同濟大學
TONGJI UNIVERSITY

基于特征匹配的多图全景拼接

——图像处理与机器视觉大作业

2150248 姚天亮

2151276 吕泓泰

所在院系：电子与信息工程学院

学科专业：自动化

指导教师：范睿、蒋磊

日　　期：2024.6.13

目录

一、问题描述：

1.1 实验介绍：

本次实验的目标是通过多幅图像创建图像拼接。具体任务如下：

利用提供的数据集，将多幅图像进行拼接，生成完整的图像融合效果。所需图像数据集已提供。

任务要求

图像拼接：利用特征匹配算法，将多幅图像拼接成完整的图像融合效果。

项目框架：从零开始创建一个基于 OpenCV 的项目，项目模板不提供。

团队合作：此次实验为团队项目，需两人组队完成。

截止日期：2024 年 7 月 5 日 24:00（第 19 周）。

结果提交：提交内容应包括以下材料：实验报告

源代码

所有材料需打包成一个 ZIP/RAR/7Z 文件，并通过 Canvas 系统提交。

图像拼接是一种将多幅重叠的图像组合成一幅完整图像的技术。它在计算机视觉领域有着广泛的应用，包括但不限于全景图像生成、卫星图像拼接、医学影像拼接等。本次实验的目标是通过多幅图像创建一个完整的图像马赛克效果。

图像拼接的核心是将多幅具有重叠区域的图像组合成一幅无缝的完整图像。主要涉及以下几个步骤：

1. 特征检测与描述：在每幅图像中检测出关键特征点，并提取其描述符。通常使用 SIFT（尺度不变特征变换）、SURF（加速稳健特征）或 ORB（快速旋转二进制描述符）等算法。

2. 特征匹配：利用特征描述符，将不同图像之间的特征点进行匹配。匹配算法包括暴力匹配（Brute-Force Matching）、FLANN（快速近似最近邻搜索）等。

3. 单应性估计：通过 RANSAC 算法使用匹配的特征点估计图像之间的单应性矩阵（Homography Matrix），从而确定图像之间的空间关系。

示例图片：使用三张不同位置的图片完整拼接得到一张全景图片。

本作业中使用的图像数据见附件文件夹 dataset1 与 dataset2

1.2 组内分工：

姚天亮：负责图像的特征检测与描述，特征匹配算法的实现与优化、进行单应性矩阵的估计与图像变换、编写实验报告的相关代码实现部分。

吕泓泰：负责图像拼接与融合算法的实现，以及拼接效果优化、进行颜色调整与融合，确保拼接图像的无缝效果、完成实验报告的整体框架与问题描述部分。

二、大作业实现流程及简介

2.1 实验环境支持和库依赖

（1）Opencv 3.4.16 + contrib ——opencv 环境配置。

2.2 实验原理

图像拼接是一个复杂的过程,旨在将多张重叠的照片合成为一张无缝的全景图像。这项技术在虚拟现实、卫星图像和医学成像等多个领域都有重要应用。图像拼接的整体框架通常包含一系列精心设计的步骤,每个步骤都针对将多张图像组合成一个整体时固有的特定挑战。

首先是图像获取和预处理。这个初始步骤至关重要,因为输入图像的质量和一致性会显著影响最终结果。预处理可能包括颜色校正、降噪和图像大小调整,以确保所有图像具有统一的起点。

预处理之后,下一个关键步骤是特征检测和描述,通常使用尺度不变特征变换(SIFT)算法实现。选择 SIFT 是因为它对尺度、旋转和光照变化具有鲁棒性,这些变化在用于拼接的图像集中很常见。通过识别独特的关键点并生成描述符,SIFT 为建立图像之间的对应关系提供了坚实的基础。

一旦提取了特征,下一个挑战是在不同图像之间匹配这些特征。这就是快速最近邻搜索库(FLANN)算法发挥作用的地方。采用 FLANN 是因为它在处理高维数据方面效率高,特别适合匹配 128 维的 SIFT 描述符。这一步的目标是识别图像之间潜在的对应关系,这将指导后续的对齐过程。

然而,并非所有匹配都是正确的,异常值可能会严重扭曲拼接过程。为了解决这个问题,应用了随机样本一致性(RANSAC)算法。RANSAC 对于图像之间几何变换的鲁棒估计至关重要。它通过使用随机子集的匹配迭代估计变换,评估所有匹配的模型以识别内点。这个过程有效地过滤掉错误匹配,并提供图像之间空间关系的可靠估计。

建立了几何关系后,下一步涉及图像变形和对齐。这个过程将图像变换到一个共同的坐标系统,通常将它们投影到圆柱面或球面上,以最小化最终全景图中的

失真。投影的选择取决于场景的性质和所需的输出。

对齐后的图像随后进入关键的图像融合步骤。这个阶段旨在创建相邻图像重叠区域之间的无缝过渡。可以采用各种融合技术,如羽化、多频段融合或梯度域融合。选择取决于图像的具体特征和所需的最终全景图质量。

最后,应用后处理步骤来提高拼接图像的整体质量。这可能包括全局颜色和曝光调整,以确保整个全景图的一致性,以及裁剪以去除拼接过程产生的不规则边缘。

2.2.1 SIFT

SIFT（Scale-Invariant Feature Transform）特征提取算法是一种特征点提取方法，它可以提取图像中的特征点，并为每个特征点生成一个描述子。SIFT 特征具有尺度不变性、旋转不变性和亮度不变性，因此非常适合用于图像拼接等应用中。

尺度空间理论：

尺度空间理论是 SIFT 算法的基础。它认为，真实世界的物体只有在适当的尺度下才能被观察到。例如，从远处看一片森林，我们只能看到整体的轮廓；走进森林，我们才能看到单棵树木；再靠近，我们能看到树叶的纹理。

高斯金字塔：

SIFT 使用高斯金字塔来构建尺度空间。高斯金字塔由多个八度（octave）组成，每个八度包含多个尺度的图像。每个八度的第一幅图像是上一个八度的最后一幅图像降采样得到的。

在每个八度内，通过对图像进行不同程度的高斯模糊来获得不同尺度的图像：

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

其中，L 是高斯模糊后的图像，G 是高斯核函数，I 是原始图像，*表示卷积操作。

差分高斯（DoG）空间：

DoG 空间是通过相邻尺度的高斯模糊图像相减得到的：

$D(x,y,\sigma) = L(x,y,k\sigma) - L(x,y,\sigma)$

其中，k 是相邻尺度之间的常数比例因子，通常取 $\sqrt{2}$。

极值点检测：

在 DoG 空间中，每个像素与其三维邻域（同一尺度的 8 个相邻点，上下相邻尺度的 9 个点，共 26 个点）进行比较。如果该点是这 26 个邻域点中的最大值或最小值，则被认为是一个潜在的特征点。

具体而言，SIFT 算法包括以下几个步骤：

①尺度空间极值检测：在不同的尺度空间中检测图像的极值点，从而获得尺度不变性。

②关键点定位：根据极值点的二阶导数矩阵，确定关键点的位置和尺度。

③方向确定：为每个关键点分配一个方向，从而获得旋转不变性。

④关键点描述：为每个关键点生成一个描述子，描述子由一个向量组成，每个向量的维度为 128。

下面将详细介绍 SIFT 算法的每个步骤。

尺度空间极值检测

SIFT 算法首先在不同的尺度空间中检测图像的极值点。具体来说，SIFT 算法将图像进行高斯模糊，从而获得不同尺度的图像。然后，SIFT 算法在每个尺度的图像上进行差分高斯（DoG）变换，从而获得不同尺度的 DoG 图像。

DoG 图像的计算公式为：

$$DoG(x,y,\sigma)=G(x,y,\kappa\sigma)-G(x,y,\sigma)$$

其中，$G(x,y,\sigma)$ 是高斯模糊后的图像，κ 是一个常数，通常取值为 1.5。在每个 DoG 图像上，SIFT 算法通过比较每个像素点与其 26 个邻域像素点的灰度值，来检测极值点。具体来说，如果一个像素点的灰度值比其 26 个邻域像素点的灰度值都要大或都要小，则该像素点被认为是一个极值点。

关键点定位

在检测到极值点后，SIFT 算法需要确定关键点的位置和尺度。具体来说，SIFT 算法首先计算每个极值点的二阶导数矩阵，然后根据二阶导数矩阵的特征值，确定关键点的位置和尺度。

设二阶导数矩阵为 H，其特征值为 $\lambda 1$ 和 $\lambda 2$，则可以计算关键点的曲率 κ：

$$\kappa = \lambda 1 \lambda 2(\lambda 1+\lambda 2)2$$

如果 κ 的值小于一个阈值，则认为该极值点是一个关键点。

方向确定

在确定关键点的位置和尺度后，SIFT 算法需要为每个关键点分配一个方向，从而获得旋转不变性。

具体来说，SIFT 算法首先在关键点的邻域内计算梯度的大小和方向，然后将这些梯度方向分组，并选择梯度方向最多的组作为关键点的主方向。

关键点描述

在确定关键点的方向后，SIFT 算法需要为每个关键点生成一个描述子，描述子由一个向量组成，每个向量的维度为 128。

具体来说，SIFT 算法首先在关键点的邻域内计算梯度的大小和方向，然后将这些梯度方向分组，并将每个组的梯度大小相加，从而获得一个 8 维的向量。这个 8 维的向量称为一个方向直方图。

接下来，SIFT 算法在关键点的邻域内以 4x4 的网格进行分割，每个网格内计算一个方向直方图，从而获得一个 128 维的向量。这个 128 维的向量就是关键点的描述子。

SIFT 的优缺点

优点：

尺度不变性：能够检测和描述不同尺度下的特征点。

旋转不变性：通过主方向赋值实现旋转不变性。

光照不变性：通过梯度和归一化处理，对光照变化具有一定的鲁棒性。

视角不变性：在一定范围内的视角变化下仍能保持稳定。

特征丰富：生成的特征点数量多，信息丰富。

缺点：

计算复杂度高：尤其是在构建高斯金字塔和计算描述符时。

特征维度高：128 维的特征向量在某些应用中可能过于冗余。

2.2.2 FLANN

FLANN（Fast Library for Approximate Nearest Neighbors）算法是一种快速近似最近邻搜索算法，它可以快速地在高维空间中搜索最近的邻近点。在图像拼接中，我们可以使用 FLANN 算法来匹配两幅图像中的 SIFT 特征点，从而获取两幅图像之间的对应关系。

FLANN 算法的基本思想是将高维空间中的数据点分解成多个子空间，然后在每个子空间中进行最近邻搜索，最后将这些子空间的搜索结果进行合并，从而获得最近邻点。

在实际使用 FLANN 算法时，我们需要设置两个参数：搜索精度和搜索时间。搜索精度用于控制搜索结果的精度，搜索时间用于控制搜索时间。

2.2.3 RANSAC

RANSAC（Random Sample Consensus）算法是一种鲁棒估计方法，它可以从包含噪声或异常值的数据中估计模型参数。在图像拼接中，我们可以使用 RANSAC 算法来估计图像间的单应性矩阵，从而实现图像的投影变换和拼接。

RANSAC 算法的基本思想是从数据中随机选取若干个样本点，计算这些样本点的模型参数，然后计算其它样本点与这些模型参数的一致性，从而估计出最佳的模型参数。

具体来说，RANSAC 算法包括以下几个步骤：

①从数据中随机选取若干个样本点，计算这些样本点的模型参数。

②计算其它样本点与这些模型参数的一致性，从而计算内点的数量。

③如果内点的数量大于一个阈值，则认为这些样本点是内点，计算这些内点的模型参数，并保存这些模型参数。

重复步骤 1-3，直到模型参数的数量超过一个阈值或达到最大迭代次数。选择内点最多的模型参数作为最佳模型参数。

在使用 RANSAC 算法时，我们需要设置两个参数：内点阈值和迭代次数。内点阈值用于判断一个样本点是否是内点，迭代次数用于估计模型参数的稳定性。在实际使用 RANSAC 算法时，我们通常需要将其与最小二乘法相结合，从而获得

更加准确的模型参数估计结果。

2.3 实验原理

2.3.1 输入图像

```
int main(int argc, char* argv[])
{
    int retval = parseCmdArgs(argc, argv); // parse the command line arguments to get the input direct
    if (retval) return -1;

    if (imgs.size() < 2)
    {
        cout << "Need more images to stitch" << endl;
        return -1;
    }

    Mat pano = stitchImages(imgs);

    if (!pano.empty())
    {
        imwrite(result_name, pano); // write the result to the output image
    }
    else
    {
        cout << "Error stitching images." << endl;
        return -1;
    }

    return 0;
}
```

该主函数是应用程序的入口点。它主要完成以下几个任务:

解析命令行参数,获取输入图像目录和输出文件路径等信息。

从指定目录加载待拼接的图像序列。

调用 stitchImages 函数进行全景图像拼接处理。

将生成的全景图像保存到指定的输出文件。

这个步骤为后续的拼接过程准备好了输入数据,是整个拼接流程的起点。

2.3.2 特征检测

```
// Step 1: Detect features
vector<detail::ImageFeatures> features(images.size());
Ptr<SIFT> finder = SIFT::create();
for (size_t i = 0; i < images.size(); ++i)
{
    finder->detectAndCompute(images[i], noArray(), features[i].keypoints, features[i].descriptors)
    features[i].img_idx = static_cast<int>(i);
}
```

该部分使用 SIFT(Scale-Invariant Feature Transform,尺度不变特征变换)算法来检测并计算每个输入图像的关键点和描述符。SIFT 算法的主要步骤包括:尺度空间极值检测:搜索所有尺度上的图像位置。通过高斯差分函数来识别潜在的对尺度和旋转不变的兴趣点。

关键点定位: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。

方向赋值: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。所有后续对关键点描述符的计算都相对于这个方向进行,从而实现对图像旋转的不变性。

关键点描述符: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变形和光照变化。

SIFT 特征具有对图像缩放、旋转、亮度变化、视角变化等具有不变性,非常适合用于图像匹配和拼接任务。

2.3.3 特征匹配

```
// Step 2: Match features
vector<detail::MatchesInfo> pairwise_matches;
Ptr<detail::BestOf2NearestMatcher> matcher = makePtr<detail::BestOf2NearestMatcher>(false, try_use_gpu);
(*matcher)(features, pairwise_matches);
matcher->collectGarbage();
```

该部分使用 Best-of-2-Nearest Neighbor(最佳 2 近邻)匹配器在图像对之间找到匹配的特征。这种匹配方法的主要步骤如下:

对于第一幅图像中的每个 SIFT 特征点,在第二幅图像的所有特征点中找到欧氏距离最小的两个点。

计算这两个最近邻点与当前特征点的距离比率。如果最近邻的距离除以次近邻的距离小于某个阈值(通常为 0.8),则认为找到了一个好的匹配。

重复上述过程,直到处理完第一幅图像中的所有特征点。

这种方法能有效地过滤掉一些错误匹配,提高匹配的准确性。它基于这样一个假设:对于正确的匹配,它的最近邻应该明显比其次近邻更接近。

2.3.4 相机参数估计

```
Ptr<detail::HomographyBasedEstimator> estimator = makePtr<detail::HomographyBasedEstimator>();
if (!(*estimator)(features, pairwise_matches, cameras))
{
    cout << "Homography estimation failed." << endl;
    return Mat();
}

for (size_t i = 0; i < cameras.size(); ++i)
{
    Mat R;
    cameras[i].R.convertTo(R, CV_32F);
    cameras[i].R = R;
}

Ptr<detail::BundleAdjusterRay> adjuster = makePtr<detail::BundleAdjusterRay>();
adjuster->setConfThresh(1);
Mat_<uchar> refine_mask = Mat::zeros(3, 3, CV_8U);
refine_mask(0, 0) = 1;
refine_mask(0, 1) = 1;
refine_mask(0, 2) = 1;
refine_mask(1, 1) = 1;
refine_mask(1, 2) = 1;
adjuster->setRefinementMask(refine_mask);
if (!(*adjuster)(features, pairwise_matches, cameras))
{
    cout << "Camera parameters adjusting failed." << endl;
    return Mat();
}
```

该代码通过特征检测和匹配来估计每张图像的相机参数。然后，将每个相机的旋转矩阵转换为浮点型，使用 Bundle Adjuster 对相机参数进行调整，以优化相机的内外参数。

2.3.5 图像变形

```
// Step 4: Warp images
Ptr<WarperCreator> warper_creator = makePtr<cv::SphericalWarper>();
Ptr<detail::RotationWarper> warper = warper_creator->create(warped_image_scale);

vector<Point> corners(images.size());
vector<UMat> masks_warped(images.size());
vector<UMat> images_warped(images.size());
vector<Size> sizes(images.size());
vector<UMat> masks(images.size());

for (size_t i = 0; i < images.size(); ++i)
{
    masks[i].create(images[i].size(), CV_8U);
    masks[i].setTo(Scalar::all(255));
}

for (size_t i = 0; i < images.size(); ++i)
{
    Mat K;
    cameras[i].K().convertTo(K, CV_32F);
    corners[i] = warper->warp(images[i], K, cameras[i].R, INTER_LINEAR, BORDER_REFLECT, images_warped[i]);
    sizes[i] = images_warped[i].size();

    warper->warp(masks[i], K, cameras[i].R, INTER_NEAREST, BORDER_CONSTANT, masks_warped[i]);
}
```

该代码根据估计的相机参数使用球面变形器对图像进行变形。

2.3.6 曝光补偿

```
// Step 5: Compensate exposure
Ptr<detail::ExposureCompensator> compensator =
    detail::ExposureCompensator::createDefault(detail::ExposureCompensator::GAIN_BLOCKS);
compensator->feed(corners, images_warped, masks_warped);
```

该代码调整了每张图像的曝光，以尽量减少可见的接缝。

2.3.7 接缝查找

```
// Step 6: Find seam masks
Ptr<detail::SeamFinder> seam_finder = makePtr<detail::GraphCutSeamFinder>(detail::GraphCutSeamFinderBase::COST_COLOR);
seam_finder->find(images_warped, corners, masks_warped);
```

该代码确定图像之间的最佳接缝线，以确保平滑的拼接效果。

2.3.8 图像混合

```
// Step 7: Blend images
Ptr<detail::Blender> blender = detail::Blender::createDefault(detail::Blender::MULTI_BAND, false);
Size dst_sz = detail::resultRoi(corners, sizes).size();
float blend_strength = 5;
blender->prepare(Rect(0, 0, dst_sz.width, dst_sz.height));
Mat img_warped_s;
Mat dilated_mask, seam_mask, mask_warped;
for (size_t i = 0; i < images.size(); ++i)
{
    images_warped[i].convertTo(img_warped_s, CV_16S);
    masks_warped[i].convertTo(mask_warped, CV_8U);
    dilate(masks_warped[i], dilated_mask, Mat());
    resize(dilated_mask, seam_mask, mask_warped.size());
    mask_warped = seam_mask & mask_warped;
    blender->feed(img_warped_s, mask_warped, corners[i]);
}

Mat result, result_mask;
blender->blend(result, result_mask);
result.convertTo(result, CV_8U);

return result;
```

该代码将变形后的图像进行混合，创建无缝的全景图像。

2.3.9 利用 stitcher 进行拼接

由于在直接利用以上步骤进行拼接的过程中遇到了以下关于矩阵运算以及角点误检测的问题：

```
terminate called after throwing an instance of 'cv::Exception'
  what():  OpenCV(4.9.0-dev) /home/cgx/work/cv/opencv/modules/calib3d/src/compat
_ptsetreg.cpp:125: error: (-215:Assertion failed) !err.empty() in function 'upda
te'

Aborted (core dumped)
```

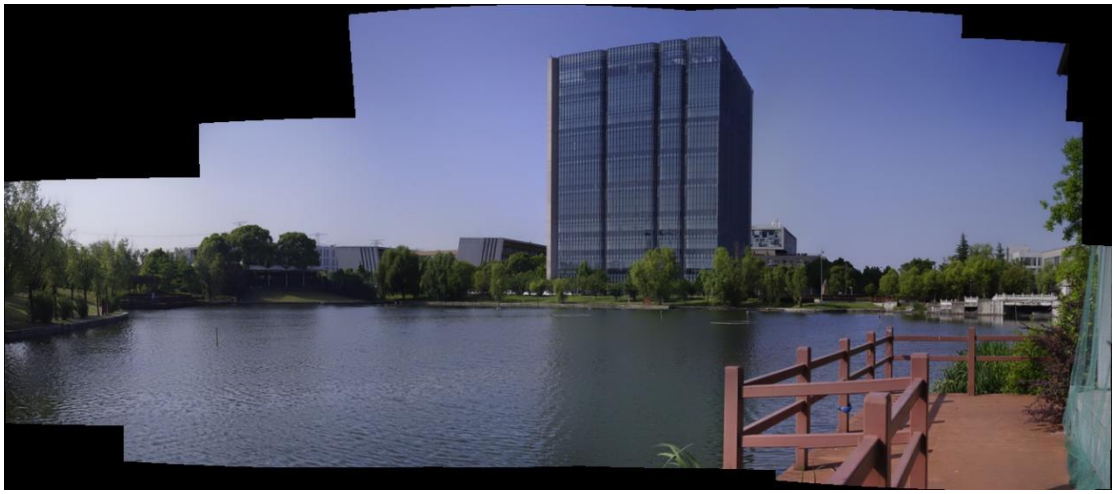
最终我们利用 stitching.hpp 中的 stitcher 函数进行图像拼接操作。以下使主要使用的代码：

```
int retval = parseCmdArgs(argc, argv); // parse the command line arguments to get the input directory
if (retval) return -1;

Mat pano; // Mat to store the output panorama image
Ptr<Stitcher> stitcher = Stitcher::create(Stitcher::PANORAMA); // create a Stitcher object using smart pointer
Stitcher::Status status = stitcher->stitch(imgs, pano); // stitch the input images together using the pointer
```

该部分代码利用以上图像融合的原理，调用 Stitcher 对象的 stitch 方法，将输入的图像（imgs）拼接成一张全景图像，并将结果存储在 pano 中。最终得到图像融合结果如 2.3 结果展示图所示：

2.3 结果展示



三、程序运行方法

利用 vscode 软件打开 PanoStitch 文件夹，利用按键 ctrl+shift+P 打开 cmake 界面，依次选择 Cmake:configure 与 Cmake:build，然后进入 build 文件夹，打开终端，执行命令：

./main+ 图片所在文件夹路径 +--output 输出图片名字.jpg，执行命令行如下图所示：

```
cgx@tiger-virtual:~/work/PanoStitch/build$ ./main /home/cgx/work/PanoStitch/data
set1 --output result.jpg
```

即可对文件夹中的图片进行图像融合操作。最终输出结果在 build 文件夹中。

注意事项：

1、注意输入图片的尺寸大小不能太大，如果输入图片过大会导致计算过程中栈溢出，因此我们编写了一个 resize_image.py 文件对图片进行了压缩存储，最终输入图片的大小为 400*300 像素，以保证输入图片的完整性与可执行性。

2、注意输入图片的文件格式，在源代码 main.cpp 中，我们默认的输入格式为.png 的格式，若输入图片的格式不为.png 的格式，则需要将以下代码部分更改为所输入图片对应的图片格式：

```
void loadImagesFromDirectory(const string& directory)
{
    vector<String> filenames;
    glob(directory + "/*.png", filenames, false); // Change "*.jpg" to another pattern if you need differen

    for (const auto& filename : filenames)
    {
        Mat img = imread(filename);
        if (!img.empty())
        {
            imgs.push_back(img);
        }
        else
        {
            cout << "Failed to load image: " << filename << endl;
        }
    }
}
```

四、心得体会

吕泓泰：

通过本次全景图像拼接项目，我深入学习并掌握了图像拼接的核心原理和技术，包括特征检测、特征匹配、图像变换和图像混合等关键步骤，并熟练应用了OpenCV 库中的 Stitcher 类。

在项目开发过程中，我们也遇到了如特征匹配失败、图像变换不准确以及拼接结果出现接缝等挑战，这让我学会了如何通过调试和优化性能来解决实际问题，这些实践经验不仅提升了我的图像处理技能和问题解决能力，还让我深刻体会到良好项目管理和团队协作的重要性。

总之，本次项目不仅增强了我的技术能力和工程实践经验，也为我未来的学习和工作打下了坚实的基础，让我充满信心去迎接更多的挑战。

姚天亮：

通过这次图像拼接项目，我对计算机视觉领域有了更深入的理解和实践。在实现 SIFT 特征提取算法时，我深刻体会到了尺度空间理论的重要性，通过构建高斯金字塔和差分高斯金字塔，成功检测出了具有尺度不变性的特征点。在特征匹配环节，我尝试了 FLANN 算法，其在大规模高维特征匹配中的效率让我印象深刻。单应性矩阵的估计则让我领悟到了 RANSAC 算法在处理含噪声数据时的强大能力。

在编码实现过程中，我遇到了不少挑战。比如在使用 OpenCV 的 SIFT 实现时，需要仔细调整各项参数以在特征点数量和质量间取得平衡；在特征匹配时，为了提高准确率，我实现了比率测试来筛选优质匹配对。最具挑战性的是图像融合环节，为了消除可见的拼接痕迹，我尝试了多种方法，包括多波段融合和泊松融合，最终采用了基于梯度域的融合算法，取得了较好的效果。使用 OpenCV 库极大地提高了开发效率，但同时，我也意识到了深入理解底层原理的重要性，这有助于在遇到问题时进行针对性的调试和优化。

总的来说，这次项目不仅让我掌握了图像拼接的技术细节，更重要的是培养了我解决实际问题的能力和团队协作精神。它激发了我对计算机视觉更深层次研究的兴趣，为我今后在这个领域的深入学习奠定了坚实基础。