作者: 293311923@qq.com
URL: https://github.com/snowflowersnowflake/Sfm-python/blob/master/revise_v2.py

实现一个基本的sfm算法

> 现行的商业算法在sfm方面十分成熟，虽然个人所做的算法效果不能比拟大量优化后的商业算法，但是对于理解sfm的核心思想，增加实践能力，十分有帮助

基本参数初始化

下面是有关代码以及注释:

```
image_dir = 'G:/3D-CVSFM/img/' #初始化sfm_img存储路径
MRT = 0.7

#下面初始化一个相机内参矩阵,其中，K[0][0]和K[1][1]代表相机焦距，而K[0][2]和K[1][2]代表图像的中心像素。
#利用np我们可以轻松构造矩阵
K = np.array([
        [2362.12, 0, 720],
        [0, 2362.12,  578],
        [0, 0, 1]])

#选择性删除所选点的范围。
x = 0.5
y = 1
```

这样我们就完成了一个参数初始化过程，包括我们sfm过程中需要的**超参数。**

需要注意的是其中MRT(matching ratio test)是一个与匹配阈值呈正相关的参数，**在特征匹配过程中我们需要对匹配率进行比照，选中对应点对的匹配率高于匹配阈值才被匹配成功。**

两张图之间的特征提取及匹配

提取

```
def extract_features(image_names):
    sift = cv2.xfeatures2d.SIFT_create(0, 3, 0.04, 10) #调用cv2中的特征检测函数
    key_points_for_all = []
    descriptor_for_all = []
    colors_for_all = []
    for image_name in image_names: #遍历图像文件夹中的图像，用cv2读入函数读入
        image = cv2.imread(image_name)

        if image is None: #如果是空图像则用continue跳过该函数，即不进行匹配
            continue
        key_points, descriptor = sift.detectAndCompute(cv2.cvtColor(image, cv2.COLOR_BGR2GRAY), None)

        if len(key_points) <= 10: #如果匹配的点对小于10，认为图像间匹配不成功，和空图像一样跳过
            continue
        #匹配成功则将描述子和点对加入匹配组中
        key_points_for_all.append(key_points)
        descriptor_for_all.append(descriptor)
        colors = np.zeros((len(key_points), 3))
        for i, key_point in enumerate(key_points): #遍历keypoint（关键点）
            p = key_point.pt #赋值坐标
            colors[i] = image[int(p[1])][int(p[0])]    #通过点在图像上的坐标提取其像素值（像素值即颜色）
        colors_for_all.append(colors) #保存匹配点对的颜色
    return np.array(key_points_for_all), np.array(descriptor_for_all), np.array(colors_for_all)
```

keypoint的数据结构

keypoint是opencv对于图片中关键点对所创建的数据结构，包括如下参数：

angle：角度，表示关键点的方向，为了保证方向不变形，SIFT算法通过对关键点周围邻域进行梯度运算，求得该点方向。-1为初值。

class_id：当要对图片进行分类时，我们可以用class_id对每个特征点进行区分，未设定时为-1，需要靠自己设定

octave：代表是从金字塔哪一层提取的得到的数据。

pt：关键点点的坐标

response：响应程度，代表该点强壮大小。

size：该点直径的大小

xfeatures2d

xfeatures2d是cv2中包涵的一个用于提取2D图像特征的库。
在上述代码中，xfeatures2d.SIFT_create则是调用了其中的sift算法。

xfeatures2d其中包括三个算法，可以用于提取图像特征：

- SIFT，基于尺度不变特征的变换算法（Scale-invariant feature transform，SIFT）
- SURF,鲁棒性特征增强算法（SpeededUp Robust Features）
- ORB，简快描述子特征提取算法

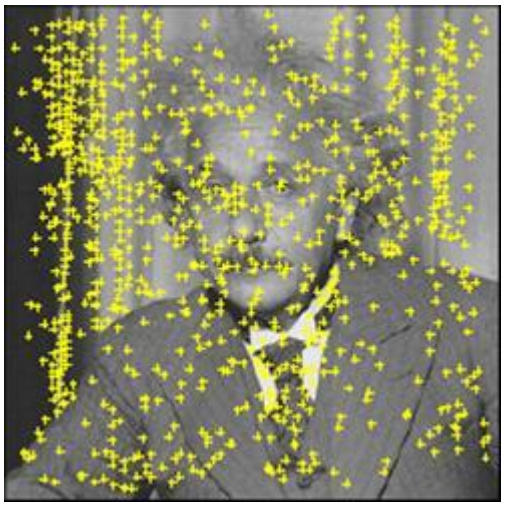
SIFT

简介

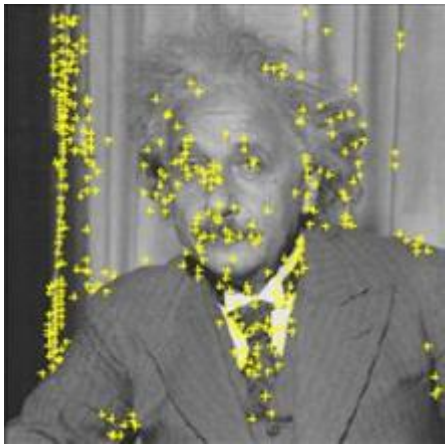
SIFT特征是基于物体上的一些局部外观的兴趣点而与影像的大小和旋转无关，通过挖掘这些不受图像本身改变的SIFT特征，来**以特征点代表图像本身的特殊性。**SIFT对于光线、噪声、微视角改变的容忍度也相当高。基于这些特性，它们是高度显著而且相对容易撷取，在母数庞大的特征数据库中，很容易辨识物体而且鲜有误认。使用SIFT特征描述对于部分物体遮蔽的侦测率也相当高，甚至只需要3个以上的SIFT物体特征就足以计算出位置与方位。在现今的电脑硬件速度下和小型的特征数据库条件下，辨识速度可接近即时运算。
SIFT特征的信息量大，适合在海量数据库中快速准确匹配。

SIFT特征检测主要包括以下4个基本步骤：

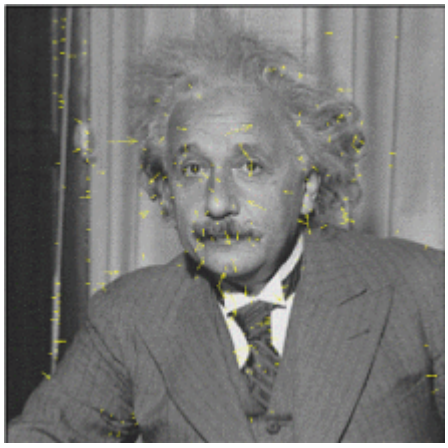
1.尺度空间极值检测：
通过构建尺度空间，检测极值点，获得尺度不变性: 搜索所有尺度上的图像位置。通过高斯微分函数来识别潜在的对于尺度和旋转不变的兴趣点。



2. 关键点定位:
   将特征点过滤并进行精确定位: 在每个候选的位置上，通过一个拟合精细的模型来确定位置和尺度。关键点的选择依据于它们的稳定程度。



3. 方向确定:
   为特征点分配方向值: 基于图像局部的梯度方向，分配给每个关键点位置一个或多个方向。所有后面的对图像数据的操作都相对于关键点的方向、尺度和位置进行变换，从而提供对于这些变换的不变性。



4. 关键点描述:
   生成特征描述子: 在每个关键点周围的邻域内，在选定的尺度上测量图像局部的梯度。这些梯度被变换成一种表示，这种表示允许比较大的局部形状的变形和光照变化。

opencv的xfeatures2d的sift.cpp库文件中，按照数学理论，创建了SIFT class,其参数和成员函数定义如下：

```
class SIFT_Impl : public SIFT
{
public:
    explicit SIFT_Impl( int nfeatures = 0, int nOctaveLayers = 3,
                        double contrastThreshold = 0.04, double
edgeThreshold = 10,
                        double sigma = 1.6);//初始化寻找特征点的权重和参数
//定义参数类型
    //! returns the descriptor size in floats (128)
    int descriptorSize() const CV_OVERRIDE;

    //! returns the descriptor type
    int descriptorType() const CV_OVERRIDE;

    //! returns the default norm type
    int defaultNorm() const CV_OVERRIDE;

    //! finds the keypoints and computes descriptors for them using SIFT
algorithm.
    //! Optionally it can compute descriptors for the user-provided keypoints
    //定义算法过程中需要的数据类型，如img，mask（掩膜），descriptors（描述
子）
    void detectAndCompute(InputArray img, InputArray mask,
                    std::vector<KeyPoint>& keypoints,
                    OutputArray descriptors,
                    bool useProvidedKeypoints = false) CV_OVERRIDE;

    void buildGaussianPyramid( const Mat& base, std::vector<Mat>& pyr, int
nOctaves ) const;  #高斯微分函数
    void buildDoGPyramid( const std::vector<Mat>& pyr, std::vector<Mat>&
dogpyr ) const;#尺度空间
    void findScaleSpaceExtrema( const std::vector<Mat>& gauss_pyr, const
std::vector<Mat>& dog_pyr, #极值点
                                std::vector<KeyPoint>& keypoints ) const;

//保护性继承各项权重
protected:
    CV_PROP_RW int nfeatures;
    CV_PROP_RW int nOctaveLayers;
    CV_PROP_RW double contrastThreshold;
    CV_PROP_RW double edgeThreshold;
```

```
    CV_PROP_RW double sigma;
};

//构造
Ptr<SIFT> SIFT::create( int _nfeatures, int _nOctaveLayers,
                        double _contrastThreshold, double _edgeThreshold, double
_sigma )
{
    return makePtr<SIFT_Impl>(_nfeatures, _nOctaveLayers, _contrastThreshold,
_edgeThreshold, _sigma);
}
```

SURF

简介

在SIFT中引入尺度不变的特征，主要的思想是每个检测到的特征点都伴随着对应的尺寸因子。当我们想匹配不同图像时，经常会遇到**图像尺度不同**的问题，不同图像中特征点的距离变得不同，物体变成不同的尺寸，如果我们通过修正特征点的大小，就会造成强度不匹配。为了解决这个问题，提出一个尺度不变的SURF特征检测，在**计算特征点的时候把尺度因素加入之中。**SURF与SIFT算法相似，SIFT算法比较稳定，检测特征点更多，但是复杂度较高，而SURF要运算简单，效率高，运算时间短一点。

可以说，SURF算法是对SIFT算法加强版，加速了具有鲁棒性（更稳定，健壮）的特征。该算法包括三个主要部分：兴趣点检测，局部邻域描述和匹配。

1.兴趣点检测：
SURF使用方形滤波器作为高斯平滑的近似

SURF使用基于海森矩阵(Hessian Matrix)的blob检测器来查找感兴趣的点。SIFT算法建立一幅图像的金字塔，在每一层进行高斯滤波并求取图像差(DOG)进行特征点的提取，而SURF则用的是海森矩阵进行特征点的提取，所以海森矩阵是SURF算法的核心。

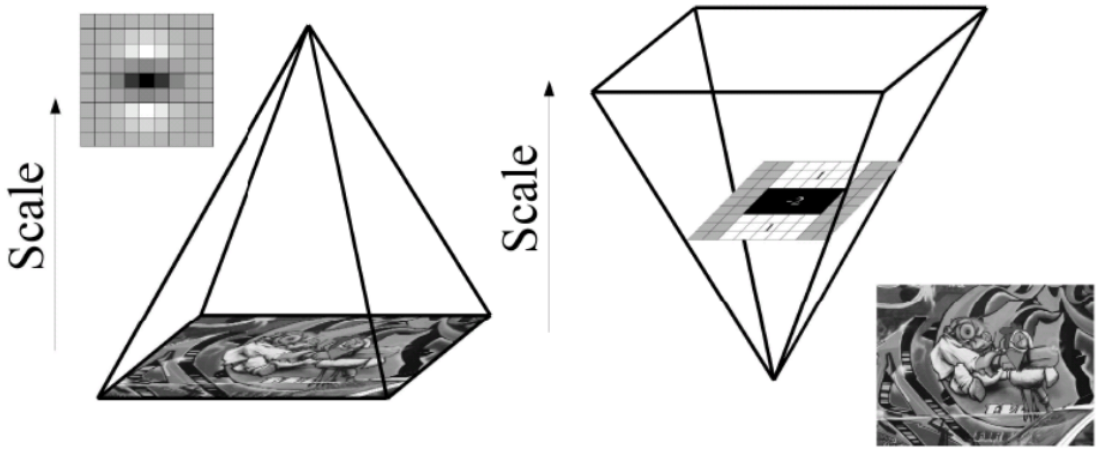
图像中某个像素点的Hessian Matrix可定义为：

其中，L（p，sigma）表示高斯函数二阶导数和图像I(x,y)在p处的反卷积

2.局部邻域描述：

图像的尺度空间是这幅图像在不同解析度下的表示。 在金字塔图像中分为很多层，每一层叫做一个octave，每一个octave中又有几张尺度不同的图片。在进行高斯模糊时，SIFT的高斯模板大小是始终不变的，只是在不同的octave之间改变图片的大小。而在SURF中，图片的大小是一直不变的，不同的octave层得到的待检测图片是改变高斯模糊尺寸大小得到

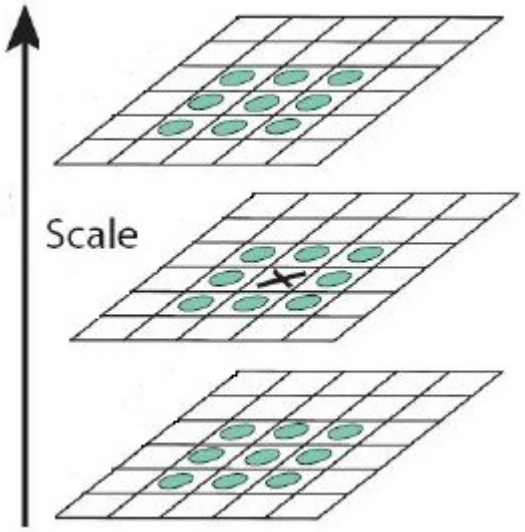
的，当然了，同一个octave中个的图片用到的高斯模板尺度也不同。
下图右边说明SURF算法使原始图像保持不变而只改变滤波器大小。SURF采用这种方法节省了降采样过程，其处理速度自然也就提上去了



传统描述子（SIFT）尺度空间保持filter不变，依赖上层结果，改变图像大小

SURF保持图像大小不变，改变filter的尺度大小，提高速度和精度

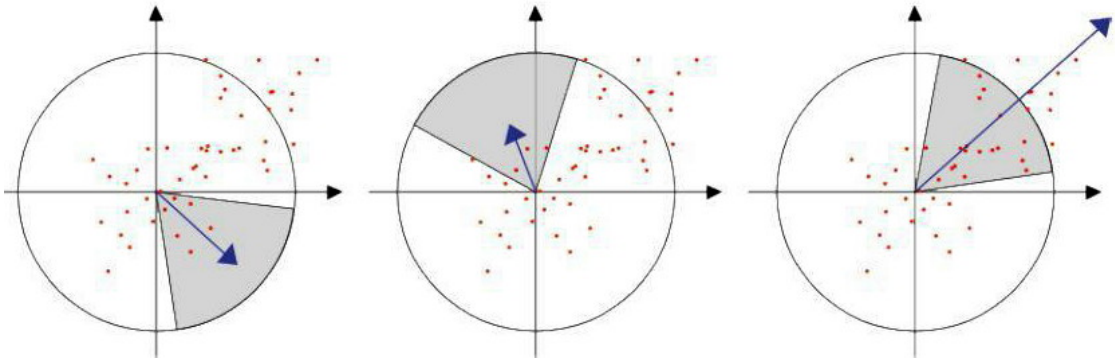
SURF算法继而在尺度层上利用非极大值抑制初步确定特征点和精确定位特征点。将经过hessian矩阵处理过的每个像素点与其3维领域的26个点进行大小比较，如果它是这26个点中的最大值或者最小值，则保留下来，当做初步的特征点。这26个点就如下图：



检测过程中使用与该尺度层图像解析度相对应大小的滤波器进行检测，以3×3的滤波器为例，该尺度层图像中9个像素点之一.如下图中检测特征点与自身尺度层中其余8个点和在其之上及之下的两个尺度层9个点进行比较，共26个点，图中标记‘x’的像素点的特征值若大于周围像素则可确定该点为该区域的特征点。然后，采用3维线性插值法得到亚像素级的特征点，同时也去掉那些值小于一定阈值的点，增加极值使检测到的特征点数量减少，最终只有几个特征最强点会被检测出来。

为了保证旋转不变性，在SURF中，不统计其梯度直方图，而是**统计特征点领域内的Harr小波特征。**即以特征点为中心，计算半径为6s(S为特征点所在的尺度值)的邻域内，统计60度扇形内所有点在x(水平)和y(垂直)方向的Haar小波响应总和(Haar小波边长取4s)，并给这些响应值赋高斯权重系数，使得靠近特征点的响应贡献大，而远离特征点的响应贡献小，然后60度范围内的响应相加以形成新的矢量，遍历整个圆形区域，选择**最长矢量的方向为该特征点的主方向。**这样，**通过特征点逐个进行计算，得到每一个特征点的主方向。**

通过该过程，**就相当于得到了具备主方向的特征点**，示意图如下：



opencv的xfeatures2d的surf.hpp库文件中，按照数学理论，创建了SURF class,其参数和成员函数定义如下：

```
class SURF_Impl : public SURF
{
public:
    //! the full constructor taking all the necessary parameters
    //相比较于SIFT，多了尺度空间相关的超参数
    explicit CV_WRAP SURF_Impl(double hessianThreshold,
                               int nOctaves = 4, int nOctaveLayers = 2,
                               bool extended = true, bool upright = false);

    //! returns the descriptor size in float's (64 or 128)
    CV_WRAP int descriptorSize() const CV_OVERRIDE;

    //! returns the descriptor type
    CV_WRAP int descriptorType() const CV_OVERRIDE;

    //! returns the descriptor type
    CV_WRAP int defaultNorm() const CV_OVERRIDE;

    void set(int, double);
    double get(int) const;

    //! finds the keypoints and computes their descriptors.
    // Optionally it can compute descriptors for the user-provided keypoints
    void detectAndCompute(InputArray img, InputArray mask,
                          CV_OUT std::vector<KeyPoint>& keypoints,
                          OutputArray descriptors,
                          bool useProvidedKeypoints = false) CV_OVERRIDE;

    void setHessianThreshold(double hessianThreshold_) CV_OVERRIDE {
hessianThreshold = hessianThreshold_; } //用于求解海森矩阵的阈值
    double getHessianThreshold() const CV_OVERRIDE { return hessianThreshold;
}
```

```
    void setNOctaves(int nOctaves_) CV_OVERRIDE { nOctaves = nOctaves_; }
    int getNOctaves() const CV_OVERRIDE { return nOctaves; }

    void setNOctaveLayers(int nOctaveLayers_) CV_OVERRIDE { nOctaveLayers =
nOctaveLayers_; }
    int getNOctaveLayers() const CV_OVERRIDE { return nOctaveLayers; }

    void setExtended(bool extended_) CV_OVERRIDE { extended = extended_; }
    bool getExtended() const CV_OVERRIDE { return extended; }

    void setUpright(bool upright_) CV_OVERRIDE { upright = upright_; }
    bool getUpright() const CV_OVERRIDE { return upright; }

    double hessianThreshold;
    int nOctaves;
    int nOctaveLayers;
    bool extended;
    bool upright;
};
```

ORB

简介

ORB（Oriented FAST and Rotated BRIEF）是Oriented FAST + Rotated BRIEF的缩写（感觉应该叫OFRB）。是目前最快速稳定的特征点检测和提取算法，许多图像拼接和目标追踪技术利用ORB特征进行实现。

- ORB = Oriented FAST（特征点） + Rotated BRIEF（特征描述）

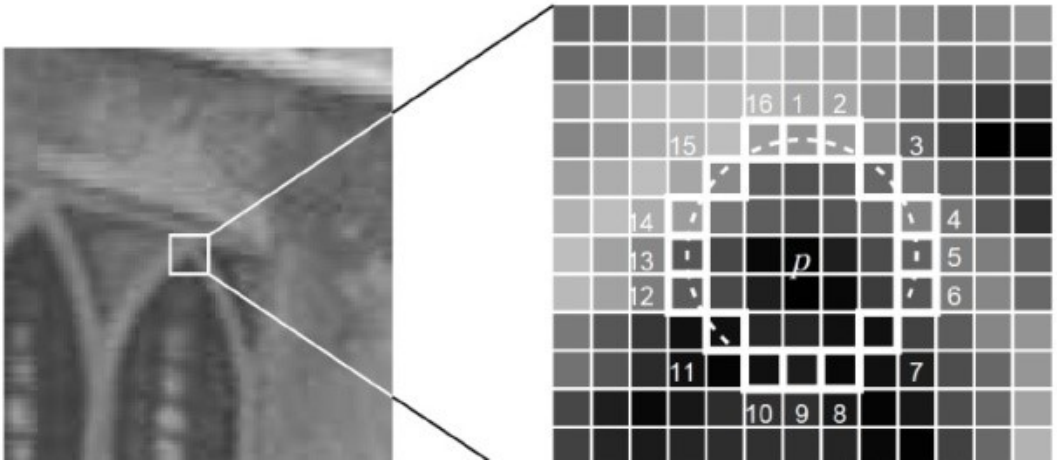
FAST

FAST，正如其名。它的出现就是为了**解决SIFT在建立特征点时速度慢的问题**。我们程序员都知道，要解Bug首先要做的是定位问题。

> SIFT进行特征点检测时需要建立尺度空间，基于局部图像的梯度直方图来计算描述子，整个算法的计算和数据存储复杂度比较高，不适用于处理实时性很强的图像。

FAST的解决方案：若某像素与其周围领域内足够多的像素点相差较大，则该像素可能是特征点

Step1: **确定候选角点**（Segment Test）



1. 选择某个像素 p ， 其像素值为 I_p 。
2. 以 p 为圆心，半径为3， 确立一个圆，圆上有16个像素，分别为 $p_1, p_2, \ldots, p_{16}$ 确定一个阈值： t (比如 Ip 的 20%)。
3. 让圆上的像素的像素值分别与 p 的像素值做差，如果存在连续n个点满足 或 $I_x - Ip < -t$ （其中 I_x 代表此圆上16个像素中的一个点），那么就把该点作为一个候选点。根据经验，一般令n=12(n 通常取 12，即为 FAST-12。其它常用的 N 取值为 9 和 11， 他们分别被称为 FAST-9，FAST-11).

Step2:**非极大值抑制**（Non-maximum suppression）

经过Step 1的海选后，还是会有很多个特征点。好在他们有个缺点：很可能大部分检测出来的点彼此之间相邻，我们要去除一部分这样的点。为了解决这一问题，可以采用非最大值抑制的算法：假设P，Q两个点相邻，分别计算两个点与其周围的16个像素点之间的差分和为V。去除V值较小的点，即把非最大的角点抑制掉。经过上述两步后FAST特征值筛选的结果就结束了。

Oriented FAST

FAST算法的缺点：无法体现出一个优良特征点的尺度不变性和旋转不变性。

Fast角点本不具有方向，由于特征点匹配需要，ORB对Fast角点进行了改进，改进后的FAST 被称为 Oriented FAST，具有**旋转和尺度**的描述:

尺度不变性：用金字塔解决:

1. 对图像做不同尺度的高斯模糊
2. 对图像做降采样(隔点采样)
3. 对每层金字塔做FAST特征点检测
4. n幅不同比例的图像提取特征点总和作为这幅图像的oFAST特征点。

旋转不变性：用质心标定方向解决:

$$m_{pq} = \sum_{x,y} x^p y^q I(x,y)$$

1. 在一个小的图像块 B 中，定义图像块的矩。

2. 通过矩可以找到图像块的质心 $$C = \left(\frac{m_{10}}{m_{00}}, \frac{m_{01}}{m_{00}} \right)$$
3. 连接图像块的几何中心 O 与质心 C，得到一个方向向量 $\overrightarrow{OC}$ ，这就是特征点的方向 $\theta = \operatorname{atan2}\left(m_{01}, m_{10}\right)$

BRIEFRIEF

BRIEFRIEF是2010年的一篇名为《BRIEF:Binary Robust Independent Elementary Features》的文章中提出，**BRIEF是对已检测到的特征点进行描述，它是一种二进制编码的描述子**，摈弃了利用区域灰度直方图描述特征点的传统方法，采用**二级制、位异或运算**，大大的加快了特征描述符建立的速度，同时也极大的降低了特征匹配的时间，是一种非常快速，很有潜力的算法。如果说FAST用来解决寻找特征点的速度问题，那么BRIEF就用来解决描述子的空间占用冗余问题。

如果说FAST用来解决寻找特征点的速度问题，那么**BRIEF就用来解决描述子的空间占用冗余问题**:

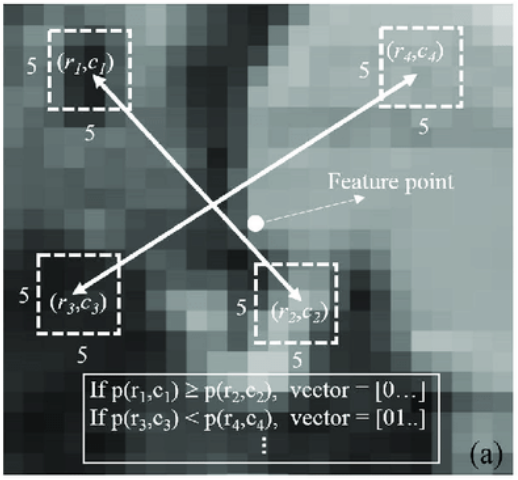
1. 为减少噪声干扰，先对图像进行高斯滤波（方差为2，高斯窗口为9x9）
2. 以特征点为中心，取SxS的邻域窗口。在窗口内随机选取一对（两个）点，比较二者像素的大小，进行如下二进制赋值：
   $$\tau(\mathbf{p};\mathbf{x},\mathbf{y}) := \begin{cases} 1 & : \mathbf{p}(\mathbf{x}) < \mathbf{p}(\mathbf{y}) \\ 0 & : \mathbf{p}(\mathbf{x}) \geq \mathbf{p}(\mathbf{y}) \end{cases}$$
   其中，p(x)，p(y)分别是随机点x=(u1,v1),y=(u2,v2)的像素值。
3. 在窗口中随机选取N对随机点，重复步骤2的二进制赋值，形成一个**二进制编码**，这个编码就是**对特征点的描述，即特征描述子。**（一般N=256）这个特征可以由n位二进制测试向量表示，BRIEF描述子：
   $$f_n(\mathbf{p}) := \sum_{1 \leq i \leq n} 2^{i-1} \tau\left(\mathbf{p};\mathbf{x}_i,\mathbf{y}_i\right)$$

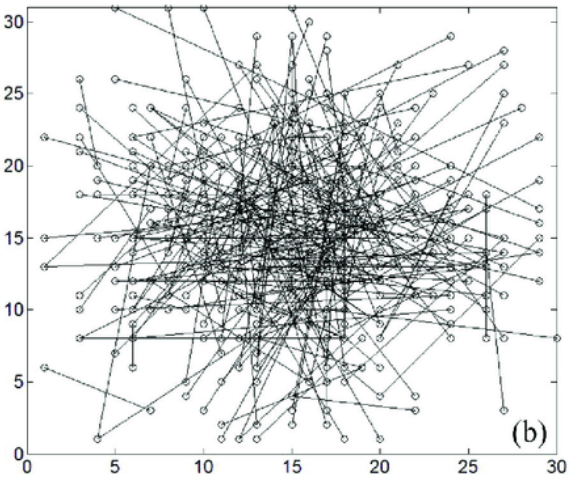
这里面，最关键的地方其实是随机特征对的选取，论文中就给出了5种方法（其中第二种比较好），分别为：

1. $\boldsymbol{x}_i, \boldsymbol{y}_i$ 都呈均匀分布 $\left[-\frac{S}{2}, \frac{S}{2}\right]$
2. ； $\boldsymbol{x}_i, \boldsymbol{y}_i$ 都呈高斯分布 $\left[0, \frac{1}{25} S^2\right]$ ，准则采样服从各向同性的同一高斯分布；
3. $\boldsymbol{x}_i$ 服从高斯分布 $\left[0, \frac{1}{25} S^2\right]$ ， $\boldsymbol{y}_i$ 服从高斯分布 ，采样分两步进行：首先在原点处为 $\boldsymbol{x}_i$ 进行高斯采样，然后在中心为 $\boldsymbol{x}_i$ 处为 $\boldsymbol{y}_i$ 进行高斯采样；
4. $\boldsymbol{x}_i, \boldsymbol{y}_i$ 在空间量化极坐标下的离散位置处进行随机采样；

5. $x_i = (0,0)^T, y_i$ 在空间量化极坐标下的离散位置处进行随机采样；

通过这些方法，可以生成类似下图中的描述子：





Rotated BRIEF

描述子是用来描述一个特征点的属性的，除了标记特征点之外，它最重要的一个功能就是要实现特征点匹配。对于BRIEF算法，其通过Hamming距离来实现特征点匹配。

```
#以d（x,y）表示两个字x,y之间的汉明距离。对两个字符串进行异或运算，并统计结果为1的个数，那么这个数就是汉明距离。
def hammingDistance(s1, s2):
    """Return the Hamming distance between equal-length sequences"""
    if len(s1) != len(s2):
        raise ValueError("Undefined for sequences of unequal length")
    return sum(el1 != el2 for el1, el2 in zip(s1, s2))
```

所以，对于BRIEF来说，描述子里不包含旋转属性，所以一旦匹配图片有稍微大点的旋转角度，按照Hamming算法，匹配度将会大幅下降。

因此，orb中优化的一个关键，就是**为BRIEF增加旋转不变性**得到Rotated BRIEF：

1、对于任意特征点，在31x31邻域内位置为 (x_i, y_i) 的n对点集，可以用2 x n的矩阵来表示：

$$S = \begin{pmatrix} x_1, \cdots, x_n \\ y_1, \cdots, y_n \end{pmatrix}$$

2、利用FAST求出的特征点的主方向 θ 和对应的旋转矩阵 R_θ ，算出旋转的 S_θ 来代表

$$S\,\theta = \operatorname{atan}2\,(m_{01}, m_{10})$$

3、计算旋转描述子（steered BRIEF）：

$$R_\theta = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

其中 $S_\theta = R_\theta S$ 为BRIEF的描述子。

$$g_n(p,\theta) := f_n(p) | (x_i, y_i) \in S_\theta$$

总结ORB的两个核心：

1. FAST是用来寻找特征点的。ORB在FAST基础上通过金字塔、质心标定解决了尺度不变和旋转不变。即oFAST。
2. BRIEF是用来构造描述子的。ORB在BRIEF基础上通过引入oFAST的旋转角度和机器学习解决了旋转特性和特征点难以区分的问题。即rBRIEF.

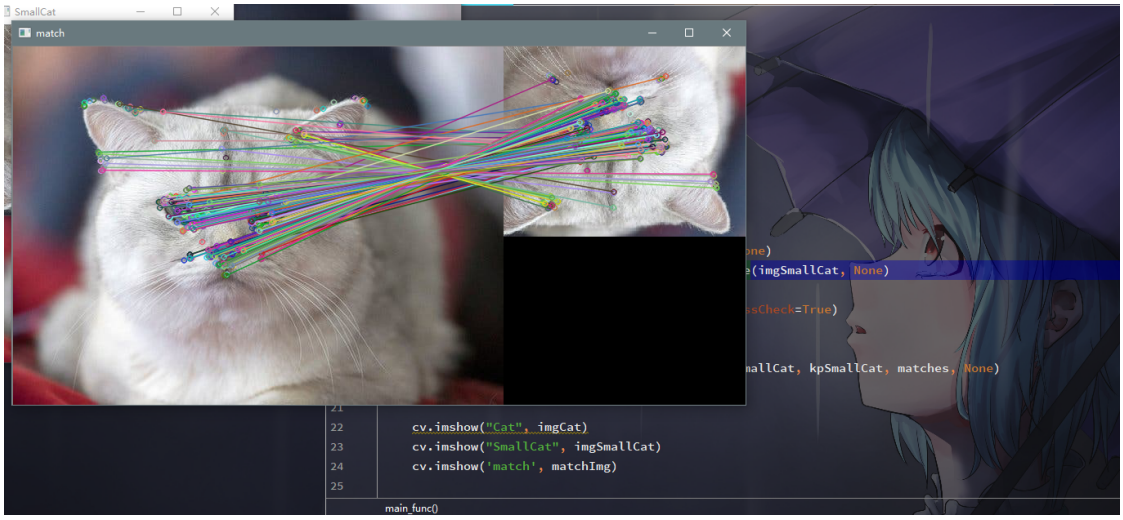
通过cv2中的orbDetector可以实现orb特征提取

```
import sys
import cv2 as cv
import numpy as np


def main_func(argv):
    imgCat = cv.imread("cat.jpg")
    imgSmallCat = cv.imread("smcat.png")

    orb = cv.ORB_create()

    kpCat, desCat = orb.detectAndCompute(imgCat,None)
    kpSmallCat, desSmallCat = orb.detectAndCompute(imgSmallCat, None)

    bf = cv.BFMatcher_create(cv.NORM_HAMMING, crossCheck=True)

    matches = bf.match(desCat, desSmallCat)
    matchImg = cv.drawMatches(imgCat, kpCat, imgSmallCat, kpSmallCat,
matches, None)


    cv.imshow("Cat", imgCat)
    cv.imshow("SmallCat", imgSmallCat)
    cv.imshow('match', matchImg)

    cv.waitKey(0)


if __name__ == '__main__':
    main_func(sys.argv)
```

c++样例代码如下：

```
#include <iostream>
#include <string>
#include <opencv2/opencv.hpp>

using namespace std;
using namespace cv;

int main(int *argv, int **argc)
{

	Mat imgCat = imread("cat.png");
	Mat imgSmallCat = imread("smallCat.png");

	auto orbDetector = ORB::create();
	vector<KeyPoint> kpCat, kpSmallCat;
	Mat descriptorCat, descriptorSmallCat;

	orbDetector->detectAndCompute(imgCat, Mat(), kpCat, descriptorCat);
	orbDetector->detectAndCompute(imgSmallCat, Mat(), kpSmallCat,
descriptorSmallCat);

	Ptr<DescriptorMatcher> matcher =
DescriptorMatcher::create(DescriptorMatcher::BRUTEFORCE);
	std::vector<DMatch> matchers;
	matcher->match(descriptorCat, descriptorSmallCat, matchers);

	Mat imgMatches;
	drawMatches(imgCat, kpCat, imgSmallCat, kpSmallCat, matchers,
imgMatches);

	imshow("Cat", imgCat);
	imshow("SmallCat", imgSmallCat);
	imshow("Match", imgMatches);

	waitKey(0);

	return true;
}
```

特征提取效果如下：



寻找图与图之间的对应相机旋转角度以及相机平移

```
def find_transform(K, p1, p2):  #发现旋转角
    focal_length = 0.5 * (K[0, 0] + K[1, 1])
    principle_point = (K[0, 2], K[1, 2])
    E,mask = cv2.findEssentialMat(p1, p2, focal_length, principle_point,
cv2.RANSAC, 0.999, 1.0) #通过findEssentialMat寻找本质矩阵
    cameraMatrix = np.array([[focal_length, 0, principle_point[0]], [0,
focal_length, principle_point[1]], [0, 0, 1]]) #相机内参矩阵
    pass_count, R, T, mask = cv2.recoverPose(E, p1, p2, cameraMatrix, mask)
    #cv::recoverPose()函数主要是用来从本质矩阵中恢复R,tR,tR,t
    return R, T, mask

def get_matched_points(p1, p2, matches): #匹配点

    src_pts = np.asarray([p1[m.queryIdx].pt for m in matches])
    dst_pts = np.asarray([p2[m.trainIdx].pt for m in matches])

    return src_pts, dst_pts

def get_matched_colors(c1, c2, matches): #匹配颜色

    color_src_pts = np.asarray([c1[m.queryIdx] for m in matches])
    color_dst_pts = np.asarray([c2[m.trainIdx] for m in matches])
```

```
    return color_src_pts, color_dst_pts
```

findEssentialMat

```
Mat cv::findEssentialMat (      InputArray      points1,
InputArray      points2,
InputArray      cameraMatrix,
int     method = RANSAC,
double  prob = 0.999,
double  threshold = 1.0,
OutputArray     mask = noArray()
)
Python:
retval, mask    =       cv.findEssentialMat(    points1, points2,
cameraMatrix[, method[, prob[, threshold[, mask]]]]       )
retval, mask    =       cv.findEssentialMat(    points1, points2[, focal[,
pp[, method[, prob[, threshold[, mask]]]]]]       )
```

- 功能 从两幅图像中的相应点计算一个基本矩阵
- 参数 points1第一幅图片的N个二维像素点. 点坐标应该是浮点（单精度或双精度）
- 参数 points2 第二幅图片的二维像素点，与points1同样大小和类型
- 参数cameraMatrix相机矩阵，请注意，此功能假设points1和points2是具有相同摄像机矩阵的摄像机的特征点。
- 参数method计算本征矩阵的方法
  RANSAC 用于RANSAC算法。
  LMedS 用于LMedS算法。

默认RANSAC方法

- 参数prob概率仅用于RANSAC或LMedS方法的参数。 它规定了估计矩阵正确的可信度（概率）
- 参数threshold用于RANSAC的参数。 它是从点到极线的最大距离（以像素为单位），超出此点时，该点被视为异常值，不用于计算最终的基本矩阵。 根据点定位精度，图像分辨率和图像噪声的不同，可将其设置为1-3。
- 参数mask输出N个元素的数组，其中每个元素对于异常值设置为0，对其他点设置为1。 该数组仅在RANSAC和LMedS方法中计算。
- 函数返回值为计算出的本质矩阵，可进一步传递给decomposeEssentialMat或recoverPose以恢复相机之间的相对位置。

使用样例:

```
import numpy as np
import cv2
#pt1和pt2
pOld = np.array(
   [[334.48077, 111.08635],
    [826.19525, 352.7404 ],
    [797.13354, 521.27057],
    [615.0971,  656.2975 ],
    [845.188,   173.10873]])
pNew = np.array(
    [[394.36942, 131.2731 ],
     [782.77637, 380.04907],
     [741.9934,  551.30444],
     [584.73315, 679.83984],
     [771.2071,  202.27649]])
#相机
K = np.array( \
    [[1.2112729e+03, 0.0000000e+00, 6.3218433e+02], \
     [0.0000000e+00, 1.2152592e+03, 3.4675201e+02], \
     [0.0000000e+00, 0.0000000e+00, 1.0000000e+00]])

E, mask = cv2.findEssentialMat(pOld, pNew, K, method=cv2.RANSAC,
threshold=1.5, prob=0.99)
'''
Result:

E:
[[-0.00627753 -0.37505678 -0.07194276]
 [ 0.35912238  0.00781432 -0.60564749]
 [ 0.07145943  0.59419955  0.00994249]
 [ 0.34932297 -0.21108792 -0.41808301]
 [-0.02788446 -0.38781955  0.48859358]
 [ 0.27005355 -0.44238482 -0.00913568]
 [ 0.00870689  0.05265274 -0.08390127]
 [ 0.01960241  0.00364806 -0.70174221]
 [ 0.08495865  0.69993091  0.0123091 ]
 [ 0.3671368  -0.0009191  -0.52399762]
 [-0.12578615 -0.3897754  -0.35040367]
 [ 0.37598167  0.39838416  0.00585678]]

Mask:
[[1]
 [1]
 [1]
```

```
 [1]
 [1]]
```

RANSAC与LMedS

都属于**线性估计**方法，常用的线性参数估计算法有LS、WLS、Ransac LS、
LMedS（Ransac的使用并不局限于线性模型，LMedS的思想也可以扩展到非线性模型）

线性估计的数学模型为：$y = \beta_0 + \beta_1 x + u.$
其中，y是被解释变量（因变量）；x是解释变量（自变量）；B0和B1是待估计的参数。

Ransac是一种随机参数估计算法。Ransac LS从样本中随机抽选出一个样本子集，使用LS对这个子集计算模型参数，然后计算所有样本与该模型的偏差，再使用一个预先设定好的阈值与偏差比较，当偏差小于阈值时，该样本点属于模型内样本点（inliers），否则为模型外样本点（outliers），记录下当前的inliers的个数，然后重复这一过程。每一次重复，都记录当前最佳的模型参数，所谓最佳，既是inliers的个数最多，此时对应的inliers个数为best_ninliers。每次迭代的末尾，都会根据期望的误差率、best_ninliers、总样本个数、当前迭代次数，计算一个迭代结束评判因子，据此决定是否迭代结束。迭代结束后，最佳模型参数就是最终的模型参数估计值。

LMedS也是一种随机参数估计算法。LMedS也从样本中随机抽选出一个样本子集，使用LS对子集计算模型参数，然后计算所有样本与该模型的偏差。但是与Ransac LS不同的是，LMedS记录的是所有样本中，偏差值居中的那个样本的偏差，称为Med偏差（这也是LMedS中Med的由来），以及本次计算得到的模型参数。由于这一变化，LMedS不需要预先设定阈值来区分inliers和outliers。重复前面的过程N次，从中N个Med偏差中挑选出最小的一个，其对应的模型参数就是最终的模型参数估计值。其中迭代次数N是由样本集子中样本的个数、期望的模型误差、事先估计的样本中outliers的比例所决定。

recoverPose

> Recover relative camera rotation and translation from an estimated essential matrix and the corresponding points in two images, using cheirality check. Returns the number of inliers which pass the check.

通常搭配findEssentialMat，在其返回的基本矩阵estimated essential matrix

```
int recoverPose( InputArray E, InputArray points1, InputArray points2,
                 InputArray cameraMatrix, OutputArray R, OutputArray t,
                 double distanceThresh, InputOutputArray mask = noArray(),
                 OutputArray triangulatedPoints = noArray());
```

E：已经求解出来的本质矩阵，它是3x3的矩阵；

points1：第一张图片中的点；

points2：第二张图片中的点；

cameraMatrix：相机内参矩阵，它是3x3的矩阵；

R：求解出来的两帧图片之间的旋转矩阵；

t：求解出来的两帧图片之间的平移向量；

focal：相机焦距；

pp：像素坐标的原点；

distanceThresh：点的距离阈值，用来滤出距离较远的点；

triangulatedPoints：通过三角化还原点；

注意：

1. 通过该函数求解出来的R,tR,tR,t，它表示的是points2到points1的变换

2.该函数求解出来的R,tR,tR,t已经是最合适的R,tR,tR,t，已经通过内部的代码去掉了另外三种错误的解

3. cv::recoverPose()中points1和points2的输入顺序，必须也要和求本质矩阵时对函数cv::findEssentialMat()输入的顺序相同