关于矩阵的操作，异常繁多，而且OpenCV提供的函数也相对很丰富，这里作一个简单的系统介绍，主要涉及矩阵的基本运算，矩阵的属性运算和一些其他常用操作。

**矩阵基本运算**

OpenCV提供了矩阵常用的一些操作包括矩阵的相加，相乘，逐元素相乘，逐元素相除，求逆等。

**Mat类内部实现**

设为A，B为Mat类型，s是Scalar类型，a是一个实数。下面列出关于Mat的常用运算：

* **矩阵加减：** A+B，A-B，A+s，A-s，s+A，s-A,-A.
* **矩阵乘以实数：** A\*a，a\*A
* **逐元素乘除：** A.mul(B)，A/B，a/A
* **矩阵乘法：** A\*BmaxVal; Point minPos,m
* **矩阵倒置：** A.t()
* **矩阵的逆：** A.inv()
* **矩阵比较：** A comp B，A comp a，a comp A。这里comp包括 >， >=，==，!=，<=，<。得出的结果是一个单通道8位的矩阵，元素的值为255或0。
* **矩阵位操作：** A logic B， A logic s，s logic A。这里logic包括：&，|，^
* **向量的差乘和内积：** A.cross(B)，A.dot(B);

这里需要注意，为防止溢出，**矩阵乘法**的矩阵元素类型至少是float，即CV\_32F以上。OpenCV除了提供上述提到的直接调用Mat成员函数(如求逆，倒置)的以及使用Mat重载操作符（如加减乘除）的方法外，还提供了外部函数实现这些功能。

**矩阵的加减乘除**

除了可以使用Mat对象的成员函数实现加减乘除运算外，还可以使用OpenCV提供的外部函数实现这些操作。

以下四个函数分别实现矩阵逐像素的加、减、乘、除：

|  |  |
| --- | --- |
| 1  2  3  4 | void add(InputArray src1, InputArray src2, OutputArray dst, InputArray mask=noArray(), int dtype=-1)`  void subtract(InputArray src1, InputArray src2, OutputArray dst, InputArray mask=noArray(), int dtype=-1)  void multiply(InputArray src1, InputArray src2, OutputArray dst, double scale=1, int dtype=-1 )  void divide(InputArray src1, InputArray src2, OutputArray dst, double scale=1, int dtype=-1) |

**src1**和**src2**可以为Mat或者Scalar，如果一个是Mat，一个Scalar，则Mat的元素的通道数要和Scalar的维数相同。**dtype**可以控制输出数组的元素类型。注意，当输出类型是CV\_32S的时候，不进行溢出处理。

此外，对于矩阵的加权和计算使用如下函数：

|  |  |
| --- | --- |
| 1 | void addWeighted(InputArray src1, double alpha, InputArray src2, double beta, double gamma, OutputArray dst, int dtype=-1) |

其计算的结果为

dst(I)=saturate(src1(I)∗alpha+src2[I]∗beta+gamma)dst(I)=saturate(src1(I)∗alpha+src2[I]∗beta+gamma)

对于矩阵乘法，使用如下函数：

|  |  |
| --- | --- |
| 1 | void gemm(InputArray src1, InputArray src2, double alpha, InputArray src3, double gamma, OutputArray dst, int flags=0 ) |

其中， \* **src1, src2**的type必须是CV\_32FC1, CV\_64FC1, CV\_32FC2, or CV\_64FC2 \* **flag**是控制矩阵转置的： \* GEMM\_1\_T 转置 src1. \* GEMM\_2\_T 转置 src2. \* GEMM\_3\_T 转置 src3.

例如，如下调用gemm(src1, src2, alpha, src3, beta, dst, GEMM\_1\_T + GEMM\_3\_T)的计算结果如下：

dst=alpha∗src1T∗src2+beta∗src3Tdst=alpha∗src1T∗src2+beta∗src3T

此外，还有计算矩阵的自身和其转置矩阵的乘积的函数，如下：

|  |  |
| --- | --- |
| 1 | void mulTransposed(InputArray src, OutputArray dst, bool aTa, InputArray delta=noArray(), double scale=1, int dtype=-1 ) |

当**aTa**为true的时候，执行如下运算：

dst=scale(src−delat)∗(src−delta)Tdst=scale(src−delat)∗(src−delta)T

否则，执行如下运算

dst=scale(src−delat)T∗(src−delta)dst=scale(src−delat)T∗(src−delta)

**矩阵的逆和转置**

和Mat::inv()类似，invert()也提供三种计算矩阵逆的方法，其函数声明如下：

|  |  |
| --- | --- |
| 1 | double invert(InputArray src, OutputArray dst, int flags=DECOMP\_LU) |

**flag**值有：

* **DECOMP\_LU：**采用的[是高斯消去法](http://zh.wikipedia.org/wiki/%E9%AB%98%E6%96%AF%E6%B6%88%E5%8E%BB%E6%B3%95)（Gaussian elimination），只能计算非奇异矩阵的逆，如果矩阵可逆，则函数返回非零值，否则返回0.
* **DECOMP\_SVD：**采用的是[奇异值分解法](http://zh.wikipedia.org/wiki/%E5%A5%87%E5%BC%82%E5%80%BC%E5%88%86%E8%A7%A3)（singular value decomposition）。对于奇异矩阵，计算伪逆矩阵，非奇异矩阵则计算逆矩阵，如果矩阵是非奇异的，则返回最小奇异值和最大奇异值的比例，否则返回0.
* **DECOMP\_CHOLESKY：**采用的是[Cholesky分解](http://en.wikipedia.org/wiki/Cholesky_decomposition).只能计算非奇异的方形矩阵，如果矩阵奇异，则返回0，否则返回非零值。

矩阵的转置使用如下函数：

|  |  |
| --- | --- |
| 1 | void transpose(InputArray src, OutputArray dst) |

**矩阵的比较**

|  |  |
| --- | --- |
| 1 | void compare(InputArray src1, InputArray src2, OutputArray dst, int cmpop) |

**cmpop**是标志位，指定比较符号：

* **CMP\_EQ** src1等于src2.
* **CMP\_GT** src1大于src2.
* **CMP\_GE** src1大于等于src2.
* **CMP\_LT** src1小于src2.
* **CMP\_LE** src1小于等于src2.
* **CMP\_NE** src1不等于src2.

如果两个矩阵的对应元素比较结果为true，则**dst**中相应的元素设置为255。

**矩阵的位运算**

位运算除了上面提到的&，|，^外，还可以使用如下外部函数：

|  |  |
| --- | --- |
| 1  2  3  4 | void bitwise\_and(InputArray src1, InputArray src2, OutputArray dst, InputArray mask=noArray()) \\与  void bitwise\_not(InputArray src, OutputArray dst, InputArray mask=noArray()) \\非  void bitwise\_or(InputArray src1, InputArray src2, OutputArray dst, InputArray mask=noArray()) \\或  void bitwise\_xor(InputArray src1, InputArray src2, OutputArray dst, InputArray mask=noArray()) \\异或 |

**矩阵的绝对值**

对于求解矩阵内每个元素的绝对值以及矩阵差的绝对值，可以使用如下函数：

|  |  |
| --- | --- |
| 1  2 | MatExpr abs(const Mat& m) //矩阵逐个元素的绝对值  void absdiff(InputArray src1, InputArray src2, OutputArray dst) //两个矩阵之差的绝对值 |

**矩阵的对数和指数**

逐元素求解矩阵的对数：

|  |  |
| --- | --- |
| 1 | void log(InputArray src, OutputArray dst) |

逐元素计算矩阵的以e为底的指数：

|  |  |
| --- | --- |
| 1 | void exp(InputArray src, OutputArray dst) |

逐元素计算矩阵的幂

|  |  |
| --- | --- |
| 1 | void pow(InputArray src, double power, OutputArray dst) |

**矩阵的范数**

矩阵的范数运算提供了多种范数的运算方法，下面是函数声明：

|  |  |
| --- | --- |
| 1  2 | double norm(InputArray src1, int normType=NORM\_L2, InputArray mask=noArray())  double norm(InputArray src1, InputArray src2, int normType=NORM\_L2, InputArray mask=noArray() ) |

其中，第二种调用是，求的src1和src2的差值的范数，各个参数意义如下：

* **src1, src2：**输入矩阵
* **normType：**范数的类型，主要有**NORM\_INF，NORM\_L1, NORM\_L2**

此外，OpenCV还提供了把归一化范数的计算，即把计算的值归一化到某个范围内：

|  |  |
| --- | --- |
| 1 | void normalize(InputArray src, OutputArray dst, double alpha=1, double beta=0, int norm\_type=NORM\_L2, int dtype=-1, InputArray mask=noArray() ) |

其中，**alpha**是范围的下限，**beta**是范围上限。

**矩阵的属性运算**

矩阵的属性主要包括矩阵(方阵)的行列式值，矩阵的迹，矩阵(方阵)的特征值和特征向量的计算等等

**矩阵对应行列式值**

|  |  |
| --- | --- |
| 1 | double determinant(InputArray mtx) |

直接返回行列式的值。输入矩阵的类型必须是CV\_32F或CV\_64F的方阵

**矩阵的迹**

求矩阵对角元素的和

|  |  |
| --- | --- |
| 1 | Scalar trace(InputArray mtx) |

直接返回结果。

**矩阵的特征值和特征向量**

|  |  |
| --- | --- |
| 1  2 | bool eigen(InputArray src, OutputArray eigenvalues, int lowindex=-1, int highindex=-1)  bool eigen(InputArray src, OutputArray eigenvalues, OutputArray eigenvectors, int lowindex=-1, int highindex=-1) |

输入矩阵的类型必须是CV\_32F或CV\_64F的方阵, **eigenvalues**存储计算的特征值，**eigenvectors**存储计算的特征向量，**lowindex**和**highindex**输出最小最大特征值对于的索引，但是目前该功能不能用。

**矩阵的元素和**

|  |  |
| --- | --- |
| 1 | Scalar sum(InputArray src) |

其中，多通道是单独计算各个通道内的元素和。

**矩阵非零元素个数**

|  |  |
| --- | --- |
| 1 | int countNonZero(InputArray src) |

函数返回矩阵的非零元素的个数。

**矩阵的元素均值和方差**

元素的均值和方差的函数如下：

|  |  |
| --- | --- |
| 1  2 | Scalar mean(InputArray src, InputArray mask=noArray()) //只计算均值  void meanStdDev(InputArray src, OutputArray mean, OutputArray stddev, InputArray mask=noArray()) //计算均值和方差 |

这里计算的方差其实就是归一化的协方差矩阵的对角线的和。

**矩阵的最大最小值**

计算元素的最大最小值，OpenCV针对不同的情况，提供了较多的函数， 最大值有：

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | MatExpr max(const Mat& a, const Mat& b)  MatExpr max(const Mat& a, double s)  MatExpr max(double s, const Mat& a)  void max(InputArray src1, InputArray src2, OutputArray dst)  void max(const Mat& src1, const Mat& src2, Mat& dst)  void max(const Mat& src1, double src2, Mat& dst) |

最小值有：

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | MatExpr min(const Mat& a, const Mat& b)  MatExpr min(const Mat& a, double s)  MatExpr min(double s, const Mat& a)  void min(InputArray src1, InputArray src2, OutputArray dst)  void min(const Mat& src1, const Mat& src2, Mat& dst)  void min(const Mat& src1, double src2, Mat& dst) |

还有能给出最大最小值的函数如下：

|  |  |
| --- | --- |
| 1  2 | void minMaxIdx(InputArray src, double\* minVal, double\* maxVal, int\* minIdx=0, int\* maxIdx=0, InputArray mask=noArray())  void minMaxLoc(InputArray src, double\* minVal, double\* maxVal=0, Point\* minLoc=0, Point\* maxLoc=0, InputArray mask=noArray()) |

这两个函数调用很类似，主要不同在于最大最小值的位置的输出不同。minMaxIdx()的最大最小值位置坐标顺序地存储在maxIdx和minIdx数组中，而minMaxLoc()则直接存储在Point中，注意minMaxIdx()存储是按照维数排列的，即行、列。而minMaxLoc()存储的Point是按照坐标x，y存储的，即列、行。

**矩阵的其他相关操作**

除了上面提到的操作外，OpenCV还提供了一些额外的相关操作。

**矩阵通道操作**

针对矩阵的不同通道进行操作，例如图像的色彩模式转换，OpenCV也提供了相对方便的函数，主要有：

* 把多通道的矩阵分拆到多个矩阵的函数

|  |  |
| --- | --- |
| 1 | void split(const Mat& src, Mat\* mvbegin) |

* 把多个矩阵合并到一个多通道矩阵

|  |  |
| --- | --- |
| 1 | void merge(const Mat\* mv, size\_t count, OutputArray dst) |

* 还有一个更通用的函数，可以把多个矩阵的相关通道分拆或合并到其他矩阵中

|  |  |
| --- | --- |
| 1  2 | void mixChannels(const Mat\* src, size\_t nsrcs, Mat\* dst, size\_t ndsts, const int\* fromTo, size\_t npairs)  void mixChannels(const vector<Mat>& src, vector<Mat>& dst, const int\* fromTo, size\_t npairs) |

这里举一个OpenCV的简单例子来说明这个函数的使用

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | Mat rgba( 100, 100, CV\_8UC4, Scalar(1,2,3,4) );  Mat bgr( rgba.rows, rgba.cols, CV\_8UC3 );  Mat alpha( rgba.rows, rgba.cols, CV\_8UC1 );  Mat out[] = { bgr, alpha };  // rgba[0] -> bgr[2], rgba[1] -> bgr[1],  // rgba[2] -> bgr[0], rgba[3] -> alpha[0]  int from\_to[] = { 0,2, 1,1, 2,0, 3,3 };  mixChannels( &rgba, 1, out, 2, from\_to, 4 ); |

针对OpenCV的老的数据结构CvMat和IplImage，OpenCV还提供了另外两个函数实现相关操作：

|  |  |
| --- | --- |
| 1  2 | void extractImageCOI(const CvArr\* arr, OutputArray coiimg, int coi=-1 )  void insertImageCOI(InputArray coiimg, CvArr\* arr, int coi=-1 ) |

其中，extractImageCOI()是实现提取arr中的某一通道，到coiimg中去；coiimg必须为单通道。insertImageCOI是把单通道的coiimg嵌入到arr中去；如果coi>=0，则提取对应coi通道的矩阵，否则，提取IplImage标记的COI通道

**查表替换矩阵元素**

OpenCV提供了一个查表替换矩阵中元素的操作，函数声明如下：

|  |  |
| --- | --- |
| 1 | void LUT(InputArray src, InputArray lut, OutputArray dst, int interpolation=0 ) |

**src**是一个8位的矩阵，**lut**是一个256的查询表，依照**lut**中对每个值的映射，替换相应的元素值，存储到**dst**中去。其实现的操作如下：

dst(I)=lut(src(I)+d)dst(I)=lut(src(I)+d)

d={0,128,如果src.depth()=CV\_8U;如果src.depth()=CV\_8S;d={0,如果src.depth()=CV\_8U;128,如果src.depth()=CV\_8S;

**极坐标的的极角和极径**

两个矩阵对应元素组成的一系列二维坐标的的极角和极径的计算可以使用如下函数：

|  |  |
| --- | --- |
| 1 | void cartToPolar(InputArray x, InputArray y, OutputArray magnitude, OutputArray angle, bool angleInDegrees=false) |

反过来，可以通过极坐标下的极角和极径，计算出对于的二维向量坐标：

|  |  |
| --- | --- |
| 1 | void polarToCart(InputArray magnitude, InputArray angle, OutputArray x, OutputArray y, bool angleInDegrees=false) |

除此之外，还可以单独计算二维坐标的极角和极径，函数调用为：

|  |  |
| --- | --- |
| 1  2 | void magnitude(InputArray x, InputArray y, OutputArray magnitude) \\计算向量的极径  void phase(InputArray x, InputArray y, OutputArray angle, bool angleInDegrees=false) \\计算向量的极角 |

**检查矩阵元素的范围**

|  |  |
| --- | --- |
| 1 | bool checkRange(InputArray a, bool quiet=true, Point\* pos=0, double minVal=-DBL\_MAX, double maxVal=DBL\_MAX ) |

检查矩阵**a**中的元素有没有超过**[minVal,maxVal]**这个范围,**pos**记录元素位置，标志位**quiet**决定是否抛出异常还是仅仅返回bool值false。

**对角拷贝**

OpenCV还提供了一种对角拷贝的操作，即把右上角的元素拷贝到左下角，或者相反，中间对角线元素不变。函数如下：

|  |  |
| --- | --- |
| 1 | void completeSymm(InputOutputArray mtx, bool lowerToUpper=false) |

其中lowerToUpper设置为true的时候，左下角拷贝到右上角，反之，右上角拷贝到左下角。

**Mat::convertTo()升级版**

除了前面[Mat详解](http://localhost:4000/2014/03/26/OpenCV(III)-How-to-use-Mat/#mat提供的常用成员函数)中介绍的Mat::convertTo()可实现类型的转换外，OpenCV还提供了如下函数，实现一个组合功能即：缩放，求绝对值，转换类型到8-bit：

|  |  |
| --- | --- |
| 1 | void convertScaleAbs(InputArray src, OutputArray dst, double alpha=1, double beta=0) |

其实现的结果是

dst(I)=saturatecast<uchar>(|src(I)∗alpha+beta|)dst(I)=saturatecast<uchar>(|src(I)∗alpha+beta|)

**旧结构体和Mat的转换**

除了前面[Mat详解](http://localhost:4000/2014/03/26/OpenCV(III)-How-to-use-Mat/#获取mat元素的方法)中介绍的直接转换方式外，OpenCV还提供了如下函数进行转换：

|  |  |
| --- | --- |
| 1 | Mat cvarrToMat(const CvArr\* arr, bool copyData=false, bool allowND=true, int coiMode=0 ) |

其中，

* **CvArr——**是CvMat，IplImage，CvMatND类型的数组
* **copyData——**决定是否拷贝数据，如果为true，则拷贝到新的结构体中，否则仅创建一个头，指向原数据。
* **allowND——**决定是否支持多维数组，如果为true，则把多维数组转换到一个二维数组中，如果不能转换，则返回错误。
* **coiMode——**这个标识符是为IplImage的指定通道设置的，类似ROI的COI(Channel of Interest)。如果coiMode=0，且IPlImage的COI是设置的，则返回错误；如果coiMode=1,则不返回错误，而是直接返回整个原始图像的头，需要自己通过函数extractImageCOI()单独提取相应的通道。

**协方差矩阵**

[协方差矩阵](http://zh.wikipedia.org/wiki/%E5%8D%8F%E6%96%B9%E5%B7%AE%E7%9F%A9%E9%98%B5)表示的是高维度随机变量中各个元素之间的协方差。计算协方差矩阵的函数为：

|  |  |
| --- | --- |
| 1  2 | void calcCovarMatrix(const Mat\* samples, int nsamples, Mat& covar, Mat& mean, int flags, int ctype=CV\_64F)  void calcCovarMatrix(InputArray samples, OutputArray covar, OutputArray mean, int flags, int ctype=CV\_64F) |

其中第一种调用其实是直接调用的第二个调用方式。对于第一种调用，**samples**是输入的随机向量指针，**nsamples**是个数。对于第二种调用，**sample**是以矩阵的形式存储随机向量，通过**flag**的值指定是列向量还是行向量。**flag**是通过以下几种值进行组合的：

首先，设一个由三个随机列向量[v1,v2,v3][v1,v2,v3]组成的矩阵M如下：

M=[v1,v2,v3]=[x1y1x2y2x3y3]M=[v1,v2,v3]=[x1x2x3y1y2y3]

* **CV\_COVAR\_SCRAMBLED：**设置该标志位时，计算协方差的矩阵的方式和正常的不同，主要用于快速PCA的计算，它是通过如下公式计算的：

Mcov=⎡⎣⎢x1−x¯¯¯x2−x¯¯¯x3−x¯¯¯y1−y¯¯¯y2−y¯¯¯y3−y¯¯¯⎤⎦⎥[x1−x¯¯¯y1−y¯¯¯x2−x¯¯¯y2−y¯¯¯x3−x¯¯¯y3−y¯¯¯]Mcov=[x1−x¯y1−y¯x2−x¯y2−y¯x3−x¯y3−y¯][x1−x¯x2−x¯x3−x¯y1−y¯y2−y¯y3−y¯]

其中x¯¯¯,y¯¯¯x¯,y¯是随机变量X,YX,Y的均值。

* **CV\_COVAR\_NORMAL：**该标志位使用的方法是普通的协方差矩阵计算方法，计算公式如下：

Mcov=[x1−x¯¯¯y1−y¯¯¯x2−x¯¯¯y2−y¯¯¯x3−x¯¯¯y3−y¯¯¯]⎡⎣⎢x1−x¯¯¯x2−x¯¯¯x3−x¯¯¯y1−y¯¯¯y2−y¯¯¯y3−y¯¯¯⎤⎦⎥Mcov=[x1−x¯x2−x¯x3−x¯y1−y¯y2−y¯y3−y¯][x1−x¯y1−y¯x2−x¯y2−y¯x3−x¯y3−y¯]

这里需要说明一点，CV\_COVAR\_SCRAMBLED和CV\_COVAR\_NORMAL这两个标志位只能设置其中一个。

* **CV\_COVAR\_USE\_AVG：**当设置该标志位时，不计算mean，而是根据输入的mean直接计算协方差矩阵。
* **CV\_COVAR\_SCALE：**当该标志位设置时，CV\_COVAR\_SCRAMBLED模式下，scale=1/nelement，nelement是每个随机向量里元素的个数，比如上例中的nelement就是2。CV\_COVAR\_NORMAL模式下，scale=1/nsample，nsample是向量的个数，比如上例中的nsample就是3。
* **CV\_COVAR\_ROWS：**该标志位只对第二种调用有用，是指定计算行向量的协方差矩阵。
* **CV\_COVAR\_COLS：**该标志位只对第二种调用有用，是指定计算列向量的协方差矩阵。

这里对第二种调用提供一个简单的示例，

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40 | #include <iostream>  #include <fstream>  #include "opencv2/highgui/highgui.hpp"  #include "opencv2/core/core.hpp"  using namespace std;  using namespace cv;  void showResults(ofstream &file, Mat covMat, Mat mean, string str)  {  file<<str<<endl;  file<<"covMat:"<<endl;  file<<covMat<<endl;  file<<"mean:"<<endl;  file<<mean<<endl;  }  int main()  {  float m[2][3] = { {2, 7, 3}, {5 ,4, 6} };  ofstream file("results.txt");  Mat M(2,3,CV\_32F,m);  file<<"M:"<<endl;  file<<M<<endl;  Mat covMat,mean;  calcCovarMatrix(M,covMat,mean,CV\_COVAR\_NORMAL+CV\_COVAR\_COLS);  showResults(file,covMat,mean,"Calculate normal colum vector:");  calcCovarMatrix(M,covMat,mean,CV\_COVAR\_NORMAL+CV\_COVAR\_ROWS);  showResults(file,covMat,mean,"Calculate normal row vector:");  calcCovarMatrix(M,covMat,mean,CV\_COVAR\_SCRAMBLED+CV\_COVAR\_COLS);  showResults(file,covMat,mean,"Calculate scrampled col vector:");  calcCovarMatrix(M,covMat,mean,CV\_COVAR\_NORMAL+CV\_COVAR\_COLS+CV\_COVAR\_SCALE);  showResults(file,covMat,mean,"Calculate normal scale col vector:");  return 0;  } |

以上例子计算的是三个列向量(2,5),(7,4),(3,6)对应的随机变量(X,Y)的协方差矩阵。这里，**Cov**对应协方差矩阵，**mean**对应(X,Y)的均值，CV\_COVAR\_NORMAL+CV\_COVAR\_COLS对应计算列向量的正常的协方差矩阵，CV\_COVAR\_NORMAL+CV\_COVAR\_ROWS对应的是计算行向量的正常协方差矩阵，CV\_COVAR\_SCRAMBLED+CV\_COVAR\_COLS对应的是计算列向量的**SCRAMPLED**的协方差矩阵,V\_COVAR\_NORMAL+CV\_COVAR\_COLS+CV\_COVAR\_SCALE对应的是计算带有scale的列向量的正常协方差矩阵。这里注意输入矩阵的数据类型至少为float型，即CV\_32F。运行结果为：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | M:  [2, 7, 3;  5, 4, 6]  Calculate normal colum vector:  covMat:  [14, -4;  -4, 2]  mean:  [4;  5]  Calculate normal row vector:  covMat:  [4.5, -4.5, 4.5;  -4.5, 4.5, -4.5;  4.5, -4.5, 4.5]  mean:  [3.5, 5.5, 4.5]  Calculate scrampled col vector:  covMat:  [4, -6, 2;  -6, 10, -4;  2, -4, 2]  mean:  [4;  5]  Calculate normal scale col vector:  covMat:  [4.666666666666666, -1.333333333333333;  -1.333333333333333, 0.6666666666666666]  mean:  [4;  5] |

**计算马氏距离**

[马氏距离](http://zh.wikipedia.org/wiki/%E9%A9%AC%E6%B0%8F%E8%B7%9D%E7%A6%BB)是计算数据的协方差距离，是一种有效的计算两组数据相似性的度量。和欧氏距离不同，马氏距离考虑各种特性之间的联系。调用函数如下：

|  |  |
| --- | --- |
| 1 | double Mahalanobis(InputArray v1, InputArray v2, InputArray icovar) |