

# 全景图拼接

## Lab 7

姓名：周炜

学号：32010103790

### 全景图拼接

柱面变化

特征匹配

SIFT算法

1. 通过高斯卷积构建图像金字塔
2. 构建高斯差分金字塔
3. 关键点定位
4. 关键点方向赋值
5. 关键点描述

### 全景图拼接

### 实验原理

椭圆变换

特征点检测

特征点匹配

计算变换

全景拼接

# 全景图拼接

全景图拼接是一项技术，通过对同一场景的多张图像进行重叠并寻找其匹配关系，从而生成整个场景的图像。这项技术包括多种方法，根据场景和相机运动的类型可分为单视点全景拼接和多视点全景拼接。

对于平面场景和相机只进行旋转的情况，可以采用计算每两幅图像之间的Homography变换来将它们映射到同一平面上的方法。此外，还可以通过恢复相机的旋转来得到最终的全景图。当相机固定只在水平方向旋转时，也可以利用柱面或球面坐标映射的方法来生成全景图。

## 柱面变化

对于每一幅图像来说，我们都可以把它们投影到一个柱面上，得到柱面上的图像。柱面图像的坐标变换为：

$$x' = r \tan^{-1} \left( \frac{x}{f} \right)$$

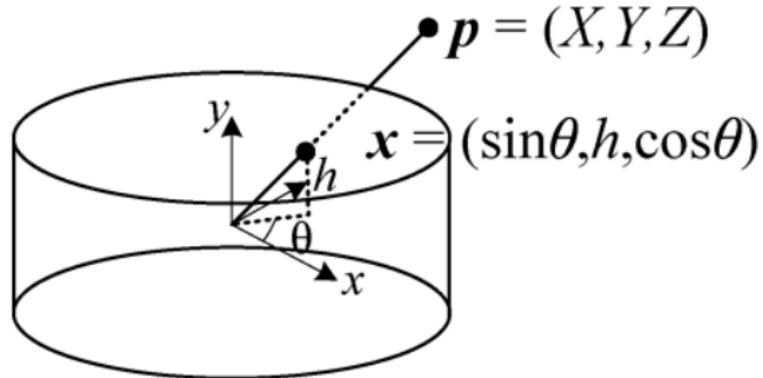
$$y' = \frac{ry}{\sqrt{x^2 + f^2}}$$

其中 $(x', y')$ 为柱面上的坐标， $(x, y)$ 为平面图像坐标，其坐标原点都已移至图像中心， $r$ 为柱面半径， $f$ 为焦距。

$$x = f \tan\left(\frac{x'}{r}\right)$$

$$y = \frac{y'}{r} \sqrt{x^2 + f^2}$$

然而为了得到柱面投影图像，我们往往需要将柱面图像上的点逆变换到平面图像上的对应像素点，进行插值，得到完整的柱面图像，逆变换的变换公式为：



## 特征匹配

对每两幅相邻的柱面图像进行特征提取和匹配（特征可以选用SIFT、ORB等，可以使用OpenCV的函数实现），寻找两幅相邻图像的对应关系。

## SIFT算法

SIFT(Scale-invariant feature transform)又称尺度不变特征转换，此算法由David Lowe在1999年所发表，2004年完善总结。SIFT主要是用来提取图像中的关键点。相比于其它角点检测算法(如Harris和shitomis)，SIFT算法具有角度和尺度不变性，换句话说就是不容易受到图像平移、旋转、缩放和噪声的影响。

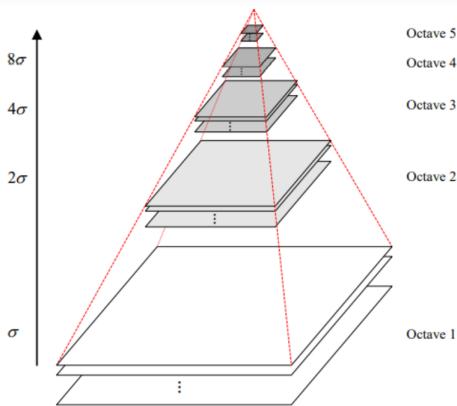
### 1. 通过高斯卷积构建图像金字塔

高斯核是唯一可以产生多尺度空间的核。因此使用高斯函数和原图像进行卷积，具体公式如下图所示：

$$G(x_i, y_i, \sigma) = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{(x - x_i)^2 + (y - y_i)^2}{2\sigma^2}\right)$$

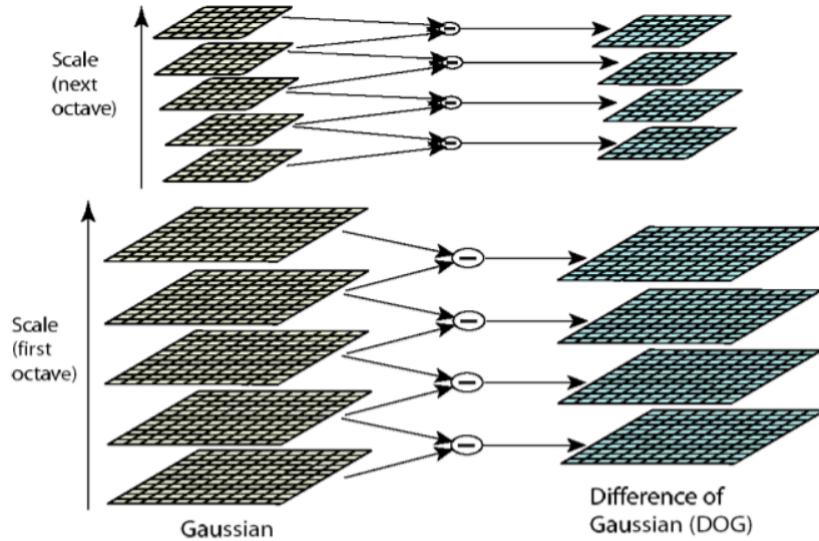
$$L(x, y, \sigma) = G(x, y, \sigma) * I(x, y)$$

用高斯卷积就可以构建出不同尺度空间的图像，这样就形成了一组图像。之后，再对这些图像进行降采样，这样就形成**图像金字塔**：



## 2. 构建高斯差分金字塔

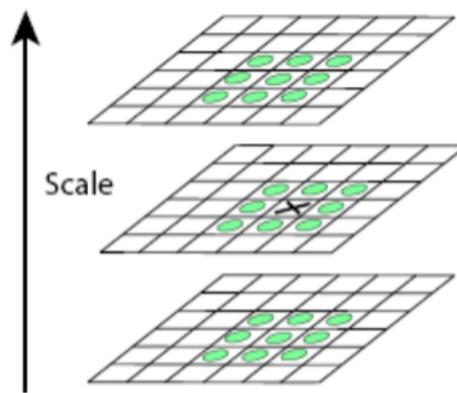
创建好图像高斯金字塔后，每一组内的相邻层相减可以得到高斯差分金字塔(DoG, Difference of Gaussian)，如下图所示：



由于DOG是通过相邻层相减得到，因此层数会比高斯金字塔少一层

## 3. 关键点定位

得到DOG之后，就可以在正数第二层和倒数第二层的范围内寻找极值点(第一层和最后一层无法和相邻两层进行比较)。此时将某个点和周围26个点进行比较，比较的示意图如下图所示，图中x为比较的点，这幅图中x首先和相邻的8个点比较，然后和上下两个尺度的 $9 \times 2 = 18$ 个点进行比较，总共需要比较26个点。如果该点符合极小值或极大值，则此点为离散空间中的极值点。



离散空间中的极值点并不是真正的极值点，因此通过离散值插值的方式，可以找到真正的极值点

## 4.关键点方向赋值

通过尺度不变性求极值点，可以使其具有缩放不变的性质；为了让其具有图像旋转不变性，需要对每个关键点方向进行赋值。每一个像素点的梯度方向和幅值计算公式如下：

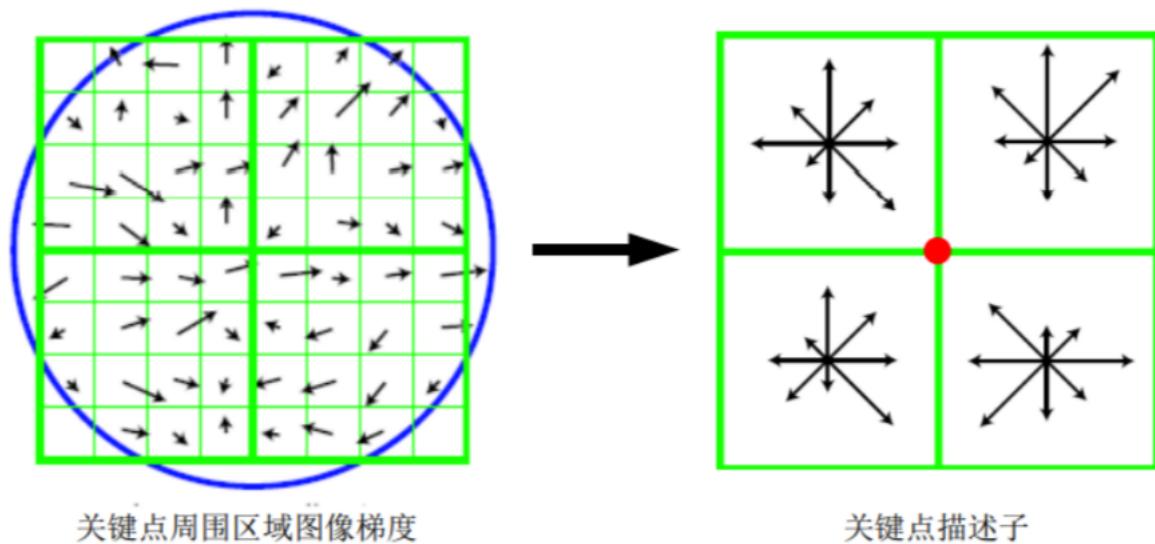
$$m(x, y) = \sqrt{(L(x+1, y) - L(x-1, y))^2 + (L(x, y+1) - L(x, y-1))^2}$$
$$\theta(x, y) = \tan^{-1}((L(x, y+1) - L(x, y-1)) / (L(x+1, y) - L(x-1, y)))$$

然而直接计算一个点的方向可能会存在误差，因此选取关键点附近的一块领域，对领域内每一个点的方向进行统计。将0-360度分成8个方向，每45度为一个方向，形成8个方向的柱状图，峰值代表关键点方向，大于峰值80%的作为辅方向。

## 5.关键点描述

关键点描述的目的是在关键点计算后，用一组向量将这个关键点描述出来，用来作为目标匹配的依据。

SIFT采用 $4 \times 4 \times 8$ 共128维向量作为特征点，取特征点周围 $8 \times 8$ 的像素范围进行梯度方向统计和高斯加权(蓝色圆圈表示加权范围)，每 $4 \times 4$ 的窗口生成8个方向。箭头方向代表了像素梯度方向，箭头长度代表该像素的幅值。每个 $4 \times 4$ 的窗口形成一个种子点，一个特征点由4个种子点的信息所组成。



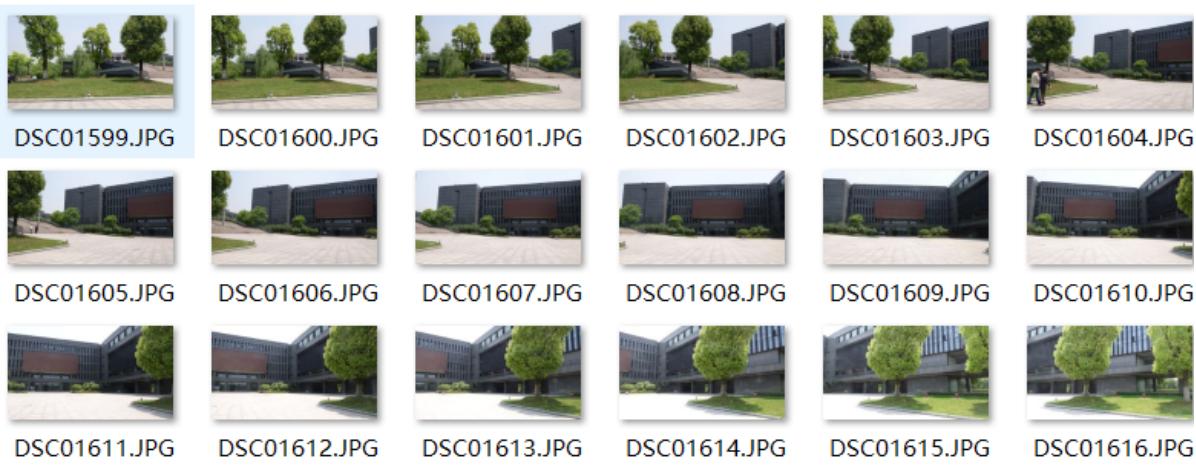
## 全景图拼接

使用上一步得到的匹配关系，求出每两幅柱面图像的一个平移变换，利用平移变换将所有图像拼接到一起。得到一幅全景图

本次实验有2组数据需要拼接，如下(DSC前缀一般是相机拍摄的)：



以及



观察后开源发现相机型号和拍摄时间，都是在临近时间拍摄的，应该是同一时间连拍的结果

### 照相机

照相机制造商 SONY  
 照相机型号 FDR-AX100E  
 光圈值 f/11  
 曝光时间 1/1000 秒  
 ISO 速度 ISO-800  
 曝光补偿 0 档光圈  
 焦距 12 毫米  
 最大光圈 3.265625  
 测光模式 图案  
 目标距离  
 闪光灯模式 无闪光，强制  
 闪光灯能量  
35mm 焦距

## 实验原理

### 椭圆变换

在全景图拼接中,第一步是将输入的平面图像投影到一个公共的柱面坐标系上。这是通过 `projectToCylinder` 函数实现的。

```
void projectToCylinder(Mat const& img, Mat& img_cyl, double f) {
    int width = img.cols, height = img.rows;
    double r = f;
    double cyl_width = r * atan(width / 2.0 / f), cyl_height = r * height / 2.0 / f;
    int cyl_cols = ceil(2 * cyl_width), cyl_rows = ceil(2 * cyl_height);

    img_cyl = Mat::zeros(cyl_rows, cyl_cols, CV_8UC3);

    for (int i = 0; i < cyl_rows; i++) {
        double y = (i - cyl_height) / r;
        for (int j = 0; j < cyl_cols; j++) {
            double x = f * tan((j - cyl_width) / r);
            ...
        }
    }
}
```

```
    double src_x = x + width / 2.0;
    double src_y = y * sqrt(f * f + x * x) + height / 2.0;
    if (src_x >= 0 && src_x < width && src_y >= 0 && src_y < height) {
        img_cyl.at<Vec3b>(i, j) = bilinearInterpolation(img, src_x,
src_y);
    }
}
}
}
```

该函数接受三个参数:输入图像 `img`、输出的柱面投影图像 `img_cyl` 和焦距 `f`。函数首先根据输入图像的尺寸和焦距计算出柱面投影图像的尺寸。然后,通过遍历柱面投影图像的每个像素,计算出其在原始平面图像中对应的坐标,并使用双线性插值的方法获取该坐标处的像素值,赋值给柱面投影图像的对应像素。

通过这一步,我们将所有输入的平面图像都投影到了一个公共的柱面坐标系上,为后续的特征点检测和匹配做好了准备。



## 特征点检测

在柱面投影图像上,我们需要检测出一些特征点,用于后续的图像匹配和拼接。

我使用了SIFT算法在 `merge` 函数中,我们首先创建了一个 `SIFT` 特征检测器,然后分别对两个待拼接的图像进行特征点检测和描述子提取。`detectAndCompute` 函数用于检测图像中的特征点,并计算出每个特征点的描述子。

## 特征点匹配

我通过CV自带的 `BFMatcher` 实现特征匹配

在 `merge` 函数中, 我们创建了一个 `BFMatcher` 对象, 然后使用 `knnMatch` 函数对两个图像的描述子进行匹配。这里使用了 k 近邻匹配策略, 对于每个描述子, 返回其最近的两个匹配点。

当两幅图像的SIFT特征向量生成以后, 就可以采用关键点描述子的欧式距离来作为两幅图像中关键点的相似性判定度量。取图像1中的某个关键点, 并找出其与图像2中欧式距离最近的前两个关键点, 在这两个关键点中, 如果最近的距离除以次近的距离小于预先设置的阈值`DistRatio`, 则接受这一对匹配点。降低阈值`DistRatio`, SIFT匹配点数目会减少, 但更加稳定。Lowe推荐的阈值为0.8。但对大量任意存在尺度、旋转和亮度变化的两幅图片进行匹配后, 结果表明`DistRatio`取值在0.4~0.6之间最佳, 小于0.4很少有匹配点, 大于0.6则存在大量错误匹配点。博主参考网络`DistRatio`的取值原则如下:

`DistRatio=0.4` 对于准确度要求高的匹配;  
`DistRatio=0.6` 对于匹配点数目要求比较多的匹配;  
`DistRatio=0.5` 一般情况下。

我这里选择了只保留距离比小于 0.5 的匹配点, 以提高匹配的准确性。

## 计算变换

有了特征点的对应关系, 我们就可以计算出两个图像之间的变换矩阵, 直接调用 `findHomography`

在 `merge` 函数中, 我们将匹配的特征点对应关系传入 `findHomography` 函数, 该函数使用 RANSAC 算法来计算出最优的单应性矩阵 `H`。如果 `H` 矩阵为空, 说明匹配的特征点不足以计算出可靠的变换矩阵, 此时返回 `false`。

```
Mat H = findHomography(pts2, pts1, RANSAC);
if (H.empty()) {
    return false;
}
```

## 全景拼接

有了变换矩阵, 我们就可以进行最终的图像拼接了。这是在 `merge` 函数的后半部分实现的。

首先, 我们使用 `warpPerspective` 函数将图像 `img2` 根据变换矩阵 `H` 进行透视变换, 得到 `img2_warp`。然后, 我们创建两个掩码 `mask` 和 `mask_out`, 分别表示 `img2_warp` 的有效区域和无效区域。

接下来, 我们将 `img1` 和 `img2_warp` 分别拷贝到 `img_out1` 和 `img_out2` 中, 并使用掩码 `mask_out` 和 `mask` 将它们拷贝到最终的输出图像 `img_out` 中。

最后, 对于重叠区域, 我们使用线性插值的方法来平滑过渡, 避免拼接处出现明显的边缘。

通过这一步, 我们完成了两个图像的拼接, 得到了最终的全景图像。

在 `makePanorama` 函数中, 我们对多个输入图像依次进行柱面投影和拼接, 最终得到完整的全景图像

```
Mat img2_warp;
warpPerspective(img2, img2_warp, H, img1.size());

Mat mask = Mat::zeros(img2_warp.size(), CV_8U);
for (int i = 0; i < img2_warp.rows; i++) {
    for (int j = 0; j < img2_warp.cols; j++) {
        if (img2_warp.at<Vec3b>(i, j) != Vec3b(0, 0, 0)) {
            mask.at<uchar>(i, j) = 255;
        }
    }
}
```

```

}

Mat img_out1, img_out2;
img1.copyTo(img_out1);
img2_warp.copyTo(img_out2, mask);

Mat mask_out;
bitwise_not(mask, mask_out);
img_out1.copyTo(img_out, mask_out);
img_out2.copyTo(img_out, mask);

Rect range1 = getValidRange(img_out1);
Rect range2 = getValidRange(img_out2);
if (range1.x > range2.x) {
    swap(range1, range2);
}

double alpha = 1.0 / (range1.x + range1.width - range2.x);
for (int i = range2.y; i < range2.y + range2.height; i++) {
    for (int j = range2.x; j < range1.x + range1.width; j++) {
        if (mask.at<uchar>(i, j) > 0 && mask_out.at<uchar>(i, j) > 0) {
            double weight = (j - range2.x) * alpha;
            img_out.at<Vec3b>(i, j) = (1 - weight) * img_out1.at<Vec3b>(i, j) +
weight * img_out2.at<Vec3b>(i, j);
        }
    }
}
return true;

```

我刚开始使用ORB算法。ORB是一种快速的特征检测器和描述符提取器,是OpenCV默认提供的,不需要额外的模块。但是效果很差,大致如下。在网上查验后发现ORB, AKAZE算法会比SIFT差一些,因此最后改用SIFT



最后结果为

data1为



data2为

