计数机制存在bug。

[编辑]

关于引用计数

引用计数应该也可以叫写时复制技术。就是在复制一个数据时，先只是简单地复制数据的 指针（地址），只有在数据被修改的时候才真的进行数据的复制操作。写时复制技术对用户 是透明的，也就是说用户可以当作数据是真的复制了。

一般数据（或者是文件，类等）都会对应创建/销毁操作。因此，采用写时复制技术的数据 一般还对应一个计数，记录该数据被别人引用的次数。数据在第一次被创建的时候被设置为1， 以后每次被重复创建则增加1，如果是被销毁则减少1。再销毁数据减少引用计数的时候，如果 记录变为0则真的执行删除数据操作，否则的话只执行逻辑删除。

这里需要注意的一点是，每个引用计数和它对应的数据是绑定的。因此，任何一个引用计数都 不应该独立于数据存在。

[编辑]

CvImage中的引用计数机制

```
class CV_EXPORTS CvImage
{
        IplImage* image;
        int* refcount;
};
```

image指向影像数据的地址，refcount指向影像数据对应的引用计数的地址。需要强调的一点是， refcount指向的引用计数并不属于哪个类，而是属于image指向影像数据！任何将影像数据 和其对应的引用计数分离的操作都是错误的。

[编辑]

CvImage(IplImage* img)陷阱

假设有下面一个段代码：

```
IplImage *pIplImg = cvLoadImage("load.tiff");
{
        CvImage cvImg(pIplImg);
}
cvSaveImage("save.tiff", pIplImg);
```

虽然逻辑上好像没有错误，但再执行到cvSaveImage语句的时候却会产生异常！跟踪调试后发现， 原来pIplImg对应的数据在cvImg析构的时候被释放了！

仔细分析后会发现，CvImage将pIplImg对应的数据和它本身的refcount绑定到一起了。pIplImg 对应的数据虽然不属于CvImage，但是它却依据refcount对其进行管理，直到(*refcount)变为0 的时候私自释放了pIplImg影像。

对于这个问题，我不建议使用引用计数，因此可以将代码修改为：

```
CvImage( IplImage* img, bool use_refcount=false) : image(img)
{
        refcount = use_refcount && image ? new int(1) : 0;
}
```

在默认的时候不使用引用计数机制，用户自己维护img内存空间。

[编辑]

attach问题

```
void attach( IplImage* img, bool use_refcount=true )
{
        if( refcount )
        {
                if( --*refcount == 0 )
                        cvReleaseImage( &image );
                delete refcount;
        }
        image = img;
        refcount = use_refcount && image ? new int(1) : 0;
}
```

attach是将一个IplImage影像绑定到CvImage。其中的一个陷阱和前面的CvImage类似：

```
IplImage *pIplImg = cvLoadImage("load.tiff");
{
        CvImage cvImg;
        cvImg.attach(pIplImg);
}
cvSaveImage("save.tiff", pIplImg);        // 异常
```

处理是方法是把参数use_refcount的默认值改为false。

除了和CvImage类型的陷阱外，attach本身还有一个bug！前面我们分析过，CvImage类中 refcount指向的空间和image指向的空间是绑在一起的。因此，if(--*refcount == 0) 语句中将**cvReleaseImage(&image)**和**delete refcount**分离的操作肯定是错误的！！

假设有以下代码：

```
IplImage *pIplImg = cvLoadImage("load.tiff");
{
        CvImage cvImg;
        cvImg.create(cvSize(600,400), 8, 1);        // 创建一个600*400的单字节单通道影像

        CvImage cvImgX(cvImg);                        // 由cvImg拷贝构造cvImgX
        cvImgX.attach(pIplImg);
```

```
}
cvSaveImage("save.tiff", pIplImg);
```

代码将在执行完cvImgX.attach(pIplImg)语句后发生异常!

分析代码可以发现，cvImg.create先创建了一个影像，同时影像还对应一个引用计数。由于cvImgX 是有cvImg拷贝构造得到，因此cvImgX也保存了和cvImg一样的image和refcount。在接着执行的attach中，cvImgX将refcount指向的空间释放（delete refcount）。注意,cvImgX和cvImg的refcount 对应同一个空间！！那么在，cvImg退出花括弧执行析构函数的时候，delete refcount语句就非法了!

修改bug后的attach代码:

```
void attach( IplImage* img, bool use_refcount=false )        // use_refcount默认值没有修改
{
        if( refcount )
        {
                if( --*refcount == 0 )
                {
                        // 同时释放

                        cvReleaseImage( &image );
                        delete refcount;
                }
        }
        image = img;
        refcount = use_refcount && image ? new int(1) : 0;
}
```

由于**CvImage**中的许多函数都基于**attach**实现，因此没有修改**use_refcount**的默认值。detach中的问题和attach相似，代码修改如下:

```
void detach()
{
        if( refcount )
        {
                if( --*refcount == 0 )
                {
                        // 同时释放

                        cvReleaseImage( &image );
                        delete refcount;
                }
                refcount = 0;
        }
        image = 0;
}
```

[编辑]

重载操作符“=”时的内存泄漏

```
CvImage& operator = (const CvImage& img)
{
        if( img.refcount )
                ++*img.refcount;
        if( refcount && !(--*refcount) )
                cvReleaseImage( &image );
        image=img.image;
        refcount=img.refcount;
        return *this;
}
```

假设有以下类似代码:

```
CvImage cvImg1, cvImg2;

cvImg1.create(cvSize(600,400), 8, 1);
cvImg2.create(cvSize(800,500), 8, 1);

cvImg1 = cvImg2;
```

虽然看着很清晰，但是该代码却存在内存泄漏！分析如下：

cvImg1先创建一个(600,400)大小的影像，默认还对应一个引用计数（refcount指向的空间）。cvImg2 也采用同样的方式创建一个类似的影像。注意：cvImg1和cvImg1中refcount指向的空间是不同的！！

下面执行“=”操作时，cvImg1的image空间被释放（cvReleaseImage(&image)），但是cvImg1的 refcount指向的空间却没有释放！然后，cvImg1的refcount指向了cvImg2的refcount。这样，cvImg1 的refcount指向内存就丢失了！

修改后的代码：

```
CvImage& operator = (const CvImage& img)
{
        if( img.refcount )
                ++*img.refcount;
        if( refcount && !(--*refcount) )
        {
                cvReleaseImage( &image );

                // 释放refcount

                delete refcount;
        }
        image=img.image;
        refcount=img.refcount;
        return *this;
}
```

[编辑]

小节

虽然讲了这么多关于CvImage的陷阱和bug，单主要目的还是为了更好地使用CvImage。这里给出一个 建议：

在将**IplImage**数据和**CvImage**进行绑定，或者是基于**IplImage**数据构造**CvImage**对象的时候， 要清楚是否需要使用**CvImage**的引用计数技术（有哪些好处**/**坏处）。

特别是**attach**默认是采用引用计数的（没改的理由前面已经说明）。

[编辑]

附：修复的**CvImage**

```
class CV_EXPORTS CvImage
{
public:

        CvImage() : image(0), refcount(0) {}

        CvImage( CvSize size, int depth, int channels )
        {
                image = cvCreateImage( size, depth, channels );
                refcount = image ? new int(1) : 0;
        }

        // 修改

        CvImage( IplImage* img, bool use_refcount=false) : image(img)
        {
                refcount = use_refcount && image ? new int(1) : 0;
        }

        ~CvImage()
        {
                if( refcount && !(--*refcount) )
                {
                        cvReleaseImage( &image );
                        delete refcount;
                }
        }
```

```
        // 修改
        void attach( IplImage* img, bool use_refcount=false )         // use_refcount默认值没有修改
        {
                if( refcount )
                {
                        if( --*refcount == 0 )
                        {
                                // 同时释放

                                cvReleaseImage( &image );
                                delete refcount;
                        }
                }
                image = img;
                refcount = use_refcount && image ? new int(1) : 0;
        }

        // 修改

        void detach()
        {
                if( refcount )
                {
                        if( --*refcount == 0 )
                        {
                                // 同时释放

                                cvReleaseImage( &image );
                                delete refcount;
                        }
                        refcount = 0;
                }
                image = 0;
        }

        // 修改

        CvImage& operator = (const CvImage& img)
        {
                if( img.refcount )
                        ++*img.refcount;

                if( refcount && !(--*refcount) )
                {
                        cvReleaseImage( &image );

                        // 释放refcount

                        delete refcount;
                }

                image=img.image;
                refcount=img.refcount;
                return *this;
        }

protected:
        IplImage* image;         // 实际影象
        int* refcount;                  // 引用计数
```

Cv中文参考手册

Wikipedia，自由的百科全书

1. 图像处理
2. 结构分析
3. 运动分析与对象跟踪
4. 模式识别
5. 照相机定标和三维重建

[编辑]

中文翻译者

本文最初版本(0.9.6/beta4)由如下人员翻译：图像处理、结构分析、运动分析和对象跟踪部分由R.Z.LIU翻译，模式识别、照相机定标与三维重建部分由H.M.ZHANG翻译， 全文由Y.C.WEI统一修改校正。

以下人员又对中文版提供了修改（如果您对本页做了修改，请把您的名字加在下面）

- 于仕琪，中科院自动化所自由软件协会
- 张兆翔，中科院自动化所自由软件协会

Cv图像处理

Wikipedia，自由的百科全书

注意:本章描述图像处理和分析的一些函数。大多数函数都是针对两维象素数组的，这里，我们称这些数组为“图像”，但是它们不一定非得是IplImage 结构，也可以是CvMat或者CvMatND结构。

目录

[隐藏]

[编辑]

梯度、边缘和角点

[编辑]

Sobel

使用扩展 Sobel 算子计算一阶、二阶、三阶或混合图像差分

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

src
    输入图像.
dst
    输出图像.
xorder
    x 方向上的差分阶数
yorder
    y 方向上的差分阶数
aperture_size

扩展 Sobel 核的大小，必须是 1, 3, 5 或 7。 除了尺寸为 1， 其它情况下， aperture_size ×aperture_size 可分离内核将用来计算差分。对 aperture_size=1的情况， 使用 3x1 或 1x3 内核 （不进行高斯平滑操作）。这里有一个特殊变量 CV_SCHARR (=-1)，对应 3x3 Scharr 滤波器，可以给出比 3x3 Sobel 滤波更精确的结果。Scharr 滤波器系数是：

$$\begin{bmatrix} -3 & 0 & 3 \\ -10 & 0 & 10 \\ -3 & 0 & 3 \end{bmatrix}$$

对 x-方向 或矩阵转置后对 y-方向。

函数 cvSobel 通过对图像用相应的内核进行卷积操作来计算图像差分：

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

由于Sobel 算子结合了 Gaussian 平滑和微分，所以，其结果或多或少对噪声有一定的鲁棒性。通常情况，函数调用采用如下参数 (xorder=1, yorder=0, aperture_size=3) 或 (xorder=0, yorder=1, aperture_size=3) 来计算一阶 x- 或 y- 方向的图像差分。第一种情况对应：

$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$ 核。

第二种对应：

$$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

或者

$$\begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

核的选则依赖于图像原点的定义 (origin 来自 IplImage 结构的定义)。由于该函数不进行图像尺度变换，所以和输入图像(数组)相比，输出图像(数组)的元素通常具有更大的绝对数值（译者注：即像素的位深）。为 防止溢出，当输入图像是 8 位的，要求输出图像是 16 位的。当然可以用函数 cvConvertScale 或 cvConvertScaleAbs 转换为 8 位的。除了 8-位 图像，函数也接受 32-位 浮点数图像。所有输入和输出图像都必须是单通道的，并且具有相同的图像尺寸或者ROI尺寸。

[编辑]

Laplace

计算图像的 Laplacian 变换

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

src
　　输入图像.
dst
　　输出图像.
aperture_size
　　核大小 (与 cvSobel 中定义一样).

函数 cvLaplace 计算输入图像的 Laplacian变换，方法是先用 sobel 算子计算二阶 x- 和 y- 差分，再求和：

$$dst(x,y) = \frac{d^2 src}{dx^2} + \frac{d^2 src}{dy^2}$$

对 aperture_size=1 则给出最快计算结果，相当于对图像采用如下内核做卷积：

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

类似于 cvSobel 函数，该函数也不作图像的尺度变换，所支持的输入、输出图像类型的组合和cvSobel一致。

[编辑]

Canny

采用 Canny 算法做边缘检测

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

image
　　单通道输入图像.
edges
　　单通道存储边缘的输出图像
threshold1
　　第一个阈值
threshold2
　　第二个阈值
aperture_size
　　Sobel 算子内核大小 (见 cvSobel).

函数 cvCanny 采用 CANNY 算法发现输入图像的边缘而且在输出图像中标识这些边缘。threshold1和threshold2 当中的小阈值用来控制边缘连接，大的阈值用来控制强边缘的初始分割。

- 注意事项：cvCanny只接受单通道图像作为输入。
- 外部链接：经典的canny自调整阈值算法的一个opencv的实现见在OpenCV中自适应确定canny算法的分割门限

[编辑]

PreCornerDetect

计算用于角点检测的特征图，

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

image
　　输入图像.
corners
　　保存候选角点的特征图
aperture_size
　　Sobel 算子的核大小(见cvSobel).

函数 cvPreCornerDetect 计算函数 $D_x^2 D_{yy} + D_y^2 D_{xx} - 2 D_x D_y D_{xy}$ 其中 $D_.$ 表示一阶图像差分，$D_{..}$ 表示二阶图像差分。 角点被认为是函数的局部最大值：

```
// 假设图像格式为浮点数
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 );
```

[编辑]

CornerEigenValsAndVecs

计算图像块的特征值和特征向量，用于角点检测

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

image
　　输入图像.
eigenvv
　　保存结果的数组。必须比输入图像宽 6 倍。
block_size
　　邻域大小 (见讨论).
aperture_size
　　Sobel 算子的核尺寸(见 cvSobel).

对每个象素，函数 cvCornerEigenValsAndVecs 考虑 block_size × block_size 大小的邻域 S(p)，然后在邻域上计算图像差分的相关矩阵:

$$M = \begin{bmatrix} \sum_{S(p)}(\frac{dI}{dx})^2 & \sum_{S(p)}(\frac{dI}{dx} \cdot \frac{dI}{dy})^2 \\ \sum_{S(p)}(\frac{dI}{dx} \cdot \frac{dI}{dy})^2 & \sum_{S(p)}(\frac{dI}{dy})^2 \end{bmatrix}$$

然后它计算矩阵的特征值和特征向量，并且按如下方式(λ1, λ2, x1, y1, x2, y2)存储这些值到输出图像中，其中

　　λ1, λ2 - M 的特征值，没有排序
　　(x1, y1) - 特征向量，对 λ1
　　(x2, y2) - 特征向量，对 λ2

[编辑]

CornerMinEigenVal

计算梯度矩阵的最小特征值，用于角点检测

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

image
　　输入图像.
eigenval
　　保存最小特征值的图像. 与输入图像大小一致
block_size
　　邻域大小 (见讨论 cvCornerEigenValsAndVecs).
aperture_size
　　Sobel 算子的核尺寸(见 cvSobel). 当输入图像是浮点数格式时，该参数表示用来计算差分固定的浮点滤波器的个数.

函数 cvCornerMinEigenVal 与 cvCornerEigenValsAndVecs 类似，但是它仅仅计算和存储每个象素点差分相关矩阵的最小特征值，即前一个函数的 min(λ1, λ2)

[编辑]

CornerHarris

哈里斯（Harris）角点检测

```
void cvCornerHarris( const CvArr* image, CvArr* harris_responce, int block_size, int aperture_size=3, double
k=0.04 );
```

image
　　输入图像。

harris_responce
存储哈里斯（Harris）检测responces的图像。与输入图像等大。
block_size
邻域大小（见关于cvCornerEigenValsAndVecs的讨论）。
aperture_size
扩展 Sobel 核的大小（见 cvSobel）。格式. 当输入图像是浮点数格式时，该参数表示用来计算差分固定的浮点滤波器的个数。
k
harris 检测器的自由参数。参见下面的公式。
函数 cvCornerHarris 对输入图像进行 Harris 边界检测。类似于 cvCornerMinEigenVal 和 cvCornerEigenValsAndVecs。对每个像素，在 block_size*block_size 大小的邻域上，计算其2*2梯度共变矩阵（或相关异变矩阵）M。然后，将
det(M) - k*trace(M)2 （这里2是平方）

保存到输出图像中。输入图像中的角点在输出图像中由局部最大值表示。

[编辑]

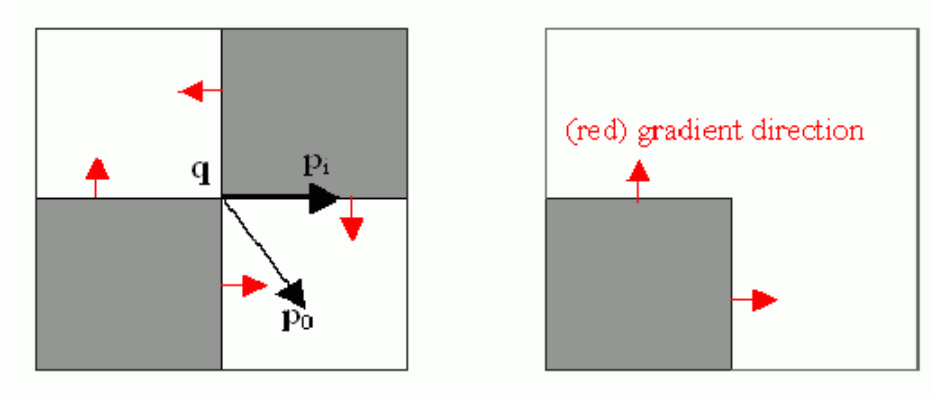
FindCornerSubPix

精确角点位置

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

image
输入图像.
corners
输入角点的初始坐标，也存储精确的输出坐标
count
角点数目
win
搜索窗口的一半尺寸。如果 win=(5,5) 那么使用 5*2+1 × 5*2+1 = 11 × 11 大小的搜索窗口
zero_zone
死区的一半尺寸，死区为不对搜索区的中央位置做求和运算的区域。它是用来避免自相关矩阵出现的某些可能的奇异性。当值为 (-1,-1) 表示没有死区。
criteria
求角点的迭代过程的终止条件。即角点位置的确定，要么迭代数大于某个设定值，或者是精确度达到某个设定值。criteria 可以是最大迭代数目，或者是设定的精确度，也可以是它们的组合。

函数 cvFindCornerSubPix 通过迭代来发现具有子象素精度的角点位置，或如图所示的放射鞍点（radial saddle points）。



子象素级角点定位的实现是基于对向量正交性的观测而实现的，即从中央点q到其邻域点p 的向量和p点处的图像梯度正交（服从图像和测量噪声）。考虑以下的表达式：

```
εi=DIpiT•(q-pi)
```

其中，DIpi表示在q的一个邻域点pi处的图像梯度，q的值通过最小化εi得到。通过将εi设为0，可以建立系统方程如下：

```
sumi(DIpi•DIpiT)•q - sumi(DIpi•DIpiT•pi) = 0
```

其中q的邻域（搜索窗）中的梯度被累加。调用第一个梯度参数G和第二个梯度参数b，得到：

```
q=G-1•b
```

该算法将搜索窗的中心设为新的中心q，然后迭代，直到找到低于某个阈值点的中心位置。

[编辑]

GoodFeaturesToTrack

确定图像的强角点

```
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
　　输入图像，8-位或浮点32-比特，单通道
eig_image
　　临时浮点32-位图像，尺寸与输入图像一致
temp_image
　　另外一个临时图像，格式与尺寸与 eig_image 一致
corners
　　输出参数，检测到的角点
corner_count
　　输出参数，检测到的角点数目
quality_level
　　最大最小特征值的乘法因子。定义可接受图像角点的最小质量因子。
min_distance
　　限制因子。得到的角点的最小距离。使用 Euclidian 距离
mask
　　ROI:感兴趣区域。函数在ROI中计算角点，如果 mask 为 NULL，则选择整个图像。 必须为单通道的灰度图，大小与输入图像相同。mask对应的点不为0，表示计算该点。

函数 cvGoodFeaturesToTrack 在图像中寻找具有大特征值的角点。该函数，首先用cvCornerMinEigenVal 计算输入图像的每一个象素点的最小特征值，并将结果存储到变量 eig_image 中。然后进行非最大值抑制（仅保留3x3邻域中的局部最大值）。下一步将最小特征值小于 quality_level•max(eig_image(x,y)) 排除掉。最后，函数确保所有发现的角点之间具有足够的距离，（最强的角点第一个保留，然后检查新的角点与已有角点之间的距离大于 min_distance ）。

[编辑]

采样、插值和几何变换

[编辑]

InitLineIterator

初始化线段迭代器

```
int cvInitLineIterator( const CvArr* image, CvPoint pt1, CvPoint pt2,
                        CvLineIterator* line_iterator, int connectivity=8 );
```

image
　　带采线段的输入图像.
pt1
　　线段起始点
pt2
　　线段结束点
line_iterator
　　指向线段迭代器状态结构的指针
connectivity
　　被扫描线段的连通数，4 或 8.

函数 cvInitLineIterator 初始化线段迭代器，并返回两点之间的象素点数目。两个点必须在图像内。当迭代器初始化后，连接两点的光栅线上所有点，都可以连续通过调用 CV_NEXT_LINE_POINT 来得到。线段上的点是使用 4－连通或8－连通利用 Bresenham 算法逐点计算的。

例子：使用线段迭代器计算彩色线上象素值的和

```
CvScalar sum_line_pixels( IplImage* image, CvPoint pt1, CvPoint pt2 )
{
    CvLineIterator iterator;
    int blue_sum = 0, green_sum = 0, red_sum = 0;
    int count = cvInitLineIterator( image, pt1, pt2, &iterator, 8 );

    for( int i = 0; i < count; i++ ){
        blue_sum += iterator.ptr[0];
        green_sum += iterator.ptr[1];
        red_sum += iterator.ptr[2];
        CV_NEXT_LINE_POINT(iterator);

        /* print the pixel coordinates: demonstrates how to calculate the coordinates */
        {
        int offset, x, y;
        /* assume that ROI is not set, otherwise need to take it into account. */
        offset = iterator.ptr - (uchar*)(image->imageData);
        y = offset/image->widthStep;
        x = (offset - y*image->widthStep)/(3*sizeof(uchar) /* size of pixel */);
        printf("(%d,%d)\n", x, y );
        }
    }
    return cvScalar( blue_sum, green_sum, red_sum );
}
```

[编辑]

SampleLine

将图像上某一光栅线上的像素数据读入缓冲区

```
int cvSampleLine( const CvArr* image, CvPoint pt1, CvPoint pt2,
                  void* buffer, int connectivity=8 );
```

image
: 输入图像

pt1
: 光栅线段的起点

pt2
: 光栅线段的终点

buffer
: 存储线段点的缓存区，必须有足够大小来存储点 max(|pt2.x-pt1.x|+1, |pt2.y-pt1.y|+1) ： 8－连通情况下，或者 |pt2.x-pt1.x|+|pt2.y-pt1.y|+1 ： 4－连通情况下.

connectivity
: 线段的连通方式, 4 or 8.

函数 cvSampleLine 实现了线段迭代器的一个特殊应用。它读取由 pt1 和 pt2 两点确定的线段上的所有图像点，包括终点，并存储到缓存中。

[编辑]

GetRectSubPix

从图像中提取象素矩形，使用子象素精度

```
void cvGetRectSubPix( const CvArr* src, CvArr* dst, CvPoint2D32f center );
```

src
: 输入图像.

dst
: 提取的矩形.

center
: 提取的象素矩形的中心，浮点数坐标。中心必须位于图像内部.

函数 cvGetRectSubPix 从图像 src 中提取矩形:

```
dst(x, y) = src(x + center.x - (width(dst)-1)*0.5, y + center.y - (height(dst)-1)*0.5)
```

其中非整数象素点坐标采用双线性插值提取。对多通道图像，每个通道独立单独完成提取。尽管函数要求矩形的中心一定要在输入图像之中，但是有可能出现 矩形的一部分超出图像边界的情况，这时，该函数复制边界的模识（hunnish:即用于矩形相交的图像边界线段的象素来代替矩形超越部分的象素）。

[编辑]

GetQuadrangleSubPix

提取象素四边形，使用子象素精度

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

src
　　输入图像.
dst
　　提取的四边形.
map_matrix
　　3 × 2 变换矩阵 [A|b] （见讨论）.

函数 cvGetQuadrangleSubPix 以子象素精度从图像 src 中提取四边形，使用子象素精度，并且将结果存储于 dst ,计算公式是:

$dst(x + width(dst) / 2, y + height(dst) / 2) = src(A_{11}x + A_{12}y + b_1, A21x + A22y + b_2)$

其中 A和 b 均来自映射矩阵(译者注：A, b为几何形变参数) ，映射矩阵为:

$$map\_matrix = \begin{bmatrix} A_{11} & A_{12} & b_1 \\ A_{21} & A_{22} & b_2 \end{bmatrix}$$

其中在非整数坐标 $A \cdot (x, y)^T + b$ 的象素点值通过双线性变换得到。当函数需要图像边界外的像素点时，使用重复边界模式（replication border mode）恢复出所需的值。多通道图像的每一个通道都单独计算。

例子：使用 cvGetQuadrangleSubPix 进行图像旋转

```
#include "cv.h"
#include "highgui.h"
#include "math.h"

int main( int argc, char** argv )
{
    IplImage* src;
    /* the first command line parameter must be image file name */
    if( argc==2 && (src = cvLoadImage(argv[1], -1))!=0)
    {
        IplImage* dst = cvCloneImage( src );
        int delta = 1;
        int angle = 0;

        cvNamedWindow( "src", 1 );
        cvShowImage( "src", src );

        for(;;)
        {
            float m[6];
            double factor = (cos(angle*CV_PI/180.) + 1.1)*3;
            CvMat M = cvMat( 2, 3, CV_32F, m );
            int w = src->width;
            int h = src->height;

            m[0] = (float)(factor*cos(-angle*2*CV_PI/180.));
            m[1] = (float)(factor*sin(-angle*2*CV_PI/180.));
            m[2] = w*0.5f;
            m[3] = -m[1];
            m[4] = m[0];
            m[5] = h*0.5f;

            cvGetQuadrangleSubPix( src, dst, &M, 1, cvScalarAll(0));

            cvNamedWindow( "dst", 1 );
            cvShowImage( "dst", dst );
```

```
            if( cvWaitKey(5) == 27 )
                break;

            angle = (angle + delta) % 360;
        }
    }
    return 0;
}
```

[编辑]

Resize

图像大小变换

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

src
　　输入图像.
dst
　　输出图像.
interpolation
　　插值方法:

- CV_INTER_NN - 最近邻插值,
- CV_INTER_LINEAR - 双线性插值 (缺省使用)
- CV_INTER_AREA - 使用象素关系重采样。当图像缩小时候，该方法可以避免波纹出现。当图像放大时，类似于 CV_INTER_NN 方法..
- CV_INTER_CUBIC - 立方插值.

函数 cvResize 将图像 src 改变尺寸得到与 dst 同样大小。若设定 ROI，函数将按常规支持 ROI.

[编辑]

WarpAffine

对图像做仿射变换

```
void cvWarpAffine( const CvArr* src, CvArr* dst, const CvMat* map_matrix,
                   int flags=CV_INTER_LINEAR+CV_WARP_FILL_OUTLIERS,
                   CvScalar fillval=cvScalarAll(0) );
```

src
　　输入图像.
dst
　　输出图像.
map_matrix
　　2×3 变换矩阵
flags
　　插值方法和以下开关选项的组合:

- CV_WARP_FILL_OUTLIERS - 填充所有输出图像的象素。如果部分象素落在输入图像的边界外，那么它们的值设定为 fillval.
- CV_WARP_INVERSE_MAP - 指定 map_matrix 是输出图像到输入图像的反变换，因此可以直接用来做象素插值。否则, 函数从 map_matrix 得到反变换。

fillval
　　用来填充边界外面的值

函数 cvWarpAffine 利用下面指定的矩阵变换输入图像: $dst(x',y') \leftarrow src(x,y)$

- 如果没有指定 CV_WARP_INVERSE_MAP ， $(x',y')^T = map\_matrix \cdot (x,y,1)^T$,
- 否则， $(x,y)^T = map\_matrix \cdot (x',y',1)^T$

函数与 cvGetQuadrangleSubPix 类似，但是不完全相同。 cvWarpAffine 要求输入和输出图像具有同样的数据类型，有更大

的资源开销（因此对小图像不太合适）而且输出图像的部分可以保留不变。而 cvGetQuadrangleSubPix 可以精确地从8位图像中提取四边形到浮点数缓存区中，具有比较小的系统开销，而且总是全部改变输出图像的内容。

要变换稀疏矩阵，使用 cxcore 中的函数 cvTransform 。

[编辑]

GetAffineTransform

由三对点计算仿射变换

```
CvMat* cvGetAffineTransform( const CvPoint2D32f* src, const CvPoint2D32f* dst, CvMat* map_matrix );
```

src
　　输入图像的三角形顶点坐标。
dst
　　输出图像的相应的三角形顶点坐标。
map_matrix
　　指向2×3输出矩阵的指针。

函数cvGetAffineTransform计算满足以下关系的仿射变换矩阵:

$$(x_i', y_i', 1)^T = map\_matrix \cdot (x_i, y_i, 1)^T$$

　　这里,$dst(i) = (x'_i, y'_i)$,$src(i) = (x_i, y_i)$,$i = 0..2$.

[编辑]

2DRotationMatrix

计算二维旋转的仿射变换矩阵

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

center
　　输入图像的旋转中心坐标
angle
　　旋转角度（度）。正值表示逆时针旋转(坐标原点假设在左上角).
scale
　　各项同性的尺度因子
map_matrix
　　输出 2×3 矩阵的指针

函数 cv2DRotationMatrix 计算矩阵:

```
[  α  β  |  (1-α)*center.x - β*center.y ]
[ -β  α  |  β*center.x + (1-α)*center.y ]

where α=scale*cos(angle), β=scale*sin(angle)
```

该变换并不改变原始旋转中心点的坐标，如果这不是操作目的，则可以通过调整平移量改变其坐标(译者注：通过简单的推导可知，仿射变换的实现是首先将 旋转中心置为坐标原点，再进行旋转和尺度变换，最后重新将坐标原点设定为输入图像的左上角，这里的平移量是center.x, center.y).

[编辑]

WarpPerspective

对图像进行透视变换

```
void cvWarpPerspective( const CvArr* src, CvArr* dst, const CvMat* map_matrix,
                        int flags=CV_INTER_LINEAR+CV_WARP_FILL_OUTLIERS,
                        CvScalar fillval=cvScalarAll(0) );
```

src

输入图像.

dst
输出图像.

map_matrix
3×3 变换矩阵

flags
插值方法和以下开关选项的组合:

- CV_WARP_FILL_OUTLIERS - 填充所有缩小图像的象素。如果部分象素落在输入图像的边界外，那么它们的值设定为 fillval.
- CV_WARP_INVERSE_MAP - 指定 matrix 是输出图像到输入图像的反变换，因此可以直接用来做象素插值。否则, 函数从 map_matrix 得到反变换。

fillval
用来填充边界外面的值

函数 cvWarpPerspective 利用下面指定矩阵变换输入图像: $dst(x',y') \leftarrow src(x,y)$

- 如果没有指定 CV_WARP_INVERSE_MAP , $(x',y')^T = map\_matrix \cdot (x,y,1)^T$,
- 否则, $(x,y)^T = map\_matrix \cdot (x',y',1)^T$

要变换稀疏矩阵，使用 cxcore 中的函数 cvTransform 。

[编辑]

WarpPerspectiveQMatrix

用4个对应点计算透视变换矩阵

```
CvMat* cvWarpPerspectiveQMatrix( const CvPoint2D32f* src,
                                 const CvPoint2D32f* dst,
                                 CvMat* map_matrix );
```

src
输入图像的四边形的4个点坐标

dst
输出图像的对应四边形的4个点坐标

map_matrix
输出的 3×3 矩阵

函数 cvWarpPerspectiveQMatrix 计算透视变换矩阵，使得:

```
(tix'i,tiy'i,ti)T=matrix•(xi,yi,1)T
```

其中 dst(i)=(x'i,y'i), src(i)=(xi,yi), i=0..3.

[编辑]

GetPerspectiveTransform

由四对点计算透射变换

```
CvMat* cvGetPerspectiveTransform( const CvPoint2D32f* src, const CvPoint2D32f* dst,
                                  CvMat* map_matrix );

#define cvWarpPerspectiveQMatrix cvGetPerspectiveTransform
```

src
输入图像的四边形顶点坐标。

dst
输出图像的相应的四边形顶点坐标。

map_matrix
指向3×3输出矩阵的指针。

函数cvGetPerspectiveTransform计算满足以下关系的透射变换矩阵:

$$(t_i x'_i, t_i y'_i, t_i)^T = map\_matrix \cdot (x_i, y_i, 1)^T$$

这里,*dst*(*i*) = (*x'*_{*i*},*y'*_{*i*}),*src*(*i*) = (*x*_{*i*},*y*_{*i*}),*i* = 0..3.

[编辑]

Remap

对图像进行普通几何变换

```
void cvRemap( const CvArr* src, CvArr* dst,
              const CvArr* mapx, const CvArr* mapy,
              int flags=CV_INTER_LINEAR+CV_WARP_FILL_OUTLIERS,
              CvScalar fillval=cvScalarAll(0) );
```

src
输入图像.
dst
输出图像.
mapx
x坐标的映射 (32fC1 image).
mapy
y坐标的映射 (32fC1 image).
flags
插值方法和以下开关选项的组合:

- CV_WARP_FILL_OUTLIERS - 填充边界外的像素. 如果输出图像的部分象素落在变换后的边界外，那么它们的值设定为 fillval。

fillval
用来填充边界外面的值.

函数 cvRemap 利用下面指定的矩阵变换输入图像:

```
dst(x,y)<-src(mapx(x,y),mapy(x,y))
```

与其它几何变换类似，可以使用一些插值方法（由用户指定，译者注：同cvResize）来计算非整数坐标的像素值。

[编辑]

LogPolar

把图像映射到极指数空间

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

src
输入图像。
dst
输出图像。
center
变换的中心，输出图像在这里最精确。
M
幅度的尺度参数，见下面公式。
flags
插值方法和以下选择标志的结合

- CV_WARP_FILL_OUTLIERS -填充输出图像所有像素，如果这些点有和外点对应的，则置零。
- CV_WARP_INVERSE_MAP - 表示矩阵由输出图像到输入图像的逆变换，并且因此可以直接用于像素插值。否

则，函数从map_matrix中寻找逆变换。

fillval
用于填充外点的值。

函数cvLogPolar用以下变换变换输入图像：

正变换 (CV_WARP_INVERSE_MAP 未置位)：

```
dst(phi,rho)<-src(x,y)
```

逆变换 (CV_WARP_INVERSE_MAP 置位)：

```
dst(x,y)<-src(phi,rho),
```

这里，

```
rho=M+log(sqrt(x2+y2))
phi=atan(y/x)
```

此函数模仿人类视网膜中央凹视力，并且对于目标跟踪等可用于快速尺度和旋转变换不变模板匹配。

Example. Log-polar transformation.

```
#include <cv.h>
#include <highgui.h>

int main(int argc, char** argv)
{
    IplImage* src;

    if(argc == 2 && ((src=cvLoadImage(argv[1],1)) != 0 ))
    {
        IplImage* dst = cvCreateImage( cvSize(256,256), 8, 3 );
        IplImage* src2 = cvCreateImage( cvGetSize(src), 8, 3 );
        cvLogPolar( src, dst, cvPoint2D32f(src->width/2,src->height/2), 40,
CV_INTER_LINEAR+CV_WARP_FILL_OUTLIERS );
        cvLogPolar( dst, src2, cvPoint2D32f(src->width/2,src->height/2), 40,
CV_INTER_LINEAR+CV_WARP_FILL_OUTLIERS+CV_WARP_INVERSE_MAP );
        cvNamedWindow( "log-polar", 1 );
        cvShowImage( "log-polar", dst );
        cvNamedWindow( "inverse log-polar", 1 );
        cvShowImage( "inverse log-polar", src2 );
        cvWaitKey();
    }
    return 0;
}
```

And this is what the program displays when opencv/samples/c/fruits.jpg is passed to it

[编辑]

形态学操作

[编辑]

CreateStructuringElementEx

创建结构元素

```
IplConvKernel* cvCreateStructuringElementEx( int cols, int rows, int anchor_x, int anchor_y,
                                             int shape, int* values=NULL );
```

cols
　结构元素的列数目
rows
　结构元素的行数目
anchor_x
　锚点的相对水平偏移量
anchor_y
　锚点的相对垂直偏移量
shape
　结构元素的形状，可以是下列值：

- CV_SHAPE_RECT, 长方形元素;
- CV_SHAPE_CROSS, 交错元素 a cross-shaped element;
- CV_SHAPE_ELLIPSE, 椭圆元素;
- CV_SHAPE_CUSTOM, 用户自定义元素。这种情况下参数 values 定义了 mask,即象素的那个邻域必须考虑。

values
　指向结构元素的指针，它是一个平面数组，表示对元素矩阵逐行扫描。(非零点表示该点属于结构元)。如果指针为空，则表示平面数组中的所有元素都是非零的，即结构元是一个长方形(该参数仅仅当shape参数是 CV_SHAPE_CUSTOM 时才予以考虑)。

函数 cv CreateStructuringElementEx 分配和填充结构 IplConvKernel, 它可作为形态操作中的结构元素。

[编辑]

ReleaseStructuringElement

删除结构元素

```
void cvReleaseStructuringElement( IplConvKernel** element );
```

element
　被删除的结构元素的指针

函数 cvReleaseStructuringElement 释放结构 IplConvKernel 。如果 *element 为 NULL, 则函数不作用。

[编辑]

Erode

使用任意结构元素腐蚀图像

```
void cvErode( const CvArr* src, CvArr* dst, IplConvKernel* element=NULL, int iterations=1 );
```

src
　输入图像.
dst
　输出图像.
element
　用于腐蚀的结构元素。若为 NULL, 则使用 3×3 长方形的结构元素
iterations
　腐蚀的次数

函数 cvErode 对输入图像使用指定的结构元素进行腐蚀，该结构元素决定每个具有最小值象素点的邻域形状：

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

函数可能是本地操作，不需另外开辟存储空间的意思。腐蚀可以重复进行 (iterations) 次. 对彩色图像，每个彩色通道单独处理。

[编辑]

Dilate

使用任意结构元素膨胀图像

```
void cvDilate( const CvArr* src, CvArr* dst, IplConvKernel* element=NULL, int iterations=1 );
```

src
　　输入图像.
dst
　　输出图像.
element
　　用于膨胀的结构元素。若为 NULL, 则使用 3×3 长方形的结构元素
iterations
　　膨胀的次数

函数 cvDilate 对输入图像使用指定的结构元进行膨胀，该结构决定每个具有最小值象素点的邻域形状：

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

函数支持（in-place）模式。膨胀可以重复进行 (iterations) 次. 对彩色图像，每个彩色通道单独处理。

[编辑]

MorphologyEx

高级形态学变换

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

src
　　输入图像.
dst
　　输出图像.
temp
　　临时图像，某些情况下需要
element
　　结构元素
operation
　　形态操作的类型:

　　CV_MOP_OPEN - 开运算
　　CV_MOP_CLOSE - 闭运算
　　CV_MOP_GRADIENT - 形态梯度
　　CV_MOP_TOPHAT - "顶帽"
　　CV_MOP_BLACKHAT - "黑帽"

iterations
　　膨胀和腐蚀次数.

函数 cvMorphologyEx 在膨胀和腐蚀基本操作的基础上，完成一些高级的形态变换：

开运算
　　dst=open(src,element)=dilate(erode(src,element),element)
闭运算
　　dst=close(src,element)=erode(dilate(src,element),element)
形态梯度
　　dst=morph_grad(src,element)=dilate(src,element)-erode(src,element)
"顶帽"
　　dst=tophat(src,element)=src-open(src,element)
"黑帽"
　　dst=blackhat(src,element)=close(src,element)-src

临时图像 temp 在形态梯度以及对“顶帽”和“黑帽”操作时的 in-place 模式下需要。

[编辑]

滤波器与色彩空间变换

[编辑]

Smooth

各种方法的图像平滑

```
void cvSmooth( const CvArr* src, CvArr* dst,
               int smoothtype=CV_GAUSSIAN,
               int param1=3, int param2=0, double param3=0, double param4=0 );
```

src
　　输入图像.
dst
　　输出图像.
smoothtype
　　平滑方法:

- CV_BLUR_NO_SCALE (简单不带尺度变换的模糊) - 对每个象素的 param1×param2 领域求和。如果邻域大小是变化的，可以事先利用函数 cvIntegral 计算积分图像。
- CV_BLUR (simple blur) - 对每个象素param1×param2邻域 求和并做尺度变换 1/(param1•param2).
- CV_GAUSSIAN (gaussian blur) - 对图像进行核大小为 param1×param2 的高斯卷积
- CV_MEDIAN (median blur) - 对图像进行核大小为param1×param1 的中值滤波 (i.e. 邻域是方的).
- CV_BILATERAL (双向滤波) - 应用双向 3x3 滤波，彩色 sigma=param1，空间 sigma=param2. 关于双向滤波，可参考 http://www.dai.ed.ac.uk/CVonline/LOCAL_COPIES/MANDUCHI1/Bilateral_Filtering.html

param1
　　平滑操作的第一个参数.
param2
　　平滑操作的第二个参数. 对于简单/非尺度变换的高斯模糊的情况，如果param2的值 为零，则表示其被设定为param1。
param3
　　对应高斯参数的 Gaussian sigma (标准差). 如果为零，则标准差由下面的核尺寸计算:

```
sigma = (n/2 - 1)*0.3 + 0.8, 其中 n=param1 对应水平核,
                                 n=param2 对应垂直核.
```

对小的卷积核 (3×3 to 7×7) 使用如上公式所示的标准 sigma 速度会快。如果 param3 不为零，而 param1 和 param2 为零，则核大小有 sigma 计算 (以保证足够精确的操作).

函数 cvSmooth 可使用上面任何一种方法平滑图像。每一种方法都有自己的特点以及局限。

没有缩放的图像平滑仅支持单通道图像，并且支持8位到16位的转换(与cvSobel和cvaplace相似)和32位浮点数到32位浮点数的变换格式。

简单模糊和高斯模糊支持 1- 或 3-通道, 8-比特 和 32-比特 浮点图像。这两种方法可以（in-place）方式处理图像。

中值和双向滤波工作于 1- 或 3-通道， 8-位图像，但是不能以 in-place 方式处理图像.

中值滤波
　　中值滤波法是一种非线性平滑技术，它将每一象素点的灰度值设置为该点某邻域窗口内的所有象素点灰度值的中值。实现方法:

1. 通过从图像中的某个采样窗口取出奇数个数据进行排序
2. 用排序后的中值取代要处理的数据即可

中值滤波法对消除椒盐噪音非常有效，在光学测量条纹图象的相位分析处理方法中有特殊作用，但在条纹中心分析方法中

作用不大。中值滤波在图像处理中,常用于用来保护边缘信息,是经典的平滑噪声的方法

中值滤波原理

中值滤波是基于排序统计理论的一种能有效抑制噪声的非线性信号处理技术，中值滤波的基本原理是把数字图像或数字序列中一点的值用该点的一个拎域中各 点值的中值代替，让周围的像素值接近的值，从而消除孤立的噪声点。方法是去某种结构的二维滑动模板，将板内像素按照像素值的大小进行排序，生成单调上升 （或下降）的为二维数据序列。二维中值滤波输出为g（x,y）=med{f(x-k,y-l),(k,l∈W)} ，其中,f(x,y)，g(x,y)分别为原始图像和处理后图像。W为二维模板，通常为2*2，3*3区域，也可以是不同的的形状，如线状，圆形，十字 形，圆环形等。

高斯滤波

高斯滤波实质上是一种信号的滤波器，其用途是信号的平滑处理，我们知道数字图像用于后期应用，其噪声是最大的问题，由于误差会累计传递等原因，很多 图像处理教材会在很早的时候介绍Gauss滤波器，用于得到信噪比SNR较高的图像（反应真实信号）。于此相关的有Gauss-Lapplace变换，其 实就是为了得到较好的图像边缘，先对图像做Gauss平滑滤波，剔除噪声，然后求二阶导矢，用二阶导的过零点确定边缘，在计算时也是频域乘积=>空 域卷积。

滤波器就是建立的一个数学模型，通过这个模型来将图像数据进行能量转化，能量低的就排除掉，噪声就是属于低能量部分

其实编程运算的话就是一个模板运算，拿图像的八连通区域来说，中间点的像素值就等于八连通区的像素值的均值，这样达到平滑的效果

若使用理想滤波器，会在图像中产生振铃现象。采用高斯滤波器的话，系统函数是平滑的，避免了振铃现象。

[编辑]

Filter2D

对图像做卷积

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

src
　　输入图像.
dst
　　输出图像.
kernel
　　卷积核, 单通道浮点矩阵. 如果想要应用不同的核于不同的通道，先用 cvSplit 函数分解图像到单个色彩通道上，然后单独处理。
anchor
　　核的锚点表示一个被滤波的点在核内的位置。 锚点应该处于核内部。缺省值 (-1,-1) 表示锚点在核中心。

函数 cvFilter2D 对图像进行线性滤波，支持 In-place 操作。当核运算部分超出输入图像时，函数从最近邻的图像内部象素插值得到边界外面的象素值。

[编辑]

CopyMakeBorder

复制图像并且制作边界。

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

src
　　输入图像。
dst
　　输出图像。
offset
　　输入图像（或者其ROI）欲拷贝到的输出图像长方形的左上角坐标（或者左下角坐标，如果以左下角为原点）。长方形的尺寸要和原图像的尺寸的ROI分之一匹配。
bordertype

已拷贝的原图像长方形的边界的类型：
IPL_BORDER_CONSTANT - 填充边界为固定值，值由函数最后一个参数指定。IPL_BORDER_REPLICATE -边界用上下行或者左右列来复制填充。（其他两种IPL边界类型， IPL_BORDER_REFLECT 和IPL_BORDER_WRAP现已不支持）。

value
如果边界类型为IPL_BORDER_CONSTANT的话，那么此为边界像素的值。

函数cvCopyMakeBorder拷贝输入2维阵列到输出阵列的内部并且在拷贝区域的周围制作一个指定类型的边界。函数可以用来模拟和嵌入在指 定算法实现中的边界不同的类型。例如：和opencv中大多数其他滤波函数一样，一些形态学函数内部使用复制边界类型，但是用户可能需要零边界或者填充为 1或255的边界。

[编辑]

Integral

计算积分图像

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

image
输入图像, W×H, 单通道，8位或浮点 (32f 或 64f).

sum
积分图像, W+1×H+1(译者注：原文的公式应该写成(W+1)×(H+1),避免误会), 单通道，32位整数或 double 精度的浮点数(64f).

sqsum
对象素值平方的积分图像，W+1×H+1(译者注：原文的公式应该写成(W+1)×(H+1),避免误会), 单通道，32位整数或 double 精度的浮点数 (64f).

tilted_sum
旋转45度的积分图像，单通道，32位整数或 double 精度的浮点数 (64f).

函数 cvIntegral 计算一次或高次积分图像：

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

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

$$tilted\_sum(X,Y) = \sum_{y < Y, |x - X| < y} image(x,y)$$

利用积分图像，可以计算在某象素的上－右方的或者旋转的矩形区域中进行求和、求均值以及标准方差的计算，并且保证运算的复杂度为O(1)。例如：

$$\sum_{x_1 \le x < x_2, y_1 \le y < y_2} image(x,y) = sum(x_2,y_2) - sum(x_1,y_2) - sum(x_2,y_1) + sum(x_1,x_1)$$

因此可以在变化的窗口内做快速平滑或窗口相关等操作。

[编辑]

CvtColor

色彩空間轉換

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

src
輸入的 8-bit , 16-bit 或 32-bit 單倍精度浮點數影像.

dst
輸出的 8-bit , 16-bit 或 32-bit 單倍精度浮點數影像.

code

色彩空間轉換，通過定義 CV_<src_color_space>2<dst_color_space> 常數 (見下面).

函數 cvCvtColor 將輸入圖像從一個色彩空間轉換為另外一個色彩空間。函數忽略 IplImage 頭中定義的 colorModel 和 channelSeq 域，所以輸入圖像的色彩空間應該正確指定 (包括通道的順序，對RGB空間而言，BGR 意味著佈局為 B0 G0 R0 B1 G1 R1 ... 層疊的 24-位格式，而 RGB 意味著佈局為 R0 G0 B0 R1 G1 B1 ... 層疊的24-位格式. 函數做如下變換:

RGB 空間內部的變換，如增加/刪除 alpha 通道，反相通道順序，到16位 RGB彩色或者15位RGB彩色的正逆轉換(Rx5:Gx6:Rx5),以及到灰度圖像的正逆轉換，使用:

```
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
```

所有可能的圖像色彩空間的相互變換公式列舉如下:

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
```

使用上面從 0° 到 360° 變化的公式計算色調（hue）值，確保它們被 2 除後能適用於8位。

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*Y1/3      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)=t1/3              for t>0.008856
      f(t)=7.787*t+16/116   for t<=0.008856
```

上面的公式可以參考 http://www.cica.indiana.edu/cica/faq/color_spaces/color.spaces.html

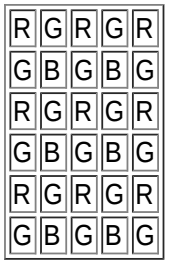
RGB=>HLS (CV_BGR2HLS, CV_RGB2HLS)

HSL 表示 hue(色相)、saturation(饱和度)、lightness(亮度)。有的地方也称为HSI，其中I表示intensity(强度)

转换公式见http://zh.wikipedia.org/wiki/HSL_%E8%89%B2%E5%BD%A9%E7%A9%BA%E9%97%B4

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 模式被廣泛應用於 CCD 和 CMOS 攝像頭. 它允許從一個單獨平面中得到彩色圖像，該平面中的 R/G/B 象素點被安排如下:



對像素輸出的RGB份量由該像素的1、2或者4鄰域中具有相同顏色的點插值得到。以上的模式可以通過向左或者向上平移一個像素點來作一些修改。轉換 常量CV_BayerC1C22{RGB|RGB}中的兩個字母C1和C2表示特定的模式類型：顏色份量分別來自於第二行，第二和第三列。比如說，上述的 模式具有很流行的"BG"類型。

[编辑]

Threshold

對數組元素進行固定閾值操作

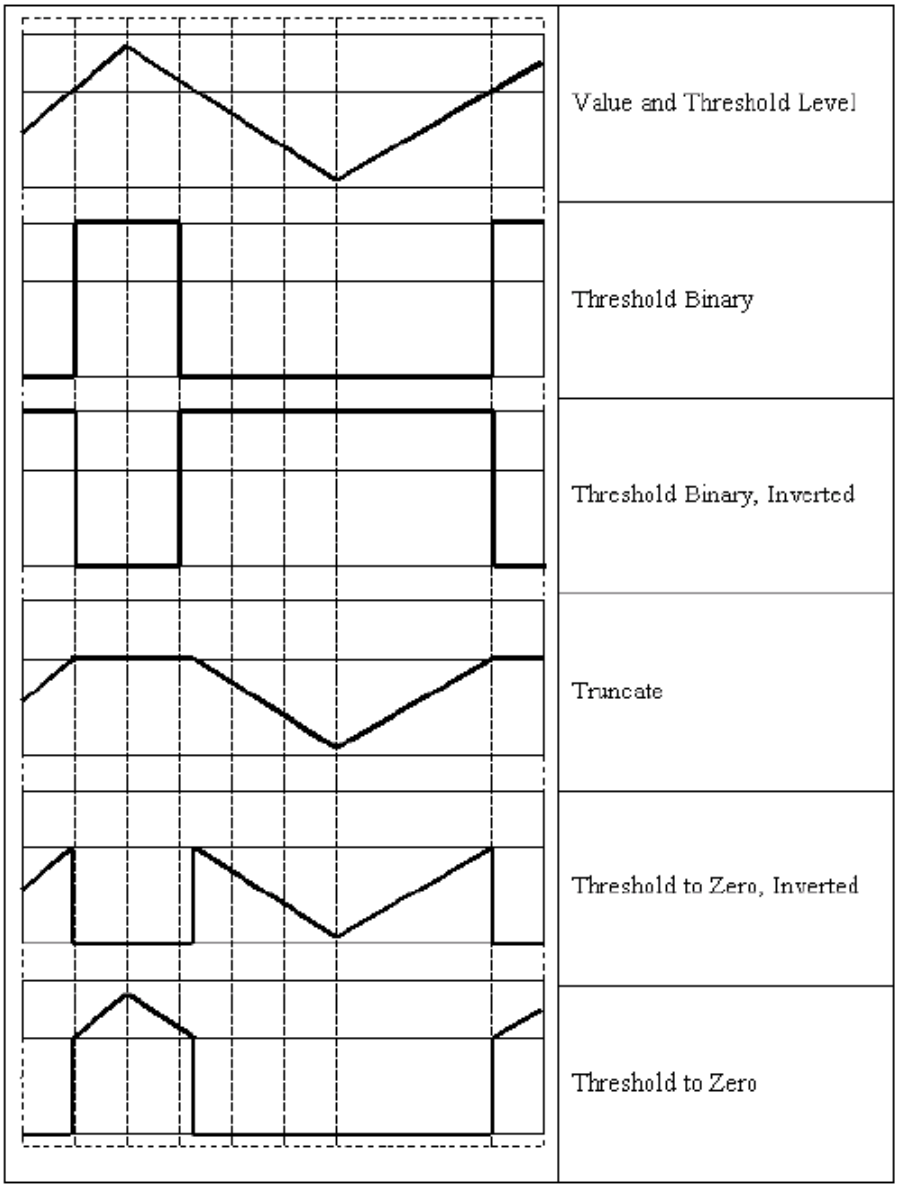
```
void cvThreshold( const CvArr* src, CvArr* dst, double threshold,
                  double max_value, int threshold_type );
```

src
　　原始數組 (單通道 , 8-bit of 32-bit 浮點數).
dst
　　輸出數組，必須與 src 的類型一致，或者為 8-bit.
threshold
　　閾值
max_value
　　使用 CV_THRESH_BINARY 和 CV_THRESH_BINARY_INV 的最大值.
threshold_type
　　閾值類型 (見討論)

函數 cvThreshold 對單通道數組應用固定閾值操作。該函數的典型應用是對灰度圖像進行閾值操作得到二值圖像。(cvCmpS 也可以達到此目的) 或者是去掉噪聲，例如過濾很小或很大象素值的圖像點。本函數支持的對圖像取閾值的方法由 threshold_type 確定:

```
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
```

下面是圖形化的閾值描述:



[编辑]

AdaptiveThreshold

自适应阈值方法

```
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
　　输入图像.
dst
　　输出图像.
max_value
　　使用 CV_THRESH_BINARY 和 CV_THRESH_BINARY_INV 的最大值.
adaptive_method
　　自适应阈值算法使用：CV_ADAPTIVE_THRESH_MEAN_C 或 CV_ADAPTIVE_THRESH_GAUSSIAN_C （见讨论）.
threshold_type
　　取阈值类型：必须是下者之一

- CV_THRESH_BINARY,
- CV_THRESH_BINARY_INV

block_size
用来计算阈值的象素邻域大小: 3, 5, 7, ...
param1
与方法有关的参数。对方法 CV_ADAPTIVE_THRESH_MEAN_C 和 CV_ADAPTIVE_THRESH_GAUSSIAN_C， 它是一个从均值或加权均值提取的常数（见讨论），尽管它可以是负数。

函数 cvAdaptiveThreshold 将灰度图像变换到二值图像，采用下面公式:

```
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
```

其中 TI 是为每一个象素点单独计算的阈值

对方法 CV_ADAPTIVE_THRESH_MEAN_C，先求出块中的均值，再减掉param1。

对方法 CV_ADAPTIVE_THRESH_GAUSSIAN_C ，先求出块中的加权和(gaussian)， 再减掉param1。

[编辑]

金字塔及其应用

[编辑]

PyrDown

图像的下采样

```
void cvPyrDown( const CvArr* src, CvArr* dst, int filter=CV_GAUSSIAN_5x5 );
```

src
输入图像.
dst
输出图像, 宽度和高度应是输入图像的一半 ,传入前必须已经完成初始化
filter
卷积滤波器的类型，目前仅支持 CV_GAUSSIAN_5x5

函数 cvPyrDown 使用 Gaussian 金字塔分解对输入图像向下采样。首先它对输入图像用指定滤波器进行卷积，然后通过拒绝偶数的行与列来下采样图像。

[编辑]

PyrUp

图像的上采样

```
void cvPyrUp( const CvArr* src, CvArr* dst, int filter=CV_GAUSSIAN_5x5 );
```

src
输入图像.
dst
输出图像, 宽度和高度应是输入图像的2倍
filter
卷积滤波器的类型，目前仅支持 CV_GAUSSIAN_5x5

函数 cvPyrUp 使用Gaussian 金字塔分解对输入图像向上采样。首先通过在图像中插入 0 偶数行和偶数列，然后对得到的图像用指定的滤波器进行高斯卷积，其中滤波器乘以4做插值。所以输出图像是输入图像的 4 倍大小。（hunnish: 原理不清楚，尚待探讨）

[编辑]

连接部件

[编辑]

CvConnectedComp

连接部件

```
typedef struct CvConnectedComp
{
    double area; /* 连通域的面积 */
    float value; /* 分割域的灰度缩放值 */
    CvRect rect; /* 分割域的 ROI */
} CvConnectedComp;
```

[编辑]

FloodFill

用指定颜色填充一个连接域

```
void cvFloodFill( CvArr* image, CvPoint seed_point, CvScalar new_val,
                  CvScalar lo_diff=cvScalarAll(0), CvScalar up_diff=cvScalarAll(0),
                  CvConnectedComp* comp=NULL, int flags=4, CvArr* mask=NULL );
#define CV_FLOODFILL_FIXED_RANGE (1 << 16)
#define CV_FLOODFILL_MASK_ONLY   (1 << 17)
```

image
: 输入的 1- 或 3-通道, 8-比特或浮点数图像。输入的图像将被函数的操作所改变，除非你选择 CV_FLOODFILL_MASK_ONLY 选项 (见下面).

seed_point
: 开始的种子点.

new_val
: 新的重新绘制的象素值

lo_diff
: 当前观察象素值与其部件领域象素或者待加入该部件的种子象素之负差(Lower difference)的最大值。对 8-比特 彩色图像，它是一个 packed value.

up_diff
: 当前观察象素值与其部件领域象素或者待加入该部件的种子象素之正差(upper difference)的最大值。 对 8-比特 彩色图像，它是一个 packed value.

comp
: 指向部件结构体的指针，该结构体的内容由函数用重绘区域的信息填充。

flags
: 操作选项. 低位比特包含连通值, 4 (缺省) 或 8, 在函数执行连通过程中确定使用哪种邻域方式。高位比特可以是 0 或下面的开关选项的组合:

  - CV_FLOODFILL_FIXED_RANGE - 如果设置，则考虑当前象素与种子象素之间的差，否则考虑当前象素与其相邻象素的差。(范围是浮点数).
  - CV_FLOODFILL_MASK_ONLY - 如果设置，函数不填充原始图像 (忽略 new_val), 但填充掩模图像 (这种情况下 MASK 必须是非空的).

mask
: 运算掩模, 应该是单通道、8-比特图像, 长和宽上都比输入图像 image 大两个象素点。若非空，则函数使用且更新掩模, 所以使用者需对 mask 内容的初始化负责。填充不会经过 MASK 的非零象素, 例如，一个边缘检测子的输出可以用来作为 MASK 来阻止填充边缘。或者有可能在多次的函数调用中使用同一个 MASK，以保证填充的区域不会重叠。注意: 因为 MASK 比欲填充图像大，所以 mask 中与输入图像(x,y)像素点相对应的点具有(x+1,y+1)坐标。

函数 cvFloodFill 用指定颜色，从种子点开始填充一个连通域。连通性由象素值的接近程度来衡量。在点 (x, y) 的象素被认为是属于重新绘制的区域，如果:

src(x',y')-lo_diff<=src(x,y)<=src(x',y')+up_diff, 灰度图像，浮动范围
src(seed.x,seed.y)-lo<=src(x,y)<=src(seed.x,seed.y)+up_diff, 灰度图像，固定范围

src(x',y')r-lo_diffr<=src(x,y)r<=src(x',y')r+up_diffr 和
src(x',y')g-lo_diffg<=src(x,y)g<=src(x',y')g+up_diffg 和
src(x',y')b-lo_diffb<=src(x,y)b<=src(x',y')b+up_diffb, 彩色图像，浮动范围

src(seed.x,seed.y)r-lo_diffr<=src(x,y)r<=src(seed.x,seed.y)r+up_diffr 和
src(seed.x,seed.y)g-lo_diffg<=src(x,y)g<=src(seed.x,seed.y)g+up_diffg 和
src(seed.x,seed.y)b-lo_diffb<=src(x,y)b<=src(seed.x,seed.y)b+up_diffb, 彩色图像，固定范围

其中 src(x',y') 是象素邻域点的值。也就是说，为了被加入到连通域中，一个象素的彩色/亮度应该足够接近于:

- 它的邻域象素的彩色/亮度值，当该邻域点已经被认为属于浮动范围情况下的连通域。
- 固定范围情况下的种子点的彩色/亮度值

[编辑]

FindContours

在二值图像中寻找轮廓

```
int cvFindContours( CvArr* image, CvMemStorage* storage, CvSeq** first_contour,
                    int header_size=sizeof(CvContour), int mode=CV_RETR_LIST,
                    int method=CV_CHAIN_APPROX_SIMPLE, CvPoint offset=cvPoint(0,0) );
```

image
: 输入的 8-比特、单通道图像. 非零元素被当成 1， 0 象素值保留为 0 - 从而图像被看成二值的。为了从灰度图像中得到这样的二值图像，可以使用 cvThreshold, cvAdaptiveThreshold 或 cvCanny. 本函数改变输入图像内容。

storage
: 得到的轮廓的存储容器

first_contour
: 输出参数：包含第一个输出轮廓的指针

header_size
: 如果 method=CV_CHAIN_CODE，则序列头的大小 >=sizeof(CvChain)，否则 >=sizeof(CvContour) .

mode
: 提取模式.

- CV_RETR_EXTERNAL - 只提取最外层的轮廓
- CV_RETR_LIST - 提取所有轮廓，并且放置在 list 中
- CV_RETR_CCOMP - 提取所有轮廓，并且将其组织为两层的 hierarchy: 顶层为连通域的外围边界，次层为洞的内层边界。
- CV_RETR_TREE - 提取所有轮廓，并且重构嵌套轮廓的全部 hierarchy

method
: 逼近方法 (对所有节点, 不包括使用内部逼近的 CV_RETR_RUNS).

- CV_CHAIN_CODE - Freeman 链码的输出轮廓. 其它方法输出多边形(定点序列).
- CV_CHAIN_APPROX_NONE - 将所有点由链码形式翻译(转化）为点序列形式
- CV_CHAIN_APPROX_SIMPLE - 压缩水平、垂直和对角分割，即函数只保留末端的象素点;
- CV_CHAIN_APPROX_TC89_L1,
- CV_CHAIN_APPROX_TC89_KCOS - 应用 Teh-Chin 链逼近算法. CV_LINK_RUNS - 通过连接为 1 的水平碎片使用完全不同的轮廓提取算法。仅有 CV_RETR_LIST 提取模式可以在本方法中应用.

offset
: 每一个轮廓点的偏移量. 当轮廓是从图像 ROI 中提取出来的时候，使用偏移量有用，因为可以从整个图像上下文来对轮廓做分析.

函数 cvFindContours 从二值图像中提取轮廓，并且返回提取轮廓的数目。指针 first_contour 的内容由函数填写。它包含第一个最外层轮廓的指针，如果指针为 NULL，则没有检测到轮廓（比如图像是全黑的）。其它轮廓可以从 first_contour 利用 h_next 和 v_next 链接访问到。 在 cvDrawContours 的样例显示如何使用轮廓来进行连通域的检测。轮廓也可以用来做形状分析和对象识别 - 见CVPR2001 教程中的 squares 样例。该教程可以在 SourceForge 网站上找到。

[编辑]

StartFindContours

初始化轮廓的扫描过程

```
CvContourScanner cvStartFindContours( CvArr* image, CvMemStorage* storage,
                                      int header_size=sizeof(CvContour),
```

```
                                int mode=CV_RETR_LIST,
                                int method=CV_CHAIN_APPROX_SIMPLE,
                                CvPoint offset=cvPoint(0,0) );
```

image
　　输入的 8-比特、单通道二值图像
storage
　　提取到的轮廓容器
header_size
　　序列头的尺寸 >=sizeof(CvChain) 若 method=CV_CHAIN_CODE，否则尺寸 >=sizeof(CvContour) .
mode
　　提取模式，见 cvFindContours.
method
　　逼近方法。它与 cvFindContours 里的定义一样，但是 CV_LINK_RUNS 不能使用。
offset
　　ROI 偏移量，见 cvFindContours.

函数 cvStartFindContours 初始化并且返回轮廓扫描器的指针。扫描器在 cvFindNextContour 使用以提取其余的轮廓。

FindNextContour

Finds next contour in the image

```
CvSeq* cvFindNextContour( CvContourScanner scanner );
```

scanner
　　被函数 cvStartFindContours 初始化的轮廓扫描器.

函数 cvFindNextContour 确定和提取图像的下一个轮廓，并且返回它的指针。若没有更多的轮廓，则函数返回 NULL.

SubstituteContour

替换提取的轮廓

```
void cvSubstituteContour( CvContourScanner scanner, CvSeq* new_contour );
```

scanner
　　被函数 cvStartFindContours 初始化的轮廓扫描器 ..
new_contour
　　替换的轮廓

函数 cvSubstituteContour 把用户自定义的轮廓替换前一次的函数 cvFindNextContour 调用所提取的轮廓，该轮廓以用户定义的模式存储在边缘扫描状态之中。轮廓，根据提取状态，被插入到生成的结构，List，二层 hierarchy, 或 tree 中。如果参数 new_contour=NULL, 则提取的轮廓不被包含入生成结构中，它的所有后代以后也不会被加入到接口中。

EndFindContours

结束扫描过程

```
 CvSeq* cvEndFindContours( CvContourScanner* scanner );
```

scanner
　　轮廓扫描的指针.

函数 cvEndFindContours 结束扫描过程，并且返回最高层的第一个轮廓的指针。

PyrSegmentation

用金字塔实现图像分割

```
void cvPyrSegmentation( IplImage* src, IplImage* dst,
                        CvMemStorage* storage, CvSeq** comp,
                        int level, double threshold1, double threshold2 );
```

src
　　输入图像.
dst
　　输出图像.
storage
　　Storage: 存储连通部件的序列结果
comp
　　分割部件的输出序列指针 components.
level
　　建立金字塔的最大层数
threshold1
　　建立连接的错误阈值
threshold2
　　分割簇的错误阈值

函数 cvPyrSegmentation 实现了金字塔方法的图像分割。金字塔建立到 level 指定的最大层数。如果 p(c(a),c(b))<threshold1,则在层 i 的象素点 a 和它的相邻层的父亲象素 b 之间的连接被建立起来，

定义好连接部件后，它们被加入到某些簇中。如果p(c(A),c(B))<threshold2，则任何两个分割 A 和 B 属于同一簇。

如果输入图像只有一个通道，那么

$p(c^1,c^2)=|c^1-c^2|$.

如果输入图像有单个通道（红、绿、兰），那么

$p(c^1,c^2)=0,3\cdot(c^1r-c^2r)+0,59\cdot(c^1g-c^2g)+0,11\cdot(c^1b-c^2b)$.

每一个簇可以有多个连接部件。 图像 src 和 dst 应该是 8-比特、单通道 或 3-通道图像，且大小一样

[编辑]

PyrMeanShiftFiltering

Does meanshift image segmentation

```
void cvPyrMeanShiftFiltering( const CvArr* src, CvArr* dst,
     double sp, double sr, int max_level=1,
     CvTermCriteria termcrit=cvTermCriteria(CV_TERMCRIT_ITER+CV_TERMCRIT_EPS,5,1));
```

src
　　输入的8-比特,3-信道图象.
dst
　　和源图象相同大小,相同格式的输出图象.
sp
　　The spatial window radius.
　　空间窗的半径
sr
　　The color window radius.
　　色彩窗的半径
max_level
　　Maximum level of the pyramid for the segmentation.

termcrit
　　Termination criteria: when to stop meanshift iterations.

The function cvPyrMeanShiftFiltering implements the filtering stage of meanshift segmentation, that is, the output of the function is the filtered "posterized" image with color gradients and fine-grain texture flattened. At every pixel (X,Y) of the input image (or down-sized input image, see below) the function executes meanshift iterations, that is, the pixel

(X,Y) neighborhood in the joint space-color hyperspace is considered:

{(x,y): X-sp≤x≤X+sp && Y-sp≤y≤Y+sp && ||(R,G,B)-(r,g,b)|| ≤ sr},where (R,G,B) and (r,g,b) are the vectors of color components at (X,Y) and (x,y), respectively (though, the algorithm does not depend on the color space used, so any 3-component color space can be used instead). Over the neighborhood the average spatial value (X',Y') and average color vector (R',G',B') are found and they act as the neighborhood center on the next iteration: (X,Y)~(X',Y'), (R,G,B)~(R',G',B').After the iterations over, the color components of the initial pixel (that is, the pixel from where the iterations started) are set to the final value (average color at the last iteration): I(X,Y) <- (R*,G*,B*).Then max_level>0, the gaussian pyramid of max_level+1 levels is built, and the above procedure is run on the smallest layer. After that, the results are propagated to the larger layer and the iterations are run again only on those pixels where the layer colors differ much (>sr) from the lower-resolution layer, that is, the boundaries of the color regions are clarified. Note, that the results will be actually different from the ones obtained by running the meanshift procedure on the whole original image (i.e. when max_level==0).

[编辑]

Watershed

做分水岭图像分割

```
void cvWatershed( const CvArr* image, CvArr* markers );
```

image
　　输入8比特3通道图像。
markers
　　输入或输出的32比特单通道标记图像。

函数cvWatershed实现在[Meyer92]描述的变量分水岭，基于非参数标记的分割算法中的一种。在把图像传给函数之前，用户需要用正指 标大致勾画出图像标记的感兴趣区域。比如，每一个区域都表示成一个或者多个像素值1，2，3的互联部分。这些部分将作为将来图像区域的种子。标记中所有的 其他像素，他们和勾画出的区域关系不明并且应由算法定义，应当被置0。这个函数的输出则是标记区域所有像素被置为某个种子部分的值，或者在区域边界则置 -1。

注：每两个相邻区域也不是必须有一个分水岭边界（-1像素）分开，例如在初始标记图像里有这样相切的部分。opencv例程文件夹里面有函数的视觉效果演示和用户例程。见watershed.cpp。

[编辑]

图像与轮廓矩

[编辑]

Moments

计算多边形和光栅形状的最高达三阶的所有矩

```
void cvMoments( const CvArr* arr, CvMoments* moments, int binary=0 );
```

arr
　　图像 (1-通道或3-通道，有COI设置) 或多边形(点的 CvSeq 或一族点的向量).
moments
　　返回的矩状态接口的指针
binary
　　(仅对图像) 如果标识为非零，则所有零象素点被当成零，其它的被看成 1.

函数 cvMoments 计算最高达三阶的空间和中心矩，并且将结果存在结构 moments 中。矩用来计算形状的重心，面积，主轴和其它的形状特征，如 7 Hu 不变量等。

[编辑]

GetSpatialMoment

从矩状态结构中提取空间矩

```
double cvGetSpatialMoment( CvMoments* moments, int x_order, int y_order );
```

moments
矩状态，由 cvMoments 计算
x_order
提取的 x 次矩, x_order >= 0.
y_order
提取的 y 次矩, y_order >= 0 并且 x_order + y_order <= 3.

函数 cvGetSpatialMoment 提取空间矩，当图像矩被定义为：

```
Mx_order,y_order=sumx,y(I(x,y)•xx_order•yy_order)
```

其中 I(x,y) 是象素点 (x, y) 的亮度值.

[编辑]

GetCentralMoment

从矩状态结构中提取中心矩

```
double cvGetCentralMoment( CvMoments* moments, int x_order, int y_order );
```

moments
矩状态结构指针
x_order
提取的 x 阶矩, x_order >= 0.
y_order
提取的 y 阶矩， y_order >= 0 且 x_order + y_order <= 3.

函数 cvGetCentralMoment 提取中心矩，其中图像矩的定义是：

```
µx_order,y_order=sumx,y(I(x,y)•(x-xc)x_order•(y-yc)y_order),
```

其中 xc=M10/M00, yc=M01/M00 - 重心坐标

[编辑]

GetNormalizedCentralMoment

从矩状态结构中提取归一化的中心矩

```
double cvGetNormalizedCentralMoment( CvMoments* moments, int x_order, int y_order );
```

moments
矩状态结构指针
x_order
提取的 x 阶矩, x_order >= 0.
y_order
提取的 y 阶矩， y_order >= 0 且 x_order + y_order <= 3.

函数 cvGetNormalizedCentralMoment 提取归一化中心矩：

```
ηx_order,y_order= µx_order,y_order/M00((y_order+x_order)/2+1)
```

[编辑]

GetHuMoments

计算 7 Hu 不变量

```
void cvGetHuMoments( CvMoments* moments, CvHuMoments* hu_moments );
```

moments
矩状态结构的指针
hu_moments
Hu 矩结构的指针.

函数 cvGetHuMoments 计算 7 个 Hu 不变量，它们的定义是：

```
h1=η20+η02
h2=(η20-η02)²+4η11²
h3=(η30-3η12)²+ (3η21-η03)²
h4=(η30+η12)²+ (η21+η03)²
h5=(η30-3η12)(η30+η12)[(η30+η12)²-3(η21+η03)²]+(3η21-η03)(η21+η03)[3(η30+η12)²-(η21+η03)²]
h6=(η20-η02)[(η30+η12)²- (η21+η03)²]+4η11(η30+η12)(η21+η03)
h7=(3η21-η03)(η21+η03)[3(η30+η12)²-(η21+η03)²]-(η30-3η12)(η21+η03)[3(η30+η12)²-(η21+η03)²]
```

这些值被证明为对图像缩放、旋转和反射的不变量。对反射，第7个除外，因为它的符号会因为反射而改变。

[编辑]

特殊图像变换

[编辑]

HoughLines

利用 Hough 变换在二值图像中找到直线

```
CvSeq* cvHoughLines2( CvArr* image, void* line_storage, int method,
                      double rho, double theta, int threshold,
                      double param1=0, double param2=0 );
```

image
: 输入 8-比特、单通道 (二值) 图像，当用CV_HOUGH_PROBABILISTIC方法检测的时候其内容会被函数改变

line_storage
: 检测到的线段存储仓. 可以是内存存储仓 (此种情况下，一个线段序列在存储仓中被创建，并且由函数返回），或者是包含线段参数的特殊类型（见下面）的具有单行/单列的矩阵(CvMat*)。矩阵 头为函数所修改，使得它的 cols/rows 将包含一组检测到的线段。如果 line_storage 是矩阵，而实际线段的数目超过矩阵尺寸，那么最大可能数目的线段被返回(对于标准hough变换，线段按照长度降序输出).

method
: Hough 变换变量，是下面变量的其中之一：

  - CV_HOUGH_STANDARD - 传统或标准 Hough 变换. 每一个线段由两个浮点数 (ρ, θ) 表示，其中 ρ 是直线与原点 (0,0) 之间的距离，θ 线段与 x-轴之间的夹角。因此，矩阵类型必须是 CV_32FC2 type.
  - CV_HOUGH_PROBABILISTIC - 概率 Hough 变换(如果图像包含一些长的线性分割，则效率更高). 它返回线段分割而不是整个线段。每个分割用起点和终点来表示，所以矩阵（或创建的序列）类型是 CV_32SC4.
  - CV_HOUGH_MULTI_SCALE - 传统 Hough 变换的多尺度变种。线段的编码方式与 CV_HOUGH_STANDARD 的一致。

rho
: 与象素相关单位的距离精度

theta
: 弧度测量的角度精度

threshold
: 阈值参数。如果相应的累计值大于 threshold， 则函数返回的这个线段.

param1
: 第一个方法相关的参数:

  - 对传统 Hough 变换，不使用(0).
  - 对概率 Hough 变换，它是最小线段长度.
  - 对多尺度 Hough 变换，它是距离精度 rho 的分母 (大致的距离精度是 rho 而精确的应该是 rho / param1).

param2
: 第二个方法相关参数:

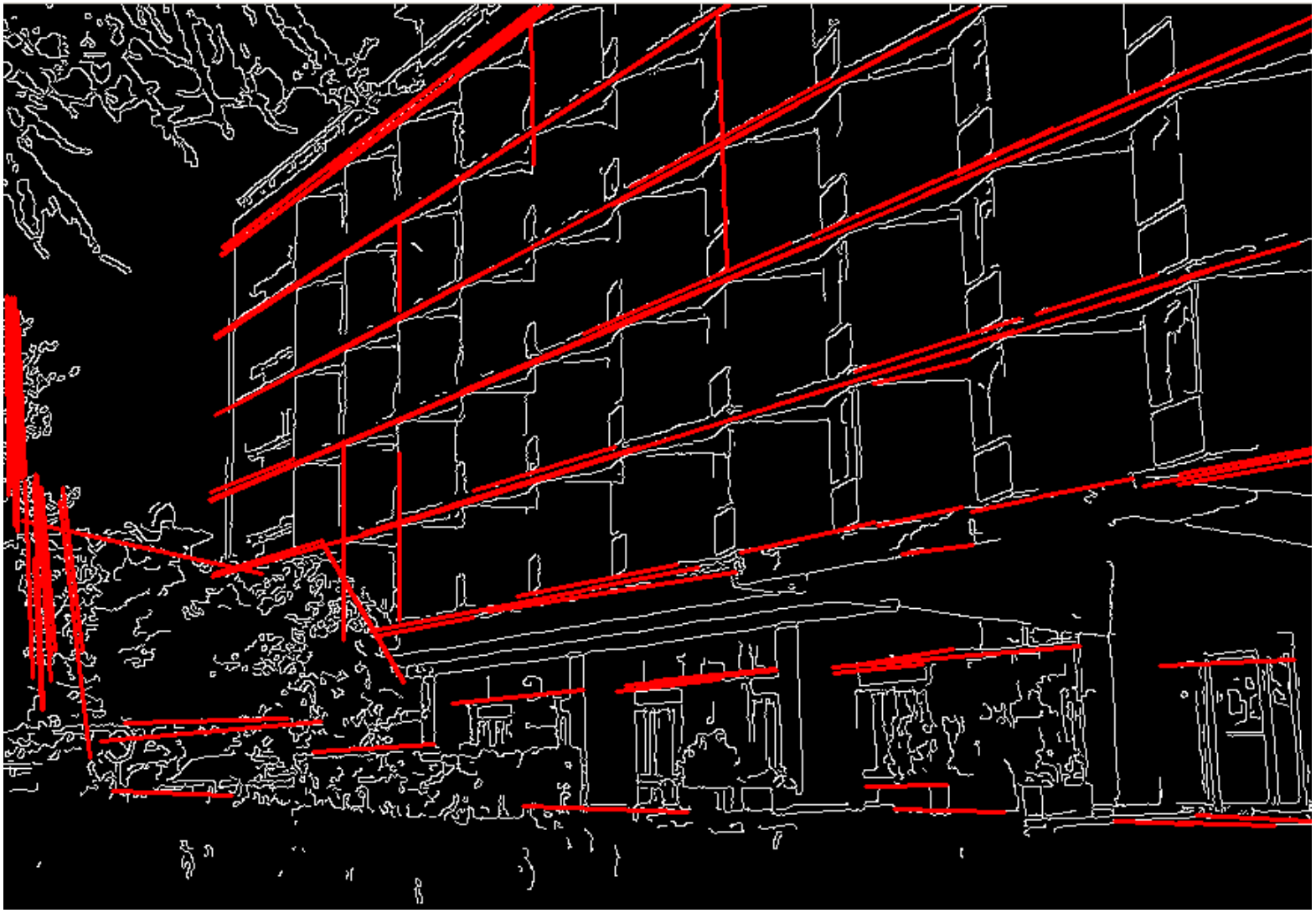
  - 对传统 Hough 变换，不使用 (0).
  - 对概率 Hough 变换，这个参数表示在同一条直线上进行碎线段连接的最大间隔值(gap)，即当同一条直线上的两条碎线段之间的间隔小于param2时，将其合二为一。
  - 对多尺度 Hough 变换，它是角度精度 theta 的分母 (大致的角度精度是 theta 而精确的角度应该是 theta / param2).

函数 cvHoughLines2 实现了用于线段检测的不同 Hough 变换方法. Example. 用 Hough transform 检测线段

```
/* This is a standalone program. Pass an image name as a first parameter
   of the program.Switch between standard and probabilistic Hough transform
   by changing "#if 1" to "#if 0" and back */
#include <cv.h>
#include <highgui.h>
#include <math.h>

int main(int argc, char** argv)
{
    IplImage* src;
    if( argc == 2 && (src=cvLoadImage(argv[1], 0))!= 0)
    {
        IplImage* dst = cvCreateImage( cvGetSize(src), 8, 1 );
        IplImage* color_dst = cvCreateImage( cvGetSize(src), 8, 3 );
        CvMemStorage* storage = cvCreateMemStorage(0);
        CvSeq* lines = 0;
        int i;
        IplImage* src1=cvCreateImage(cvSize(src->width,src->height),IPL_DEPTH_8U,1);

        cvCvtColor(src, src1, CV_BGR2GRAY);
        cvCanny( src1, dst, 50, 200, 3 );

        cvCvtColor( dst, color_dst, CV_GRAY2BGR );
#if 1
        lines = cvHoughLines2( dst, storage, CV_HOUGH_STANDARD, 1, CV_PI/180, 150, 0, 0 );

        for( i = 0; i < lines->total; i++ )
        {
            float* line = (float*)cvGetSeqElem(lines,i);
            float rho = line[0];
            float theta = line[1];
            CvPoint pt1, pt2;
            double a = cos(theta), b = sin(theta);
            if( fabs(a) < 0.001 )
            {
                pt1.x = pt2.x = cvRound(rho);
                pt1.y = 0;
                pt2.y = color_dst->height;
            }
            else if( fabs(b) < 0.001 )
            {
                pt1.y = pt2.y = cvRound(rho);
                pt1.x = 0;
                pt2.x = color_dst->width;
            }
            else
            {
                pt1.x = 0;
                pt1.y = cvRound(rho/b);
                pt2.x = cvRound(rho/a);
                pt2.y = 0;
            }
            cvLine( color_dst, pt1, pt2, CV_RGB(255,0,0), 3, 8 );
        }
#else
        lines = cvHoughLines2( dst, storage, CV_HOUGH_PROBABILISTIC, 1, CV_PI/180, 80, 30, 10 );
        for( i = 0; i < lines->total; i++ )
        {
            CvPoint* line = (CvPoint*)cvGetSeqElem(lines,i);
            cvLine( color_dst, line[0], line[1], CV_RGB(255,0,0), 3, 8 );
        }
#endif
        cvNamedWindow( "Source", 1 );
        cvShowImage( "Source", src );

        cvNamedWindow( "Hough", 1 );
        cvShowImage( "Hough", color_dst );

        cvWaitKey(0);
    }
}
```

这是函数所用的样本图像:

下面是程序的输出，采用概率 Hough transform ("#if 0" 的部分):



[编辑]

HoughCircles

利用 Hough 变换在灰度图像中找圆

```
CvSeq* cvHoughCircles( CvArr* image, void* circle_storage,
                       int method, double dp, double min_dist,
                       double param1=100, double param2=100,
                       int min_radius=0, int max_radius=0 );
```

image
　　输入 8-比特、单通道灰度图像.

circle_storage
　　检测到的圆存储仓. 可以是内存存储仓 (此种情况下，一个线段序列在存储仓中被创建，并且由函数返回）或者是包含圆参数的特殊类型的具有单行/单列的CV_32FC3型矩阵(CvMat*). 矩阵头为函数所修改，使得它的 cols/rows 将包含一组检测到的圆。如果 circle_storage 是矩阵，而实际圆的数目超过矩阵尺寸，那么最大可能数目的圆被返回

. 每个圆由三个浮点数表示：圆心坐标(x,y)和半径.

method
　　Hough 变换方式，目前只支持CV_HOUGH_GRADIENT, which is basically 21HT, described in [Yuen03].

dp
　　累加器图像的分辨率。这个参数允许创建一个比输入图像分辨率低的累加器。（这样做是因为有理由认为图像中存在的圆会自然降低到与图像宽 高相同数量的范畴）。如果dp设置为1，则分辨率是相同的；如果设置为更大的值（比如2），累加器的分辨率受此影响会变小（此情况下为一半）。dp的值不 能比1小。

Resolution of the accumulator used to detect centers of the circles. For example, if it is 1, the accumulator will have the

same resolution as the input image, if it is 2 - accumulator will have twice smaller width and height, etc.

min_dist
　　该参数是让算法能明显区分的两个不同圆之间的最小距离。

Minimum distance between centers of the detected circles. If the parameter is too small, multiple neighbor circles may be falsely detected in addition to a true one. If it is too large, some circles may be missed.

param1
　　用于Canny的边缘阀值上限，下限被置为上限的一半。

The first method-specific parameter. In case of CV_HOUGH_GRADIENT it is the higher threshold of the two passed to Canny edge detector (the lower one will be twice smaller).

param2
　　累加器的阀值。

The second method-specific parameter. In case of CV_HOUGH_GRADIENT it is accumulator threshold at the center detection stage. The smaller it is, the more false circles may be detected. Circles, corresponding to the larger accumulator values, will be returned first.

min_radius
　　最小圆半径。

Minimal radius of the circles to search for.

max_radius
　　最大圆半径。

Maximal radius of the circles to search for. By default the maximal radius is set to max(image_width, image_height).

The function cvHoughCircles finds circles in grayscale image using some modification of Hough transform.

Example. Detecting circles with Hough transform.

```
#include <cv.h>
#include <highgui.h>
#include <math.h>

int main(int argc, char** argv)
{
    IplImage* img;
    if( argc == 2 && (img=cvLoadImage(argv[1], 1))!= 0)
    {
        IplImage* gray = cvCreateImage( cvGetSize(img), 8, 1 );
        CvMemStorage* storage = cvCreateMemStorage(0);
        cvCvtColor( img, gray, CV_BGR2GRAY );
        cvSmooth( gray, gray, CV_GAUSSIAN, 9, 9 ); // smooth it, otherwise a lot of false circles may be
detected
        CvSeq* circles = cvHoughCircles( gray, storage, CV_HOUGH_GRADIENT, 2, gray->height/4, 200, 100 );
        int i;
        for( i = 0; i < circles->total; i++ )
        {
             float* p = (float*)cvGetSeqElem( circles, i );
             cvCircle( img, cvPoint(cvRound(p[0]),cvRound(p[1])), 3, CV_RGB(0,255,0), -1, 8, 0 );
             cvCircle( img, cvPoint(cvRound(p[0]),cvRound(p[1])), cvRound(p[2]), CV_RGB(255,0,0), 3, 8, 0 );
        }
        cvNamedWindow( "circles", 1 );
        cvShowImage( "circles", img );
    }
    return 0;
}
```


DistTransform

计算输入图像的所有非零元素对其最近零元素的距离

```
void cvDistTransform( const CvArr* src, CvArr* dst, int distance_type=CV_DIST_L2,
                      int mask_size=3, const float* mask=NULL );
```

src

输入 8-比特、单通道 (二值) 图像.

dst

含计算出的距离的输出图像(32-比特、浮点数、单通道).

distance_type

距离类型; 可以是 CV_DIST_L1, CV_DIST_L2, CV_DIST_C 或 CV_DIST_USER.

mask_size

距离变换掩模的大小，可以是 3 或 5. 对 CV_DIST_L1 或 CV_DIST_C 的情况，参数值被强制设定为 3, 因为 3×3 mask 给出 5×5 mask 一样的结果，而且速度还更快。

mask

用户自定义距离情况下的 mask。 在 3×3 mask 下它由两个数(水平/垂直位量，对角线位移量）组成， 5×5 mask 下由三个数组成(水平/垂直位移量，对角位移和 国际象棋里的马步(马走日))

函数 cvDistTransform 二值图像每一个象素点到它最邻近零象素点的距离。对零象素，函数设置 0 距离，对其它象素，它寻找由基本位移（水平、垂直、对角线或knight's move，最后一项对 5×5 mask 有用）构成的最短路径。 全部的距离被认为是基本距离的和。由于距离函数是对称的，所有水平和垂直位移具有同样的代价 (表示为 a), 所有的对角位移具有同样的代价 (表示为 b), 所有的 knight's 移动具有同样的代价 (表示为 c). 对类型 CV_DIST_C 和 CV_DIST_L1，距离的计算是精确的，而类型 CV_DIST_L2 (欧式距离) 距离的计算有某些相对误差 (5×5 mask 给出更精确的结果), OpenCV 使用 [Borgefors86] 推荐的值:

CV_DIST_C (3×3):
a=1, b=1

CV_DIST_L1 (3×3):
a=1, b=2

CV_DIST_L2 (3×3):
a=0.955, b=1.3693

CV_DIST_L2 (5×5):
a=1, b=1.4, c=2.1969

下面用户自定义距离的的距离域示例 (黑点 (0) 在白色方块中间）：用户自定义 3×3 mask (a=1, b=1.5)

4.5	4	3.5	3	3.5	4	4.5
4	3	2.5	2	2.5	3	4
3.5	2.5	1.5	1	1.5	2.5	3.5
3	2	1	0	1	2	3
3.5	2.5	1.5	1	1.5	2.5	3.5
4	3	2.5	2	2.5	3	4
4.5	4	3.5	3	3.5	4	4.5

用户自定义 5×5 mask (a=1, b=1.5, c=2)

4.5	3.5	3	3	3	3.5	4.5
3.5	3	2	2	2	3	3.5
3	2	1.5	1	1.5	2	3
3	2	1	0	1	2	3
3	2	1.5	1	1.5	2	3
3.5	3	2	2	2	3	3.5
4	3.5	3	3	3	3.5	4

典型的使用快速粗略距离估计 CV_DIST_L2, 3×3 mask , 如果要更精确的距离估计，使用 CV_DIST_L2, 5×5 mask。

When the output parameter labels is not NULL, for every non-zero pixel the function also finds the nearest connected component consisting of zero pixels. The connected components themselves are found as contours in the beginning of the function.

In this mode the processing time is still O(N), where N is the number of pixels. Thus, the function provides a very fast way to compute approximate Voronoi diagram for the binary image.

[编辑]

Inpaint

修复图像中选择区域。

```
void cvInpaint( const CvArr* src, const CvArr* mask, CvArr* dst,
                int flags, double inpaintRadius );
```

src
　　输入8比特单通道或者三通道图像。
mask
　　修复图像的掩饰，8比特单通道图像。非零像素表示该区域需要修复。
dst
　　输出图像，和输入图像相同格式相同大小。
flags
　　修复方法,以下之一:
　　CV_INPAINT_NS - 基于Navier-Stokes的方法。
　　CV_INPAINT_TELEA - Alexandru Telea[Telea04]的方法。
inpaintRadius
　　算法考虑的每个修复点的圆形领域的半径。

函数cvInpaint从选择图像区域边界的像素重建该区域。函数可以用来去除扫描相片的灰尘或者刮伤，或者从静态图像或者视频中去除不需要的物体。

[编辑]

直方图

[编辑]

CvHistogram

多维直方图

```
typedef struct CvHistogram
{
    int     type;
    CvArr*  bins;
    float   thresh[CV_MAX_DIM][2]; /* for uniform histograms */
    float** thresh2; /* for non-uniform histograms */
    CvMatND mat; /* embedded matrix header for array histograms */
}
CvHistogram;
```

bins：用于存放直方图每个灰度级数目的数组指针，数组在cvCreateHist 的时候创建，其维数由cvCreateHist 确定（一般以一维比较常见）

[编辑]

CreateHist

创建直方图

```
CvHistogram* cvCreateHist( int dims, int* sizes, int type,
                           float** ranges=NULL, int uniform=1 );
```

dims
　　直方图维数的数目
sizes
　　直方图维数尺寸的数组
type
　　直方图的表示格式: CV_HIST_ARRAY 意味着直方图数据表示为多维密集数组 CvMatND; CV_HIST_TREE 意味着直方图数据表示为多维稀疏数组 CvSparseMat.
ranges

图中方块范围的数组. 它的内容取决于参数 uniform 的值。这个范围的用处是确定何时计算直方图或决定反向映射（backprojected ），每个方块对应于输入图像的哪个/哪组值。

uniform

归一化标识。 如果不为0，则ranges[i]（0<=i<cDims，译者注：cDims为直方图的维数，对于灰度图为1，彩色图为3）是包含两个元素 的范围数组，包括直方图第i维的上界和下界。在第i维上的整个区域 [lower,upper]被分割成 dims[i] 个相等的块（译者注：dims[i]表示直方图第i维的块数），这些块用来确定输入象素的第 i 个值（译者注：对于彩色图像，i确定R, G,或者B）的对应的块；如果为0，则ranges[i]是包含dims[i]+1个元素的范围数组，包括lower0, upper0, lower1, upper1 == lower2, ..., upperdims[i]-1, 其中lowerj 和upperj分别是直方图第i维上第 j 个方块的上下界（针对输入象素的第 i 个值）。任何情况下，输入值如果超出了一个直方块所指定的范围外，都不会被 cvCalcHist 计数，而且会被函数 cvCalcBackProject 置零。

函数 cvCreateHist 创建一个指定尺寸的直方图，并且返回创建的直方图的指针。 如果数组的 ranges 是 0, 则直方块的范围必须由函数 cvSetHistBinRanges 稍后指定。虽然 cvCalcHist 和 cvCalcBackProject 可以处理 8-比特图像而无需设置任何直方块的范围，但它们都被假设等分 0..255 之间的空间。

[编辑]

SetHistBinRanges

设置直方块的区间

```
void cvSetHistBinRanges( CvHistogram* hist, float** ranges, int uniform=1 );
```

hist

直方图.

ranges

直方块范围数组的数组，见 cvCreateHist.

uniform

归一化标识，见 cvCreateHist.

函数 cvSetHistBinRanges 是一个独立的函数，完成直方块的区间设置。更多详细的关于参数 ranges 和 uniform 的描述，请参考函数 cvCalcHist , 该函数也可以初始化区间。直方块的区间的设置必须在计算直方图之前，或 在计算直方图的反射图之前。

[编辑]

ReleaseHist

释放直方图结构

```
void cvReleaseHist( CvHistogram** hist );
```

hist

被释放的直方图结构的双指针.

函数 cvReleaseHist 释放直方图 (头和数据). 指向直方图的指针被函数所清空。如果 *hist 指针已经为 NULL, 则函数不做任何事情。

[编辑]

ClearHist

清除直方图

```
void cvClearHist( CvHistogram* hist );
```

hist

直方图.

函数 cvClearHist 当直方图是稠密数组时将所有直方块设置为 0，当直方图是稀疏数组时，除去所有的直方块。

[编辑]

MakeHistHeaderForArray

从数组中创建直方图

```
CvHistogram*  cvMakeHistHeaderForArray( int dims, int* sizes, CvHistogram* hist,
                                        float* data, float** ranges=NULL, int uniform=1 );
```

dims
　　直方图维数.
sizes
　　直方图维数尺寸的数组
hist
　　为函数所初始化的直方图头
data
　　用来存储直方块的数组
ranges
　　直方块范围，见 cvCreateHist.
uniform
　　归一化标识，见 cvCreateHist.

函数 cvMakeHistHeaderForArray 初始化直方图，其中头和直方块为用户所分配。以后不需要调用 cvReleaseHist 只有稠密直方图可以采用这种方法，函数返回 hist.

[编辑]

QueryHistValue_1D

查询直方块的值

```
#define cvQueryHistValue_1D( hist, idx0 ) \
    cvGetReal1D( (hist)->bins, (idx0) )
#define cvQueryHistValue_2D( hist, idx0, idx1 ) \
    cvGetReal2D( (hist)->bins, (idx0), (idx1) )
#define cvQueryHistValue_3D( hist, idx0, idx1, idx2 ) \
    cvGetReal3D( (hist)->bins, (idx0), (idx1), (idx2) )
#define cvQueryHistValue_nD( hist, idx ) \
    cvGetRealND( (hist)->bins, (idx) )
```

hist
　　直方图
idx0, idx1, idx2, idx3
　　直方块的下标索引
idx
　　下标数组

宏 cvQueryHistValue_*D 返回 1D, 2D, 3D 或 N-D 直方图的指定直方块的值。对稀疏直方图，如果方块在直方图中不存在，函数返回 0, 而且不创建新的直方块。

[编辑]

GetHistValue_1D

返回直方块的指针

```
#define cvGetHistValue_1D( hist, idx0 ) \
    ((float*)(cvPtr1D( (hist)->bins, (idx0), 0 ))
#define cvGetHistValue_2D( hist, idx0, idx1 ) \
    ((float*)(cvPtr2D( (hist)->bins, (idx0), (idx1), 0 ))
#define cvGetHistValue_3D( hist, idx0, idx1, idx2 ) \
    ((float*)(cvPtr3D( (hist)->bins, (idx0), (idx1), (idx2), 0 ))
#define cvGetHistValue_nD( hist, idx ) \
    ((float*)(cvPtrND( (hist)->bins, (idx), 0 ))
```

hist
　　直方图.
idx0, idx1, idx2, idx3
　　直方块的下标索引.

idx
　　下标数组

宏 cvGetHistValue_*D 返回 1D, 2D, 3D 或 N-D 直方图的指定直方块的指针。对稀疏直方图，函数创建一个新的直方块，且设置其为 0，除非它已经存在。

[编辑]

GetMinMaxHistValue

发现最大和最小直方块

```
void cvGetMinMaxHistValue( const CvHistogram* hist,
                           float* min_value, float* max_value,
                           int* min_idx=NULL, int* max_idx=NULL );
```

hist
　　直方图
min_value
　　直方图最小值的指针
max_value
　　直方图最大值的指针
min_idx
　　数组中最小坐标的指针
max_idx
　　数组中最大坐标的指针

函数 cvGetMinMaxHistValue 发现最大和最小直方块以及它们的位置。任何输出变量都是可选的。在具有同样值几个极值中，返回具有最小下标索引（以字母排列顺序定）的那一个。

[编辑]

NormalizeHist

归一化直方图

```
void cvNormalizeHist( CvHistogram* hist, double factor );
```

hist
　　直方图的指针.
factor
　　归一化因子

函数 cvNormalizeHist 通过缩放来归一化直方块，使得所有块的和等于 factor.

[编辑]

ThreshHist

对直方图取阈值

```
void cvThreshHist( CvHistogram* hist, double threshold );
```

hist
　　直方图的指针.
threshold
　　阈值大小

函数 cvThreshHist 清除那些小于指定阈值得直方块

[编辑]

CompareHist

比较两个稠密直方图

```
double cvCompareHist( const CvHistogram* hist1, const CvHistogram* hist2, int method );
```

hist1
　　第一个稠密直方图
hist2
　　第二个稠密直方图
method
　　比较方法，采用其中之一：

- CV_COMP_CORREL
- CV_COMP_CHISQR
- CV_COMP_INTERSECT
- CV_COMP_BHATTACHARYYA

　　函数 cvCompareHist 采用下面指定的方法比较两个稠密直方图(H_1 表示第一个， H_2表示第二个):

- Correlation (method=CV_COMP_CORREL):

$$d(H1,H2) = \sum_i \frac{H'_1(i) \cdot H'_2(i)}{\sqrt{(\sum_j H'_1(j)^2) \cdot (\sum_j H'_2(j)^2)}}$$

　　其中

$$H'_k(i) = H_k(i) - \frac{1}{N}\sum_j H_k(j)$$ (N 是number of histogram bins)

- Chi-square(method=CV_COMP_CHISQR):

$$d(H_1,H_2) = \sum_i \frac{H_1(i) - H_2(i)}{H_1(i) + H_2(i)}$$

- 交叉 (method=CV_COMP_INTERSECT):

$$d(H_1,H_2) = \sum_i \min(H_1(i),H_2(i))$$

- Bhattacharyya 距离 (method=CV_COMP_BHATTACHARYYA):

$$d(H_1,H_2) = \sqrt{1 - \sum_i \sqrt{H_1(i) \cdot H_2(i)}}$$

函数返回 d(H1,H2) 的值。

注意：Bhattacharyya 距离只能应用到规一化后的直方图。

为了比较稀疏直方图或更一般的加权稀疏点集(译者注：直方图匹配是图像检索中的常用方法)，考虑使用函数 cvCalcEMD 。

[编辑]

CopyHist

拷贝直方图

```
void cvCopyHist( const CvHistogram* src, CvHistogram** dst );
```

src
　　输入的直方图
dst

输出的直方图指针

函数 cvCopyHist 对直方图作拷贝。如果第二个直方图指针 *dst 是 NULL, 则创建一个与 src 同样大小的直方图。否则，两个直方图必须大小和类型一致。然后函数将输入的直方块的值复制到输出的直方图中，并且设置取值范围与 src 的一致。

[编辑]

CalcHist

计算图像image(s) 的直方图

```
void cvCalcHist( IplImage** image, CvHistogram* hist,
                 int accumulate=0, const CvArr* mask=NULL );
```

image
输入图像s (虽然也可以使用 CvMat**).

hist
直方图指针

accumulate
累计标识。如果设置，则直方图在开始时不被清零。这个特征保证可以为多个图像计算一个单独的直方图，或者在线更新直方图。

mask
操作 mask, 确定输入图像的哪个象素被计数

函数 cvCalcHist 计算一张或多张单通道图像的直方图（译者注：若要计算多通道，可像以下例子那样用多个单通道图来表示）。 用来增加直方块的数组元素可从相应输入图像的同样位置提取。 Sample. 计算和显示彩色图像的 2D 色调－饱和度图像

```
#include <cv.h>
#include <highgui.h>

int main( int argc, char** argv )
{
    IplImage* src;
    if( argc == 2 && (src=cvLoadImage(argv[1], 1))!= 0)
    {
        IplImage* h_plane = cvCreateImage( cvGetSize(src), 8, 1 );
        IplImage* s_plane = cvCreateImage( cvGetSize(src), 8, 1 );
        IplImage* v_plane = cvCreateImage( cvGetSize(src), 8, 1 );
        IplImage* planes[] = { h_plane, s_plane };
        IplImage* hsv = cvCreateImage( cvGetSize(src), 8, 3 );
        int h_bins = 30, s_bins = 32;
        int hist_size[] = {h_bins, s_bins};
        /* hue varies from 0 (~0°red) to 180 (~360°red again) */
        float h_ranges[] = { 0, 180 };
        /* saturation varies from 0 (black-gray-white) to 255 (pure spectrum color) */
        float s_ranges[] = { 0, 255 };
        float* ranges[] = { h_ranges, s_ranges };
        int scale = 10;
        IplImage* hist_img = cvCreateImage( cvSize(h_bins*scale,s_bins*scale), 8, 3 );
        CvHistogram* hist;
        float max_value = 0;
        int h, s;

        cvCvtColor( src, hsv, CV_BGR2HSV );
        cvCvtPixToPlane( hsv, h_plane, s_plane, v_plane, 0 );
        hist = cvCreateHist( 2, hist_size, CV_HIST_ARRAY, ranges, 1 );
        cvCalcHist( planes, hist, 0, 0 );
        cvGetMinMaxHistValue( hist, 0, &max_value, 0, 0 );
        cvZero( hist_img );

        for( h = 0; h < h_bins; h++ )
        {
            for( s = 0; s < s_bins; s++ )
            {
                float bin_val = cvQueryHistValue_2D( hist, h, s );
                int intensity = cvRound(bin_val*255/max_value);
                cvRectangle( hist_img, cvPoint( h*scale, s*scale ),
                             cvPoint( (h+1)*scale - 1, (s+1)*scale - 1),
                             CV_RGB(intensity,intensity,intensity), /* graw a grayscale histogram.
                                                                       if you have idea how to do it
                                                                       nicer let us know */
                             CV_FILLED );
            }
        }

        cvNamedWindow( "Source", 1 );
        cvShowImage( "Source", src );
```

```
        cvNamedWindow( "H-S Histogram", 1 );
        cvShowImage( "H-S Histogram", hist_img );

        cvWaitKey(0);
    }
}
```

[编辑]

CalcBackProject

计算反向投影

```
void cvCalcBackProject( IplImage** image, CvArr* back_project, const CvHistogram* hist );
```

image
　　输入图像 (也可以传递 CvMat**).
back_project
　　反向投影图像，与输入图像具有同样类型.
hist
　　直方图

函数 cvCalcBackProject 计算直方图的反向投影. 對於所有输入的单通道图像同一位置的象素数组，该函数根据相应的象素数组(RGB)，放置其对应的直方块的值到输出图像中。用统计学术语，输出图像象素点 的值是观测数组在某个分布（直方图）下的概率。 例如，为了发现图像中的红色目标，可以这么做：

1. 对红色物体计算色调直方图，假设图像仅仅包含该物体。则直方图有可能有极值，对应着红颜色。
2. 对将要搜索目标的输入图像，使用直方图计算其色调平面的反向投影，然后对图像做阈值操作。
3. 在产生的图像中发现连通部分，然后使用某种附加准则选择正确的部分，比如最大的连通部分。

这是 Camshift 彩色目标跟踪器中的一个逼进算法，除了第三步，CAMSHIFT 算法使用了上一次目标位置来定位反向投影中的目标。

[编辑]

CalcBackProjectPatch

用直方图比较来定位图像中的模板

```
void cvCalcBackProjectPatch( IplImage** image, CvArr* dst,
                             CvSize patch_size, CvHistogram* hist,
                             int method, double factor );
```

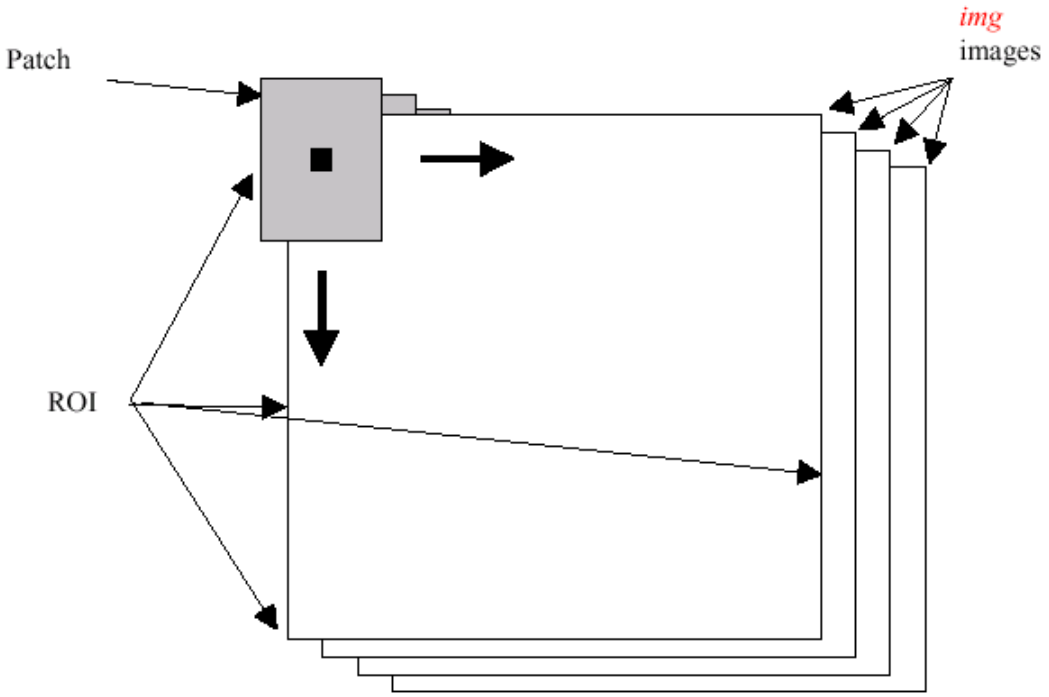
image
　　输入图像 (可以传递 CvMat**)
dst
　　输出图像.
patch_size
　　扫描输入图像的补丁尺寸
hist
　　直方图
method
　　比较方法，传递给 cvCompareHist (见该函数的描述).
factor
　　直方图的归一化因子，将影响输出图像的归一化缩放。如果为 1，则不定。 /*归一化因子的类型实际上是double，而非float*/

函数 cvCalcBackProjectPatch 通过输入图像补丁的直方图和给定直方图的比较，来计算反向投影。提取图像在 ROI 中每一个位置的某种测量结果产生了数组 image. 这些结果可以是色调, x 差分, y 差分, Laplacian 滤波器, 有方向 Gabor 滤波器等中 的一个或多个。每种测量输出都被划归为它自己的单独图像。 image 图像数组是这些测量图像的集合。一个多维直方图 hist 从这些图像数组中被采样创建。最后直方图被归一化。直方图 hist 的维数通常很大等于图像数组 image 的元素个数。

在选择的 ROI 中，每一个新的图像被测量并且转换为一个图像数组。在以锚点为“补丁”中心的图像 image 区域中计算直方图 （如下图所示）。用参数 norm_factor 来归一化直方图，使得它可以与 hist 互相比较。计算出的直方图与直方图模型互相比较， （hist 使用函数 cvCompareHist ，比较方法是 method=method). 输出结果被放置到概率图像 dst 补丁锚点的对应位置上。这个过程随着补丁滑过整个 ROI 而重复进行。迭代直方图的更新可以通过在原直方图中减除“补丁”已覆盖的尾

象素点或者加上新覆盖的象素点来实现，这种更新方式可以节省大量的操作，尽管目 前在函数体中还没有实现。

Back Project Calculation by Patches



[编辑]

CalcProbDensity

两个直方图相除

```
void  cvCalcProbDensity( const CvHistogram* hist1, const CvHistogram* hist2,
                         CvHistogram* dst_hist, double scale=255 );
```

hist1
　　第一个直方图(分母).
hist2
　　第二个直方图
dst_hist
　　输出的直方图
scale
　　输出直方图的尺度因子

函数 cvCalcProbDensity 从两个直方图中计算目标概率密度:

```
dist_hist(I)=0       if hist1(I)==0
             scale   if hist1(I)!=0 && hist2(I)>hist1(I)
             hist2(I)*scale/hist1(I) if hist1(I)!=0 && hist2(I)<=hist1(I)
```

所以输出的直方块小于尺度因子。

[编辑]

EqualizeHist

灰度图象直方图均衡化

```
void cvEqualizeHist( const CvArr* src, CvArr* dst );
```

src
　　输入的 8-比特 单信道图像
dst

输出的图像与输入图像大小与数据类型相同

函数 cvEqualizeHist 采用如下法则对输入图像进行直方图均衡化：

1. 计算输入图像的直方图 H
2. 直方图归一化，因此直方块和为255
$$H'(i) = \sum_{0 \le j \le i} H(j)$$
3. 计算直方图积分：
4. 采用H'作为查询表：dst(x,y)=H'(src(x,y))进行图像变换。

该方法归一化图像亮度和增强对比度。

例：彩色图像的直方图均衡化

```
int i;
IplImage *pImageChannel[4] = { 0, 0, 0, 0 };
pSrcImage = cvLoadImage( "test.jpg", 1 ) ;
IplImage *pImage = cvCreateImage(cvGetSize(pSrcImage), pSrcImage->depth, pSrcImage->nChannels);
if( pSrcImage )
{
        for( i = 0; i < pSrcImage->nChannels; i++ )
        {
                pImageChannel[i] = cvCreateImage( cvGetSize(pSrcImage), pSrcImage->depth, 1 );
        }
        // 信道分离
        cvSplit( pSrcImage, pImageChannel[0], pImageChannel[1],
                pImageChannel[2], pImageChannel[3] );
        for( i = 0; i < pImage->nChannels; i++ )
        {
                // 直方图均衡化
                cvEqualizeHist( pImageChannel[i], pImageChannel[i] );
        }
        // 信道组合
        cvMerge( pImageChannel[0], pImageChannel[1], pImageChannel[2],
                pImageChannel[3], pImage );
        // ……图像显示代码（略）
        // 释放资源
        for( i = 0; i < pSrcImage->nChannels; i++ )
        {
                if ( pImageChannel[i] )
                {
                        cvReleaseImage( &pImageChannel[i] );
                        pImageChannel[i] = 0;
                }
        }
        cvReleaseImage( &pImage );
        pImage = 0;
}
```

[编辑]

匹配

[编辑]

MatchTemplate

比较模板和重叠的图像区域

```
void cvMatchTemplate( const CvArr* image, const CvArr* templ,
                      CvArr* result, int method );
```

image
　　欲搜索的图像。它应该是单通道、8-比特或32-比特 浮点数图像
templ
　　搜索模板，不能大于输入图像，且与输入图像具有一样的数据类型
result
　　比较结果的映射图像。单通道、32-比特浮点数. 如果图像是 W×H 而 templ 是 w×h ， 则 result 一定是 (W-w+1）×(H-h+1）.
method
　　指定匹配方法：

函数 cvMatchTemplate 与函数 cvCalcBackProjectPatch 类似。它滑动过整个图像 image, 用指定方法比较 templ 与图像尺

寸为 w×h 的重叠区域，并且将比较结果存到 result 中。 下面是不同的比较方法，可以使用其中的一种 (I 表示图像，T - 模板, R - 结果. 模板与图像重叠区域 x'=0..w-1, y'=0..h-1 之间求和):

method=CV_TM_SQDIFF:

$$R(x,y) = \sum_{x',y'} [T(x',y') - I(x + x',y + y')]^2$$

method=CV_TM_SQDIFF_NORMED:

$$R(x,y) = \sum_{x',y'} [T(x',y') - I(x+x',y+y')]^2 / \sqrt{\sum_{x',y'} T(x',y')^2 \cdot \sum_{x',y'} I(x+x',y+y')^2}$$

method=CV_TM_CCORR:

$$R(x,y) = \sum_{x',y'} [T(x',y') \cdot I(x+x',y+y')]$$

method=CV_TM_CCORR_NORMED:

$$R(x,y) = \sum_{x',y'} [T(x',y') \cdot I(x+x',y+y')] / \sqrt{\sum_{x',y'} T(x',y')^2 \cdot \sum_{x',y'} I(x+x',y+y')^2}$$

method=CV_TM_CCOEFF:

$$R(x,y) = \sum_{x',y'} [T'(x',y') \cdot I'(x+x',y+y')]$$

,

其中

$$T'(x',y') = T(x',y') - \frac{1}{w \cdot h} \sum_{x'',y''} T(x'',y'')$$

(mean template brightness=>0)

$$I'(x+x',y+y') = I(x+x',y+y') - \frac{1}{w \cdot h} \sum_{x'',y''} I(x+x'',y+y'')$$

(mean patch brightness=>0)

method=CV_TM_CCOEFF_NORMED:

$$R(x,y) = \sum_{x',y'} [T'(x',y') \cdot I'(x+x',y+y')] / \sqrt{\sum_{x',y'} T'(x',y')^2 \cdot \sum_{x',y'} I'(x+x',y+y')^2}$$

函数完成比较后，通过使用cvMinMaxLoc找全局最小值CV_TM_SQDIFF*) 或者最大值 (CV_TM_CCORR* and CV_TM_CCOEFF*)。

[编辑]

MatchShapes

比较两个形状

```
double cvMatchShapes( const void* object1, const void* object2,
                      int method, double parameter=0 );
```

object1
　　第一个轮廓或灰度图像
object2
　　第二个轮廓或灰度图像

method
比较方法，其中之一 CV_CONTOUR_MATCH_I1, CV_CONTOURS_MATCH_I2 or CV_CONTOURS_MATCH_I3.
parameter
比较方法的参数 (目前不用).

函数 cvMatchShapes 比较两个形状。 三个实现方法全部使用 Hu 矩 (见 cvGetHuMoments) (A ~ object1, B - object2):

method=CV_CONTOUR_MATCH_I1:

$$I_1(A,B) = \sum_{i=1}^{7} \left| \frac{1}{m^{A_i}} - \frac{1}{m^{B_i}} \right|$$

method=CV_CONTOUR_MATCH_I2:

$$I_2(A,B) = \sum_{i=1}^{7} \left| m^{A_i} - m^{B_i} \right|$$

method=CV_CONTOUR_MATCH_I3:

$$I_3(A,B) = \sum_{i=1}^{7} \frac{\left| m^{A_i} - m^{B_i} \right|}{\left| m^{A_i} \right|}$$

其中

$$m^{A_i} = sign(h^{A_i}) \cdot \log(h^{A_i}),$$
$$m^{B_i} = sign(h^{B_i}) \cdot \log(h^{B_i}),$$
h^{A_i}, h^{B_i} 是A 和 B的Hu矩.

[编辑]

CalcEMD2

两个加权点集之间计算最小工作距离

```
float cvCalcEMD2( const CvArr* signature1, const CvArr* signature2, int distance_type,
                  CvDistanceFunction distance_func=NULL, const CvArr* cost_matrix=NULL,
                  CvArr* flow=NULL, float* lower_bound=NULL, void* userdata=NULL );
typedef float (*CvDistanceFunction)(const float* f1, const float* f2, void* userdata);
```

signature1
第一个签名，大小为 size1×(dims+1) 的浮点数矩阵，每一行依次存储点的权重和点的坐标。矩阵允许只有一列（即仅有权重），如果使用用户自定义的代价矩阵。
signature2
第二个签名，与 signature1 的格式一样size2×(dims+1)，尽管行数可以不同(列数要相同)。当一个额外的虚拟点加入 signature1 或 signature2 中的时候，权重也可不同。
distance_type
使用的准则， CV_DIST_L1, CV_DIST_L2, 和 CV_DIST_C 分别为标准的准则。 CV_DIST_USER 意味着使用用户自定义函数 distance_func 或预先计算好的代价矩阵 cost_matrix 。
distance_func
用户自定义的距离函数。用两个点的坐标计算两点之间的距离。
cost_matrix
自定义大小为 size1×size2 的代价矩阵。 cost_matrix 和 distance_func 两者至少有一个必须为 NULL. 而且，如果使用代价函数，下边界无法计算，因为它需要准则函数。
flow
产生的大小为 size1×size2 流矩阵（flow matrix）：flowij 是从 signature1 的第 i 个点到 signature2 的第 j 个点的流(flow)。
lower_bound
可选的输入/输出参数：两个签名之间的距离下边界，是两个质心之间的距离。如果使用自定义代价矩阵，点集的所有权重不等，或者有签名只 包含权重（即该签名矩阵只有单独一列），则下边界也许不会计算。用户必须初始化

*lower_bound. 如果质心之间的距离大于获等于 *lower_bound (这意味着签名之间足够远）， 函数则不计算 EMD. 任何情况下，函数返回时 *lower_bound 都被设置为计算出来的质心距离。因此如果用户想同时计算质心距离和T EMD, *lower_bound 应该被设置为 0.

userdata
传输到自定义距离函数的可选数据指针

函数 cvCalcEMD2 计算两个加权点集之间的移动距离或距离下界。在 [RubnerSept98] 中所描述的其中一个应用就是图像提取得多维直方图比较。 EMD 是一个使用某种单纯形算法（simplex algorithm）来解决的交通问题。其计算复杂度在最坏情况下是指数形式的，但是平均而言它的速度相当快。对实的准则，下边界的计算可以更快（使用线 性时间算法），且它可用来粗略确定两个点集是否足够远以至无法联系到同一个目标上。

Cv结构分析

Wikipedia，自由的百科全书

目录

[隐藏]

[编辑]

轮廓处理函数

[编辑]

ApproxChains

用多边形曲线逼近 Freeman 链

```
CvSeq* cvApproxChains( CvSeq* src_seq, CvMemStorage* storage,
                       int method=CV_CHAIN_APPROX_SIMPLE,
                       double parameter=0, int minimal_perimeter=0, int recursive=0 );
```

src_seq
　　涉及其它链的链指针
storage
　　存储多边形线段位置的缓存
method
　　逼近方法 (见函数 cvFindContours 的描述).
parameter
　　方法参数(现在不用).
minimal_perimeter
　　仅逼近周长大于 minimal_perimeter 轮廓。其它的链从结果中除去。
recursive
　　如果非 0, 函数从 src_seq 中利用 h_next 和 v_next links 连接逼近所有可访问的链。如果为 0, 则仅逼近单链。

这是一个单独的逼近程序。 对同样的逼近标识，函数 cvApproxChains 与 cvFindContours 的工作方式一模一样。它返回发现的第一个轮廓的指针。其它的逼近模块，可以用返回结构中的 v_next 和 v_next 域来访问

[编辑]

StartReadChainPoints

初始化链读取

```
void cvStartReadChainPoints( CvChain* chain, CvChainPtReader* reader );
```

chain
　　链的指针

reader
　　链的读取状态

函数 cvStartReadChainPoints 初始化一个特殊的读取器 (参考 Dynamic Data Structures 以获得关于集合与序列的更多内容).

[编辑]

ReadChainPoint

得到下一个链的点

```
CvPoint cvReadChainPoint( CvChainPtReader* reader );
```

reader
　　链的读取状态

函数 cvReadChainPoint 返回当前链的点，并且更新读取位置。

[编辑]

ApproxPoly

用指定精度逼近多边形曲线

```
CvSeq* cvApproxPoly( const void* src_seq, int header_size, CvMemStorage* storage,
                     int method, double parameter, int parameter2=0 );
```

src_seq
: 点集数组序列

header_size
: 逼近曲线的头尺寸

storage
: 逼近轮廓的容器。如果为 NULL， 则使用输入的序列

method
: 逼近方法。目前仅支持 CV_POLY_APPROX_DP ， 对应 Douglas-Peucker 算法.

parameter
: 方法相关参数。对 CV_POLY_APPROX_DP 它是指定的逼近精度

parameter2
: 如果 src_seq 是序列，它表示要么逼近单个序列，要么在 src_seq 的同一个或低级层次上逼近所有序列 (参考 cvFindContours 中对轮廓继承结构的描述). 如果 src_seq 是点集的数组 (CvMat*) ， 参数指定曲线是闭合 (parameter2!=0) 还是非闭合 (parameter2=0).

函数 cvApproxPoly 逼近一个或多个曲线，并返回逼近结果。对多个曲线的逼近，生成的树将与输入的具有同样的结构。(1:1 的对应关系).

[编辑]

BoundingRect

计算点集的最外面（up-right）矩形边界

```
CvRect cvBoundingRect( CvArr* points, int update=0 );
```

points
: 二维点集，点的序列或向量 (CvMat)

update
: 更新标识。下面是轮廓类型和标识的一些可能组合:

- update=0, contour ~ CvContour*: 不计算矩形边界，但直接由轮廓头的 rect 域得到。
- update=1, contour ~ CvContour*: 计算矩形边界，而且将结果写入到轮廓头的 rect 域中 header.
- update=0, contour ~ CvSeq* or CvMat*: 计算并返回边界矩形
- update=1, contour ~ CvSeq* or CvMat*: 产生运行错误 （runtime error is raised）

函数 cvBoundingRect 返回二维点集的最外面 （up-right）矩形边界。

[编辑]

ContourArea

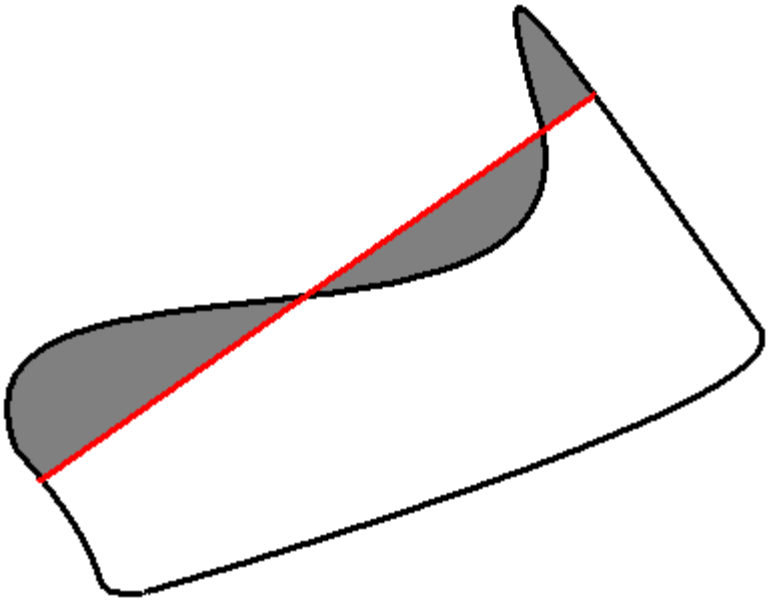
计算整个轮廓或部分轮廓的面积

```
double cvContourArea( const CvArr* contour, CvSlice slice=CV_WHOLE_SEQ );
```

contour
: 轮廓 (边界点的序列或数组).

slice
: 感兴趣轮廓部分的起始点，缺省是计算整个轮廓的面积。

函数 cvContourArea 计算整个轮廓或部分轮廓的面积。 对后面的情况，面积表示轮廓部分和起始点连线构成的封闭部分的面积。如下图所示:



备注: 轮廓的方向影响面积的符号。因此函数也许会返回负的结果。应用函数 fabs() 得到面积的绝对值。

[编辑]

ArcLength

计算轮廓周长或曲线长度

```
double cvArcLength( const void* curve, CvSlice slice=CV_WHOLE_SEQ, int is_closed=-1 );
```

curve
　　曲线点集序列或数组
slice
　　曲线的起始点，缺省是计算整个曲线的长度
is_closed
　　表示曲线是否闭合，有三种情况：

- is_closed=0 - 假设曲线不闭合
- is_closed>0 - 假设曲线闭合
- is_closed<0 - 若曲线是序列，检查 ((CvSeq*)curve)->flags 中的标识 CV_SEQ_FLAG_CLOSED 来确定曲线是否闭合。否则 (曲线由点集的数组 (CvMat*) 表示) 假设曲线不闭合。

函数 cvArcLength 通过依次计算序列点之间的线段长度，并求和来得到曲线的长度。

[编辑]

CreateContourTree

创建轮廓的继承表示形式

```
CvContourTree* cvCreateContourTree( const CvSeq* contour, CvMemStorage* storage, double threshold
);
```

contour
　　输入的轮廓
storage
　　输出树的容器
threshold
　　逼近精度

函数 cvCreateContourTree 为输入轮廓 contour 创建一个二叉树，并返回树根的指针。如果参数 threshold 小于或等于 0 ,则函数创建一个完整的二叉树。如果 threshold 大于 0 ， 函数用 threshold 指定的精度创建二叉树：如果基线的截断区域顶点小于threshold，该数就停止生长并作为函数的最终结果返回。

[编辑]

ContourFromContourTree

由树恢复轮廓

```
CvSeq* cvContourFromContourTree( const CvContourTree* tree, CvMemStorage* storage,
                                 CvTermCriteria criteria );
```

tree
　　轮廓树
storage
　　重构的轮廓容器
criteria
　　停止重构的准则

函数 cvContourFromContourTree 从二叉树恢复轮廓。参数 criteria 决定了重构的精度和使用树的数目及层次。所以它可建立逼近的轮廓。 函数返回重构的轮廓。

[编辑]

MatchContourTrees

用树的形式比较两个轮廓

```
double cvMatchContourTrees( const CvContourTree* tree1, const CvContourTree* tree2,
                            int method, double threshold );
```

tree1
　　第一个轮廓树
tree2
　　第二个轮廓树
method
　　相似度。仅支持 CV_CONTOUR_TREES_MATCH_I1 。
threshold
　　相似度阈值

函数 cvMatchContourTrees 计算两个轮廓树的匹配值。从树根开始通过逐层比较来计算相似度。如果某层的相似度小于 threshold, 则中断比较过程，且返回当前的差值。

[编辑]

计算几何

[编辑]

MaxRect

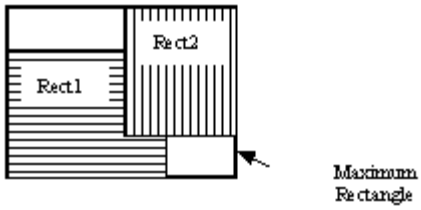
对两个给定矩形，寻找矩形边界

```
CvRect cvMaxRect( const CvRect* rect1, const CvRect* rect2 );
```

rect1
　　第一个矩形
rect2

第二个矩形

函数 cvMaxRect 寻找包含两个输入矩形的具有最小面积的矩形边界。



[编辑]

CvBox2D

旋转的二维盒子

```
typedef struct CvBox2D
{
    CvPoint2D32f center;  /* 盒子的中心 */
    CvSize2D32f  size;    /* 盒子的长和宽 */
    float angle;          /* 水平轴与第一个边的夹角，用角度度表示*/
}
CvBox2D;
```

[编辑]

PointSeqFromMat

从点向量中初始化点序列头部

```
CvSeq* cvPointSeqFromMat( int seq_kind, const CvArr* mat,
                          CvContour* contour_header,
                          CvSeqBlock* block );
```

seq_kind

点序列的类型：一系列点(0),曲线(CV_SEQ_KIND_CURVE),封闭曲线(CV_SEQ_KIND_CURVE+CV_SEQ_FLAG_CLOSED) 等等。

mat
: 输入矩阵。输入应该是连续的一维点向量，类型也应该是CV_32SC2或者CV_32FC2.

contour_header
: 轮廓头部，被函数初始化。

block
: 序列块头部，被函数初始化。

函数cvPointSeqFromMat 初始化序列头部，用来创建一个将给定矩阵中的元素形成的"虚拟"序列。没有数据被拷贝。被初始化的头部可以传递给其他任何包含输入点序列的函数。没有额外 的元素加入序列，但是一些可能被移除。函数是cvMakeSeqHeaderForArray 的一个特别的变量，然后在内部使用。它返回初始化头部的指针。需要注意的是，包含的边界矩形（CvContour 的rect字段）没有被初始化，如果你需要使用，需要自己调用cvBoundingRect。

以下是使用例子。

```
CvContour header;
CvSeqBlock block;
CvMat* vector = cvCreateMat( 1, 3, CV_32SC2 );
```

```
CV_MAT_ELEM( *vector, CvPoint, 0, 0 ) = cvPoint(100,100);
CV_MAT_ELEM( *vector, CvPoint, 0, 1 ) = cvPoint(100,200);
CV_MAT_ELEM( *vector, CvPoint, 0, 2 ) = cvPoint(200,100);

IplImage* img = cvCreateImage( cvSize(300,300), 8, 3 );
cvZero(img);

cvDrawContours( img, cvPointSeqFromMat(CV_SEQ_KIND_CURVE+CV_SEQ_FLAG_CLOSED,
   vector, &header, &block), CV_RGB(255,0,0), CV_RGB(255,0,0), 0, 3, 8, cvPoint(0,0));
```

[编辑]

BoxPoints

寻找盒子的顶点

```
void cvBoxPoints( CvBox2D box, CvPoint2D32f pt[4] );
```

box
　　盒子
pt
　　顶点数组

函数 cvBoxPoints 计算输入的二维盒子的顶点。下面是函数代码:

```
void cvBoxPoints( CvBox2D box, CvPoint2D32f pt[4] )
{
    double angle = box.angle*CV_PI/180.
    float a = (float)cos(angle)*0.5f;
    float b = (float)sin(angle)*0.5f;

    pt[0].x = box.center.x - a*box.size.height - b*box.size.width;
    pt[0].y = box.center.y + b*box.size.height - a*box.size.width;
    pt[1].x = box.center.x + a*box.size.height - b*box.size.width;
    pt[1].y = box.center.y - b*box.size.height - a*box.size.width;
    pt[2].x = 2*box.center.x - pt[0].x;
    pt[2].y = 2*box.center.y - pt[0].y;
    pt[3].x = 2*box.center.x - pt[1].x;
    pt[3].y = 2*box.center.y - pt[1].y;
}
```

[编辑]

FitEllipse

二维点集的椭圆拟合

```
CvBox2D cvFitEllipse2( const CvArr* points );
```

points
　　点集的序列或数组

函数 cvFitEllipse 对给定的一组二维点集作椭圆的最佳拟合(最小二乘意义上的)。返回的结构与 cvEllipse 中的意义类似，除了 size 表示椭圆轴的整个长度，而不是一半长度。

[编辑]

FitLine

2D 或 3D 点集的直线拟合

```
void  cvFitLine( const CvArr* points, int dist_type, double param,
                 double reps, double aeps, float* line );
```

points
　　2D 或 3D 点集，32-比特整数或浮点数坐标

dist_type
　　拟合的距离类型 （见讨论）.
param
　　对某些距离的数字参数，如果是 0， 则选择某些最优值
reps, aeps
　　半径 (坐标原点到直线的距离) 和角度的精度，一般设为0.01。
line
　　输出的直线参数。2D 拟合情况下，它是包含 4 个浮点数的数组 (vx, vy, x0, y0)，其中 (vx, vy) 是线的单位向量而 (x0, y0) 是线上的某个点. 对 3D 拟合，它是包含 6 个浮点数的数组 (vx, vy, vz, x0, y0, z0)， 其中 (vx, vy, vz) 是线的单位向量，而 (x0, y0, z0) 是线上某点。

函数 cvFitLine 通过求 $sum_i:\rho(r_i)$ 的最小值方法，用 2D 或 3D 点集拟合直线，其中 r_i 是第 i 个点到直线的距离， ρ(r) 是下面的距离函数之一:

dist_type=CV_DIST_L2 (L_2): $\rho(r)=r^2/2$ (最简单和最快的最小二乘法)

dist_type=CV_DIST_L1 (L_1): $\rho(r)=r$

dist_type=CV_DIST_L12 (L_1-L_2): $\rho(r)=2\bullet[\sqrt{1+r^2/2} - 1]$

dist_type=CV_DIST_FAIR (Fair): $\rho(r)=C^2\bullet[r/C - \log(1 + r/C)]$, C=1.3998

dist_type=CV_DIST_WELSCH (Welsch): $\rho(r)=C^2/2\bullet[1 - \exp(-(r/C)^2)]$, C=2.9846

dist_type=CV_DIST_HUBER (Huber): $\rho(r)= r^2/2$, if r < C; $C\bullet(r-C/2)$, otherwise; C=1.345

[编辑]

ConvexHull2

发现点集的凸外形

```
CvSeq* cvConvexHull2( const CvArr* input, void* hull_storage=NULL,
                      int orientation=CV_CLOCKWISE, int return_points=0 );
```

points
　　2D 点集的序列或数组，32-比特整数或浮点数坐标
hull_storage
　　输出的数组(CvMat*) 或内存缓存 (CvMemStorage*)，用以存储凸外形。 如果是数组，则它应该是一维的，而且与输入的数组/序列具有同样数目的元素。输出时，通过修改头结构将数组裁减到凸外形的尺寸。
orientation
　　凸外形的旋转方向: 逆时针或顺时针 （CV_CLOCKWISE or CV_COUNTER_CLOCKWISE）
return_points
　　如果非零，hull_storage 为数组情况下,点集将以外形 (hull) 存储，而不是顶点形式 (indices)。如果 hull_storag 为内存存储模式下则存储为点集形式(points)。

函数 cvConvexHull2 使用 Sklansky 算法计算 2D 点集的凸外形。如果 hull_storage 是内存存储仓， 函数根据 return_points 的值，创建一个包含外形的点集或指向这些点的指针的序列。

例子. 由点集序列或数组创建凸外形

```
#include "cv.h"
#include "highgui.h"
#include <stdlib.h>

#define ARRAY  0 /* switch between array/sequence method by replacing 0<=>1 */
```

```
void main( int argc, char** argv )
{
    IplImage* img = cvCreateImage( cvSize( 500, 500 ), 8, 3 );
    cvNamedWindow( "hull", 1 );

#if !ARRAY
        CvMemStorage* storage = cvCreateMemStorage();
#endif

    for(;;)
    {
        int i, count = rand()%100 + 1, hullcount;
        CvPoint pt0;
#if !ARRAY
        CvSeq* ptseq = cvCreateSeq( CV_SEQ_KIND_GENERIC|CV_32SC2, sizeof(CvContour),
                                    sizeof(CvPoint), storage );
        CvSeq* hull;

        for( i = 0; i < count; i++ )
        {
            pt0.x = rand() % (img->width/2) + img->width/4;
            pt0.y = rand() % (img->height/2) + img->height/4;
            cvSeqPush( ptseq, &pt0 );
        }
        hull = cvConvexHull2( ptseq, 0, CV_CLOCKWISE, 0 );
        hullcount = hull->total;
#else
        CvPoint* points = (CvPoint*)malloc( count * sizeof(points[0]));
        int* hull = (int*)malloc( count * sizeof(hull[0]));
        CvMat point_mat = cvMat( 1, count, CV_32SC2, points );
        CvMat hull_mat = cvMat( 1, count, CV_32SC1, hull );

        for( i = 0; i < count; i++ )
        {
            pt0.x = rand() % (img->width/2) + img->width/4;
            pt0.y = rand() % (img->height/2) + img->height/4;
            points[i] = pt0;
        }
        cvConvexHull2( &point_mat, &hull_mat, CV_CLOCKWISE, 0 );
        hullcount = hull_mat.cols;
#endif
        cvZero( img );
        for( i = 0; i < count; i++ )
        {
#if !ARRAY
            pt0 = *CV_GET_SEQ_ELEM( CvPoint, ptseq, i );
#else
            pt0 = points[i];
#endif
            cvCircle( img, pt0, 2, CV_RGB( 255, 0, 0 ), CV_FILLED );
        }

#if !ARRAY
        pt0 = **CV_GET_SEQ_ELEM( CvPoint*, hull, hullcount - 1 );
#else
        pt0 = points[hull[hullcount-1]];
#endif

        for( i = 0; i < hullcount; i++ )
        {
#if !ARRAY
            CvPoint pt = **CV_GET_SEQ_ELEM( CvPoint*, hull, i );
#else
            CvPoint pt = points[hull[i]];
#endif
            cvLine( img, pt0, pt, CV_RGB( 0, 255, 0 ));
            pt0 = pt;
        }

        cvShowImage( "hull", img );

        int key = cvWaitKey(0);
        if( key == 27 ) // 'ESC'
            break;

#if !ARRAY
        cvClearMemStorage( storage );
#else
        free( points );
        free( hull );
```

```
#endif
    }
}
```

[编辑]

CheckContourConvexity

测试轮廓的凸性

```
int cvCheckContourConvexity( const CvArr* contour );
```

contour
被测试轮廓 (点序列或数组).

函数 cvCheckContourConvexity 输入的轮廓是否为凸的。必须是简单轮廓，比如没有自交叉。

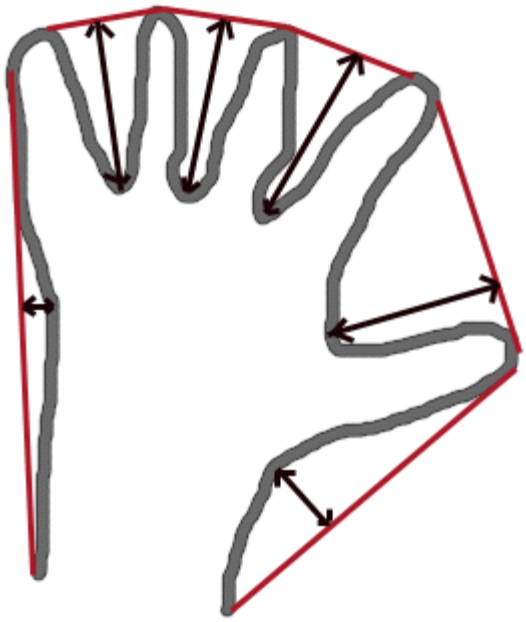
[编辑]

CvConvexityDefect

用来描述一个简单轮廓凸性缺陷的结构体

```
typedef struct CvConvexityDefect
{
    CvPoint* start; /* 缺陷开始的轮廓点 */
    CvPoint* end; /* 缺陷结束的轮廓点 */
    CvPoint* depth_point; /* 缺陷中距离凸形最远的轮廓点(谷底) */
    float depth; /* 谷底距离凸形的深度*/
} CvConvexityDefect;
```

Picture. 手部轮廓的凸形缺陷.



[编辑]

ConvexityDefects

发现轮廓凸形缺陷

```
CvSeq* cvConvexityDefects( const CvArr* contour, const CvArr* convexhull,
                           CvMemStorage* storage=NULL );
```

contour

输入轮廓

convexhull

用 cvConvexHull2 得到的凸外形，它应该包含轮廓的定点的指针或下标，而不是外形点的本身，即cvConvexHull2 中的参数 return_points 应该设置为 0.

storage

凸性缺陷的输出序列容器。如果为 NULL, 使用轮廓或外形的存储仓。

函数 cvConvexityDefects 发现输入轮廓的所有凸性缺陷，并且返回 CvConvexityDefect 结构序列。

[编辑]

PointPolygonTest

测试点是否在多边形中

```
double cvPointPolygonTest( const CvArr* contour,
                           CvPoint2D32f pt, int measure_dist );
```

contour

输入轮廓.

pt

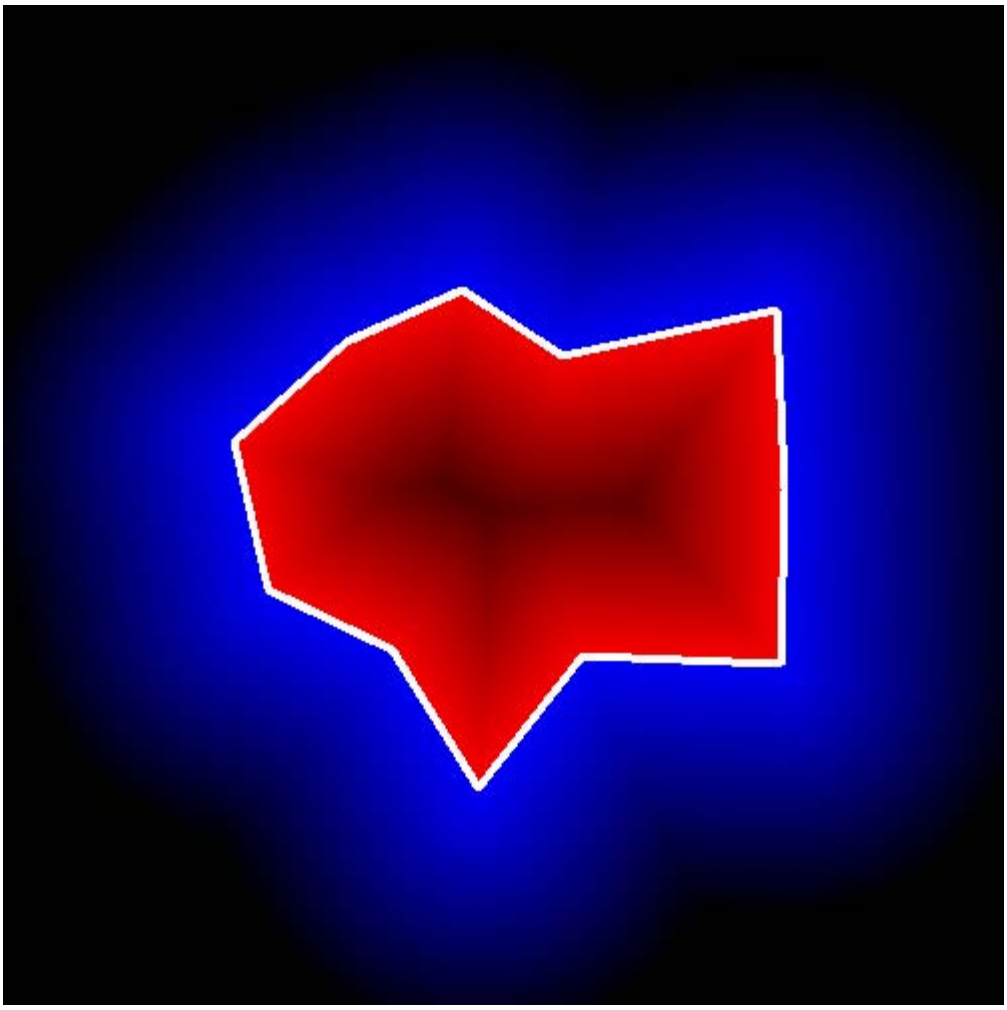
针对轮廓需要测试的点。

measure_dist

如果非0，函数将估算点到轮廓最近边的距离。

函数cvPointPolygonTest 决定测试点是否在轮廓内，轮廓外，还是轮廓的边上（或者共边的交点上），它的返回值是正负零，相对应的，当measure_dist=0时，返回值是1, -1,0, 同样当 measure_dist≠0 ，它是返回一个从点到最近的边的带符号距离。

下面是函数输出的结果，用图片的每一个象素去测试轮廓的结果。



[编辑]

MinAreaRect2

对给定的 2D 点集，寻找最小面积的包围矩形

```
CvBox2D  cvMinAreaRect2( const CvArr* points, CvMemStorage* storage=NULL );
```

points
　　点序列或点集数组
storage
　　可选的临时存储仓

函数 cvMinAreaRect2 通过建立凸外形并且旋转外形以寻找给定 2D 点集的最小面积的包围矩形.

Picture. Minimal-area bounding rectangle for contour

[编辑]

MinEnclosingCircle

对给定的 2D 点集，寻找最小面积的包围圆形

```
int cvMinEnclosingCircle( const CvArr* points, CvPoint2D32f* center, float* radius );
```

points
　　点序列或点集数组
center
　　输出参数：圆心
radius
　　输出参数：半径

函数 cvMinEnclosingCircle 对给定的 2D 点集迭代寻找最小面积的包围圆形。如果产生的圆包含所有点，返回非零。否则返回零（算法失败）。

CalcPGH

计算轮廓的 pair-wise 几何直方图

```
void cvCalcPGH( const CvSeq* contour, CvHistogram* hist );
```

contour
　　输入轮廓，当前仅仅支持具有整数坐标的点集
hist
　　计算出的直方图，必须是两维的。

函数 cvCalcPGH 计算轮廓的 2D pair-wise 几何直方图 （2D pair-wise geometrical histogram： PGH), 算法描述见 [Iivarinen97]. 算法考虑的每一对轮廓边缘。计算每一对边缘之间的夹角以及最大最小距离。具体做法是，轮流考虑每一个边缘做为基准，函数循环遍历所有其他的边缘。在考虑基 准边缘和其它边缘的时候， 选择非基准线上的点到基准线上的最大和最小距离。边缘之间的角度定义了直方图的行，而在其中增加对应计算出来的最大和最小距离的所有直方块， (即直方图是 [Iivarninen97] 定义中的转置). 该直方图用来做轮廓匹配。

平面划分

CvSubdiv2D

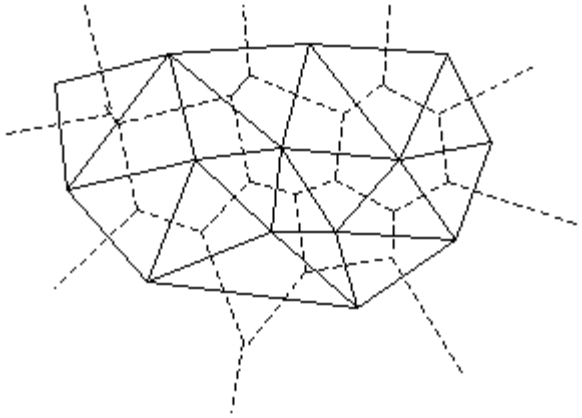
平面划分

```
#define CV_SUBDIV2D_FIELDS()    \
    CV_GRAPH_FIELDS()           \
    int  quad_edges;            \
    int  is_geometry_valid;     \
    CvSubdiv2DEdge recent_edge; \
    CvPoint2D32f  topleft;      \
    CvPoint2D32f  bottomright;

typedef struct CvSubdiv2D
{
    CV_SUBDIV2D_FIELDS()
}
CvSubdiv2D;
```

平面划分是将一个平面分割为一组互不重叠的能够覆盖整个平面的区域P(facets)。上面结构描述了建立在 2D 点集上的划分结构，其中点集互相连接并且构成平面图形，该图形通过结合一些无限连接外部划分点(称为凸形点)的边缘，将一个平面用边按照其边缘划分成很多 小区域(facets)。

对于每一个划分操作，都有一个对偶划分与之对应，对偶的意思是小区域和点(划分的顶点)变换角色，即在对偶划分中，小区域被当做一个顶点(以下称之为虚拟点)，而原始的划分顶点被当做小区域。在如下所示的图例中，原始的划分用实线来表示，而对偶划分用点线来表示。

OpenCV 使用Delaunay's 算法将平面分割成小的三角形区域。分割的实现通过从一个假定的三角形(该三角形确保包括所有的分割点)开始不断迭代来完成。在这种情况下，对偶划分就是输 入的2d点集的 Voronoi图表。这种划分可以用于对一个平面的3d分段变换、形态变换、平面点的快速定位以及建立特定的图结构 (比如NNG,RNG等等)。



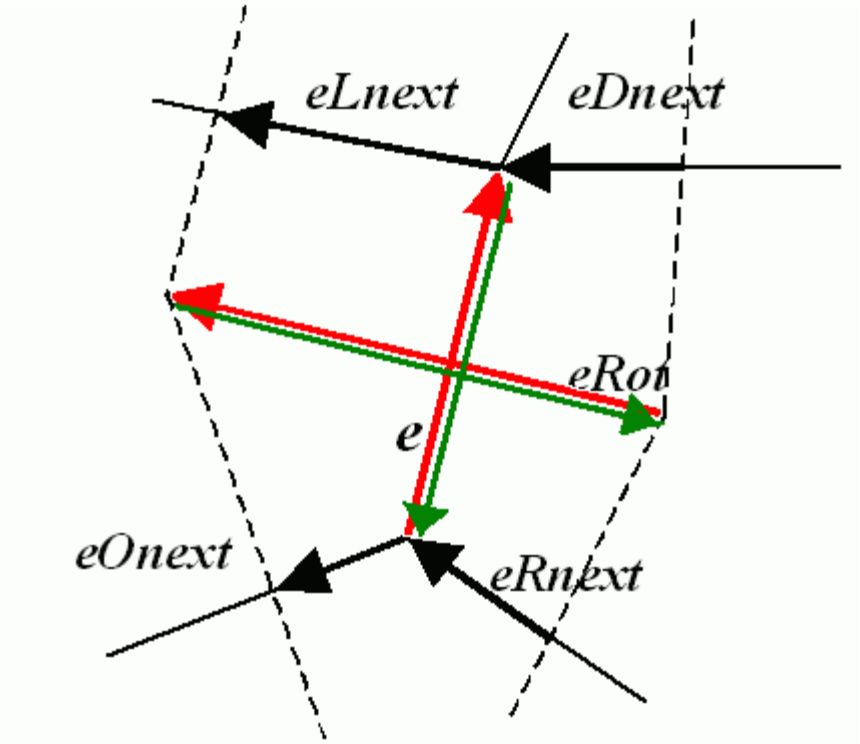
[编辑]

CvQuadEdge2D

平面划分中的Quad-edge(四方边缘结构)

```
/* quad-edge中的一条边缘，低两位表示该边缘的索引号，其它高位表示边缘指针。 */
typedef long CvSubdiv2DEdge;

/* 四方边缘的结构场 */
#define CV_QUADEDGE2D_FIELDS()     \
    int flags;                     \
    struct CvSubdiv2DPoint* pt[4]; \
    CvSubdiv2DEdge  next[4];

typedef struct CvQuadEdge2D
{
    CV_QUADEDGE2D_FIELDS()
}
CvQuadEdge2D;
```

Quad-edge(译者注：以下称之为四方边缘结构)是平面划分的基元，其中包括四个边缘 (e, eRot(红色) 以及它们的逆（绿色）)。



[编辑]

CvSubdiv2DPoint

原始和对偶划分点

```
#define CV_SUBDIV2D_POINT_FIELDS()\
    int            flags;      \
    CvSubdiv2DEdge first;      \
    CvPoint2D32f   pt;

#define CV_SUBDIV2D_VIRTUAL_POINT_FLAG (1 << 30)

typedef struct CvSubdiv2DPoint
{
    CV_SUBDIV2D_POINT_FIELDS()
}
CvSubdiv2DPoint;
```

[编辑]

Subdiv2DGetEdge

返回给定的边缘之一

```
CvSubdiv2DEdge  cvSubdiv2DGetEdge( CvSubdiv2DEdge edge, CvNextEdgeType type );
#define cvSubdiv2DNextEdge( edge ) cvSubdiv2DGetEdge( edge, CV_NEXT_AROUND_ORG )
```

edge
: 划分的边缘 (并不是四方边缘结构)

type
: 确定函数返回哪条相关边缘，是下面几种之一:

- CV_NEXT_AROUND_ORG - 边缘原点的下一条 (eOnext on the picture above if e is the input edge)
- CV_NEXT_AROUND_DST - 边缘顶点的下一条 (eDnext)
- CV_PREV_AROUND_ORG - 边缘原点的前一条 (reversed eRnext)
- CV_PREV_AROUND_DST - 边缘终点的前一条 (reversed eLnext)
- CV_NEXT_AROUND_LEFT - 左区域的下一条 (eLnext)

- CV_NEXT_AROUND_RIGHT - 右区域的下一条(eRnext)
- CV_PREV_AROUND_LEFT - 左区域的前一条 (reversed eOnext)
- CV_PREV_AROUND_RIGHT - 右区域的前一条 (reversed eDnext)

函数 cvSubdiv2DGetEdge 返回与输入边缘相关的边缘

[编辑]

Subdiv2DRotateEdge

返回同一个四方边缘结构中的另一条边缘

```
CvSubdiv2DEdge  cvSubdiv2DRotateEdge( CvSubdiv2DEdge edge, int rotate );
```

edge
: 划分的边缘 (并不是四方边缘结构)

type
: 确定函数根据输入的边缘返回同一四方边缘结构中的哪条边缘，是下面几种之一:

- 0 - 输入边缘 (上图中的e，如果e是输入边缘)
- 1 - 旋转边缘 (eRot)
- 2 -逆边缘 (e的反向边缘)
- 3 - 旋转边缘的反向边缘(eRot的反向边缘， 图中绿色)

函数 cvSubdiv2DRotateEdge 根据输入的边缘返回四方边缘结构中的一条边缘

[编辑]

Subdiv2DEdgeOrg

返回边缘的原点

```
CvSubdiv2DPoint* cvSubdiv2DEdgeOrg( CvSubdiv2DEdge edge );
```

edge
: 划分的边缘 (并不是四方边缘结构)

函数 cvSubdiv2DEdgeOrg 返回边缘的原点。如果该边缘是从对偶划分得到并且虚点坐标还没有计算出来，可能返回空指针。虚点可以用函数来cvCalcSubdivVoronoi2D计算。

[编辑]

Subdiv2DEdgeDst

Returns edge destination

```
CvSubdiv2DPoint* cvSubdiv2DEdgeDst( CvSubdiv2DEdge edge );
```

edge
: 划分的边缘 (并不是四方边缘结构)

函数 cvSubdiv2DEdgeDst 返回边缘的终点。如果该边缘是从对偶划分得到并且虚点坐标还没有计算出来，可能返回空指针。虚点可以用函数来cvCalcSubdivVoronoi2D计算。

[编辑]

CreateSubdivDelaunay2D

生成的空Delaunay 三角测量

```
CvSubdiv2D* cvCreateSubdivDelaunay2D( CvRect rect, CvMemStorage* storage );
```

rect
: Rectangle包括所有待加入划分操作的2d点的四方形。

storage
: 划分操作的存储器

函数 cvCreateSubdivDelaunay2D 生成一个空的Delaunay 划分, 其中2d points可以进一步使用函数 cvSubdivDelaunay2DInsert来添加。所有的点一定要在指定的四方形中添加，否则就会报运行错误。

[编辑]

SubdivDelaunay2DInsert

向 Delaunay三角测量中插入一个点

```
CvSubdiv2DPoint*  cvSubdivDelaunay2DInsert( CvSubdiv2D* subdiv, CvPoint2D32f pt);
```

subdiv
: 通过函数 cvCreateSubdivDelaunay2D.生成的Delaunay划分

pt
: 待插入的点

函数 cvSubdivDelaunay2DInsert 向划分的结构中插入一个点并且正确地改变划分的拓朴结构。如果划分结构中已经存在一个相同的坐标点，则不会有新点插入。该函数返回指向已插入点的指针。在这个截断，不计算任何虚点坐标。

[编辑]

Subdiv2DLocate

在 Delaunay三角测量中定位输入点

```
CvSubdiv2DPointLocation  cvSubdiv2DLocate( CvSubdiv2D* subdiv, CvPoint2D32f pt,
                                           CvSubdiv2DEdge* edge,
                                           CvSubdiv2DPoint** vertex=NULL );
```

subdiv
: Delaunay 或者是其它分割结构.

pt
: 待定位的输入点

edge
: 与输入点对应的输入边缘(点在其上或者其右)

vertex
: 与输入点对应的输出顶点坐标(指向double类型)，可选。

函数 cvSubdiv2DLocate 在划分中定位输入点，共有5种类型:

- 输入点落入某小区域内。 函数返回参数 CV_PTLOC_INSIDE 且*edge 中包含小区域的边缘之一。
- 输入点落p在边缘之上。 函数返回参数 CV_PTLOC_ON_EDGE 且 *edge 包含此边缘。
- 输入点与划分的顶点之一相对应。 函数返回参数 CV_PTLOC_VERTEX 且 *vertex 中包括指向该顶点的指针;
- 输入点落在划分的参考区域之外。 函数返回参数 CV_PTLOC_OUTSIDE_RECT且不填写任何指针。
- 输入参数之一有误。函数报运行错误(如果已经选则了沉默或者父母出错模式，则函数返回CV_PTLOC_ERROR) 。

[编辑]

FindNearestPoint2D

根据输入点，找到其最近的划分顶点

```
CvSubdiv2DPoint* cvFindNearestPoint2D( CvSubdiv2D* subdiv, CvPoint2D32f pt );
```

subdiv
: Delaunay或者其它划分方式

pt
: 输入点

函数 cvFindNearestPoint2D 是另一个定位输入点的函数。该函数找到输入点的最近划分顶点。尽管划分出的小区域(facet)被用来作为起始点，但是输入点不一定非得在最终找到的顶点所在的小区域之内。该函数返回指向找到的划分顶点的指针。

[编辑]

CalcSubdivVoronoi2D

计算Voronoi图表的细胞结构

```
void cvCalcSubdivVoronoi2D( CvSubdiv2D* subdiv );
```

subdiv
: Delaunay 划分,其中所有的点已经添加 。

函数 cvCalcSubdivVoronoi2D 计算虚点的坐标，所有与原划分中的某顶点相对应的虚点形成了(当他们相互连接时)该顶点的Voronoi 细胞的边界。

[编辑]

ClearSubdivVoronoi2D

移除所有的虚点

```
void cvClearSubdivVoronoi2D( CvSubdiv2D* subdiv );
```

subdiv
: Delaunay 划分

函数 cvClearSubdivVoronoi2D 移除所有的虚点。当划分的结果被函数cvCalcSubdivVoronoi2D的前一次调用更改时，该函数被cvCalcSubdivVoronoi2D内部调用 。

还有一些其它的底层处理函数与平面划分操作协同工作，参见 cv.h 及源码。生成 delaunay.c 三角测量以及2d随机点集的Voronoi 图表的演示代码可以在 opencv/samples/c目录下的delaunay.c 文件中找到。

Cv运动分析与对象跟踪

Wikipedia，自由的百科全书

目录

[隐藏]

[编辑]

背景统计量的累积

[编辑]

Acc

将帧叠加到累积器（accumulator）中

```
void cvAcc( const CvArr* image, CvArr* sum, const CvArr* mask=NULL );
```

image
: 输入图像, 1- 或 3-通道, 8-比特或32-比特浮点数. (多通道的每一个通道都单独处理).

sum
　　同一个输入图像通道的累积，32-比特或64-比特浮点数数组.
mask
　　可选的运算 mask.

函数 cvAcc 将整个图像 image 或某个选择区域叠加到 sum 中:

```
sum(x,y)=sum(x,y)+image(x,y) if mask(x,y)!=0
```

[编辑]

SquareAcc

叠加输入图像的平方到累积器中

```
void cvSquareAcc( const CvArr* image, CvArr* sqsum, const CvArr* mask=NULL );
```

image
　　输入图像, 1- 或 3-通道, 8-比特或32-比特浮点数 (多通道的每一个通道都单独处理)
sqsum
　　同一个输入图像通道的累积，32-比特或64-比特浮点数数组.
mask
　　可选的运算 mask.

函数 cvSquareAcc 叠加输入图像 image 或某个选择区域的二次方，到累积器 sqsum 中

```
sqsum(x,y)=sqsum(x,y)+image(x,y)2 if mask(x,y)!=0
```

[编辑]

MultiplyAcc

将两幅输入图像的乘积叠加到累积器中

```
void cvMultiplyAcc( const CvArr* image1, const CvArr* image2, CvArr* acc, const CvArr* mask=NULL
);
```

image1
　　第一个输入图像, 1- or 3-通道, 8-比特 or 32-比特 浮点数 (多通道的每一个通道都单独处理)
image2
　　第二个输入图像, 与第一个图像的格式一样
acc
　　同一个输入图像通道的累积，32-比特或64-比特浮点数数组.
mask
　　可选的运算 mask.

函数 cvMultiplyAcc 叠加两个输入图像的乘积到累积器 acc:

```
acc(x,y)=acc(x,y) + image1(x,y)•image2(x,y) if mask(x,y)!=0
```

[编辑]

RunningAvg

更新 running average 滑动平均（ Hunnish: 不知道 running average 如何翻译才恰当）

```
void cvRunningAvg( const CvArr* image, CvArr* acc, double alpha, const CvArr* mask=NULL );
```

image
　　输入图像, 1- or 3-通道, 8-比特 or 32-比特 浮点数 (each channel of multi-channel image is processed independently).

acc
　　同一个输入图像通道的累积，32-比特或64-比特浮点数数组.
alpha
　　输入图像权重
mask
　　可选的运算 mask

函数 cvRunningAvg 计算输入图像 image 的加权和，以及累积器 acc 使得 acc 成为帧序列的一个 running average：

```
acc(x,y)=(1-α)•acc(x,y) + α•image(x,y) if mask(x,y)!=0
```

其中 α (alpha) 调节更新速率 (累积器以多快的速率忘掉前面的帧).

[编辑]

运动模板

[编辑]

UpdateMotionHistory

去掉影像(silhouette) 以更新运动历史图像

```
void cvUpdateMotionHistory( const CvArr* silhouette, CvArr* mhi,
                            double timestamp, double duration );
```

silhouette
　　影像 mask，运动发生地方具有非零象素
mhi
　　运动历史图像(单通道, 32-比特 浮点数)，为本函数所更新
timestamp
　　当前时间，毫秒或其它单位
duration
　　运动跟踪的最大持续时间，用 timestamp 一样的时间单位

函数 cvUpdateMotionHistory 用下面方式更新运动历史图像：

```
mhi(x,y)=timestamp  if silhouette(x,y)!=0
         0          if silhouette(x,y)=0 and mhi(x,y)<timestamp-duration
         mhi(x,y)   otherwise
```

也就是，MHI（motion history image） 中在运动发生的象素点被设置为当前时间戳，而运动发生较久的象素点被清除。

[编辑]

CalcMotionGradient

计算运动历史图像的梯度方向

```
void cvCalcMotionGradient( const CvArr* mhi, CvArr* mask, CvArr* orientation,
                           double delta1, double delta2, int aperture_size=3 );
```

mhi
　　运动历史图像
mask
　　Mask 图像；用来标注运动梯度数据正确的点，为输出参数。
orientation
　　运动梯度的方向图像，包含从 0 到 360 角度

delta1, delta2
函数在每个象素点 (x,y) 邻域寻找 MHI 的最小值 (m(x,y)) 和最大值 (M(x,y))，并且假设梯度是正确的，当且仅当：

min(delta1,delta2) <= M(x,y)-m(x,y) <= max(delta1,delta2).

aperture_size
函数所用微分算子的开孔尺寸 CV_SCHARR, 1, 3, 5 or 7 (见 cvSobel).

函数 cvCalcMotionGradient 计算 MHI 的差分 Dx 和 Dy ，然后计算梯度方向如下式：

orientation(x,y)=arctan(Dy(x,y)/Dx(x,y))

其中都要考虑 Dx(x,y)' 和 Dy(x,y)' 的符号 (如 cvCartToPolar 类似). 然后填充 mask 以表示哪些方向是正确的(见 delta1 和delta2 的描述).

[编辑]

CalcGlobalOrientation

计算某些选择区域的全局运动方向

```
double cvCalcGlobalOrientation( const CvArr* orientation, const CvArr* mask, const CvArr* mhi,
                                double timestamp, double duration );
```

orientation
运动梯度方向图像，由函数 cvCalcMotionGradient 得到
mask
Mask 图像. 它可以是正确梯度 mask (由函数 cvCalcMotionGradient 得到)与区域 mask 的结合，其中区域 mask 确定哪些方向需要计算。
mhi
运动历史图象
timestamp
当前时间（单位毫秒或其它）最好在传递它到函数 cvUpdateMotionHistory 之前存储一下以便以后的重用，因为对大图像运行 cvUpdateMotionHistory 和 cvCalcMotionGradient 会花费一些时间
duration
运动跟踪的最大持续时间，用法与 cvUpdateMotionHistory 中的一致

函数 cvCalcGlobalOrientation 在选择的区域内计算整个运动方向，并且返回 0° 到 360° 之间的角度值。首先函数创建运动直方图，寻找基本方向做为直方图最大值的坐标。然后函数计算与基本方向的相对偏移量，做为所有方向向量的加权和：运行越 近，权重越大。得到的角度是基本方向和偏移量的循环和。

[编辑]

SegmentMotion

将整个运动分割为独立的运动部分

```
CvSeq* cvSegmentMotion( const CvArr* mhi, CvArr* seg_mask, CvMemStorage* storage,
                        double timestamp, double seg_thresh );
```

mhi
运动历史图像
seg_mask
发现应当存储的 mask 的图像, 单通道, 32bits， 浮点数.
storage
包含运动连通域序列的内存存储仓
timestamp

当前时间，毫秒单位

seg_thresh

分割阈值，推荐等于或大于运动历史“每步”之间的间隔。

函数 cvSegmentMotion 寻找所有的运动分割，并且在seg_mask 用不同的单独数字(1,2,...)标识它们。它也返回一个具有 CvConnectedComp 结构的序列，其中每个结构对应一个运动部件。在这之后，每个运动部件的运动方向就可以被函数 cvCalcGlobalOrientation 利用提取的特定部件的掩模(mask)计算出来(使用 cvCmp)

[编辑]

对象跟踪

[编辑]

MeanShift

在反向投影图中发现目标中心

```
int cvMeanShift( const CvArr* prob_image, CvRect window,
                 CvTermCriteria criteria, CvConnectedComp* comp );
```

prob_image

目标直方图的反向投影(见 cvCalcBackProject).

window

初始搜索窗口

criteria

确定窗口搜索停止的准则

comp

生成的结构，包含收敛的搜索窗口坐标 (comp->rect 字段) 与窗口内部所有象素点的和 (comp->area 字段).

函数 cvMeanShift 在给定反向投影和初始搜索窗口位置的情况下，用迭代方法寻找目标中心。当搜索窗口中心的移动小于某个给定值时或者函数已经达到最大迭代次数时停止迭代。 函数返回迭代次数。

[编辑]

CamShift

发现目标中心，尺寸和方向

```
int cvCamShift( const CvArr* prob_image, CvRect window, CvTermCriteria criteria,
                CvConnectedComp* comp, CvBox2D* box=NULL );
```

prob_image

目标直方图的反向投影 (见 cvCalcBackProject).

window

初始搜索窗口

criteria

确定窗口搜索停止的准则

comp

生成的结构，包含收敛的搜索窗口坐标 (comp->rect 字段) 与窗口内部所有象素点的和 (comp->area 字段).

box

目标的带边界盒子。如果非 NULL, 则包含目标的尺寸和方向。

函数 cvCamShift 实现了 CAMSHIFT 目标跟踪算法([Bradski98]). 首先它调用函数 cvMeanShift 寻找目标中心，然后计算目标尺寸和方向。最后返回函数 cvMeanShift 中的迭代次数。

CvCamShiftTracker 类在 cv.hpp 中被声明，函数实现了彩色目标的跟踪。

[编辑]

SnakeImage

改变轮廓位置使得它的能量最小

```
void cvSnakeImage( const IplImage* image, CvPoint* points, int length,
                   float* alpha, float* beta, float* gamma, int coeff_usage,
                   CvSize win, CvTermCriteria criteria, int calc_gradient=1 );
```

image
　　输入图像或外部能量域
points
　　轮廓点 (snake).
length
　　轮廓点的数目
alpha
　　连续性能量的权 Weight[s]，单个浮点数或长度为 length 的浮点数数组，每个轮廓点有一个权
beta
　　曲率能量的权 Weight[s]，与 alpha 类似
gamma
　　图像能量的权 Weight[s]，与 alpha 类似
coeff_usage
　　前面三个参数的不同使用方法：

- CV_VALUE 表示每个 alpha, beta, gamma 都是指向为所有点所用的一个单独数值;
- CV_ARRAY 表示每个 alpha, beta, gamma 是一个指向系数数组的指针，snake 上面各点的系数都不相同。因此，各个系数数组必须与轮廓具有同样的大小。所有数组必须与轮廓具有同样大小

win
　　每个点用于搜索最小值的邻域尺寸，两个 win.width 和 win.height 都必须是奇数
criteria
　　终止条件
calc_gradient
　　梯度符号。如果非零，函数为每一个图像象素计算梯度幅值，且把它当成能量场，否则考虑输入图像本身。

函数 cvSnakeImage 更新 snake 是为了最小化 snake 的整个能量，其中能量是依赖于轮廓形状的内部能量(轮廓越光滑，内部能量越小)以及依赖于能量场的外部能量之和，外部能量通常在哪些局部能量极值点中达到 最小值(这些局部能量极值点与图像梯度表示的图像边缘相对应)。

参数 criteria.epsilon 用来定义必须从迭代中除掉以保证迭代正常运行的点的最少数目。

如果在迭代中去掉的点数目小于 criteria.epsilon 或者函数达到了最大的迭代次数 criteria.max_iter ，则终止函数。

[编辑]

光流

[编辑]

CalcOpticalFlowHS

计算两幅图像的光流

```
void cvCalcOpticalFlowHS( const CvArr* prev, const CvArr* curr, int use_previous,
                          CvArr* velx, CvArr* vely, double lambda,
                          CvTermCriteria criteria );
```

prev
: 第一幅图像, 8-比特, 单通道.

curr
: 第二幅图像, 8-比特, 单通道.

use_previous
: 使用以前的 (输入) 速度域

velx
: 光流的水平部分，与输入图像大小一样, 32-比特，浮点数, 单通道.

vely
: 光流的垂直部分，与输入图像大小一样, 32-比特， 浮点数, 单通道.

lambda
: Lagrangian 乘子

criteria
: 速度计算的终止条件

函数 cvCalcOpticalFlowHS 为输入图像的每一个象素计算光流，使用 Horn & Schunck 算法 [Horn81].

[编辑]

CalcOpticalFlowLK

计算两幅图像的光流

```
void cvCalcOpticalFlowLK( const CvArr* prev, const CvArr* curr, CvSize win_size,
                          CvArr* velx, CvArr* vely );
```

prev
: 第一幅图像, 8-比特, 单通道.

curr
: 第二幅图像, 8-比特, 单通道.

win_size
: 用来归类象素的平均窗口尺寸 （Size of the averaging window used for grouping pixels）

velx
: 光流的水平部分，与输入图像大小一样, 32-比特， 浮点数, 单通道.

vely
: 光流的垂直部分，与 输入图像大小一样, 32-比特， 浮点数, 单通道.

函数 cvCalcOpticalFlowLK 为输入图像的每一个象素计算光流，使用 Lucas & Kanade 算法 [Lucas81].

[编辑]

CalcOpticalFlowBM

用块匹配方法计算两幅图像的光流

```
void cvCalcOpticalFlowBM( const CvArr* prev, const CvArr* curr, CvSize block_size,
                          CvSize shift_size, CvSize max_range, int use_previous,
                          CvArr* velx, CvArr* vely );
```

prev
: 第一幅图像, 8-比特, 单通道.

curr
: 第二幅图像, 8-比特, 单通道.

block_size
: 比较的基本块尺寸

shift_size

块坐标的增量

max_range
块周围象素的扫描邻域的尺寸

use_previous
使用以前的 (输入) 速度域

velx
光流的水平部分，尺寸为 floor((prev->width - block_size.width)/shiftSize.width) × floor((prev->height - block_size.height)/shiftSize.height) ，32-比特，浮点数, 单通道.

vely
光流的垂直部分，与 velx 大小一样，32-比特，浮点数, 单通道.

函数 cvCalcOpticalFlowBM 为重叠块 block_size.width×block_size.height 中的每一个象素计算光流，因此其速度域小于整个图像的速度域。对每一个在图像 prev 中的块，函数试图在 curr 中某些原始块或其偏移 (velx(x0,y0),vely(x0,y0)) 块的邻域里寻找类似的块，如同在前一个函数调用中所计算的类似(如果 use_previous=1)

[编辑]

CalcOpticalFlowPyrLK

计算一个稀疏特征集的光流，使用金字塔中的迭代 Lucas-Kanade 方法

```
void cvCalcOpticalFlowPyrLK( const CvArr* prev, const CvArr* curr, CvArr* prev_pyr, CvArr*
curr_pyr,
                             const CvPoint2D32f* prev_features, CvPoint2D32f* curr_features,
                             int count, CvSize win_size, int level, char* status,
                             float* track_error, CvTermCriteria criteria, int flags );
```

prev
在时间 t 的第一帧

curr
在时间 t + dt 的第二帧

prev_pyr
第一帧的金字塔缓存. 如果指针非 NULL ，则缓存必须有足够的空间来存储金字塔从层 1 到层 #level 的内容。尺寸 (image_width+8)*image_height/3 比特足够了

curr_pyr
与 prev_pyr 类似， 用于第二帧

prev_features
需要发现光流的点集

curr_features
包含新计算出来的位置的 点集

count
特征点的数目

win_size
每个金字塔层的搜索窗口尺寸

level
最大的金字塔层数。如果为 0 , 不使用金字塔 (即金字塔为单层), 如果为 1 , 使用两层，下面依次类推。

status
数组。如果对应特征的光流被发现，数组中的每一个元素都被设置为 1， 否则设置为 0。

error
双精度数组，包含原始图像碎片与移动点之间的差。为可选参数，可以是 NULL .

criteria
准则，指定在每个金字塔层，为某点寻找光流的迭代过程的终止条件。

flags
其它选项:

- CV_LKFLOW_PYR_A_READY , 在调用之前，第一帧的金字塔已经准备好

- CV_LKFLOW_PYR_B_READY，在调用之前，第二帧的金字塔已经准备好
- CV_LKFLOW_INITIAL_GUESSES，在调用之前，数组 B 包含特征的初始坐标 （Hunnish: 在本节中没有出现数组 B，不知是指的哪一个）

函数 cvCalcOpticalFlowPyrLK 实现了金字塔中 Lucas-Kanade 光流计算的稀疏迭代版本 ([Bouguet00])。 它根据给出的前一帧特征点坐标计算当前视频帧上的特征点坐标。 函数寻找具有子象素精度的坐标值。

两个参数 prev_pyr 和 curr_pyr 都遵循下列规则： 如果图像指针为 0, 函数在内部为其分配缓存空间，计算金字塔，然后再处理过后释放缓存。 否则，函数计算金字塔且存储它到缓存中，除非设置标识 CV_LKFLOW_PYR_A[B]_READY 。 图像应该足够大以便能够容纳 Gaussian 金字塔数据。 调用函数以后，金字塔被计算而且相应图像的标识可以被设置，为下一次调用准备就绪 (比如：对除了第一个图像的所有图像序列，标识 CV_LKFLOW_PYR_A_READY 被设置).

[编辑]

预估器

[编辑]

CvKalman

Kalman 滤波器状态

```
typedef struct CvKalman
{
    int MP;                     /* 测量向量维数 */
    int DP;                     /* 状态向量维数 */
    int CP;                     /* 控制向量维数 */

    /* 向后兼容字段 */
#if 1
    float* PosterState;         /* =state_pre->data.fl */
    float* PriorState;          /* =state_post->data.fl */
    float* DynamMatr;           /* =transition_matrix->data.fl */
    float* MeasurementMatr;     /* =measurement_matrix->data.fl */
    float* MNCovariance;        /* =measurement_noise_cov->data.fl */
    float* PNCovariance;        /* =process_noise_cov->data.fl */
    float* KalmGainMatr;        /* =gain->data.fl */
    float* PriorErrorCovariance;/* =error_cov_pre->data.fl */
    float* PosterErrorCovariance;/* =error_cov_post->data.fl */
    float* Temp1;               /* temp1->data.fl */
    float* Temp2;               /* temp2->data.fl */
#endif

    CvMat* state_pre;           /* 预测状态 (x'(k)):
                                   x(k)=A*x(k-1)+B*u(k) */
    CvMat* state_post;          /* 矫正状态 (x(k)):
                                   x(k)=x'(k)+K(k)*(z(k)-H*x'(k)) */
    CvMat* transition_matrix;   /* 状态传递矩阵 state transition matrix (A) */
    CvMat* control_matrix;      /* 控制矩阵 control matrix (B)
                                   (如果没有控制，则不使用它)*/
    CvMat* measurement_matrix;  /* 测量矩阵 measurement matrix (H) */
    CvMat* process_noise_cov;   /* 过程噪声协方差矩阵
                                   process noise covariance matrix (Q) */
    CvMat* measurement_noise_cov; /* 测量噪声协方差矩阵
                                     measurement noise covariance matrix (R) */
    CvMat* error_cov_pre;       /* 先验误差计协方差矩阵
                                     priori error estimate covariance matrix (P'(k)):
                                   P'(k)=A*P(k-1)*At + Q)*/
    CvMat* gain;                /* Kalman 增益矩阵 gain matrix (K(k)):
                                   K(k)=P'(k)*Ht*inv(H*P'(k)*Ht+R)*/
    CvMat* error_cov_post;      /* 后验错误估计协方差矩阵
                                     posteriori error estimate covariance matrix (P(k)):
                                   P(k)=(I-K(k)*H)*P'(k) */
    CvMat* temp1;               /* 临时矩阵 temporary matrices */
    CvMat* temp2;
    CvMat* temp3;
    CvMat* temp4;
    CvMat* temp5;
}
CvKalman;
```

结构 CvKalman 用来保存 Kalman 滤波器状态。它由函数 cvCreateKalman 创建，由函数f cvKalmanPredict 和 cvKalmanCorrect 更新，由 cvReleaseKalman 释放. 通常该结构是为标准 Kalman 所使用的 (符号和公式都借自非常优秀的 Kalman 教程 [Welch95]):

系统运动方程: $x_k = A \cdot x_{k-1} + B \cdot u_k + w_k$
系统观测方程: $z_k = H \cdot x_k + v_k$

其中:

$x_k(x_{k-1})$ - 系统在时刻 k (k-1) 的状态向量 （state of the system at the moment k (k-1)）
z_k - 在时刻 k 的系统状态测量向量 （measurement of the system state at the moment k）
u_k - 应用于时刻 k 的外部控制 (external control applied at the moment k)
w_k 和 v_k 分别为正态分布的运动和测量噪声

p(w) ~ N(0,Q)
p(v) ~ N(0,R),
即,
Q - 运动噪声的相关矩阵，常量或变量
R - 测量噪声的相关矩阵，常量或变量

对标准 Kalman 滤波器，所有矩阵: A, B, H, Q 和 R 都是通过 cvCreateKalman 在分配结构 CvKalman 时初始化一次。但是，同样的结构和函数，通过在当前系统状态邻域中线性化扩展 Kalman 滤波器方程，可以用来模拟扩展 Kalman 滤波器，在这种情况下， A, B, H (也许还有 Q 和 R) 在每一步中都被更新。

[编辑]

CreateKalman

分配 Kalman 滤波器结构

```
CvKalman* cvCreateKalman( int dynam_params, int measure_params, int control_params=0 );
```

dynam_params
　　状态向量维数
measure_params
　　测量向量维数
control_params
　　控制向量维数

函数 cvCreateKalman 分配 CvKalman 以及它的所有矩阵和初始参数

[编辑]

ReleaseKalman

释放 Kalman 滤波器结构

```
void cvReleaseKalman( CvKalman** kalman );
```

kalman
　　指向 Kalman 滤波器结构的双指针

函数 cvReleaseKalman 释放结构 CvKalman 和里面所有矩阵

[编辑]

KalmanPredict

估计后来的模型状态

```
const CvMat* cvKalmanPredict( CvKalman* kalman, const CvMat* control=NULL );
#define cvKalmanUpdateByTime cvKalmanPredict
```

kalman
: Kalman 滤波器状态

control
: 控制向量 (uk), 如果没有外部控制 (control_params=0) 应该为 NULL

函数 cvKalmanPredict 根据当前状态估计后来的随机模型状态，并存储于 kalman->state_pre:

$$x'_k = A \cdot x_{k-1} + B \cdot u_k$$
$$P'_k = A \cdot P_{k-1} \cdot A^T + Q ,$$

其中

: x'_k 是预测状态 (kalman->state_pre),
x_{k-1} 是前一步的矫正状态 (kalman->state_post)，应该在开始的某个地方初始化，即缺省为零向量，
u_k 是外部控制(control 参数),
P'_k 是先验误差相关矩阵 (kalman->error_cov_pre)
P_{k-1} 是前一步的后验误差相关矩阵(kalman->error_cov_post),应该在开始的某个地方初始化，即缺省为单位矩阵.

函数返回估计得到的状态值

KalmanCorrect

调节模型状态

```
const CvMat* cvKalmanCorrect( CvKalman* kalman, const CvMat* measurement );
#define cvKalmanUpdateByMeasurement cvKalmanCorrect
```

kalman
: 被更新的 Kalman 结构的指针

measurement
: 指向测量向量的指针，向量形式为 CvMat

函数 cvKalmanCorrect 在给定的模型状态的测量基础上，调节随机模型状态:

$$K_k = P'_k \cdot H^T \cdot (H \cdot P'_k \cdot H^T + R)^{-1}$$
$$x_k = x'_k + K_k \cdot (z_k - H \cdot x'_k)$$
$$P_k = (I - K_k \cdot H) \cdot P'_k$$

其中

: z_k - 给定测量(mesurement parameter)
K_k - Kalman "增益" 矩阵

函数存储调节状态到 kalman->state_post 中并且输出时返回它。

例子. 使用 Kalman 滤波器跟踪一个旋转的点

```
#include "cv.h"
#include "highgui.h"
#include <math.h>

int main(int argc, char** argv)
{
    /* A matrix data */
    const float A[] = { 1, 1, 0, 1 };

    IplImage* img = cvCreateImage( cvSize(500,500), 8, 3 );
    CvKalman* kalman = cvCreateKalman( 2, 1, 0 );
    /* state is (phi, delta_phi) - angle and angle increment */
    CvMat* state = cvCreateMat( 2, 1, CV_32FC1 );
    CvMat* process_noise = cvCreateMat( 2, 1, CV_32FC1 );
    /* only phi (angle) is measured */
    CvMat* measurement = cvCreateMat( 1, 1, CV_32FC1 );
    CvRandState rng;
    int code = -1;

    cvRandInit( &rng, 0, 1, -1, CV_RAND_UNI );

    cvZero( measurement );
    cvNamedWindow( "Kalman", 1 );

    for(;;)
    {
        cvRandSetRange( &rng, 0, 0.1, 0 );
        rng.disttype = CV_RAND_NORMAL;

        cvRand( &rng, state );

        memcpy( kalman->transition_matrix->data.fl, A, sizeof(A));
        cvSetIdentity( kalman->measurement_matrix, cvRealScalar(1) );//初始化带尺度的单位矩阵
        cvSetIdentity( kalman->process_noise_cov, cvRealScalar(1e-5) );
        cvSetIdentity( kalman->measurement_noise_cov, cvRealScalar(1e-1) );
        cvSetIdentity( kalman->error_cov_post, cvRealScalar(1));
        /* choose random initial state */
        cvRand( &rng, kalman->state_post );

        rng.disttype = CV_RAND_NORMAL;

        for(;;)
        {
            #define calc_point(angle)                                      \
                cvPoint( cvRound(img->width/2 + img->width/3*cos(angle)),  \
                         cvRound(img->height/2 - img->width/3*sin(angle)))

            float state_angle = state->data.fl[0];
            CvPoint state_pt = calc_point(state_angle);

            /* predict point position */
            const CvMat* prediction = cvKalmanPredict( kalman, 0 );
            float predict_angle = prediction->data.fl[0];
            CvPoint predict_pt = calc_point(predict_angle);
            float measurement_angle;
            CvPoint measurement_pt;

            cvRandSetRange( &rng, 0, sqrt(kalman->measurement_noise_cov->data.fl[0]), 0 );
            cvRand( &rng, measurement );

            /* generate measurement */
            cvMatMulAdd( kalman->measurement_matrix, state, measurement, measurement );

            measurement_angle = measurement->data.fl[0];
            measurement_pt = calc_point(measurement_angle);

            /* plot points */
            #define draw_cross( center, color, d )                                \
                cvLine( img, cvPoint( center.x - d, center.y - d ),               \
                             cvPoint( center.x + d, center.y + d ), color, 1, 0 ); \
                cvLine( img, cvPoint( center.x + d, center.y - d ),               \
                             cvPoint( center.x - d, center.y + d ), color, 1, 0 )

            cvZero( img );
            draw_cross( state_pt, CV_RGB(255,255,255), 3 );
            draw_cross( measurement_pt, CV_RGB(255,0,0), 3 );
            draw_cross( predict_pt, CV_RGB(0,255,0), 3 );
            cvLine( img, state_pt, predict_pt, CV_RGB(255,255,0), 3, 0 );
```

```
            /* adjust Kalman filter state */
            cvKalmanCorrect( kalman, measurement );

            cvRandSetRange( &rng, 0, sqrt(kalman->process_noise_cov->data.fl[0]), 0 );
            cvRand( &rng, process_noise );
            cvMatMulAdd( kalman->transition_matrix, state, process_noise, state );

            cvShowImage( "Kalman", img );
            code = cvWaitKey( 100 );

            if( code > 0 ) /* break current simulation by pressing a key */
                break;
        }
        if( code == 27 ) /* exit by ESCAPE */
            break;
    }

    return 0;
}
```

[编辑]

CvConDensation

ConDensaation 状态

```
typedef struct CvConDensation
{
    int MP;      // 测量向量的维数:  Dimension of measurement vector
    int DP;      // 状态向量的维数:  Dimension of state vector
    float* DynamMatr;        // 线性动态系统矩阵: Matrix of the linear Dynamics system
    float* State;            // 状态向量:  Vector of State
    int SamplesNum;          // 粒子数:  Number of the Samples
    float** flSamples;       // 粒子向量数组:  array of the Sample Vectors
    float** flNewSamples;    // 粒子向量临时数组:  temporary array of the Sample Vectors
    float* flConfidence;     // 每个粒子的置信度(译者注: 也就是粒子的权值): Confidence for each Sample
    float* flCumulative;     // 权值的累计:  Cumulative confidence
    float* Temp;             // 临时向量: Temporary vector
    float* RandomSample;     // 用来更新粒子集的随机向量:  RandomVector to update sample set
    CvRandState* RandS;      // 产生随机向量的结构数组:  Array of structures to generate random vectors
} CvConDensation;
```

结构 CvConDensation中条件概率密度传播(译者注：粒子滤波的一种特例)（Con-Dens-Ation: 单词 CONditional DENSity propagATION 的缩写）跟踪器的状态。该算法描述可参考 http://www.dai.ed.ac.uk/CVonline/LOCAL_COPIES/ISARD1/condensation.html

[编辑]

CreateConDensation

分配 ConDensation 滤波器结构

```
CvConDensation* cvCreateConDensation( int dynam_params, int measure_params, int sample_count );
```

dynam_params
　　状态向量的维数
measure_params
　　测量向量的维数
sample_count
　　粒子数

函数 cvCreateConDensation 创建结构 CvConDensation 并且返回结构指针。

[编辑]

ReleaseConDensation

释放 ConDensation 滤波器结构

```
void cvReleaseConDensation( CvConDensation** condens );
```

condens
　　要释放结构的双指针

函数 cvReleaseConDensation 释放结构 CvConDensation (见cvConDensation) 并且清空所有事先被开辟的内存空间。

[编辑]

ConDensInitSampleSet

初始化 ConDensation 算法中的粒子集

```
void cvConDensInitSampleSet( CvConDensation* condens, CvMat* lower_bound, CvMat* upper_bound );
```

condens
　　需要初始化的结构指针
lower_bound
　　每一维的下界向量
upper_bound
　　每一维的上界向量

函数 cvConDensInitSampleSet 在指定区间内填充结构 CvConDensation 中的样例数组。

[编辑]

ConDensUpdateByTime

估计下个模型状态

```
void cvConDensUpdateByTime( CvConDensation* condens );
```

condens
　　要更新的结构指针

函数 cvConDensUpdateByTime 从当前状态估计下一个随机模型状态。

Cv模式识别

Wikipedia，自由的百科全书

目录

[隐藏]

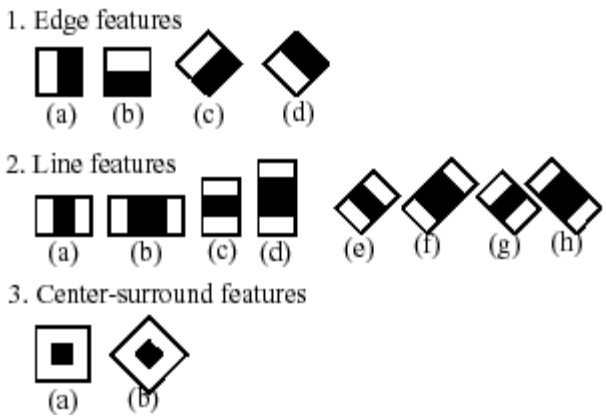
[编辑]

目标检测

目标检测方法最初由Paul Viola [Viola01]提出，并由Rainer Lienhart [Lienhart02]对这一方法进行了改善. 首先，利用样本（大约几百幅样本图片）的 harr 特征进行分类器训练，得到一个级联的boosted分类器。训练样本分为正例样本和反例样本，其中正例样本是指待检目标样本(例如人脸或汽车等)，反例样 本指其它任意图片，所有的样本图片都被归一化为同样的尺寸大小(例如，20x20)。

分类器训练完以后，就可以应用于输入图像中的感兴趣区域(与训练样本相同的尺寸)的检测。检测到目标区域(汽车或人脸)分类器输出为1，否 则输出为0。为了检测整副图像，可以在图像中移动搜索窗口，检测每一个位置来确定可能的目标。 为了搜索不同大小的目标物体，分类器被设计为可以进行尺寸改变，这样比改变待检图像的尺寸大小更为有效。所以，为了在图像中检测未知大小的目标物体，扫描 程序通常需要用不同比例大小的搜索窗口对图片进行几次扫描。

分类器中的“级联”是指最终的分类器是由几个简单分类器级联组成。在图像检测中，被检窗口依次通过每一级分类器， 这样在前面几层的检测中大部分的候选区域就被排除了，全部通过每一级分类器检测的区域即为目标区域。 目前支持这种分类器的boosting技术有四种： Discrete Adaboost, Real Adaboost, Gentle Adaboost and Logitboost。"boosted" 即指级联分类器的每一层都可以从中选取一个boosting算法(权重投票)，并利用基础分类器的自我训练得到。基础分类器是至少有两个叶结点的决策树分 类器。 Haar特征是基础分类器的输入，主要描述如下。目前的算法主要利用下面的Harr特征。



每个特定分类器所使用的特征用形状、感兴趣区域中的位置以及比例系数（这里的比例系数跟检测时候采用的比例系数是不一样的，尽管最后会取两 个系数的乘积值）来定义。例如在第二行特征(2c)的情况下，响应计算为覆盖全部特征整个矩形框(包括两个白色矩形框和一个黑色矩形框)象素的和减去黑色 矩形框内象素和的三倍。每个矩形框内的象素和都可以通过积分图象很快的计算出来。(察看下面和对cvIntegral的描述).

通过HaarFaceDetect 的演示版可以察看目标检测的工作情况。

下面只是检测部分的参考手册。 haartraining是它的一个单独的应用，可以用来对系列样本训练级联的boosted分类器。详细察看opencv/apps/haartraining。

CvHaarFeature, CvHaarClassifier, CvHaarStageClassifier, CvHaarClassifierCascade

Boosted Haar 分类器结构

```
#define CV_HAAR_FEATURE_MAX  3

/* 一个 harr 特征由 2－3 个具有相应权重的矩形组成 */
/* a haar feature consists of 2-3 rectangles with appropriate weights */
typedef struct CvHaarFeature
{
    int  tilted;  /* 0 means up-right feature, 1 means 45--rotated feature */

    /* 2-3 rectangles with weights of opposite signs and
       with absolute values inversely proportional to the areas of the rectangles.
       if rect[2].weight !=0, then
       the feature consists of 3 rectangles, otherwise it consists of 2 */
    struct
    {
        CvRect r;
        float weight;
    } rect[CV_HAAR_FEATURE_MAX];
}
CvHaarFeature;

/* a single tree classifier (stump in the simplest case) that returns the response for the
feature
   at the particular image location (i.e. pixel sum over subrectangles of the window) and gives
out
   a value depending on the responce */
typedef struct CvHaarClassifier
{
    int count;  /* number of nodes in the decision tree */

    /* these are "parallel" arrays. Every index i
       corresponds to a node of the decision tree (root has 0-th index).

       left[i] - index of the left child (or negated index if the left child is a leaf)
       right[i] - index of the right child (or negated index if the right child is a leaf)
       threshold[i] - branch threshold. if feature responce is <= threshold, left branch
                      is chosen, otherwise right branch is chosed.
       alpha[i] - output value correponding to the leaf. */
    CvHaarFeature* haar_feature;
    float* threshold;
    int* left;
    int* right;
    float* alpha;
}
CvHaarClassifier;

/* a boosted battery of classifiers(=stage classifier):
   the stage classifier returns 1
   if the sum of the classifiers' responces
   is greater than threshold and 0 otherwise */
typedef struct CvHaarStageClassifier
{
    int  count;  /* number of classifiers in the battery */
    float threshold; /* threshold for the boosted classifier */
    CvHaarClassifier* classifier; /* array of classifiers */

    /* these fields are used for organizing trees of stage classifiers,
```

```
       rather than just stright cascades */
    int next;
    int child;
    int parent;
}
CvHaarStageClassifier;

typedef struct CvHidHaarClassifierCascade CvHidHaarClassifierCascade;

/* cascade or tree of stage classifiers */
typedef struct CvHaarClassifierCascade
{
    int  flags; /* signature */
    int  count; /* number of stages */
    CvSize orig_window_size; /* original object size (the cascade is trained for) */

    /* these two parameters are set by cvSetImagesForHaarClassifierCascade */
    CvSize real_window_size; /* current object size */
    double scale; /* current scale */
    CvHaarStageClassifier* stage_classifier; /* array of stage classifiers */
    CvHidHaarClassifierCascade* hid_cascade; /* hidden optimized representation of the cascade,
                                                created by cvSetImagesForHaarClassifierCascade */
}
CvHaarClassifierCascade;
```

所有的结构都代表一个级联boosted Haar分类器。级联有下面的等级结构:

Cascade:

Stage1:

Classifier11:

Feature11

Classifier12:

Feature12

...

Stage2:

Classifier21:

Feature21

...

...

整个等级可以手工构建，也可以利用函数cvLoadHaarClassifierCascade从已有的磁盘文件或嵌入式基中导入。

[编辑]

cvLoadHaarClassifierCascade

从文件中装载训练好的级联分类器或者从OpenCV中嵌入的分类器数据库中导入

```
CvHaarClassifierCascade* cvLoadHaarClassifierCascade(
                         const char* directory,
                         CvSize orig_window_size );
```

directory

训练好的级联分类器的路径

orig_window_size

级联分类器训练中采用的检测目标的尺寸。因为这个信息没有在级联分类器中存储，所以要单独指出。

函数 cvLoadHaarClassifierCascade 用于从文件中装载训练好的利用哈尔特征的级联分类器，或者从OpenCV中嵌入的分类器数据库中导入。分类器的训练可以应用函数 haartraining(详细察看opencv/apps/haartraining) 这个数值是在训练分类器时就确定好的，修改它并不能改变检测的范围或精度。

需要注意的是，这个函数已经过时了。现在的目标检测分类器通常存储在 XML 或 YAML 文件中,而不是通过路径导入。从文件中导入分类器，可以使用函数 cvLoad 。

[编辑]

cvReleaseHaarClassifierCascade

释放haar classifier cascade。

```
void cvReleaseHaarClassifierCascade( CvHaarClassifierCascade** cascade );
```

cascade
双指针类型指针指向要释放的cascade. 指针由函数声明。

函数 cvReleaseHaarClassifierCascade 释放cascade的动态内存，其中cascade的动态内存或者是手工创建，或者通过函数 cvLoadHaarClassifierCascade 或 cvLoad分配。

[编辑]

cvHaarDetectObjects

检测图像中的目标

```
typedef struct CvAvgComp
{
    CvRect rect; /* bounding rectangle for the object (average rectangle of a group) */
    int neighbors; /* number of neighbor rectangles in the group */
}
CvAvgComp;
```

```
CvSeq* cvHaarDetectObjects( const CvArr* image, CvHaarClassifierCascade* cascade,
                            CvMemStorage* storage, double scale_factor=1.1,
                            int min_neighbors=3, int flags=0,
                            CvSize min_size=cvSize(0,0) );
```

image
被检图像
cascade
harr 分类器级联的内部标识形式
storage
用来存储检测到的一序列候选目标矩形框的内存区域。
scale_factor
在前后两次相继的扫描中，搜索窗口的比例系数。例如1.1指将搜索窗口依次扩大10%。
min_neighbors
构成检测目标的相邻矩形的最小个数(缺省－1)。如果组成检测目标的小矩形的个数和小于min_neighbors-1 都会被排除。如果min_neighbors 为 0, 则函数不做任何操作就返回所有的被检候选矩形框，这种设定值一般用在用户自定义对检测结果的组合程序上。
flags
操作方式。当前唯一可以定义的操作方式是 CV_HAAR_DO_CANNY_PRUNING。如果被设定，函数利用Canny边缘检测器来排除一些边缘很少或者很多的图像区域，因为这样的区域一 般不含被检目标。人脸检测中通过设定阈值使用了这种方法，并因此提高了检测速度。
min_size
检测窗口的最小尺寸。缺省的情况下被设为分类器训练时采用的样本尺寸(人脸检测中缺省大小是~20×20)。

函数 cvHaarDetectObjects 使用针对某目标物体训练的级联分类器在图像中找到包含目标物体的矩形区域，并且将这些区域作为一序列的矩形框返回。函数以不同比例大小的扫描窗口对图像进 行几次搜索(察看cvSetImagesForHaarClassifierCascade)。 每次都要对图像中的这些重叠区域利用cvRunHaarClassifierCascade进行检测。 有时候也会利用某些继承（heuristics）技术以减少分析的候选区域，例如利用 Canny 裁减 （prunning）方法。 函数在处理和收集到候选的方框(全部通过级联分类器各层的区域)之后，接着对这些区域进行组合并且返回一系列各个足够大的组合中的平均矩形。调节程序中的 缺省参数(scale_factor=1.1, min_neighbors=3, flags=0)用于对目标进行更精确同时也是耗时较长的进一步检测。为了能对视频图像进行更快的实时检测，参数设置通常 是：scale_factor=1.2, min_neighbors=2, flags=CV_HAAR_DO_CANNY_PRUNING, min_size=<minimum possible face size> (例如, 对于视频会议的图像区域).

例子:利用级联的Haar classifiers寻找检测目标(e.g. faces).

```
#include "cv.h"
#include "highgui.h"

CvHaarClassifierCascade* load_object_detector( const char* cascade_path )
{
    return (CvHaarClassifierCascade*)cvLoad( cascade_path );
}

void detect_and_draw_objects( IplImage* image,
                              CvHaarClassifierCascade* cascade,
                              int do_pyramids )
{
    IplImage* small_image = image;
    CvMemStorage* storage = cvCreateMemStorage(0);
    CvSeq* faces;
    int i, scale = 1;

    /* if the flag is specified, down-scale the 输入图像 to get a
       performance boost w/o loosing quality (perhaps) */
    if( do_pyramids )
    {
        small_image = cvCreateImage( cvSize(image->width/2,image->height/2), IPL_DEPTH_8U, 3 );
        cvPyrDown( image, small_image, CV_GAUSSIAN_5x5 );
        scale = 2;
    }

    /* use the fastest variant */
    faces = cvHaarDetectObjects( small_image, cascade, storage, 1.2, 2, CV_HAAR_DO_CANNY_PRUNING
);

    /* draw all the rectangles */
    for( i = 0; i < faces->total; i++ )
    {
        /* extract the rectanlges only */
        CvRect face_rect = *(CvRect*)cvGetSeqElem( faces, i, 0 );
        cvRectangle( image, cvPoint(face_rect.x*scale,face_rect.y*scale),
                     cvPoint((face_rect.x+face_rect.width)*scale,
                             (face_rect.y+face_rect.height)*scale),
                     CV_RGB(255,0,0), 3 );
    }

    if( small_image != image )
        cvReleaseImage( &small_image );
    cvReleaseMemStorage( &storage );
}

/* takes image filename and cascade path from the command line */
int main( int argc, char** argv )
{
    IplImage* image;
    if( argc==3 && (image = cvLoadImage( argv[1], 1 )) != 0 )
    {
        CvHaarClassifierCascade* cascade = load_object_detector(argv[2]);
        detect_and_draw_objects( image, cascade, 1 );
        cvNamedWindow( "test", 0 );
        cvShowImage( "test", image );
        cvWaitKey(0);
        cvReleaseHaarClassifierCascade( &cascade );
        cvReleaseImage( &image );
    }
```

```
    return 0;
}
```

[编辑]

cvSetImagesForHaarClassifierCascade

为隐藏的cascade(hidden cascade)指定图像

```
void cvSetImagesForHaarClassifierCascade( CvHaarClassifierCascade* cascade,
                                          const CvArr* sum, const CvArr* sqsum,
                                          const CvArr* tilted_sum, double scale );
```

cascade
: 隐藏 Harr 分类器级联 （Hidden Haar classifier cascade），由函数 cvCreateHidHaarClassifierCascade生成

sum
: 32－比特，单通道图像的积分图像（Integral (sum) 单通道 image of 32-比特 integer format）. 这幅图像以及随后的两幅用于对快速特征的评价和亮度/对比度的归一化。 它们都可以利用函数 cvIntegral从8-比特或浮点数 单通道的输入图像中得到。

sqsum
: 单通道64比特图像的平方和图像

tilted_sum
: 单通道32比特整数格式的图像的倾斜和（Tilted sum）

scale
: cascade的窗口比例. 如果 scale=1, 就只用原始窗口尺寸检测 (只检测同样尺寸大小的目标物体) - 原始窗口尺寸在函数cvLoadHaarClassifierCascade中定义 (在 "<default_face_cascade>"中缺省为24x24), 如果scale=2, 使用的窗口是上面的两倍 (在face cascade中缺省值是48x48)。 这样尽管可以将检测速度提高四倍，但同时尺寸小于48x48的人脸将不能被检测到。

函数 cvSetImagesForHaarClassifierCascade 为hidden classifier cascade 指定图像 and/or 窗口比例系数。 如果图像指针为空，会继续使用原来的图像(i.e. NULLs 意味这"不改变图像")。比例系数没有 "protection" 值,但是原来的值可以通过函数 cvGetHaarClassifierCascadeScale 重新得到并使用。这个函数用于对特定图像中检测特定目标尺寸的cascade分类器的设定。函数通过cvHaarDetectObjects进行内部调 用，但当需要在更低一层的函数cvRunHaarClassifierCascade中使用的时候，用户也可以自行调用。

[编辑]

cvRunHaarClassifierCascade

在给定位置的图像中运行 cascade of boosted classifier

```
int cvRunHaarClassifierCascade( CvHaarClassifierCascade* cascade,
                                CvPoint pt, int start_stage=0 );
```

cascade
: Haar 级联分类器

pt
: 待检测区域的左上角坐标。待检测区域大小为原始窗口尺寸乘以当前设定的比例系数。当前窗口尺寸可以通过cvGetHaarClassifierCascadeWindowSize重新得到。

start_stage
: 级联层的初始下标值（从0开始计数）。函数假定前面所有每层的分类器都已通过。这个特征通过函数cvHaarDetectObjects内部调用，用于更好的处理器高速缓冲存储器。

函数 cvRunHaarClassifierCascade 用于对单幅图片的检测。在函数调用前首先利用 cvSetImagesForHaarClassifierCascade设定积分图和合适的比例系数 (=> 窗口尺寸)。当分析的矩形框全部通过级联分类器每一层的时返回正值(这是一个候选目标)，否则返回0或负值。

Cv照相机定标和三维重建

Wikipedia，自由的百科全书

目录

[隐藏]

[编辑]

针孔相机模型和变形

这一节里的函数都使用针孔摄像机模型，这就是说，一幅视图是通过透视变换将三维空间中的点投影到图像平面。投影公式如下：

$s \cdot m' = A \cdot [R|t] \cdot M'$ 或者

$$s \cdot \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} fx & 0 & cx \\ 0 & fy & cy \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \cdot \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

这儿(X, Y, Z)是一个点的世界坐标，(u, v)是点投影在图像平面的坐标，以像素为单位。A被称作摄像机矩阵，或者内参数矩阵。(cx, cy)是基准点（通常在图像的中心），fx, fy是以像素为单位的焦距。所以如果因为某些因素对来自于摄像机的一幅图像升采样或者降采样，所有这些参数(fx, fy, cx和cy)都将被缩放（乘或者除）同样的尺度。内参数矩阵不依赖场景的视图，一旦计算出，可以被重复使用（只要焦距固定）。旋转－平移矩阵[R|t]被 称作外参数矩阵，它用来描述相机相对于一个固定场景的运动，或者相反，物体围绕相机的的刚性运动。也就是[R|t]将点(X, Y, Z)的坐标变换到某个坐标系，这个坐标系相对于摄像机来说是固定不变的。上面的变换等价与下面的形式（z≠0）：

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = R \cdot \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} + t$$

$x' = x / z$

$y' = y / z$

$$u = fx \cdot x' + cx$$

$$v = fy \cdot y' + cy$$

真正的镜头通常有一些形变，主要的变形为径向形变，也会有轻微的切向形变。所以上面的模型可以扩展为：

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = R \cdot \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} + t$$

$x' = x / z$

$y' = y / z$

$$x'' = x' \cdot (1 + k_1 \cdot r^2 + k_2 \cdot r^4) + 2 \cdot p_1 \cdot x' \cdot y' + p_2 \cdot (r_2 + 2x'^2)$$

$$y'' = y' \cdot (1 + k_1 \cdot r^2 + k_2 \cdot r^4) + p_1 \cdot (r_2 + 2 \cdot y'^2) + 2 \cdot p_2 \cdot x' \cdot y'$$

这儿 $r^2 = x'^2 + y'^2$

$$u = fx \cdot x'' + cx$$

$$v = fy \cdot y'' + cy$$

k_1和k_2是径向形变系数，p_1和p_1是切向形变系数。OpenCV中没有考虑高阶系数。形变系数跟拍摄的场景无关，因此它们是内参数，而且与拍摄图像的分辨率无关。

后面的函数使用上面提到的模型来做如下事情：

- 给定内参数和外参数，投影三维点到图像平面。
- 给定内参数、几个三维点坐标和其对应的图像坐标，来计算外参数。
- 根据已知的定标模式，从几个角度（每个角度都有几个对应好的3D-2D点对）的照片来计算相机的外参数和内参数。

[编辑]

照相机定标

[编辑]

ProjectPoints2

投影三维点到图像平面

```
void cvProjectPoints2( const CvMat* object_points, const CvMat* rotation_vector,
                       const CvMat* translation_vector, const CvMat* intrinsic_matrix,
                       const CvMat* distortion_coeffs, CvMat* image_points,
                       CvMat* dpdrot=NULL, CvMat* dpdt=NULL, CvMat* dpdf=NULL,
                       CvMat* dpdc=NULL, CvMat* dpddist=NULL );
```

object_points
　　物体点的坐标，为3xN或者Nx3的矩阵，这儿N是视图中的所有所有点的数目。
rotation_vector
　　旋转向量，1x3或者3x1。
translation_vector
　　平移向量，1x3或者3x1。
intrinsic_matrix
　　摄像机内参数矩阵A：$\begin{bmatrix} fx & 0 & cx \\ 0 & fy & cy \\ 0 & 0 & 1 \end{bmatrix}$
distortion_coeffs
　　形变参数向量，4x1或者1x4，为$[k_1,k_2,p_1,p_2]$。如果是NULL，所有形变系数都设为0。
image_points
　　输出数组，存储图像点坐标。大小为2xN或者Nx2，这儿N是视图中的所有点的数目。
dpdrot
　　可选参数，关于旋转向量部分的图像上点的导数，Nx3矩阵。
dpdt
　　可选参数，关于平移向量部分的图像上点的导数，Nx3矩阵。
dpdf
　　可选参数，关于fx和fy的图像上点的导数，Nx2矩阵。
dpdc
　　可选参数，关于cx和cy的图像上点的导数，Nx2矩阵。
dpddist
　　可选参数，关于形变系数的图像上点的导数，Nx4矩阵。

函数cvProjectPoints2通过给定的内参数和外参数计算三维点投影到二维图像平面上的坐标。另外，这个函数可以计算关于投影参数的图像 点偏导数的雅可比矩阵。雅可比矩阵可以用在cvCalibrateCamera2和cvFindExtrinsicCameraParams2函数的全局 优化中。这个函数也可以用来计算内参数和外参数的反投影误差。 注意，将内参数和（或）外参数设置为特定值，这个函数可以用来计算外变换（或内变换）。

[编辑]

FindHomography

计算两个平面之间的透视变换

```
void cvFindHomography( const CvMat* src_points,
                       const CvMat* dst_points,
                       CvMat* homography );
```

src_points
　　原始平面的点坐标，大小为2xN，Nx2，3xN或者 Nx3矩阵（后两个表示齐次坐标），这儿N表示点的数目。
dst_points
　　目标平面的点坐标大小为2xN，Nx2，3xN或者 Nx3矩阵（后两个表示齐次坐标）。
homography
　　输出的3x3的homography矩阵。

函数cvFindHomography计算源平面和目标平面之间的透视变换$H = \left[h_{ij}\right]_{i,j}$.

$$s_i \begin{bmatrix} x'_i \\ y'_i \\ 1 \end{bmatrix} \approx H \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix}$$

使得反投影错误最小:

$$\sum_i ((x'_i - \frac{h_{11}x_i + h_{12}y_i + h_{13}}{h_{31}x_i + h_{32}y_i + h_{33}})^2 + (y'_i - \frac{h_{21}x_i + h_{22}y_i + h_{23}}{h_{31}x_i + h_{32}y_i + h_{33}})^2)$$

这个函数可以用来计算初始的内参数和外参数矩阵。由于Homography矩阵的尺度可变，所以它被规一化使得h_{33} = 1

CalibrateCamera2

利用定标来计算摄像机的内参数和外参数

```
void cvCalibrateCamera2( const CvMat* object_points, const CvMat* image_points,
                         const CvMat* point_counts, CvSize image_size,
                         CvMat* intrinsic_matrix, CvMat* distortion_coeffs,
                         CvMat* rotation_vectors=NULL,
                         CvMat* translation_vectors=NULL,
                         int flags=0 );
```

object_points
　　定标点的世界坐标，为3xN或者Nx3的矩阵，这里N是所有视图中点的总数。
image_points
　　定标点的图像坐标，为2xN或者Nx2的矩阵，这里N是所有视图中点的总数。
point_counts
　　向量，指定不同视图里点的数目，1xM或者Mx1向量，M是视图数目。
image_size
　　图像大小，只用在初始化内参数时。
intrinsic_matrix
　　输出内参矩阵(A) $\begin{bmatrix} fx & 0 & cx \\ 0 & fy & cy \\ 0 & 0 & 1 \end{bmatrix}$，如果指定CV_CALIB_USE_INTRINSIC_GUESS和（或）CV_CALIB_FIX_ASPECT_RATION，fx、 fy、 cx和cy部分或者全部必须被初始化。
distortion_coeffs
　　输出大小为4x1或者1x4的向量，里面为形变参数[k1, k2, p1, p2]。
rotation_vectors
　　输出大小为3xM或者Mx3的矩阵，里面为旋转向量（旋转矩阵的紧凑表示方式，具体参考函数cvRodrigues2）
translation_vectors
　　输出大小为3xM或Mx3的矩阵，里面为平移向量。
flags
　　不同的标志，可以是0，或者下面值的组合:

- CV_CALIB_USE_INTRINSIC_GUESS - 内参数矩阵包含fx，fy，cx和cy的初始值。否则，(cx, cy)被初始化到图像中心（这儿用到图像大小），焦距用最小平方差方式计算得到。注意，如果内部参数已知，没有必要使用这个函数，使用 cvFindExtrinsicCameraParams2则可。
- CV_CALIB_FIX_PRINCIPAL_POINT - 主点在全局优化过程中不变，一直在中心位置或者在其他指定的位置（当CV_CALIB_USE_INTRINSIC_GUESS设置的时候）。
- CV_CALIB_FIX_ASPECT_RATIO - fx fy fx/fy

优化过程中认为 和 中只有一个独立变量，保持比例 不变，fx/fy的值跟内参数矩阵初始化时的值一样。在这种情况下， (fx, fy)的实际初始值或者从输入内存矩阵中读取（当CV_CALIB_USE_INTRINSIC_GUESS被指定时），或者采用估计值（后者情况中fx 和fy可能被设置为任意值，只有比值被使用）。

- CV_CALIB_ZERO_TANGENT_DIST – 切向形变参数(p1, p2)被设置为0，其值在优化过程中保持为0。

函数cvCalibrateCamera2从每个视图中估计相机的内参数和外参数。3维物体上的点和它们对应的在每个视图的2维投影必须被指定。这 些可以通过使用一个已知几何形状且具有容易检测的特征点的物体来实现。这样的一个物体被称作定标设备或者定标模式，OpenCV有内建的把棋盘当作定标设 备方法（参考cvFindChessboardCorners）。目前，传入初始化的内参数（当 CV_CALIB_USE_INTRINSIC_GUESS不被设置时）只支持平面定标设备（物体点的Z坐标必须为全0或者全1）。不过3维定标设备依然 可以用在提供初始内参数矩阵情况。在内参数和外参数矩阵的初始值都计算出之后，它们会被优化用来减小反投影误差（图像上的实际坐标跟 cvProjectPoints2计算出的图像坐标的差的平方和）。

[编辑]

FindExtrinsicCameraParams2

计算指定视图的摄像机外参数

```
void cvFindExtrinsicCameraParams2( const CvMat* object_points,
                                   const CvMat* image_points,
                                   const CvMat* intrinsic_matrix,
                                   const CvMat* distortion_coeffs,
                                   CvMat* rotation_vector,
                                   CvMat* translation_vector );
```

object_points
　　定标点的坐标，为3xN或者Nx3的矩阵，这里N是视图中的个数。
image_points
　　定标点在图像内的坐标，为2xN或者Nx2的矩阵，这里N是视图中的个数。
intrinsic_matrix
　　内参矩阵(A) $\begin{bmatrix} fx & 0 & cx \\ 0 & fy & cy \\ 0 & 0 & 1 \end{bmatrix}$ 。
distortion_coeffs
　　大小为4x1或者1x4的向量，里面为形变参数$[k_1,k_2,p_1,p_2]$。如果是NULL，所有的形变系数都为0。
rotation_vector
　　输出大小为3x1或者1x3的矩阵，里面为旋转向量（旋转矩阵的紧凑表示方式，具体参考函数cvRodrigues2）。
translation_vector
　　大小为3x1或1x3的矩阵，里面为平移向量。

函数cvFindExtrinsicCameraParams2使用已知的内参数和某个视图的外参数来估计相机的外参数。3维物体上的点坐标和相应的2维投影必须被指定。这个函数也可以用来最小化反投影误差。

[编辑]

Rodrigues2

进行旋转矩阵和旋转向量间的转换

```
int  cvRodrigues2( const CvMat* src, CvMat* dst, CvMat* jacobian=0 );
```

src
　　输入的旋转向量（3x1或者1x3）或者旋转矩阵（3x3）。

dst
　　输出的旋转矩阵（3x3）或者旋转向量（3x1或者1x3）
jacobian
　　可选的输出雅可比矩阵（3x9或者9x3），关于输入部分的输出数组的偏导数。

函数转换旋转向量到旋转矩阵，或者相反。旋转向量是旋转矩阵的紧凑表示形式。旋转向量的方向是旋转轴，向量的长度是围绕旋转轴的旋转角。旋转矩阵R，与其对应的旋转向量r，通过下面公式转换：

$$\theta \leftarrow norm(r)$$

$$r \leftarrow r/\theta$$

$$R = \cos(\theta) I + (1 - \cos(\theta)) r r^T + \sin(\theta) \begin{bmatrix} 0 & -r_z & r_y \\ r_z & 0 & -r_x \\ -r_y & r_x & 0 \end{bmatrix}$$

反变换也可以很容易的通过如下公式实现：

$$\sin(\theta) \begin{bmatrix} 0 & -r_z & r_y \\ r_z & 0 & -r_x \\ -r_y & r_x & 0 \end{bmatrix} = \frac{R - R^T}{2}$$

旋转向量是只有3个自由度的旋转矩阵一个方便的表示，这种表示方式被用在函数cvFindExtrinsicCameraParams2和cvCalibrateCamera2内部的全局最优化中。

[编辑]

Undistort2

校正图像因相机镜头引起的变形

```
void cvUndistort2( const CvArr* src, CvArr* dst,
                   const CvMat* intrinsic_matrix,
                   const CvMat* distortion_coeffs );
```

src
　　原始图像（已经变形的图像）。只能变换32fC1的图像。
dst
　　结果图像（已经校正的图像）。
intrinsic_matrix
　　相机内参数矩阵，格式为 $\begin{bmatrix} fx & 0 & cx \\ 0 & fy & cy \\ 0 & 0 & 1 \end{bmatrix}$。
distortion_coeffs
　　四个变形系数组成的向量，大小为4x1或者1x4，格式为[k_1,k_2,p_1,p_2]。

函数cvUndistort2对图像进行变换来抵消径向和切向镜头变形。相机参数和变形参数可以通过函数cvCalibrateCamera2取 得。使用本节开始时提到的公式，对每个输出图像像素计算其在输入图像中的位置，然后输出图像的像素值通过双线性插值来计算。如果图像得分辨率跟定标时用得 图像分辨率不一样，fx、fy、cx和cy需要相应调整，因为形变并没有变化。

[编辑]

InitUndistortMap

计算形变和非形变图像的对应（map）

```
void cvInitUndistortMap( const CvMat* intrinsic_matrix,
                         const CvMat* distortion_coeffs,
                         CvArr* mapx, CvArr* mapy );
```

intrinsic_matrix
　　摄像机内参数矩阵(A) [fx 0 cx; 0 fy cy; 0 0 1].
distortion_coeffs
　　形变系数向量[k1, k2, p1, p2]，大小为4x1或者1x4。
mapx
　　x坐标的对应矩阵。
mapy
　　y坐标的对应矩阵。

函数cvInitUndistortMap预先计算非形变对应－正确图像的每个像素在形变图像里的坐标。这个对应可以传递给cvRemap函数（跟输入和输出图像一起）。

FindChessboardCorners

寻找棋盘图的内角点位置

```
int cvFindChessboardCorners( const void* image, CvSize pattern_size,
                             CvPoint2D32f* corners, int* corner_count=NULL,
                             int flags=CV_CALIB_CB_ADAPTIVE_THRESH );
```

image
　　输入的棋盘图，必须是8位的灰度或者彩色图像。
pattern_size
　　棋盘图中每行和每列角点的个数。
corners
　　检测到的角点
corner_count
　　输出，角点的个数。如果不是NULL，函数将检测到的角点的个数存储于此变量。
flags
　　各种操作标志，可以是0或者下面值的组合：

- CV_CALIB_CB_ADAPTIVE_THRESH - 使用自适应阈值（通过平均图像亮度计算得到）将图像转换为黑白图，而不是一个固定的阈值。
- CV_CALIB_CB_NORMALIZE_IMAGE - 在利用固定阈值或者自适应的阈值进行二值化之前，先使用cvNormalizeHist来均衡化图像亮度。
- CV_CALIB_CB_FILTER_QUADS - 使用其他的准则（如轮廓面积，周长，方形形状）来去除在轮廓检测阶段检测到的错误方块。

函数cvFindChessboardCorners试图确定输入图像是否是棋盘模式，并确定角点的位置。如果所有角点都被检测到且它们都被以一定 顺序排布（一行一行地，每行从左到右），函数返回非零值，否则在函数不能发现所有角点或者记录它们地情况下，函数返回0。例如一个正常地棋盘图右8x8个 方块和7x7个内角点，内角点是黑色方块相互联通地位置。这个函数检测到地坐标只是一个大约地值，如果要精确地确定它们的位置，可以使用函数 cvFindCornerSubPix。

DrawChessBoardCorners

绘制检测到的棋盘角点

```
void cvDrawChessboardCorners( CvArr* image, CvSize pattern_size,
                              CvPoint2D32f* corners, int count,
                              int pattern_was_found );
```

image
　　结果图像，必须是8位彩色图像。
pattern_size
　　每行和每列地内角点数目。
corners
　　检测到地角点数组。
count
　　角点数目。
pattern_was_found
　　指示完整地棋盘被发现(≠0)还是没有发现(=0)。可以传输cvFindChessboardCorners函数的返回值。

当棋盘没有完全检测出时，函数cvDrawChessboardCorners以红色圆圈绘制检测到的棋盘角点；如果整个棋盘都检测到，则用直线连接所有的角点。

姿态估计

CreatePOSITObject

初始化包含对象信息的结构

```
CvPOSITObject* cvCreatePOSITObject( CvPoint3D32f* points, int point_count );
```

points
　　指向三维对象模型的指针
point_count
　　对象的点数

函数 cvCreatePOSITObject 为对象结构分配内存并计算对象的逆矩阵。

预处理的对象数据存储在结构CvPOSITObject中，只能在OpenCV内部被调用，即用户不能直接读写数据结构。用户只可以创建这个结构并将指针传递给函数。

对象是在某坐标系内的一系列点的集合，函数 cvPOSIT计算从照相机坐标系中心到目标点points[0] 之间的向量。

一旦完成对给定对象的所有操作，必须使用函数cvReleasePOSITObject释放内存。

POSIT

执行POSIT算法

```
void cvPOSIT( CvPOSITObject* posit_object, CvPoint2D32f* image_points,
              double focal_length,
              CvTermCriteria criteria, CvMatr32f rotation_matrix,
              CvVect32f translation_vector );
```

posit_object
　　指向对象结构的指针

image_points
　　指针，指向目标像素点在二维平面图上的投影。
focal_length
　　使用的摄像机的焦距
criteria
　　POSIT迭代算法程序终止的条件
rotation_matrix
　　旋转矩阵
translation_vector
　　平移矩阵.

函数 cvPOSIT 执行POSIT算法。图像坐标在摄像机坐标系统中给出。焦距可以通过摄像机标定得到。算法每一次迭代都会重新计算在估计位置的透视投影。

两次投影之间的范式差值是对应点中的最大距离。如果差值过小，参数criteria.epsilon就会终止程序。

ReleasePOSITObject

释放3D对象结构

```
void cvReleasePOSITObject( CvPOSITObject** posit_object );
```

posit_object
　　指向 CvPOSIT 结构指针的指针。

函数 cvReleasePOSITObject 释放函数 cvCreatePOSITObject分配的内存。

CalcImageHomography

计算长方形或椭圆形平面对象(例如胳膊)的Homography矩阵

```
void cvCalcImageHomography( float* line, CvPoint3D32f* center,
                            float* intrinsic, float* homography );
```

line
　　对象的主要轴方向，为向量(dx,dy,dz).
center
　　对象坐标中心 ((cx,cy,cz)).
intrinsic
　　摄像机内参数 (3x3 matrix).
homography
　　输出的Homography矩阵(3x3).

函数 cvCalcImageHomography 为从图像平面到图像平面的初始图像变化(defined by 3D oblong object line)计算Homography矩阵。

对极几何(双视几何)

FindFundamentalMat

由两幅图像中对应点计算出基本矩阵

```
int cvFindFundamentalMat( const CvMat* points1,
                          const CvMat* points2,
                          CvMat* fundamental_matrix,
                          int    method=CV_FM_RANSAC,
                          double param1=1.,
                          double param2=0.99,
                          CvMat* status=NULL);
```

points1
: 第一幅图像点的数组，大小为2xN/Nx2 或 3xN/Nx3 (N 点的个数)，多通道的1xN或Nx1也可以。点坐标应该是浮点数(双精度或单精度)。:

points2
: 第二副图像的点的数组，格式、大小与第一幅图像相同。

fundamental_matrix
: 输出的基本矩阵。大小是 3x3 或者 9x3 ，(7-点法最多可返回三个矩阵).

method
: 计算基本矩阵的方法

- CV_FM_7POINT – 7-点算法，点数目＝ 7
- CV_FM_8POINT – 8-点算法，点数目 >= 8
- CV_FM_RANSAC – RANSAC 算法，点数目 >= 8
- CV_FM_LMEDS - LMedS 算法，点数目 >= 8

param1
: 这个参数只用于方法RANSAC 或 LMedS 。它是点到对极线的最大距离，超过这个值的点将被舍弃，不用于后面的计算。通常这个值的设定是0.5 or 1.0 。

param2
: 这个参数只用于方法RANSAC 或 LMedS 。 它表示矩阵正确的可信度。例如可以被设为0.99 。

status
: 具有N个元素的输出数组，在计算过程中没有被舍弃的点，元素被被置为1；否则置为0。这个数组只可以在方法RANSAC and LMedS 情况下使用；在其它方法的情况下，status一律被置为1。这个参数是可选参数。

对极几何可以用下面的等式描述:

$$p_2^T \cdot F \cdot p_1 = 0$$

其中 F 是基本矩阵，p_1 和 p_2 分别是两幅图上的对应点。

函数 FindFundamentalMat 利用上面列出的四种方法之一计算基本矩阵，并返回基本矩阵的值：没有找到矩阵，返回0，找到一个矩阵返回1，多个矩阵返回3。 计算出的基本矩阵可以传递给函数cvComputeCorrespondEpilines来计算指定点的对极线。

例子1：使用 RANSAC 算法估算基本矩阵。

```
int    numPoints = 100;
CvMat* points1;
CvMat* points2;
CvMat* status;
CvMat* fundMatr;
points1 = cvCreateMat(2,numPoints,CV_32F);
points2 = cvCreateMat(2,numPoints,CV_32F);
status  = cvCreateMat(1,numPoints,CV_32F);

/* 在这里装入对应点的数据... */

fundMatr = cvCreateMat(3,3,CV_32F);
int num = cvFindFundamentalMat(points1,points2,fundMatr,CV_FM_RANSAC,1.0,0.99,status);
if( num == 1 )
    printf("Fundamental matrix was found\n");
else
    printf("Fundamental matrix was not found\n");
```

例子2: 7点算法（3个矩阵）的情况。

```
CvMat* points1;
CvMat* points2;
CvMat* fundMatr;
points1 = cvCreateMat(2,7,CV_32F);
points2 = cvCreateMat(2,7,CV_32F);

/* 在这里装入对应点的数据... */

fundMatr = cvCreateMat(9,3,CV_32F);
int num = cvFindFundamentalMat(points1,points2,fundMatr,CV_FM_7POINT,0,0,0);
printf("Found %d matrixes\n",num);
```

[编辑]

ComputeCorrespondEpilines

为一幅图像中的点计算其在另一幅图像中对应的对极线。

```
void cvComputeCorrespondEpilines( const CvMat* points,
                                  int which_image,
                                  const CvMat* fundamental_matrix,
                                  CvMat* correspondent_lines);
```

points
　　输入点，是2xN 或者 3xN 数组 (N为点的个数)
which_image
　　包含点的图像指数(1 or 2)
fundamental_matrix
　　基本矩阵
correspondent_lines
　　计算对极点, 3xN数组

函数 ComputeCorrespondEpilines 根据外级线几何的基本方程计算每个输入点的对应外级线。如果点位于第一幅图像(which_image=1),对应的对极线可以如下计算：

$$l_2 = F \cdot p_1$$

其中F是基本矩阵，p_1 是第一幅图像中的点， l_2 - 是与第二幅对应的对极线。如果点位于第二副图像中 which_image=2)，计算如下:

$$l_1 = F^T \cdot p_2$$

其中p_2 是第二幅图像中的点，l_1 是对应于第一幅图像的对极线，每条对极线都可以用三个系数表示 a, b, c:

$$a \cdot x + b \cdot y + c = 0$$

归一化后的对极线系数存储在correspondent_lines 中。

[编辑]

ConvertPointsHomogenious

Convert points to/from homogenious coordinates

```
void cvConvertPointsHomogenious( const CvMat* src, CvMat* dst );
```

src
　　The input point array, 2xN, Nx2, 3xN, Nx3, 4xN or Nx4 (where N is the number of points). Multi-channel 1xN or Nx1 array is also acceptable.
dst
　　The output point array, must contain the same number of points as the input; The dimensionality

must be the same, 1 less or 1 more than the input, and also within 2..4.

The function cvConvertPointsHomogenious converts 2D or 3D points from/to homogenious coordinates, or simply copies or transposes the array. In case if the input array dimensionality is larger than the output, each point coordinates are divided by the last coordinate:

(x,y[,z],w) -> (x',y'[,z'])
其中

x' = x/w
y' = y/w
z' = z/w (if output is 3D)

If the output array dimensionality is larger, an extra 1 is appended to each point.

(x,y[,z]) -> (x,y[,z],1)

Otherwise, the input array is simply copied (with optional tranposition) to the output. Note that, because the function accepts a large variety of array layouts, it may report an error when input/output array dimensionality is ambiguous. It is always safe to use the function with number of points N>=5, or to use multi-channel Nx1 or 1xN arrays.

HighGUI中文参考手册

Wikipedia，自由的百科全书

HighGUI概述

Wikipedia，自由的百科全书

OpenCV为了用于生产级别的应用而设计的。HighHGUI只是用来建立快速软件原形或是试验用的。它的设计意图是为用户提供简单易用的图形用户接口。

通常，你需要读入源图像到你的程序或者输出结果图像到磁盘。此外，需要简单的方法显示图像到监视器并且向允许（受限的）用户提供输入。

注：在**HighGUI**中没有任何的方法工具能够为流畅的用户界面提供产品级的错误处理。如果你试图创建最终用户的应用，请不要使用HighGUI 。 相对来说，应当为你的目标系统参考特定的函数库。比如：HighGUI中的摄像头输入方法（cvCreateCameraCapture）是为了易用而设计的。然而，并不意味着它能够对热插拔作出反应等等。

HighGUI简单图形界面

Wikipedia，自由的百科全书

HighHGUI只是用来建立快速软件原形或是试验用的。它提供了简单易用的图形用户接口，但是功能并不强大，也不是很灵活。

目录

[隐藏]

[编辑]

cvNamedWindow

创建窗口

```
int cvNamedWindow( const char* name, int flags=CV_WINDOW_AUTOSIZE );
```

name
: 窗口的名字，它被用来区分不同的窗口，并被显示为窗口标题。

flags
: 窗口属性标志，为1时表示会根据图像自动调整窗口大小。目前唯一支持的标志是CV_WINDOW_AUTOSIZE。当这个标志被设置后，用户不能手动改变窗口大小，窗口大小会自动调整以适合被显示图像（参考cvShowImage）。

函数cvNamedWindow创建一个可以放置图像和trackbar的窗口。被创建的窗口可以通过它们的名字被引用。

如果已经存在这个名字的窗口，这个函数将不做任何事情。

[编辑]

cvDestroyWindow

销毁一个窗口

```
void cvDestroyWindow( const char* name );
```

name

要被销毁的窗口的名字。

函数cvDestroyWindow销毁指定名字的窗口。

[编辑]

cvDestroyAllWindows

销毁所有HighGUI窗口

```
void cvDestroyAllWindows(void);
```

函数cvDestroyAllWindows销毁所有已经打开的HighGUI窗口。

[编辑]

cvResizeWindow

设定窗口大小

```
void cvResizeWindow( const char* name, int width, int height );
```

name
　　将被设置窗口的名字。
width
　　新的窗口宽度。
height
　　新的窗口高度。

函数cvResizeWindow改变窗口的大小。

[编辑]

cvMoveWindow

设定窗口的位置

```
void cvMoveWindow( const char* name, int x, int y );
```

name
　　将被设置的窗口的名字。
x
　　窗口左上角的x坐标。
y
　　窗口左上角的y坐标。

函数cvMoveWindow改变窗口的位置。

[编辑]

cvGetWindowHandle

通过名字获取窗口句柄

```
void* cvGetWindowHandle( const char* name );
```

name
　　窗口名字。

函数cvGetWindowHandle返回原始的窗口句柄（在Win32情况下返回HWND，GTK+情况下返回GtkWidget）

[编辑]

cvGetWindowName

通过句柄获取窗口的名字

```
const char* cvGetWindowName( void* window_handle );
```

window_handle
　　窗口句柄。

给定窗口的句柄（在Win32情况下是HWND，GTK+情况下是GtkWidget），返回窗口的名字。

[编辑]

cvShowImage

在指定窗口中显示图像

```
void cvShowImage( const char* name, const CvArr* image );
```

name
　　窗口的名字。
image
　　被显示的图像。

函数cvShowImage 在指定窗口中显示图像。如果窗口创建的时候被设定标志CV_WINDOW_AUTOSIZE，那么图像将以原始尺寸显示；否则，图像将被伸缩以适合窗口大小。

[编辑]

cvCreateTrackbar

创建trackbar并将它添加到指定的窗口。

```
CV_EXTERN_C_FUNCPTR( void (*CvTrackbarCallback)(int pos) );

int cvCreateTrackbar( const char* trackbar_name, const char* window_name,
                      int* value, int count, CvTrackbarCallback on_change );
```

trackbar_name
　　被创建的trackbar名字。
window_name
　　窗口名字，这个窗口将为被创建trackbar的父对象。
value
　　整数指针，它的值将反映滑块的位置。这个变量指定创建时的滑块位置。
count
　　滑块位置的最大值。最小值一直是0。
on_change
　　每次滑块位置被改变的时候，被调用函数的指针。这个函数应该被声明为void Foo(int); 如果没有回调函数，这个值可以设为NULL。

函数cvCreateTrackbar用指定的名字和范围来创建trackbar（滑块或者范围控制），指定与trackbar位置同步的变量，并且指定当trackbar位置被改变的时候调用的回调函数。被创建的trackbar显示在指定窗口的顶端。

[编辑]

cvGetTrackbarPos

获取trackbar的位置

```
int cvGetTrackbarPos( const char* trackbar_name, const char* window_name );
```

trackbar_name
　　trackbar的名字。
window_name
　　trackbar父窗口的名字。

函数cvGetTrackbarPos返回指定trackbar的当前位置。

[编辑]

cvSetTrackbarPos

设置trackbar位置

```
void cvSetTrackbarPos( const char* trackbar_name, const char* window_name, int pos );
```

trackbar_name
　　trackbar的名字。
window_name
　　trackbar父窗口的名字。
pos
　　新的位置。

函数cvSetTrackbarPos设置指定trackbar的位置。

[编辑]

cvSetMouseCallback

设置鼠标事件的回调函数

```
#define CV_EVENT_MOUSEMOVE      0
#define CV_EVENT_LBUTTONDOWN    1
#define CV_EVENT_RBUTTONDOWN    2
#define CV_EVENT_MBUTTONDOWN    3
#define CV_EVENT_LBUTTONUP      4                    放开鼠标左键
#define CV_EVENT_RBUTTONUP      5
#define CV_EVENT_MBUTTONUP      6
#define CV_EVENT_LBUTTONDBLCLK  7
#define CV_EVENT_RBUTTONDBLCLK  8
#define CV_EVENT_MBUTTONDBLCLK  9

#define CV_EVENT_FLAG_LBUTTON   1
#define CV_EVENT_FLAG_RBUTTON   2
#define CV_EVENT_FLAG_MBUTTON   4
#define CV_EVENT_FLAG_CTRLKEY   8
#define CV_EVENT_FLAG_SHIFTKEY  16
#define CV_EVENT_FLAG_ALTKEY    32

CV_EXTERN_C_FUNCPTR( void (*CvMouseCallback )(int event, int x, int y, int flags, void* param) );

void cvSetMouseCallback( const char* window_name, CvMouseCallback on_mouse, void* param=NULL );
```

window_name
　　窗口的名字。
on_mouse
　　指定窗口里每次鼠标事件发生的时候，被调用的函数指针。这个函数的原型应该为

```
void Foo(int event, int x, int y, int flags, void* param);
```

其中event是 CV_EVENT_*变量之一， x和y是鼠标指针在图像坐标系的坐标（不是窗口坐标系），flags是CV_EVENT_FLAG的组合(即上面的一些有关现在动作状态的预定义，现在鼠标没任何操作时为0），param是用户定义的传递到cvSetMouseCallback函数调用的参数。

param
　　用户定义的传递到回调函数的参数。

函数cvSetMouseCallback设定指定窗口鼠标事件发生时的回调函数。详细使用方法，请参考opencv/samples/c/ffilldemo.c demo。

[编辑]

cvWaitKey

等待按键事件

```
int cvWaitKey( int delay=0 );
```

delay
　　延迟的毫秒数。

函数cvWaitKey无限制的等待按键事件（delay<=0时）；或者延迟"delay"毫秒。返回值为被按键的值，如果超过指定时间则返回-1。

注释：这个函数是HighGUI中唯一能够获取和操作事件的函数，所以在一般的事件处理中，它需要周期地被调用，除非HighGUI被用在某些能够处理事件的环境中。

译者注：比如在MFC环境下，这个函数不起作用。

HighGUI读取与保存图像

Wikipedia，自由的百科全书

[编辑]

cvLoadImage

从文件中读取图像

需要include "highgui.h"

```
/* 8 bit, color or gray - deprecated, use CV_LOAD_IMAGE_ANYCOLOR */
#define CV_LOAD_IMAGE_UNCHANGED  -1
/* 8 bit, gray */
#define CV_LOAD_IMAGE_GRAYSCALE   0
/* 8 bit unless combined with CV_LOAD_IMAGE_ANYDEPTH, color */
#define CV_LOAD_IMAGE_COLOR       1
/* any depth, if specified on its own gray */
#define CV_LOAD_IMAGE_ANYDEPTH    2
/* by itself equivalent to CV_LOAD_IMAGE_UNCHANGED
   but can be modified with CV_LOAD_IMAGE_ANYDEPTH */
#define CV_LOAD_IMAGE_ANYCOLOR    4

IplImage* cvLoadImage( const char* filename, int flags=CV_LOAD_IMAGE_COLOR );
```

filename
: 要被读入的文件的文件名。

flags
: 指定读入图像的颜色和深度：

- 指定的颜色可以将输入的图片转为3信道(CV_LOAD_IMAGE_COLOR)也即彩色（>0），单信道(CV_LOAD_IMAGE_GRAYSCALE)也即灰色（=0），或者保持不变(CV_LOAD_IMAGE_ANYCOLOR)(<0)。
- 深度指定输入的图像是否转为每个颜色信道每象素8位，（OpenCV的早期版本一样），或者同输入的图像一样保持不变。
- 选中CV_LOAD_IMAGE_ANYDEPTH，则输入图像格式可以为8位无符号，16位无符号，32位有符号或者32位浮点型。
- 如果输入有冲突的标志，将采用较小的数字值。比如CV_LOAD_IMAGE_COLOR | CV_LOAD_IMAGE_ANYCOLOR 将载入3信道图。CV_LOAD_IMAGE_ANYCOLOR和CV_LOAD_IMAGE_UNCHANGED是等值的。但是，CV_LOAD_IMAGE_ANYCOLOR有着可以和CV_LOAD_IMAGE_ANYDEPTH同时使用的优点，所以 CV_LOAD_IMAGE_UNCHANGED不再使用了。
- 如果想要载入最真实的图像，选择CV_LOAD_IMAGE_ANYDEPTH | CV_LOAD_IMAGE_ANYCOLOR。

函数cvLoadImage从指定文件读入图像，返回读入图像的指针。目前支持如下文件格式：

- Windows位图文件 - BMP, DIB；
- JPEG文件 - JPEG, JPG, JPE；
- 便携式网络图片 - PNG；
- 便携式图像格式 - PBM，PGM，PPM；
- Sun rasters - SR，RAS；
- TIFF文件 - TIFF，TIF；
- OpenEXR HDR 图片 - EXR；
- JPEG 2000 图片- jp2。

[编辑]

cvSaveImage

保存图像到文件

需要include "highgui.h"

```
int cvSaveImage( const char* filename, const CvArr* image );
```

filename
　　文件名，如果对应的文件已经存在，则将被覆盖。
image
　　要保存的图像。

函数cvSaveImage保存图像到指定文件。图像格式的的选择依赖于filename的扩展名，请参考cvLoadImage。只有8位单通道 或者3通道（通道顺序为'BGR'）可以使用这个函数保存。如果格式，深度或者通道不符合要求，请先用cvCvtScale 和cvCvtColor转换；或者使用通用的cvSave保存图像为XML或者YAML格式。

HighGUI视频读写函数

Wikipedia，自由的百科全书

目录

[隐藏]

[编辑]

CvCapture

视频获取结构

```
typedef struct CvCapture CvCapture;
```

结构CvCapture 没有公共接口，它只能被用来作为视频获取函数的一个参数。

[编辑]

cvCreateFileCapture

初始化从文件中获取视频

```
CvCapture* cvCreateFileCapture( const char* filename );
```

filename
　　视频文件名。

函数cvCreateFileCapture给指定文件中的视频流分配和初始化CvCapture结构。

当分配的结构不再使用的时候，它应该使用cvReleaseCapture函数释放掉。

[编辑]

cvCreateCameraCapture

初始化从摄像头中获取视频

```
CvCapture* cvCreateCameraCapture( int index );
```

index
　　要使用的摄像头索引。如果只有一个摄像头或者用哪个摄像头也无所谓，那使用参数-1应该便可以。

函数cvCreateCameraCapture给从摄像头的视频流分配和初始化CvCapture结构。目前在Windows下可使用两种接 口：Video for Windows（VFW）和Matrox Imaging Library（MIL）； Linux下也有两种接口：V4L和FireWire（IEEE1394）。

释放这个结构，使用函数cvReleaseCapture。

[编辑]

cvReleaseCapture

释放CvCapture结构

```
void cvReleaseCapture( CvCapture** capture );
```

capture
　　视频获取结构指针。

函数cvReleaseCapture释放由函数cvCreateFileCapture或者cvCreateCameraCapture分配的CvCapture结构。

注:若从capture中使用cvQueryFrame获取图像指针，在releaseCapture的时候同时函数释放图像指针，用户不用再自己释放。

[编辑]

cvGrabFrame

从摄像头或者视频文件中抓取帧

```
int cvGrabFrame( CvCapture* capture );
```

capture
　　视频获取结构。

函数cvGrabFrame从摄像头或者文件中抓取帧。被抓取的帧在内部被存储。这个函数的目的是快速的抓取帧，这一点对同时从几个摄像头读取数据 的同步是很重要的。被抓取的帧可能是压缩的格式（由摄像头／驱动定义），所以没有被公开出来。如果要取回获取的帧，请使用 cvRetrieveFrame。

[编辑]

cvRetrieveFrame

取回由函数cvGrabFrame抓取的图像

```
IplImage* cvRetrieveFrame( CvCapture* capture );
```

capture
　　视频获取结构。

函数cvRetrieveFrame返回由函数cvGrabFrame 抓取的图像的指针。返回的图像不可以被用户释放或者修改。

[编辑]

cvQueryFrame

从摄像头或者文件中抓取并返回一帧

```
IplImage* cvQueryFrame( CvCapture* capture );
```

capture
　　视频获取结构。

函数cvQueryFrame从摄像头或者文件中抓取一帧，然后解压并返回这一帧。这个函数仅仅是函数cvGrabFrame和函数cvRetrieveFrame在一起调用的组合。返回的图像不可以被用户释放或者修改。 抓取后，capture被指向下一帧，可用cvSetCaptureProperty调整capture到合适的帧。

注意： cvQueryFrame返回的指针总是指向同一块内存。建议cvQueryFrame后拷贝一份。而且返回的帧需要FLIP后才符合OPENCV的坐标系。 若返回值为NULL，说明到了视频的最后一帧。

cvGetCaptureProperty

获得视频获取结构的属性

```
double cvGetCaptureProperty( CvCapture* capture, int property_id );
```

capture
　　视频获取结构。
property_id
　　属性标识。可以是下面之一：

　　CV_CAP_PROP_POS_MSEC - 影片目前位置，为毫秒数或者视频获取时间戳
　　CV_CAP_PROP_POS_FRAMES - 将被下一步解压／获取的帧索引，以0为起点
　　CV_CAP_PROP_POS_AVI_RATIO - 视频文件的相对位置（0 - 影片的开始，1 - 影片的结尾)
　　CV_CAP_PROP_FRAME_WIDTH - 视频流中的帧宽度
　　CV_CAP_PROP_FRAME_HEIGHT - 视频流中的帧高度
　　CV_CAP_PROP_FPS - 帧率
　　CV_CAP_PROP_FOURCC - 表示codec的四个字符
　　CV_CAP_PROP_FRAME_COUNT - 视频文件中帧的总数

函数cvGetCaptureProperty获得摄像头或者视频文件的指定属性。

译者注：有时候这个函数在cvQueryFrame被调用一次后，再调用cvGetCaptureProperty才会返回正确的数值。这是一个bug，建议在调用此函数前先调用cvQueryFrame。

cvSetCaptureProperty

设置视频获取属性

```
int cvSetCaptureProperty( CvCapture* capture, int property_id, double value );
```

capture
　　视频获取结构。
property_id
　　属性标识符。可以是下面之一：

　　CV_CAP_PROP_POS_MSEC - 从文件开始的位置，单位为毫秒
　　CV_CAP_PROP_POS_FRAMES - 单位为帧数的位置（只对视频文件有效）
　　CV_CAP_PROP_POS_AVI_RATIO - 视频文件的相对位置（0 - 影片的开始，1 - 影片的结尾)
　　CV_CAP_PROP_FRAME_WIDTH - 视频流的帧宽度（只对摄像头有效）

CV_CAP_PROP_FRAME_HEIGHT - 视频流的帧高度（只对摄像头有效）
CV_CAP_PROP_FPS - 帧率（只对摄像头有效）
CV_CAP_PROP_FOURCC - 表示codec的四个字符（只对摄像头有效）

value
属性的值。

函数cvSetCaptureProperty设置指定视频获取的属性。目前这个函数对视频文件只支持：
CV_CAP_PROP_POS_MSEC, CV_CAP_PROP_POS_FRAMES, CV_CAP_PROP_POS_AVI_RATIO

[编辑]

cvCreateVideoWriter

创建视频文件写入器

```
typedef struct CvVideoWriter CvVideoWriter;
CvVideoWriter* cvCreateVideoWriter( const char* filename, int fourcc, double fps, CvSize
frame_size, int is_color=1 );
```

filename
输出视频文件名。
fourcc
四个字符用来表示压缩帧的codec 例如，CV_FOURCC('P','I','M','1')是MPEG-1 codec，CV_FOURCC('M','J','P','G')是motion-jpeg codec等。 在Win32下，如果传入参数-1，可以从一个对话框中选择压缩方法和压缩参数。
fps
被创建视频流的帧率。
frame_size
视频流的大小。
is_color
如果非零，编码器将希望得到彩色帧并进行编码；否则，是灰度帧（只有在Windows下支持这个标志）。

函数cvCreateVideoWriter创建视频写入器结构。

[编辑]

cvReleaseVideoWriter

释放视频写入器

```
void cvReleaseVideoWriter( CvVideoWriter** writer );
```

writer
指向视频写入器的指针。

函数cvReleaseVideoWriter结束视频文件的写入并且释放这个结构。

[编辑]

cvWriteFrame

写入一帧到一个视频文件中

```
int cvWriteFrame( CvVideoWriter* writer, const IplImage* image );
```

writer

视频写入器结构。

image

被写入的帧。

函数cvWriteFrame写入／附加到视频文件一帧。

返回:

成功返回1,不成功返回0。

HighGUI实用函数与系统函数

Wikipedia，自由的百科全书

[编辑]

cvInitSystem

初始化HighGUI

```
int cvInitSystem( int argc, char** argv );
```

argc
　　命令行参数个数。
argv
　　命令行参数数组。

函数cvInitSystem初始化HighGUI。如果在第一个窗口被创建前这个函数不能被用户显式地调用，这个函数将以参数 argc=0，argv=NULL隐式地被调用。在Win32下，没有必要显式调用这个函数。在X Window下，参数可以被用来自定义HighGUI窗口和控件的外观。

[编辑]

cvConvertImage

把一幅图像转换为另外一幅，并可以选择同时对其进行垂直翻转

```
void cvConvertImage( const CvArr* src, CvArr* dst, int flags=0 );
```

src
　　输入图像。
dst
　　目标图像。必须为单通道或者3通道8位图像。
flags
　　操作标志：

- CV_CVTIMG_FLIP - 垂直翻转图像。
- CV_CVTIMG_SWAP_RB - 交换红蓝通道。在OpenCV中，彩色图像的通道顺序是 BGR 然而在一些系统中，在显示图像之前通道顺序应该被翻转（cvShowImage能够自动转换）。

函数cvConvertImage转换一幅图像到另一幅图像，如果需要的话可以垂直翻转图像。这个函数被cvShowImage使用。

OpenCV 编码样式指南

Wikipedia，自由的百科全书

目录

[隐藏]

[编辑]

前言

本文档是对OpenCV中代码风格的简短说明，因为OpenCV的核心库（cv,cvaux）是用C和C++编写的，所以本文档仅对用C和C++代码的编写有效。

[编辑]

文件命名

所有cv和cvaux库文件的命名必须服从于以下规则：

1. 所有的CV库文件名前缀为cv
2. 混合的C/C++接口头文件扩展名为 .h
3. 纯C++接口头文件扩展名为 .hpp
4. 实现文件扩展名为 .cpp
5. 为了与POSIX兼容，文件名都以小写字符组成

[编辑]

文件结构

每个文件以BSD兼容的许可声明(模板在Contributors_BSD_Licsense.htm文件中可以找到)开头；其它头文件和实现文件的规则包括:

1. 一行最多90个字符，不包括行结束符
2. 不使用制表符
3. 缩进为4个空格符，所以制表符应该用1-4个空格替换（依据开始列确定）

头文件必须使用保护宏，防止文件被重复包含。混合C/C++接口头文件用extern “C” { } 包含C语言定义。为了使预编译头机制在Visual C++中工作正常，源文件必须在其它头文件前包含precomp.h头文件。 同时，请参见头文件和实现文件的示例。

[编辑]

命名约定

OpenCV中使用大小写混合样式来标识外部函数、数据类型和类方法。宏全部使用大写字符，词间用下划线分隔。

所有的外部或内部名称，若在多个文件中可见，则必须含有前缀：

1. 外部函数使用前缀cv
2. 内部函数使用前缀Icv
3. 数据结构(C结构体、枚举、联合体、类)使用前缀Cv
4. 外部或某些内部宏使用前缀CV_
5. 内部宏使用前缀ICV_

[编辑]

函数接口设计

为了保持库的一致性，以如下方式设计接口非常重要。函数接口元素包括：

1. 功能
2. 名称
3. 返回值
4. 参数类型
5. 参数顺序
6. 参数默认值

函数功能必须定义良好并保持精简。函数应该容易镶入到使用其它OpenCV和IPL函数的不同处理过程。函数名称应该简单并能体现函数的功能。以下是OpenCV中的一些基本命名模式：

1. 大多数函数名形式：cv<ActionName>
2. 有时候函数以它实现的算法名或它产生的对象的名称命名。如：cvSobel, cvCanny, cvRodrigues, cvSqrt, cvGoodFeaturesToTrack.
3. 在对容器元素操作时，函数名以容器类型开头，紧跟着是动作名；在这种情况下，函数名可以当作方法名。例如：SeqPush, GraphAddEdgeByIdx。

返回值应该选择能简化功能的用法。通常一个函数创建一个对象并返回该对象。对于函数，处理动态数据结构

或标量，这是一个好的方法。然而在图片处理函数中会经常分配和回收大内存块，所以图片处理函数不能创建和返回图像结果而是修改输出一个作为参数传入的图像。

函数不应该用关于严重错误（例如空指针，0除数，错误参数范围,不支持的图像格式等）的信号作为返回值。在这种情况下，可以用一种类似于IPL中特殊的错误处理机制。相反，使用期望的运行时情况信号作为返回值比较好。（例如，跟踪图像移动到屏幕外）。

参数类型选择已经存在于OpenCV中的类型更适宜：IplImage用于光栅图像,CvMat用于矩阵，CvSeq用于轮廓线等。建议不使用简单指针和计数，因为有许多函数参数，它降低了库接口并使程序更难读。

一个一致的参数顺序很重要，因为它使参数易于记住顺序并且帮助程序员避免错误和使用错误的参数顺序联接函数。

1. 对于简单过程函数（在命名模式列表中的第一种和第二种类型）典型的顺序是：输入参数，输出参数，标记或可选参数。
2. 对于容器元素方法，顺序是：容器，元素位置，标记或可选参数。

输入参数经常用const修饰符。可选参数经常简化函数用法。因为C++允许在参数列表后跟随可选参数,它也可能影响决定以参数顺序，最重要的是标记位于最前,次重要的随后。在函数声明中用CV_DEFAULT宏指定可选参数的默认值.它使声明与C相兼容。

示例函数声明请参见cvexample.h和cv.h、cvaux.h.

函数实现

本节主要关注以下几点:

1. 参数类型检查
2. 错误产生和处理
3. 内存管理和资源回收
4. 调用低级函数

如前面所说，OpenCV函数广泛使用高级数据类型传送和返回参数。它简化了函数的使用,但是增加了使用错误的参数组合调用函数的可能性（例如浮点图像代替位图，或两个不同大小的图像）。检查标准类型参数存在标准的方法。

IplImage图像能通过CV_CHECK_IMAGE宏被检查。该宏检查传给IplImage的指针和潜在的图像数据指针不为空,图像有像素顺序,没有ROI掩码或冗余信息。

CV_CHECK_MASK_IMAGE用于检查掩码图像,二值图和灰度图。除了CV_CHECK_IMAGE检查的条件外,它还能确保图像有 8位深度和单通道。并且，所有的输入和输出图像在进行深度\通道数和尺寸组合前应该被检查。随后，应该在调用cvGetImageRawData函数返回 的图像ROI尺寸后应该被检查。输入等高线和其它动态数据结构能够用CV_IS_CONTOUR和相关的宏进行检查。

任何时候,当传入一个错误的参数或在函数执行时发生其它严重错误时,应该通过cvError函数抛出一个错误.OpenCV中与几乎所有标准的低级C库的IPL类似,有一个错误处理机制.那就是存在一个全局错误状态代替返回错误码:可以通过以下实现:

1. 使用cvError函数设置
2. 使用cvClearErrStatus清除
3. 使用cvGetErrorStatus读取

除了设置错误状态和指定值外,cvError还能进行额外的操作,依据错误处理模式而不同,错误处理模式可以通

过cvSetErrorMode调整.在silent模式或parent模式下cvError立即返回.在子模式下,它打印出错误消息并终止应用程序。

为了更方便使用. 可以通过使用如下宏来代替以上函数:

1. CV_ERROR和 OPENCV_ERROR代替cvError
2. CV_CALL和OPENCV_CALL代替调用函数和检查状态

CV_*宏需要在函数中定义”FuncName”字符串变量和”exit”标签，OPENCV_*宏则不需要。

> 在OpenCV中临时缓存用cvAlloc和cvFree函数分配和回收.函数应注意适当对齐,对未释放的内存保持跟踪，检查溢出。

当程序运行出内存泛围时,cvAlloc抛出一个错误.函数能够调用能由用户赋予完全控制内存分配的低级函数.因此强烈建议使用这些函数.以上 描述仅对简单缓存有效.临时图像,内存存储和其它结构使用cvCreate<Object>和cvRelease<Object>的方式分配和回收.

如果错误发生,并且CV_ERROR或CV_CALL宏被调用,控制转到exit标签处.同在在程序流中可以通过EXIT宏跳转控制.标签可以 通过手动或__BEGIN__和宏被定义.此标签引入是为了资源回收.尽管执行分支,当程序流进行时,还是经常发生内存泄漏.这种情况通常发生在分支语句 中使用返回语句.

使用库中的技术,可以帮助程序员避免大多数内存泄漏.在函数开始所有的指针被清除(通常在初始化中).在”exit”标签后,对每个指针调用 cvFree函数.cvFree函数可以安全处理空指针.在函数内部,返回语句用EXIT宏代替.这样,我们可以确保内存的回收.当然,我们可能忘记对某 些块调用cvFree函数,函数执行时,仅仅只是内存泄漏发发生,并且易于捕捉.

OpenCV中的低级函数象IPP中那样主要是是C语言实现原始操作的.它们不同于前面讨论的接口级高级函数(他们使用简单的指针和数值,几乎 不用结构体)和错误处理方法(它们返回错误代码而不是全局错误状态).方便并安全的调用这些函数的方法是使用IPPI_CALL宏.

函数实现示例请参见cvexample.cpp文件。

[编辑]

代码布局

> 在OpenCV中有一个单独的字符串规则：每个文件必须使用一致格式样式。
> 当前使用在OpenCV中，并推荐使用的样式如下:

```
if( a > 5 )
{
  int b = a*a;
  c = c > b ? c : b + 1;
}
else if( abs(a) < 5 )
{
  c--;
}
else
{
  printf( "a=%d is far to negative\n", a );
}
```

> 在符合以上样式的前提下其它样式也可能接受。也就是说，如果一个人修改别人的代码，他应该使用相同的代码样式。

[编辑]

移植性

所要代码必须符合以下标准:

1. ANSI C 第一个语言标准ISO/IEC 9899:1990
2. C9X (1999年修订的新标准) – ISO/IEC 9899.
3. C++ 标准 – ISO/IEC 14882-1998.

你应该去除编译器依赖或平台依赖和系统调用,例如:

1. 编译器: pragma's
2. 特定关键字: __stdcall, __inline, __int64(or long long).使用CV_INLINE, CV_STDCALL, int64分别代替。
3. 编译器扩展，例如?<和<?宏表示最小和最大，重载宏等。
4. 内联汇编
5. Unix或Win32调用，如：bcopy, readdir, CreateFile, WaitForSingleObject 等。
6. 用sizeof代替具体的数据大小(如sizeof(int)而不是4),字节顺序(*(int*)"\x1\x2\x3\x4"是0x01020304或0x04030201或什么?),用简单的字符有符号字符或无符号字符处理数据（不是字符串）。使用短形式，uchar表示 unsigned char和schar表示signed char。使用预处理指令包含非可移植性代码片段。不要试途使用标准元素，主要编译器制造商几乎不支持这些。

[编辑]

函数文档编写

文档以HTML格式提供，因为HTML格式提供文本格式化功能和超级链接，同时它非常简单，易用和易于维护。每个函数的文档或相关函数组的文档放入不同文件中，该文可以从主页链接到。

这是一个包含链接到函数示例文档的主页原型，实现在cvexample.cpp中。

看一下这些页面和相应的HTML代码，包括详细的格式化注释。你对拷贝函数文档HTML,适当的更改和加入一个链接到索引页中。

函数文档HTML文件包知以下基本元素(以如下顺序):

1. 页面标题——显示在浏览器标题栏中并且表示一个扩展的函数名或函数组名。
2. 关键字列表——用于搜索引擎和各种工具检索文档。
3. 可见的页面标题——简单的重复页面标题。
4. 函数名——真实的函数名，但是不包括前缀。它应该标签化以便从文档的其它位置引用它。
5. Blurb——单行函数描述。
6. 函数声明——按它在头文件中的形式，用＜pre＞和＜/pre＞包围。除了OPENCVAPI被忽略外，默认参数的普通C++语法用CV_DEFAULT宏代替。
7. 函数参数描述——一个<参数名，描述>对的列表。
8. 讨论——描述函数功能的节。允许或支持的参数组合的限制，算法参考(链接或标题)。
9. 使用示例——一个可选的代码片断或伪代码段.几个相关函数可以共用相同示例.
10. 请参见——包含零个或多个相关函数链接。

[编辑]

函数测试实现

每个测试实现为从文本文件输入和输出结果到另外一个文本文件的C/C++函数。这样，函数就有如下接口:

```
bool <TestName>( const char* inputfile, const char* output file );
```

(

输入输出文件的格式没有定义。然而，如果测试系统函数用于从文件中读取或写入高级数据 矩阵，文件名，轮廓等),那么文件的格式应与函数兼容。

外部或主测试函数执行所有或选择的测试并且与标准结果相比较，它可以由其它程序或以前执行的相同测试创建。

使用这种设计可以实现检查在一个函数上检查几个特殊的数据集和测式比较函数在武断的数据上的输出和标准输出。在这种情况下输出文件能以不同的两个输出和从前面段的标准结果将全部是零。

测试系统API使测试更容易，它包括:

1. 系统测试内核(测试注册，文件管理，用异常处理能力控制测试)
2. 从文本文件取得矩阵、文件名和其它数据和写入数据到文本文件的函数。
3. 检查数组中的特殊值。
4. 内存管理函数帮助捕捉内存泄漏和缓冲区越界。
5. 随机数据生成。
6. 简单算法函数(矩阵操作)
7. 可视化函数
8. 很多功能实现在OpenCV和HighGUI API上的瘦层.

以下是一步一步描述怎样实现测试的示例:

```
//创建一个测试体文件:
//
// skeleton_test.cpp
//
#include "opencv_tst.h"
// 测试函数
bool  test_skeleton( const char* input, const char* output )
{
// 从文本文件中加载一个图片.
IplImage* img = tstLoadImage( input );
// 运行函数(参见cvexample.cpp)
cvRasterSkeleton( img, CV_SKEL_PAVLIDIS );
//保存结果
tstSaveImage( output );
}
// 注册测试
OPENCV_REGISTER_TEST( test_skeleton, "cvRasterSkeleton" )
// 以上宏扩展了如下代码:
//
//     static  CvTstReg( test_skeleton, "cvRasterSkeleton",
//                       "test_skeleton", "skeleton_test.cpp",
//                       20 /* code line number */,
//                       "test_skeleton.in" /* input data file name */
//                       "test_skeleton.out"/* output data file name */
//                       "test_skeleton.0"  /* etalon data file name */
//                      );
//
// The line calls constructor of CvTstReg class that links the test to
// the master test list.
```

将测试文件加入到测试系统项目中.有main()函数的主测试已经包含在项目中,所以不需要再编写执行代码.

1. 创建输入和标准数据.首先你可能需要手工创建.标准数据能够通过用-c(--create-etalon)传输测命令行选项标识生成。并且,这种情况可以在测试函数中用tstIsEtalonMode()函数(在这种情况下,给定可靠值给另外一个算法的变量)处理。
2. 输入输出数据存放在opencv_tst/testdata文件夹中.
3. 测试数据文件的名字由测试体文件和特定的扩展名构成.(输入数据文件的扩展名为".in",输出数据文件的扩展名为".out",标准数据文件的扩展名为".0").

随后,测试系统将在不同的模式下执行.

[编辑]

提示

某些OpenCV函数使用小结构体作为输入.使结构体的名称类型为CvSomething.然后cvSomething通常是内联的,从参数列表构成对象.使用这些内联的构造器,使代码更容易写和读.

使用cvRound,cvFloor和cvCeil快速转换浮点数为相近的整形数或到负无穷在或正无穷大.在x86架构上,这些函数比简单转换 操作运行更快.在C9X标准中有几个标准的函数做相同的事情,但是它们现在很少被支持.当情况发生时,以上函数将转换为内联宏.

[编辑]

附录

[编辑]

附录**A:** 参考

本文不是完整的样式参考,更多详细和写得更好相关主题列表如下:

1. 推荐C样式和标准编码(Indian Hill C样式和编码标准更新版) . Henry Spencer et al. Rev. 6.0, 1990.
2. C++编程规则与建议 Mats Henricson, Erik Nyquist. Ellemtel Communications Systems Laboratories. 1990-1992.
3. GNU编码样式, Richard Stallman. GNU Project – Free Software Foundation. 2000
4. 用C语言编写可移植的程序 A. Dolenc, A. Lemmke, D. Keppel, G.V. Reilly. 1990。类似文档可以在http://www.softpanorama.org/Lang/c.html 上找到。

[编辑]

附录**B:** 规则简述列表

1. 文件名以小写的cv前缀开始.头文件扩展名为.h或.hpp
2. 实现文件扩展名为.cpp
3. 每一个文件在开头包含BSD兼容许可
4. 头文件有保护宏和extern “C” { } 保护C语言部分接口
5. 源文件包含”precomp.h”作为第一个头文件
6. 外部函数名和数据类型名写成大小写混合类型.外部宏写作大写字符
7. 外部函数以cv前缀开始
8. 内部函数前缀为icv
9. 数据类型前缀为Cv
10. 外部宏前缀为CV
11. 函数声明形式为:

```
OPENCVAPI <returnType>  <functionName>( arguments );
```

1. 函数实现形式为:

```
CV_IMPL <returnType>
<functionName>( arguments )
{
CV_FUNCNAME("<functionName>");
...
__BEGIN__;
...
__END__;

// cleanup ...
```

```
return ...;
}
```

1. 使用CV_ERROR/OPENCV_ERROR抛出错误,使用CV_CALL/OPENCV_CALL调用高级函数,使用IPPI_CALL调用低级函数
2. 使用cvAlloc和cvFree分配和回收临时缓冲区
3. 在每一个源文件中保持一致的格式
4. 与C/C++标准兼容,避免编译器依赖性,系统依赖性和平台依赖性