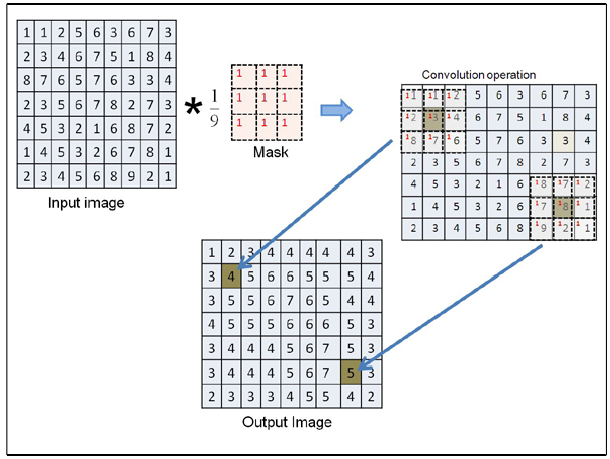
关于图片的基本知识：OpenCV中的图片大多数是以矩阵Mat来表示的，Mat中的每一个点在图片中都代表一个像素，其值的表示方法为一个double[]，数组元素个数取决于图片本身的通道数，即RGBA。也就是，一张RGBA四通道的图片，其矩阵的点是一个四元素的double[],而数组中值的大小取决于图片的位数。这在初始创建图片时可以确定，如设置type为CvType.CV\_8UC4时，该图片为一张8位4通道图片，它有RGBA四个值，每个值的范围是0-255，而type为CvType.CV\_16UC4时，该图片为16位四通道图片，每个值的范围为0-65535，不过此时其存储数据类型是无符号short int型，而CvType.CV\_16SC4，作为有符号的short int型，其范围为-32768-32767，以此类推。

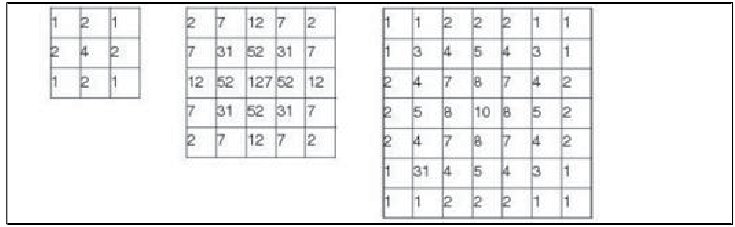
1. 线性过滤是实现图片模糊化的一种主要方式，其算法中大多数会用到名为”kernel（内核）”的数组，其作用相当于一个过滤窗口，使原图进行所需的变化。过程如下：



卷积是线性滤波中最常用的操作之一，而线性滤波常用来去噪。

1. 模糊化：方法Imgproc.blur(src, src, new Size(3,3)) src为原图矩阵，Size（3，3）为kernel数组，值越大模糊程度越高。

方法Imgproc.GaussianBlur(src, src, new Size(3,3), 0); 高斯模糊方法，最后一个参数为（sigma值）不同大小的kernel数组如下所示



方法Imgproc.medianBlur(src, src, 3); 为中值模糊，将图片像素值正序或反序排列，取中间值已获得去噪效果（没有利用卷积）。

1. 锐化：创建合适的kernel 矩阵

Mat kernel = new Mat(3,3,CvType.CV\_16SC1); //单像素16bit，单通道

kernel.put(0, 0, 0, -1, 0, -1, 5, -1, 0, -1, 0);

方法 Imgproc.filter2D(src, src, src.depth(), kernel);以此对图片进行了锐化处理，得到更清晰的图片

1. 膨胀化：设置合适的kernel，用相应的kernel以指定区域的偏大值将指定区域替换（例如1占多数即让0被1替换） Open CV提供的kernel有MORPH\_RECT, MORPH\_CROSS和MORPH\_ELLIPSE ，即将指定区域的内容以方形，十字或椭圆的结构替换，例如

Mat kernelDilate = Imgproc.getStructuringElement( Imgproc.MORPH\_RECT, new Size(3, 3));

Imgproc.dilate(src, src, kernelDilate);可以得到图片相应的膨胀效果

1. 腐蚀化：设置合适的kernel，用相应的kernel以指定区域的偏小值将指定区域替换，例

Mat kernelErode = Imgproc.getStructuringElement(Imgproc.MORPH\_ELLIPSE, new Size(5, 5));

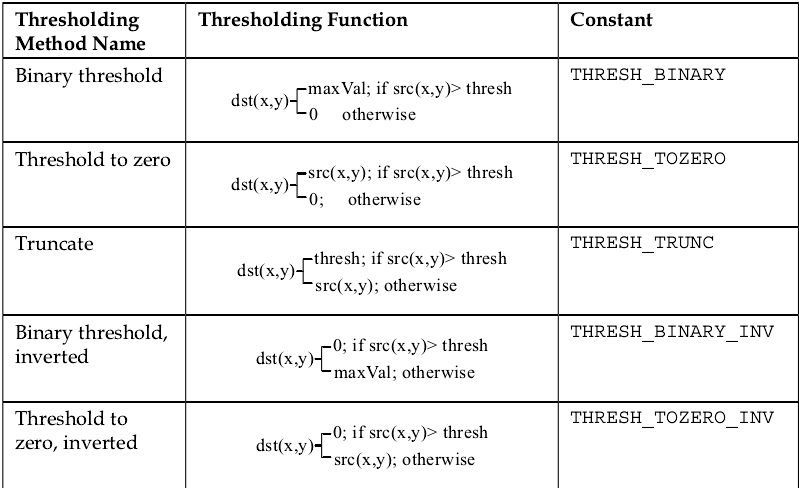
Imgproc.erode(src, src, kernelErode);可以得到图片相应的腐蚀效果，（会有微缩效果）。

1. 二值化：通过设置临界值和最大值将图片像素的值修改为两个值（由函数常数参数决定）

方法：Imgproc.threshold(src, src, 100, 255, Imgproc.THRESH\_CONSTANT);

100为临界值，255为最大值（如二进制二值化时，max Val的值即为255，所以为白色）

常数参数各值意义如下：



适配二值化：通过设置周边值确定二值化值

参数：

ADAPTIVE\_THRESH\_MEAN\_C: 二值化值为周边值的平均值

ADAPTIVE\_THRESH\_GAUSSIAN\_C: 二值化值为高斯kernel权重的总值

Imgproc.adaptiveThreshold(src,src,255,Imgproc.ADAPTIVE\_THRESH\_GAUSSIAN\_C,Imgproc.THRESH\_BINARY, 3, 0);//二值化值为高斯kernel权重的总值，二进制二值化，kernel为3\*3矩阵，值偏差值为0。

1. 检测边缘和角落：其检测原理为边缘处像素值落差较大,而角落为边缘相交点或边缘明显的边缘方向聚合点。

高斯差分（DoG）算法：先将图片转化为灰度图，然后用两个不同半径的高斯模糊对图像进行处理，这样一来边缘特征会受到两种不同程度的削弱，而非边缘处变化较小，再将两次模糊所得结果相减，便可以得到特征削弱较大的点，即可区分出边缘。

Core.absdiff(blur1, blur2,DoG); //两个高斯模糊相减得到含特征信息的矩阵

Core.multiply(DoG,new Scalar(100), DoG);//特征矩阵与梯度阵相乘

Imgproc.threshold(DoG,DoG,50,255 ,Imgproc.THRESH\_BINARY\_INV);//二值化注意这里为BINARY\_INV，因为得到的DoG是一个差值，故需要反向赋值才能够使之值排列与原图一致。

坎尼边缘检测算法（The Canny Edge Detector）：首先对图片作恰当的模糊化处理，然后计算图片梯度值，接着对非梯度正、负方向极值（即非边缘值）进行削减，实现边缘细化，最后检查边缘值是否足以进行图像输出，除去不够可靠的边缘值。

方法Imgproc.Canny(grayMat, cannyEdges,10, 100); 第一个参数为灰度图矩阵，第二参数为输出矩阵，第三和第四个参数为设定阈值，由两个阈值确定三种情况，分别为suppressed

points, weak edge points和strong edge points，较大者绝对为绝对可靠的边缘值，值在上限阈值之上，而中间值为需要判断的值，若其八个像素内有可靠的边缘值存在，即将其计为边缘值的一部分，较小值直接被忽略。

索贝尔边缘检测算法（The Sobel Operator）：先得到目标图片的灰度图，通过横向矩阵与纵向矩阵求出图片两个方向的梯度强度，然后将两个方向经操作后的矩阵相加，即可得到边缘化的图片矩阵。

方法

Imgproc.Sobel(grayMat,grad\_x,CvType.CV\_16S,1,0,3,1,0);

Imgproc.Sobel(grayMat,grad\_y,CvType.CV\_16S,0,1,3,1,0);

Core.convertScaleAbs(grad\_x,abs\_gard\_x);

Core.convertScaleAbs(grad\_y,abs\_gard\_y);

Core.addWeighted(abs\_gard\_x,0.5,abs\_gard\_y,0.5,1,sobel);

哈瑞斯角落检测（Harris Corner detection）：用一个移动窗口计算图片值强度变化，通过横向和纵向强度变化和判断变化程度，两者大到一定数值即认为其为角落。

Imgproc.cornerHarris(grayMat,tempDst,2,3,0.04);//参数分别为，原始图的灰度图，用以记录特征点的矩阵，区域大小，Sobel算法的孔径参数（kernel的大小），harris公式的一个系数

Core.normalize(tempDst, tempDst, 0, 255, Core.NORM\_MINMAX);//归一化输入数值，使其输出数值在一定范围内，NORM\_MINMAX线性归一化，将数值平移或缩放到一定数值

Core.convertScaleAbs(tempDst,corners);//取绝对值

霍夫转换：（该方法无法识别比较复杂的图像）

（1）霍夫直线：通过取已知边缘点排布值代入直线公式（遍历所有系数），若得到的计算结果拟合度大于一定值，则判定其为直线。Open CV提供的两种模式分别为standard

Hough lines 和probabilistic Hough lines，后者只随机取部分边缘点计算判断，故速度较快。

方法：Imgproc.HoughLines(edges, lines, 1, Math.PI / 180, 50, 20, 20);//各系数分别为 所求得的边缘矩阵，输出直线矩阵，像素分辨率圆半径，角度值，直线的临界值和最小值。得到的矩阵通过get方法得到的数组按角标排列为起始点x，y，终点x，y。

houghLines.create(edges.rows(),edges.cols(),CvType.CV\_8UC1);

1. 霍夫圆：判断原理同上，但参数代入的是圆方程。

方法：Imgproc.HoughCircles(edges, circles, Imgproc.CV\_HOUGH\_GRADIENT, 1, edges.rows() / 15);//得到的矩阵通过get方法得到的数组，按角标排列为所在点x坐标，y坐标，半径r

houghCircles.create(edges.rows(),edges.cols(),CvType.CV\_8UC1);

轮廓线检测：类似于边缘检测，但是其差异在轮廓线必定是闭合的，其输入参数也是边缘矩阵。

方法：

Imgproc.findContours(edges,contourList,hierarchy,Imgproc.RETR\_LIST,Imgproc.CHAIN\_APPROX\_SIMPLE);//该方法得到的结果是一个轮廓线数组，存在第二个参数中，第三个参数用于存储轮廓层级，第四个参数用于指出用户需要的本质层级。

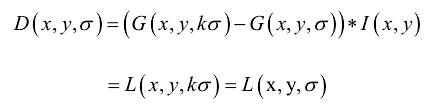
contours.create(edges.rows() ,edges.cols(),CvType.CV\_8UC3);//用于承载轮廓特征的矩阵 该方法不要求输出图像为灰度图。

Imgproc.drawContours(contours,contourList,i,new Scalar(r.nextInt(255),r.nextInt(255),r.nextInt(255)), -1);//i是数组角标，为遍历过程中的参数，最后一个参数表示轮廓线厚度，-1代表轮廓线内部完全填充 。

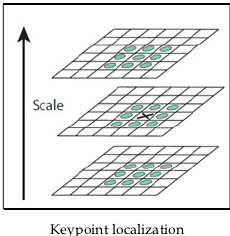
1. 特征检测算法：

SIFT(Scale Invariant Feature Transform):尺度不变特征转换，使用最多的特征探测算法（不能用于商用），其实现步骤如下：

1. 尺度空间极值检测：设置多张尺寸渐变的图（推荐为四张），通过多层高斯模糊（推荐为五层），使图像细节减少，同时去除噪声。为了更精确地得到边缘信息，这里还使用了高斯拉普拉斯算子，对模糊过的图片进行求二次导数，来确保边缘和角落足够得到好的特征点。为了获得高斯拉普拉斯图像，可用GoD方法近似获得，且这样一来获得的图片也获得了尺度不变特性。

（高斯拉普拉斯算子），k值代表了模糊层数，对不同尺寸数值不同。

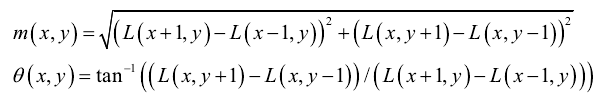
1. 关键点位置检测：由上述预处理过程中取三种尺寸的同层模糊图像，以中间尺寸的中心点向周围26个点幅散（如图所示），由这27个点中取出最大或最小值（理论上应该全体比较一遍，但通常只需要比较几次就能得到相应值），当其关键点在两个像素之间时对其进行泰勒二阶展开，将点周围数代入，当其二阶项为0时，即得到了确定的关键点位置



该算法推荐用两张极值图像进行关键点确定，故需要四张DoG图像，五张高斯模糊图像，由实践得到 σ = 1.6 and k = 2时得到的结果最好。

得到关键点之后，还需要取出一些对比不够明显的点，于是需比较当前关键点像素的值与预选的阈值，将低于阈值的除去。除此之外，为了不让边缘作为噪声影响结果，需要计算关键点正交方向的两个梯度值，当两个梯度值都很高时（即所需要的角落特征），即确定下该关键点为所求点，计算算法为Hessian matrix，直接计算两梯度值的比值。

1. 方向标记：通过下列公式分别算出关键点值的大小和方向角度



画出关于角度和大小的直方图，对于赋值峰值点作为新的关键点，大小权重高于80%的点将其作为辅助关键点（权重分母为梯度值总和），并记录其方向。

1. 关键点描述符：以关键点为中心，取周围16个4\*4格子，流失一些精度的条件下，可以将每个格子内个方向向量计入一个8位直方图，每一位45度，将这128个方向量归一化，可以得到16个特征向量，由于引入特征向量，为保证旋转独立性，将每个方向减去原关键点的方向，为保证光照独立性，将所取值进行二值化，所有大于0.2的值都视为0.2。

具体相关代码：

FeatureDetector detector;

MatOfKeyPoint keypoints1, keypoints2;

DescriptorExtractor descriptorExtractor;

Mat descriptors1, descriptors2;

detector = FeatureDetector. create(FeatureDetector.SIFT);

descriptorExtractor = DescriptorExtractor. create(DescriptorExtractor.SIFT);

detector.detect(src2, keypoints2);

detector.detect(src1, keypoints1);//keypoint需要初始化

descriptorExtractor.compute(src1, keypoints1, descriptors1);

descriptorExtractor.compute(src2, keypoints2, descriptors2);//此处所有操作都是对两张图像的检测

1. 根据特征点匹配或检测物体

有两种算法：

1. 、Brute-force matcher：将目标特征点和待匹配特征点分别放入两个集合，采用距离算法，特征点间相距最近的作为结果返回。

方法 descriptorMatcher = DescriptorMatcher.create(DescriptorMatcher.BRUTEFORCE\_SL2);//其中参数一为特征点类型指定，SIFT和SURF为NORM\_L2,而当descriptor 基于二进制字符串，例如ORB和BRISK时,采用NORM\_HAMMING。第二个参数为可选参数，= = 。。。。

算法运用前要先声明对象 DescriptorMatcher descriptorMatcher; MatOfDMatch matches = new MatOfDMatch(); 下同

1. FLANN based matcher：用于对数据量大的特征点。运用该匹配方式时需要通过两种字典：IndexParams和SearchParams，前者用于指定所用算法和相关参数，后者指定了索引中的树状结构需要递归遍历的时间，时间越长越精确。

方法 descriptorMatcher = DescriptorMatcher.create(DescriptorMatcher.FLANNBASED);

匹配有两种方法，分别为match（）和knnMatch（），前者返回所有匹配点，而后者返回指定k匹配点，绘图也同样有drawMatches（）和drawMatchesknn（）两种方法

方法 descriptorMatcher.match(descriptors1, descriptors2, matches);

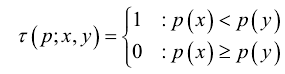
Features2d.drawMatches(im1, key1, im2, key2, matches, out);

1. 非专利的特征点检测算法

Oriented FAST and Rotated BRIEF（ORB算法）：该算法可用于代替SIFT和SURF算法，并且更加高效，基于FAST特征点检测和BRIEF标记描述。

FAST算法以其计算速度快著称，但其单独使用时并不支持唯一性标记，且对于旋转的图像检测十分弱。其使用时需要获得一个参数，即探测中心点和一定圆环范围内的中间阈值，为取得较好的结果，一般选用圆环半径为九个像素。FAST算法同时也会取得边缘附近的特征点，为消除这些影响，可使用Harris角落探测方法排列这些特征点。且FAST算法得到的特征点是没有方向的，故假设其点相对于图像中心点有一定程度的偏移，用指向中心的向量来表识其特征点的方向。FAST算法无法处理多尺度图像，需要构造图像金字塔，层层处理。

BRIEF算法同样以计算速度快著称，但其也不支持旋转不变性，故通过下列公式为其标注向量方向：



同时计算时通过特征点方向来引导BRIEF，具体过程为将二元测试中的基于x,y的矩阵与一个特征点方向相关的旋转矩阵相乘，并将旋转角度θ离散化（以12°为单位），形成一个可查的表，进行计算时通过查表方式找到与当前特征点方向一致的数据，用此时的旋转矩阵量参与标记描述。

BRIFE算法得到的特征点特性之一为方差较大且平均值接近0.5，而方差越大则其相关性越低，结果也就越好。于是在Steered BRIFE算法中，将所有可能二元测试点枚举出来画在一个31\*31像素的画布上，再从中取一个5\*5的子窗口，共26^2种取法，再从其中每次取一对二元测试点，将每一个点与其余所有点进行比较，以其与平均值0.5的差距进行排序并构造向量集T。接下来是贪婪学习算法部分：将第一个点取出存入向量集R，并从T中除去该向量，从T中取出第二个向量，与向量集R中的所有向量进行比较，如果相关性低于一定阈值，则添加进R中；重复以上过程，直到R中拥有256个向量，如果向量数少于256个，则提高阈值重新选取。

方法实现：（完整实现见测试程序代码）

首先设定特征点检测者和描述者的算法并创建ORB相关匹配器：

detector=FeatureDetector.create(FeatureDetector.ORB);

descriptorExtractor= DescriptorExtractor.create(DescriptorExtractor.ORB);

descriptorMatcher = DescriptorMatcher.create(DescriptorMatcher.BRUTEFORCE\_HAMMING);

然后检测特征点：

detector.detect(src1,keypoint01);

detector.detect(src2,keypoint02);

接着进行特征点描述，并根据描述进行匹配：

descriptorExtractor.compute(src1, keypoint01, descriptors1);

descriptorExtractor.compute(src2, keypoint02, descriptors2);

descriptorMatcher.match(descriptors1, descriptors2, matches);

（略过中间一段对匹配点排序并减少匹配对数的操作）

最后画匹配线：

Mat src3=drawMatches(src1,keypoint01,src2,keypoint02,matches,false);//该方法中调用了Feature2D的静态方法，会将配对的特征点间连上线，并且将两张对比图片横向粘合，返回结果的mat。

Bitmap bmp=Bitmap.createBitmap(src3.cols(),src3.rows(),Bitmap.Config.ARGB\_8888);

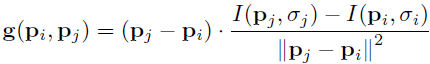
Utils.matToBitmap(src3,bmp);

（本部分可参考文章http://www.mamicode.com/info-detail-940675.html）

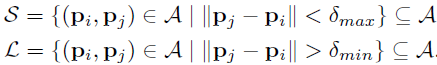
Binary Robust Invariant Scalable KeyPoint（BRISK）：二进制健壮性的尺度不变关键点算法，其诞生的目的是提升计算效率及健壮性，用于取代目前最先进的特征检测算法。其特征检测器主要是基于AGAST检测器，这是一种FAST特征检测器的延伸性检测算法。为了得到尺度不变性，BRISK选择以FAST数值为标准寻找尺度空间内的极大值。BRISK的尺度空间由n个八度与n个内八度组成，n一般推荐为4.原图设置为第一个八度，其余八度依次为前一个八度的半采样，而内八度为两个对应八度间值，为八度的降采样。

FAST 16-9检测法即在一个16个像素的圆内根据FAST的标准需要有至少9个像素点亮于或暗于中心点，而每个中心点一定范围内若有其他的待定特征点，则以周围点FAST值的差值绝对值进行比较，若响应最强，则确定为为特征点。利用FAST 16-9检测法得到的特征点，数值需要比三层八度范围内的所有相邻点（即26个点）都要大，才能作为粗略确定的特征点，在极值点所在层及其上下层所对应的位置，对FAST得分值（共3个）进行二维二次函数插值（x、y方向），得到真正意义上的得分极值点及其精确的坐标位置（作为特征点位置）；再对尺度方向进行一维插值，得到极值点所对应的尺度（作为特征点尺度）。

BRISK的特征点描述方法为：以特征点为中心从周围取N个采样点，为避免降采样带来的混叠效应，对其进行高斯模糊。N个采样点两两组合所有组合方式的集合称作采样点对，用集合IMG_256表示，其中像素分别是IMG_257、IMG_258，δ表示尺度。用IMG_259表示特征点局部梯度集合，则有：

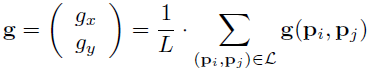


定义短距离点对子集、长距离点对子集（L个）：



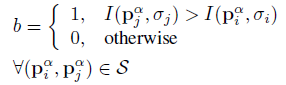
其中，IMG_262，IMG_263，t是特征点所在的尺度。

现在要利用上面得到的信息，来计算特征点的主方向（注意：此处只用到了长距离点对子集），如下：



IMG_265

 要解决旋转不变性，则需要对特征点周围的采样区域进行旋转到主方向，旋转后得到新的采样区域，采样模式同上。BRISK描述子是二进制的特征，由采样点集合可得到N(N-1)/2对采样点对，就可以得到N(N-1)/2个距离的集合（包含长、短距离子集），考虑其中短距离子集中的512个短距离点对，进行二进制编码，判断方式如下：

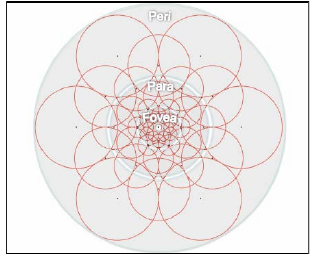


其中，IMG_257带有上标，表示经过旋转a角度后的，新的采样点。由此可得到，512Bit的二进制编码，也就是64个字节（BRISK64）。

方法：同ORB算法，只需要定义特征点探测器和描述符为BRISK即可。

Fast Retina Keypoint（FREAK）：视网膜快速关键点描述算法，其参照了人类视网膜的工作方式，对于唯一标识性的关键点和进程中识别有很好的鲁棒性，所需时间和内存更少。

其点采样分布跟BRISK相似，越接近中心点越密集，且采样范围都为圆形，但不同的是FREAK方法点偏离中心时密度是呈指数形式下降的。FREAK方法也需要进行高斯模糊来除去噪声的影响，但与BRISK和ORB不同，其每个点高斯模糊的核大小是和参数σ成正比的。同时FREAK方法遵循了ORB的方法，通过最大化点值的方差来降低点之间的相关性。



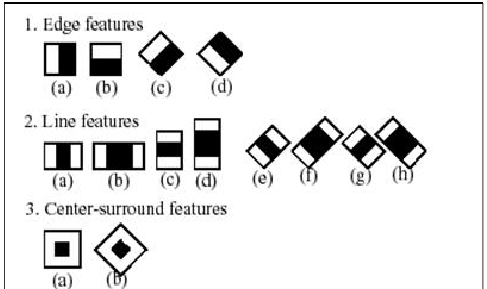
其采样点描述过程为：根据图中43个采样点建立采样对描述符矩阵，共43\*42/2=903行，计算每一列的均值（由于描述符是由二进制编码0、1构成的），按照均值接近0.5的程度由高到低排序，取前512列作为精确的描述符。该方法先取出较大范围的特征点对，再根据已经取出的对进行进一步筛选，原理类似视网膜先确定目标，再仔细分辨目标特质的工作方式。

并且根据人类扫视的特征，设置Fovea区域为高精度区域，而Para区域为低精度区域，在搜索描述符时先比较前16个字节的粗略信息，汉明距离小于某阈值时，在搜索之后更精确的结果，由此提高比较的效率，称为级联搜索。

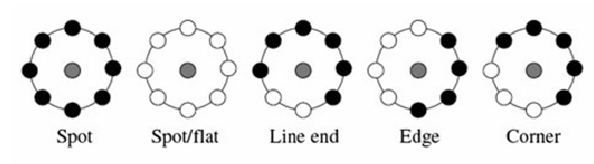
关于特征点方向，与BRISK算法类似，但FREAK没有使用长距离对，而是采用了包括45个采样对点的预定义集。

1. 级联分级器

Haar cascades ：哈尔分级器，最早的人脸识别算法之一，采用识别人脸固有结构和特征的方法，利用卷积核（有边缘，带状，中心，对角等黑白矩形模板，如图）遍历整个图片，根据白色区域像素量和黑色区域像素量的差算出特征值（中间使用了积分图方法，用一次遍历即得到的图片所有的特征值，大大提高了计算速度），若其特征值超过一定阈值（阈值由级联分类器给出），则认为已识别到。实际使用时，分类器由弱分类器经过训练成为强分类器（即判断阈值的逐步准确）再经过多个强分类器进行级联得到。可以直接用OpenCV提供的已训练的分类器。



LBP cascades ：局部二值模式（Local Binary Patterns）分级器，具有良好的灰度不变性，可以描述图片局部纹理信息。初始LBP值是由划分出图片的一个3\*3区域，比较中心点和周围八个像素点值大小，大的为1，小的为0，然后将得到的八位二进制转化为十进制而得到的。为提升其旋转不变性，可以将一个LBP点向各向旋转，得到八个LBP值，取其中最小值。而为了简化数据量（多个采样点的二进制组合指数式增长），可将LBP对应的二进制类分为等价模式类（0、1跃变次数不多于两次）和混合模式类。统计时可以将图片区域划分为若干块， 由得到的LBP特征值建立统计直方图，则图片可有这若干个统计直方图描述，再利用各种分类器进行匹配。



方法：具体代码见项目，其中需要注意

cascade.detectMultiScale(mGray, faces, 1.1, 2, 2, new Size(50, 50), new Size());//Size决定了能识别多大的脸，如果分辨率不够的话需要把数值调小，但调小后画面帧数可能会降低

HOG Descriptor：方向梯度直方图（Histogram of Oriented Gradients）描述算子是一种根据方向梯度强度和边缘方向来描述特征点的算法。其计算过程是把图像分解为各个单元，然后计算每个单元的直方图，再将所有直方图结合得到一个描述算子。该算法与SIFT相似，但SIFT是以关键点位置来得到描述算子的，而该算法选择用一个移动窗口来检测物体。HOG具有优良的几何与光照转换特性，但没有旋转不变性。

其检测过程如下：选择一个方向（或者两个都选）的导数掩模计算每个像素点的梯度值，可以使用Sobel算子或者Prewitt算法，但原算法推荐使用一维的导数掩码 [-1,0,1]。然后将得到的梯度值分为0-180或0-360的bin值。对每个单元计算直方图后，组成归一化直方图，最终HOG算子就是归一化直方图的向量。最后将HOG得到的特征向量输入筛选算法，用于检测目标。

方法：

HOGDescriptor hog=new HOGDescriptor();  
hog.setSVMDetector(HOGDescriptor.getDefaultPeopleDetector()); //SVM，支持向量机，一种分类器  
MatOfRect faces =new MatOfRect();  
MatOfDouble weights=new MatOfDouble();  
hog.detectMultiScale(grayMat,faces,weights);//所有人脸位置信息都存在faces中

（代码实现效果极差，具体问题待考究）

1. 图像追踪

Optical Flow（光流法）：光流法是一种能够描述出前一刻和后一刻物体移动向量的算法，其功能建立在两个假设上：一、连续画面的像素强度基本是连续的。二、目标点周围的像素与其运动基本一致。由此可以将图像变化视为一个平稳过程，从而写出其数学描述公式，由于其方程数不足以得出所有未知数，故有不同的解。（中间数学过程暂略）根据其具体实现来看，其实就是将图像移动的特征点分别取出做位置匹配，从而得出变化轨迹（此时初始特征点是固定的）。

方法：（具体看项目文件夹代码）

Video.calcOpticalFlowPyrLK(mPreGary,mGray,preFeature,nextFeature,status,err);//主要实现函数，其中preFeature为原特征点位置，nextFeature则包含了移动后的特征点位置信息。

为了弥补光流法假设对图像追踪造成的限制，采取图像金字塔的方式，层层扩大视觉范围，来保证物体运动距离长时也能被探知到。其中构建图像金字塔有两种方式三个方法，分别为高斯金字塔和拉普拉斯金字塔，前者可以对图片进行放大或缩小（默认为两倍）并进行模糊化处理，后者在OpenCV中没有固定的函数，可以由前者两个方法的结果相减得到，表现为物体的轮廓线。

方法：Imgproc.pyrDown(src, srcRes);

Imgproc.pyrUp(srcRes, srcRes);

Core.absdiff(src, srcRes, srcRes);

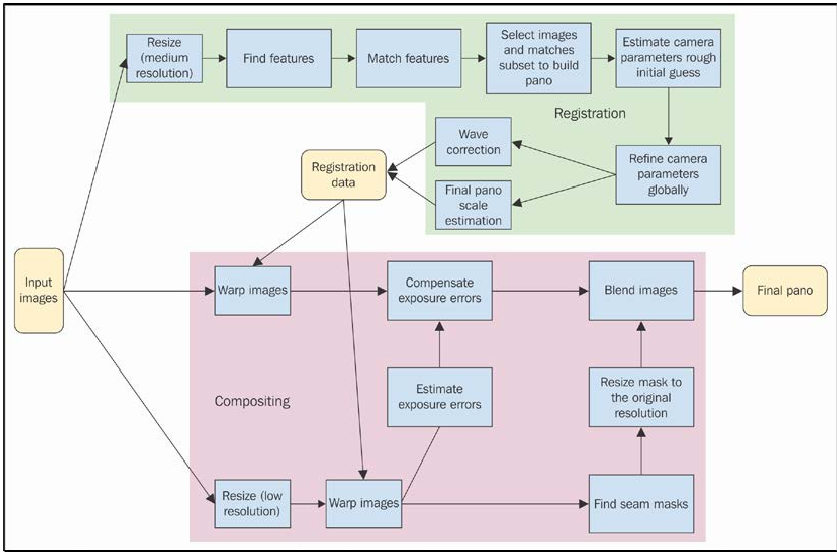
KLT追踪器（Kanade-Lucas-Tomasi Tracker）:KLT算法是光流法中的一种，其基本步骤为：① 寻找初始帧画面中可靠的特征点（一般为角落点） ②用光流法和二维变换投影检测每个特征点的移动 ③将帧与帧间的移动向量联系起来，得到特征点的移动轨迹 ④ 创建一定数量的新特征点来取代被遮住或移出屏幕的特征点 ⑤继续检测新旧特征点的移动轨迹

方法：Imgproc.goodFeaturesToTrack(mGray,features,10,0.01,10);

Video.calcOpticalFlowPyrLK(mPreGary,mGray,preFeature,nextFeature,status,err);

1. 图像的对齐和拼接

为解决相机的视场缺陷，实现全景拍摄，需要应用图像的拼接方法，其主要步骤如下：①、特征提取和匹配：运用特征点检测算法，获得特征点尺度、方向和光强等属性，保证其不变性，再通过比较重叠特征将图像大致匹配起来。接下来利用RANSAC算法计算若干含重叠特征点图像的单应性（线性射影变换，即三维物体在不同视角下的平面投影），筛选出一对单应性匹配度最高的图像视为邻居图像，当这对图像匹配概率足够高时则确认其相关性。②、匹配完成后，需要对图片进行全景校直，其基本方法即通过寻找水平方向的正交向量和同向向量对图像进行旋转。③、之后需要对图像进行增益补偿（光敏感度补偿），主要利用重叠特征点的误差对比。④、最后对图像作多频段融合，为图像分配权重，由中心依次降低，接下来看不懂。。。具体全过程如图：



方法：最重要的部分是C++代码，实现的时候会出现内存问题，书中建议是将原始数据交给服务器处理再发回，待日后研究。

1. 机器学习和字符识别

①、KNN算法（最近相邻比较算法）：其主要原理为对比与目标最近的K个样本值的特征，根据样本分析归类，然后选择相似数量最多的样本类别作为目标的类别。需要注意的是进行对比的样本和测试数据都是以行向量的方式存储和比较的，故实际运用时会有许多矩阵转化成行单向量矩阵的过程。书中直接利用了已训练好的数据集作为对比样本，同时附带了数据集的标签，这些都需要进行预先导入。

代码：

for(int i=0;i<totalImages;i++){  
 byte[] image=new byte[px\_count];  
 imageReader.read(image,0,px\_count);  
 traningImage.put(i,0,image);  
}  
traningImage.convertTo(traningImage,CvType.CV\_32FC1);//totalImage为训练用图片的总数量，可以由下载到的数据帧头中获得该信息，px\_count为计算得到的单张图片像素，traningImage表示训练用图片集的矩阵。这个循环代码完成了将训练用图片转化为行向量的功能，一张图片一行。

labelsReader.read(header,0,8);//先将指向label文件的指针移到属性信息之后

labelsReader.read(labelsData, 0, totalImages);

tempLabels.put(0, 0, labelsData);//将所有图片的标签信息都存到相应矩阵中，每个信息一字节

Core.transpose(tempLabels, traningLabels);//此处做一个转置，使一行n列的矩阵变成了n行一列的矩阵，则标签与训练图片相对应。

knn=new CvKNearest();

knn.train(traningImage,traningLabels,new Mat(),false,10,false);//通过对训练数据的训练，得到一个初步拥有辨别能力的KNN模型。

Imgproc.dilate(testImage, testImage, Imgproc.getStructuringElement(Imgproc.CV\_SHAPE\_CROSS, new Size(3, 3)));//膨胀化，降低噪声影响，得到更准确的数据。

Imgproc.resize(testImage, testImage, new Size(width, height));//将维度降到与训练图片一致

Imgproc.cvtColor(testImage,testImage,Imgproc.COLOR\_BGR2GRAY); Imgproc.adaptiveThreshold(testImage,testImage,255,Imgproc.ADAPTIVE\_THRESH\_MEAN\_C,Imgproc.THRESH\_BINARY\_INV,15,2);//自适应二值化，以上预处理都是为了使测试图片更接近训练图片

Mat test=new Mat(1,testImage.cols()\*testImage.rows(),CvType.CV\_32FC1);

int count=0;

//将测试图片的矩阵转化成一个行向量，存在test矩阵中

for(int i=0;i<testImage.rows();i++){

for(int j=0;j<testImage.cols();j++){

test.put(0,count,testImage.get(i,j)[0]);

count++;

}

}

Mat result=new Mat(1,1,CvType.CV\_8U);

knn.find\_nearest(test,10,result,new Mat(),new Mat());//最后分类结果存在result中

Mat中的submat方法通过输入起始行列得到一个父矩阵的子矩阵，如果对子矩阵进行

/\* OpenCV中Mat函数创建时传入的参数rows和cols指的是行和列的数量，即rows代表高，而cols代表宽。\*/