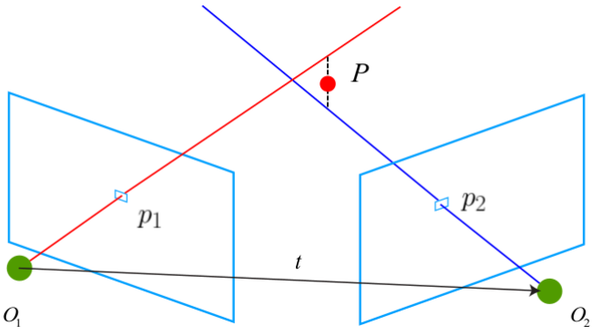
SLAM 课程项目文档

学号	小组成员
2253744	林觉凯
2153085	马立欣
2151422	武芷朵
2253215	刘垚
2250397	秦成

1. 三角化估计深度

在单目 SLAM 中，我们仅通过单张图像是无法获得像素的深度信息，这就需要通过三角测量（Triangulation）（三角化）的方法来估计地图点的深度。即从多个视角观测一个点，通过几何的方法来求出该点到相机的实际距离。这是视觉 SLAM 中将图像点转换为世界坐标最核心的一步。具体的三角化深度估计图示如下：



考虑图像 I_1 和 I_2，以作图为参考，右图的变换矩阵为 T。相机光心为 O_1 和 O_2。在 I_1 中有特征点 p_1，对应 I_2 中有特征点 p_2。理论上直线 O_1p_1 与 O_2p_2 在场景中会相交于一点 P。该点即两个特征点所对应的地图点在三维场景中的位置。由于噪声的影响，这两条直线往往无法相交。因此，通过最小二乘求解。

按照对极几何中的定义，设 x_1, x_2 为两个特征点的归一化坐标，那么它们满足：

$$s_1 x_1 = s_2 R x_2 + t$$
$$s_2 x_2 = s_1 R x_1 + t$$

通过对极约束知道了 R, t，想要求解的是两个特征点的深度 s_1, s_2。从几何上来看，可以在射线 O_1p_1 上寻找 3D 点，使其投影位置接近 p_2，同理，也可以在 O_1p_1 上找，或者在两条线的中间线。当然这两个深度是可以分开求的，例如，我们希望先求 s_2 ，那么对上式左乘一个 $x_1^\wedge$；我们希望先求 s_1 ，那么对上式左乘一个 $x_2^\wedge$，得：

$$s_1 x_1^\wedge x_1 = 0 = s_2 x_1^\wedge R x_2 + x_1^\wedge t$$
$$s_2 x_2^\wedge x_2 = 0 = s_1 x_2^\wedge R x_1 + x_2^\wedge t$$

该式左侧为零，右侧可看成 s_2 的一个方程，可以根据它直接求得 s_2。有了 s_2，s_1也非常容易求出。于是，得到了两帧下的点的深度，确定了它们的空间坐标。由于噪声的存在，估得的 R, t 不一定精确使上式为零，所以常见的做法是求最小二乘解而不是零解。

2. 项目介绍

2.1. 项目目录结构

todo

2.2. 实验环境搭建

- **硬件与运行环境**

  本项目在远程服务器环境中完成实验与模型部署，具体软硬件配置如下：

  - **软件环境（镜像配置）**
    - 操作系统：Ubuntu 20.04
    - Python 版本：3.8
    - 深度学习框架：PyTorch 1.11.0
    - CUDA 版本：11.3
  - **硬件资源**
    - **GPU**：NVIDIA RTX 3090（24GB 显存）× 1
    - **CPU**：Intel(R) Xeon(R) Gold 6330，14 核 vCPU，主频 2.00GHz
    - **内存**：60GB
  - **硬盘存储**：
    - 系统盘：30GB
    - 数据盘：50GB SSD

  该服务器环境为本项目中的 SLAM 算法训练与测试提供了充足的计算资源，确保了大规模数据处理与GPU加速下的实时性能评估。

- **C++ 环境**

本项目的基础部分基于C++语言进行开发与实现，为确保三种特征匹配算法在Linux环境下顺利编译运行，需完成一系列基础依赖配置与环境搭建工作。

- **基础开发环境**

  1. 安装编译工具链

     ```
     sudo apt-get update
     sudo apt-get install -y build-essential cmake git
     ```

  2. 安装OpenCV依赖项，为后续编译OpenCV准备必要的依赖库

     ```
     sudo apt-get install -y \
         libgtk2.0-dev \
         pkg-config \
         libavcodec-dev \
         libavformat-dev \
         libswscale-dev \
         libtbb2 \
         libtbb-dev \
         libjpeg-dev \
         libpng-dev \
         libtiff-dev \
         libdc1394-22-dev
     ```

- **安装OpenCV（含contrib模块）**

  由于项目中使用了SIFT和SURF特征检测器，这些都在OpenCV contrib模块中，所以必须编译安装OpenCV contrib。

  1. 下载OpenCV源码与contrib模块

     ```
     git clone https://github.com/opencv/opencv.git
     git clone https://github.com/opencv/opencv_contrib.git
     ```

  2. 编译并安装OpenCV

     ```
     cd ~/opencv
     mkdir build && cd build

     cmake -D CMAKE_BUILD_TYPE=Release \
           -D CMAKE_INSTALL_PREFIX=/usr/local \
           -D OPENCV_EXTRA_MODULES_PATH=~/opencv_contrib/modules \
           -D OPENCV_ENABLE_NONFREE=ON \
           -D BUILD_EXAMPLES=ON ..

     make -j$(nproc)
     sudo make install
     sudo ldconfig
     ```

  3. 验证安装

     ```
     # 验证OpenCV安装
     pkg-config --modversion opencv4

     # 检查是否包含xfeatures2d模块
     opencv_version -v
     ```

     注意这里推荐使用 OpenCV 的版本为 4.2.0，因为它默认集成了 SIFT 和 SURF 特征检测器（在 `opencv_contrib` 模块中）OpenCV 4.2.0 版本相对稳定，且支持同时启用 `SIFT` 和 `SURF`，因此项目推荐使用这个版本。

- **编译运行项目**

  在完成以上环境的搭建之后，你就可以运行这个项目了，具体的步骤如下：

  1. 进入项目目录并编译

```
cd Final_SLAM/Required_Section
mkdir -p build && cd build
cmake ..
make
```

2. 运行程序

   编译后将生成三个可执行文件，分别对应三种特征提取算法：

```
# 运行ORB特征提取和深度估计
./ORB
# 运行SIFT特征提取和深度估计
./SIFT
# 运行SURF特征提取和深度估计
./SURF
```

- **Python 环境**

  todo

2.3. 关于数据集

本项目使用的数据集为 https://sun3d.cs.princeton.edu/data/mit_w85k1/whole_apartment/ 数据集，SUN3D 是普林斯顿大学计算机视觉实验室发布的大规模 RGB-D 视频数据集，专为研究 3D 重建、室内场景理解、SLAM 建图和多视角配准等任务设计。

- **获取数据集**

  1. 打开终端，更新并安装 `wget`

```
sudo apt update
sudo apt install wget
```

  2. 使用 `wget` 下载整个数据集目录

```
wget -r -np -nH --cut-dirs=3 -R "index.html*"
https://sun3d.cs.princeton.edu/data/mit_w85k1/whole_apartment/
```

     使用 `wget` 下载该数据集时，我们采用递归模式 `-r` 来抓取整个目录结构，配合 `-np` 防止访问父目录，`-nH` 避免创建主机名目录。通过 `--cut-dirs=3` 去除 URL 前 3 层路径，仅保留 `whole_apartment/` 目录下的内容，`-R "index.html*"` 则用于排除网站自动生成的网页文件。最终目标链接为：`https://sun3d.cs.princeton.edu/data/mit_w85k1/whole_apartment/`。

     注意：这个数据集比较大，在本项目的服务器上差不多需要 4-5 小时进行下载。

- **数据集说明**

  我们本次项目使用的 whole_apartment 数据集该是一套完整扫描的 MIT W85K1 公寓全景数据。数据通过 RGB-D 摄像头录制并处理，包括 RGB 图像、深度图、相机参数和缩略图。下载后的数据集结构目录如下：

```
whole_apartment/
├── depth/
├── image/
├── extrinsics/
├── intrinsics.txt
├── thumbnail/
```

  - **depth** 文件夹：存储深度图像，每张与 RGB 图对应。一共有 10692 张深度图像。
  - **image** 文件夹：存储对应的彩色 RGB 图像，一共有 10642 张彩色 RGB 图像。
  - **extrinsics** 文件夹：包含相机外参（位姿）信息。
  - **intrinsics.txt** : 相机内参矩阵，如焦距、主点，用于像素→相机坐标转换。
  - **thumbnail** 文件夹：存储缩略图，仅供快速预览或 Web 页面使用。

2.4. 评估指标

在对基于三角化方法估计的深度值进行定量评估时，我们这里使用的误差指标包括：绝对误差平均值（Abs）、均方根误差（RMSE）以及对数均方根误差（RMSE log）。这些指标能够从不同角度反映深度估计的准确性与鲁棒性。设：

- y_i 表示第 i 个像素点的预测深度值
- y_i^*表示该点的真实深度值
- N 表示参与评估的像素点总数

则各评估指标定义如下：

1. **平均绝对误差（Abs）**

   该指标反映了预测深度与真实深度之间的平均绝对差异，计算公式如下：

$$\text{Abs} = \frac{1}{N}\sum_{i=1}^{N}|y_i - y_i^*|$$

2. **均方根误差（Root Mean Square Error, RMSE）**

   RMSE 是评估误差平方的均值再开方，它对较大的误差更加敏感，适合衡量整体估计的偏差程度：

$$\text{RMSE} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(y_i - y_i^*)^2}$$

3. **对数均方根误差（RMSE log）**

   RMSE log 使用预测值和真实值的对数差异进行评价，减弱了对大深度值误差的敏感性，更关注相对误差，适合于尺度变化大的深度估计任务:

$$\text{RMSE}_{\log} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(\log y_i - \log y_i^*)^2}$$

3. 必做部分 - 传统特征检测器

三角化是在已知两个相机的姿态与内参的前提下，通过图像上某点在两帧中的匹配点位置，恢复该点在三维空间中的坐标。因此特征点匹配在三角化过程中的作用至关重要，是三角化的前置条件和直接输入，它的本质作用是：建立两帧图像中“同一个三维点”在不同图像中的对应关系。本项目按照要求完成了 ORB 算法、SIFT 算法和 SURF 算法特征检测器的三角化过程。

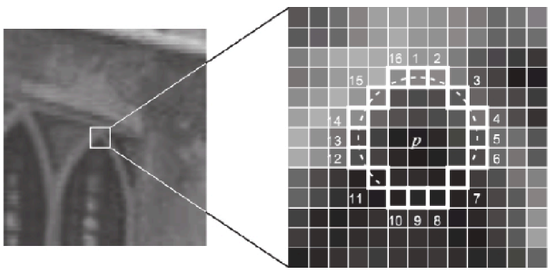
3.1. ORB 算法

3.1.1. ORB 算法原理

ORB（Oriented FAST and Rotated BRIEF）是一种高效、快速的特征点提取与匹配算法，广泛应用于 SLAM、图像匹配、目标识别等领域。它结合了 FAST 关键点检测器和 BRIEF 描述子的优点，并增强了其方向不变性与尺度鲁棒性。ORB 可以看作是对传统 FAST + BRIEF 的改进，主要包括以下步骤：

- **FAST 角点检测（关键点提取）**

  ORB 使用 FAST 算法检测图像中的角点（关键点），通过在每个像素周围构建圆形邻域，比较邻域像素与中心像素的亮度差值，判断是否为角点。为了提高质量，ORB 使用 Harris 角点评分函数对 FAST 提取的点进行排序，筛选响应强度最高的前 N 个关键点。



- **构建金字塔（实现尺度不变性）**

  为获得尺度不变性，ORB 在图像上构建图像金字塔，在不同分辨率下进行 FAST 角点检测。这确保了无论特征大小如何，都能被有效检测。

- 每一层图像缩放后独立检测角点；
- 关键点坐标、尺度随图像层级进行适当变换。

- **主方向分配（实现旋转不变性）**

  原始 BRIEF 特征对旋转敏感，因此 ORB 引入主方向分配来解决这一问题：

  在每个关键点的邻域中，根据灰度质心法（Intensity Centroid）计算该点的主方向。像素点的图像矩定义为：

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

  其中 $I(x,y)$ 是图像在坐标 (x,y) 处的灰度值，m_{10}、m_{01} 是一阶矩，关键点方向由下式计算：

  $$\theta = \arctan\left(\frac{m_{01}}{m_{10}}\right)$$

  该方向用于后续 BRIEF 描述子的旋转，确保描述子在旋转下保持一致。

- **BRIEF 描述子生成（特征表示）**

  BRIEF 是一种使用二值对比的特征描述子，通过在关键点邻域内随机采样像素对，比较灰度值形成二进制串：

  - 若第一个像素亮度大于第二个像素，取值为1，否则为0；
  - 一个典型的 BRIEF 描述子包含 256 位或 512 位二进制串；
  - ORB 通过将 BRIEF 模板旋转到主方向，形成 rBRIEF（Rotated BRIEF），实现旋转不变性。

- **特征点匹配**

  由于 ORB 描述子是二进制形式，使用 Hamming 距离可以快速进行特征点匹配。Hamming 距离是两位串之间不同位数的个数，计算高效、速度快，适用于实时场景。

3.1.2. 实验代码实现过程

接下来基于 ORB 特征检测器的三角化估计深度的具体实验代码的实现过程。

首先读取我们上面所示的数据集中的指定文件夹下所有图像路径，并按文件名排序，保证图像对顺序一致。使用 ORB 特征提取与匹配，其中包括了使用 ORB 算法提取图像关键点（Oriented FAST）、计算 BRIEF 描述子和使用 Hamming 距离进行暴力匹配。

```
// 使用 ORB 算法在两张图像中提取并匹配特征点
void find_ORB_feature_matches(const Mat &img_1, const Mat &img_2,
                              std::vector<KeyPoint> &keypoints_1,
                              std::vector<KeyPoint> &keypoints_2,
                              std::vector<DMatch> &matches)
{
  // 用于存储图像的描述子（特征向量）
  Mat descriptors_1, descriptors_2;
  // 创建 ORB 特征检测器（同时也作为描述子提取器）
  Ptr<FeatureDetector> detector = ORB::create();
  Ptr<DescriptorExtractor> descriptor = ORB::create();
  // 创建 Hamming 距离的暴力匹配器（适用于二进制描述子 ORB）
  Ptr<DescriptorMatcher> matcher = DescriptorMatcher::create("BruteForce-Hamming");
  // 使用 ORB 检测图像 1 的关键点
  detector->detect(img_1, keypoints_1);
  // 使用 ORB 检测图像 2 的关键点
  detector->detect(img_2, keypoints_2);
  // 计算图像 1 的描述子（基于检测到的关键点）
  descriptor->compute(img_1, keypoints_1, descriptors_1);
  // 计算图像 2 的描述子
  descriptor->compute(img_2, keypoints_2, descriptors_2);
  // 使用 BruteForce-Hamming 匹配器对描述子进行匹配
  matcher->match(descriptors_1, descriptors_2, matches);
  // 输出：关键点 keypoints_1、keypoints_2，匹配结果 matches
}
```

之后进行相机位姿估计（使用五点法 + 本质矩阵），通过图像中的匹配点计算两帧图像之间的相对位姿（旋转矩阵 R 和平移向量 t），本质矩阵是从匹配点与内参矩阵 K 中推导而来的。

```
// 根据匹配的特征点估计两帧图像之间的相机相对姿态（R 和 t）
void pose_estimation_2d2d(const std::vector<KeyPoint> &keypoints_1,
                          const std::vector<KeyPoint> &keypoints_2,
                          const std::vector<DMatch> &matches,
                          Mat &R, Mat &t)
{
  // 构建相机内参矩阵 K
  Mat K = (Mat_<double>(3, 3) << fx, 0, cx,
                                 0, fy, cy,
                                 0, 0, 1);

  // 提取匹配点对应的像素坐标
  vector<Point2f> points1, points2;
  for (int i = 0; i < (int)matches.size(); i++) {
    points1.push_back(keypoints_1[matches[i].queryIdx].pt);
    points2.push_back(keypoints_2[matches[i].trainIdx].pt);
  }
  // 计算本质矩阵（Essential Matrix）
  // 本质矩阵编码了两个相机之间的几何约束
  Mat essential_matrix = findEssentialMat(points1, points2, K);
  // 根据本质矩阵恢复相机的相对旋转 R 和位移 t
  recoverPose(essential_matrix, points1, points2, K, R, t);
}
```

接下来进行像素坐标 → 相机坐标的转换，将图像像素点坐标转换为归一化的相机坐标系下的点，用于三角化计算。

```
// 将图像像素坐标 p 转换为归一化相机坐标
Point2f pixel2cam(const Point2d &p, const Mat &K)
{
  return Point2f(
    (p.x - K.at<double>(0, 2)) / K.at<double>(0, 0), // (x - cx) / fx
    (p.y - K.at<double>(1, 2)) / K.at<double>(1, 1)  // (y - cy) / fy
  );
}
```

最后就是使用三角化重建 3D 点云了，利用两帧图像的相对位姿（R、t）与匹配点，通过线性三角化方法计算每对点在三维空间中的坐标。

```
// 使用两帧图像之间的位姿将匹配点三角化为3D点
void triangulation(const vector<KeyPoint> &keypoint_1,
                   const vector<KeyPoint> &keypoint_2,
                   const std::vector<DMatch> &matches,
                   const Mat &R, const Mat &t,
                   vector<Point3d> &points,
                   vector<int> &query_indices)
{
  // 定义第一帧的投影矩阵
  Mat T1 = (Mat_<float>(3, 4) << 1, 0, 0, 0,
                                 0, 1, 0, 0,
                                 0, 0, 1, 0);
  // 构建第二帧的投影矩阵
  Mat T2 = (Mat_<float>(3, 4) <<
            R.at<double>(0, 0), R.at<double>(0, 1), R.at<double>(0, 2), t.at<double>
(0),
            R.at<double>(1, 0), R.at<double>(1, 1), R.at<double>(1, 2), t.at<double>
(1),
            R.at<double>(2, 0), R.at<double>(2, 1), R.at<double>(2, 2), t.at<double>
(2));
  // 将匹配点转换为归一化相机坐标
  vector<Point2f> pts_1, pts_2;
  for (DMatch m : matches) {
    pts_1.push_back(pixel2cam(keypoint_1[m.queryIdx].pt, K));
    pts_2.push_back(pixel2cam(keypoint_2[m.trainIdx].pt, K));
```

```
    query_indices.push_back(m.queryIdx); // 保存索引方便后续匹配
  }
  // 三角化求解：将2D点对重建为4D齐次坐标点
  Mat pts_4d;
  triangulatePoints(T1, T2, pts_1, pts_2, pts_4d);
  // 齐次坐标 → 非齐次坐标转换，并保存为 Point3d
  for (int i = 0; i < pts_4d.cols; i++) {
    Mat x = pts_4d.col(i);
    x /= x.at<float>(3, 0); // 齐次归一化
    points.push_back(Point3d(
      x.at<float>(0, 0),
      x.at<float>(1, 0),
      x.at<float>(2, 0)
    ));
  }
}
```

在此之后，我们就得到了 3D 空间中的点云坐标，便可以从三角化结果中得到的 Z 与深度图中真实 Z 值做差，记录每个匹配点的深度误差，计算误差指标计算与输出了，得到下面小节中的实验结果了。

3.1.3. 实验结果

三角化（Triangulation）需要两帧图像来估计深度，具体来说，是需要两帧之间的对应点和这两帧的相对位姿，才能进行深度估计。在实验过程中，我们划分了四种数据规模，因为在这么多数据集中，并不是每一个图像对都是有效的，有些可能因为模糊、角度突然变化导致深度误差太过于不合理，因此在这里我们设置最大图像对分别为 200, 500, 1000, 3000 pairs，目的是观察在数据量增长的情况下，ORB 特征在三角化深度估计中的稳定性和误差表现；同时，帧间隔（interval）影响三角化中视差的大小，从而直接关系到重建精度。我们从 `interval = 1` 到 `interval = 10` 逐步增大帧间隔，探索视差变化对深度估计误差的影响，看看是否存在一个较优帧间隔，使得误差小、效率优。以下是我们的实验结果：

- **设置最大图像对为 200 pairs**
  - 总体实验指标结果

Max 200 pairs	Avg Abs	Avg RMSE	Avg RMSE log	Time	HQ(High Quality)
interval = 1 frames	0.926	4.024	1.154	9.348	23
interval = 3 frames	**0.883**	**3.796**	**1.101**	9.812	**24**
interval = 5 frames	0.952	3.904	1.172	**9.191**	20
interval = 10 frames	0.966	4.192	1.198	9.917	21

  - ORB 筛选出高质量匹配对指标结果

	Avg Abs	Avg RMSE	Avg RMSE log
interval = 1 frames	0.382	0.912	0.483
interval = 3 frames	**0.367**	0.861	**0.472**
interval = 5 frames	0.386	**0.827**	0.488
interval = 10 frames	0.394	0.954	0.496

- **设置最大图像对为 500 pairs**
  - 总体实验指标结果

Max 500 pairs	Avg Abs	Avg RMSE	Avg RMSE log	Time	HQ(High Quality)
interval = 1 frames	0.948	4.064	1.187	26.728	**51**

Max 500 pairs	Avg Abs	Avg RMSE	Avg RMSE log	Time	HQ(High Quality)
interval = 3 frames	**0.912**	3.813	**1.152**	25.213	46
interval = 5 frames	0.954	**3.801**	1.166	26.326	48
interval = 10 frames	0.979	4.117	1.198	**25.192**	48

- ORB 筛选出高质量匹配对指标结果

	Avg Abs	Avg RMSE	Avg RMSE log
interval = 1 frames	0.389	**0.932**	0.491
interval = 3 frames	**0.377**	0.941	**0.477**
interval = 5 frames	0.394	0.928	0.494
interval = 10 frames	0.399	0.984	0.502

- **设置最大图像对为 1000 pairs**
  - 总体实验指标结果

Max 1000 pairs	Avg Abs	Avg RMSE	Avg RMSE log	Time	HQ(High Quality)
interval = 1 frames	0.963	4.104	1.207	53.278	87
interval = 3 frames	**0.937**	3.983	**1.174**	**51.345**	**92**
interval = 5 frames	0.958	**3.978**	1.191	54.263	89
interval = 10 frames	0.989	4.217	1.228	55.824	84

  - ORB 筛选出高质量匹配对指标结果

	Avg Abs	Avg RMSE	Avg RMSE log
interval = 1 frames	0.392	0.976	0.512
interval = 3 frames	**0.381**	**0.973**	**0.507**
interval = 5 frames	0.402	0.995	0.524
interval = 10 frames	0.419	1.014	0.541

- **设置最大图像对为 3000 pairs**
  - 总体实验指标结果

Max 3000 pairs	Avg Abs	Avg RMSE	Avg RMSE log	Time	HQ(High Quality)
interval = 1 frames	1.028	4.311	1.278	203.142	255
interval = 3 frames	**0.989**	4.279	**1.252**	211.556	253

Max 3000 pairs	Avg Abs	Avg RMSE	Avg RMSE log	Time	HQ(High Quality)
interval = 5 frames	0.998	**4.271**	1.267	**204.634**	**262**
interval = 10 frames	1.039	4.483	1.314	215.645	241

- ORB 筛选出高质量匹配对指标结果

	Avg Abs	Avg RMSE	Avg RMSE log
interval = 1 frames	0.432	1.036	0.551
interval = 3 frames	**0.407**	**1.013**	**0.534**
interval = 5 frames	0.441	1.025	0.550
interval = 10 frames	0.463	1.078	0.597

以上我们做了 16 组关于 ORB 算法的对比实验，从两个关键因素展开分析，一是图像对的数据规模，二是图像间的帧间隔。我们此处在统一算法 ORB 框架下进行内部对比，三种算法的对比将在 3.4 节进行分析。

1. 首先是图像对数量对误差与稳定性的影响。从宏观上看，随着最大图像对数量的提升，整体误差相对稳定但略有上升：从200到3000对图像的实验来看，Avg Abs 整体略有上升趋势，尤其是在 interval 较小时表现明显。可能的原因在于更多图像对虽然提高了冗余度，但也引入了更多不稳定或误差较大的匹配，拉高了整体误差。
2. 然后是帧间隔对重建误差的影响。帧间隔决定了视差的大小，帧间隔不是越大越好也不是越小越好。从实验结果来看，interval = 3 或 interval = 5 的实验结果在精度与时间之间达到较好平衡。例如在 500 对图像中，interval = 3下的 Avg Abs 为 0.912 ，Avg RMSE为 3.813，优于其他间隔设置。而高质量匹配指标也显示interval = 3 时误差最低。

综合有关于 ORB 算法的实验结果，在精度表现上来看，interval = 3或 5 相对均衡，是较为理想的配置。interval = 1 的误差稍微高一些，interval = 10 的匹配不稳定。同时，随着图像对数量各项指标稍微上涨也是合理的，体现出 ORB 匹配筛选算法的有效性和一定的稳定性。

3.2. SIFT 算法

3.2.1. SIFT 算法原理

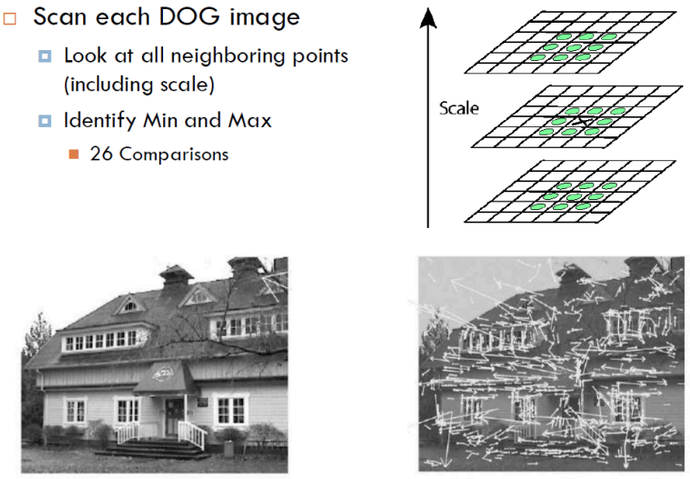
SIFT（Scale-Invariant Feature Transform，尺度不变特征变换）是一种用于检测和描述图像局部特征的经典算法，由 David Lowe 在1999 年提出，广泛应用于计算机视觉领域。SIFT 算法具有尺度不变性和旋转不变性，能够在图像发生缩放、旋转以及光照变化时仍可靠地检测关键点，并生成独特的特征描述符。其主要流程包括构造尺度空间检测极值点、关键点精确定位、分配方向以及生成描述符。通过计算关键点邻域的梯度方向分布，SIFT 为每个关键点生成 128 维的特征向量，用于描述图像局部特征。由于这些特征描述符具有较高的独特性和鲁棒性，SIFT 被广泛应用于图像匹配、目标检测、物体跟踪、3D 重建和图像拼接等任务中。

- **尺度空间极值检测**

  高斯模糊：对图像进行高斯模糊处理，生成不同尺度的图像。

  差分高斯（DoG）：计算相邻高斯模糊图像的差分，生成 DoG 金字塔。

  极值点检测：在 DoG 金字塔中，检测每个像素点在空间和尺度上的极值。即在 3x3x3 的邻域内，找到局部最大或最小值。



- **关键点定位**

  精确定位：对检测到的极值点进行精确定位，使用泰勒展开对极值点进行拟合，去除不稳定的边缘响应点和低对比度点。

  去除低对比度点：通过计算极值点的对比度，去除对比度低于某个阈值的点。

  去除边缘响应：通过计算 Hessian 矩阵的主曲率，去除边缘响应强的点。

- **方向匹配**

  梯度计算：在关键点的邻域内，计算每个像素的梯度幅值和方向。

  方向直方图：将梯度方向分成 36 个方向（每 10 度一个方向），计算每个方向的梯度幅值和。

  主方向确定：选择幅值和最大的方向作为关键点的主方向。对于幅值和超过主方向 80% 的其他方向，也可以分配一个次方向。

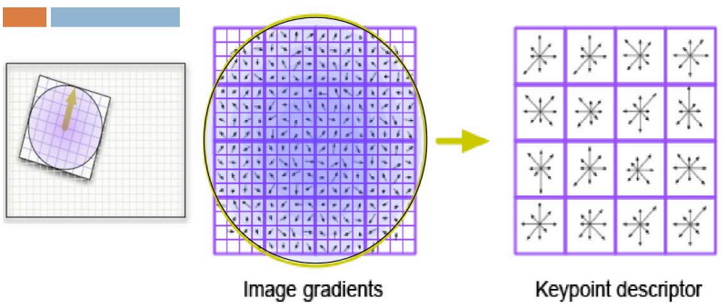
- **特征描述符生成**

  邻域划分：在关键点的邻域内，划分为 4x4 的网格。

  梯度直方图：在每个网格中，计算 8 个方向的梯度直方图。

  特征向量：将所有网格的梯度直方图串联成一个 128 维的特征向量。

  归一化：对特征向量进行归一化处理，以增强对光照变化的鲁棒性。



SIFT 生成的 128 维浮点型向量通常使用欧氏距离（L2 距离）进行匹配，匹配结果精度高但计算开销相对较大。

3.2.2. 实验代码实现过程

接下来基于 SIFT 特征检测器的三角化估计深度的具体实验代码的实现过程。

首先读取我们上面所示的数据集中的指定文件夹下所有图像路径，并按文件名排序，保证图像对顺序一致。然后进行 SIFT 过程，这个过程包含四个关键步骤：首先，使用 OpenCV 的 SIFT 实现创建检测器，并在两幅图像上检测关键点；其次，对这些关键点计算 SIFT 描述子，形成 128 维特征向量描述局部图像区域的梯度信息；接着，通过暴力匹配器 (BruteForce) 使用 L2 欧氏距离匹配两幅图像中的描述子；最后，应用距离阈值（最小距离的两倍或 30.0，取较大值）过滤匹配点，保留高质量的特征对应。这一流程提供了具有旋转、尺度和亮度不变性的特征匹配，为后续的姿态估计奠定基础。

```
void find_SIFT_feature_matches(const Mat &img_1, const Mat &img_2,
                               std::vector<KeyPoint> &keypoints_1,
                               std::vector<KeyPoint> &keypoints_2,
                               std::vector<DMatch> &matches)
{
  //-- 初始化
  Mat descriptors_1, descriptors_2;
```

```
  Ptr<SIFT> sift = SIFT::create();

  //-- 第一步:检测 SIFT 关键点位置
  sift->detect(img_1, keypoints_1);
  sift->detect(img_2, keypoints_2);

  //-- 第二步:根据关键点位置计算 SIFT 描述子
  sift->compute(img_1, keypoints_1, descriptors_1);
  sift->compute(img_2, keypoints_2, descriptors_2);

  //-- 第三步:对两幅图像中的SIFT描述子进行匹配，使用 L2 距离
  Ptr<DescriptorMatcher> matcher = DescriptorMatcher::create("BruteForce");
  vector<DMatch> match;
  matcher->match(descriptors_1, descriptors_2, match);

  //-- 第四步:匹配点对筛选
  double min_dist = 10000, max_dist = 0;

  // 找出所有匹配之间的最小距离和最大距离
  for (int i = 0; i < descriptors_1.rows; i++)
  {
    double dist = match[i].distance;
    if (dist < min_dist)
      min_dist = dist;
    if (dist > max_dist)
      max_dist = dist;
  }

  // 筛选好的匹配点
  for (int i = 0; i < descriptors_1.rows; i++)
  {
    if (match[i].distance <= max(2 * min_dist, 30.0))
    {
      matches.push_back(match[i]);
    }
  }
}
```

然后进行基于特征匹配的相机姿态估计，利用匹配的特征点计算两相机间的几何关系。首先，从 SIFT 匹配对中提取对应的像素坐标点；然后，使用这些点对和已知的相机内参矩阵 K 计算本质矩阵 E，该矩阵编码了两个相机视角之间的几何约束；最后，通过从本质矩阵中分解出旋转矩阵 R 和平移向量 t。

```
void pose_estimation_2d2d(
    const std::vector<KeyPoint> &keypoints_1,
    const std::vector<KeyPoint> &keypoints_2,
    const std::vector<DMatch> &matches,
    Mat &R, Mat &t)
{
  // 提供的内参矩阵
  Mat K = (Mat_<double>(3, 3) << 570.34220474152971291914582252502441406 25, 0, 320,
           0, 570.34220474152971291914582252502441406 25, 240,
           0, 0, 1);
  //-- 把匹配点转换为vector<Point2f>的形式
  vector<Point2f> points1;
  vector<Point2f> points2;
  for (int i = 0; i < (int)matches.size(); i++)
  {
    points1.push_back(keypoints_1[matches[i].queryIdx].pt);
    points2.push_back(keypoints_2[matches[i].trainIdx].pt);
  }
  //-- 计算本质矩阵
  Mat essential_matrix;
  essential_matrix = findEssentialMat(points1, points2, K);
  //-- 从本质矩阵中恢复旋转和平移信息.
  recoverPose(essential_matrix, points1, points2, K, R, t);
```

```
}
```

最后就是使用三角化重建 3D 点云了，它首先设置两相机投影矩阵，将像素坐标转为归一化坐标，然后通过 OpenCV 的三角测量算法计算点的空间位置，最后转换为3D欧式坐标。

```
void triangulation(
    const vector<KeyPoint> &keypoint_1,
    const vector<KeyPoint> &keypoint_2,
    const std::vector<DMatch> &matches,
    const Mat &R, const Mat &t,
    vector<Point3d> &points,
    vector<int> &query_indices)
{
  Mat T1 = (Mat_<float>(3, 4) << 1, 0, 0, 0,
            0, 1, 0, 0,
            0, 0, 1, 0);
  Mat T2 = (Mat_<float>(3, 4) << R.at<double>(0, 0), R.at<double>(0, 1), R.at<double>
(0, 2), t.at<double>(0, 0),
            R.at<double>(1, 0), R.at<double>(1, 1), R.at<double>(1, 2), t.at<double>
(1, 0),
            R.at<double>(2, 0), R.at<double>(2, 1), R.at<double>(2, 2), t.at<double>
(2, 0));
  // 提供的内参矩阵
  Mat K = (Mat_<double>(3, 3) << 570.342204741529712919145822525024414062 5, 0, 320,
          0, 570.342204741529712919145822525024414062 5, 240,
          0, 0, 1);
  vector<Point2f> pts_1, pts_2;
  query_indices.clear(); // 清空索引向量
  for (DMatch m : matches)
  {
    // 将像素坐标转换至相机坐标
    pts_1.push_back(pixel2cam(keypoint_1[m.queryIdx].pt, K));
    pts_2.push_back(pixel2cam(keypoint_2[m.trainIdx].pt, K));
    query_indices.push_back(m.queryIdx); // 记录第一帧中的特征点索引
  }
  Mat pts_4d;
  cv::triangulatePoints(T1, T2, pts_1, pts_2, pts_4d);
  // 转换成非齐次坐标
  points.clear(); // 确保点集是空的
  for (int i = 0; i < pts_4d.cols; i++)
  {
    Mat x = pts_4d.col(i);
    x /= x.at<float>(3, 0); // 归一化
    Point3d p(
        x.at<float>(0, 0),
        x.at<float>(1, 0),
        x.at<float>(2, 0));
    points.push_back(p);
  }
}
```

在此之后，我们就得到了 3D 空间中的点云坐标，便可以从三角化结果中得到的 Z 与深度图中真实 Z 值做差，记录每个匹配点的深度误差，计算误差指标计算与输出了，得到下面小节中的实验结果了。

3.2.3. 实验结果

三角化（Triangulation）需要两帧图像来估计深度，具体来说，是需要两帧之间的对应点和这两帧的相对位姿，才能进行深度估计。在实验过程中，我们划分了四种数据规模，因为在这么多数据集中，并不是每一个图像对都是有效的，有些可能因为模糊、角度突然变化导致深度误差太过于不合理，因此在这里我们设置最大图像对分别为 200, 500, 1000, 3000 pairs，目的是观察在数据量增长的情况下，SIFT 特征在三角化深度估计中的稳定性和误差表现；同时，帧间隔（interval）影响三角化中视差的大小，从而直接关系到重建精度。我们从 `interval = 1` 到 `interval = 10` 逐步增大帧间隔，探索视差变化对深度估计误差的影响，看看是否存在一个较优帧间隔，使得误差小、效率优。以下是我们的实验结果：

- **设置最大图像对为 200 pairs**

- 总体实验指标结果

Max 200 pairs	Avg Abs	Avg RMSE	Avg RMSE log	Time	HQ(High Quality)
interval = 1 frames	0.872	3.536	0.964	45.523	32
interval = 3 frames	**0.849**	**3.318**	**0.933**	46.142	**36**
interval = 5 frames	0.883	3.512	0.972	**44.546**	29
interval = 10 frames	0.904	3.546	1.007	45.635	31

- SIFT 筛选出高质量匹配对指标结果

	Avg Abs	Avg RMSE	Avg RMSE log
interval = 1 frames	0.301	0.812	0.447
interval = 3 frames	**0.297**	**0.785**	**0.441**
interval = 5 frames	0.312	0.796	0.455
interval = 10 frames	0.334	0.837	0.463

- **设置最大图像对为 500 pairs**
  - 总体实验指标结果

Max 500 pairs	Avg Abs	Avg RMSE	Avg RMSE log	Time	HQ(High Quality)
interval = 1 frames	0.886	3.601	0.989	109.453	59
interval = 3 frames	**0.851**	**3.413**	**0.973**	113.438	67
interval = 5 frames	0.867	3.502	0.982	**104.658**	**69**
interval = 10 frames	0.912	3.596	1.017	107.687	63

  - SIFT 筛选出高质量匹配对指标结果

	Avg Abs	Avg RMSE	Avg RMSE log
interval = 1 frames	0.317	0.856	0.463
interval = 3 frames	**0.304**	**0.811**	**0.452**
interval = 5 frames	0.321	0.826	0.471
interval = 10 frames	0.376	0.889	0.487

- **设置最大图像对为 1000 pairs**
  - 总体实验指标结果

Max 1000 pairs	Avg Abs	Avg RMSE	Avg RMSE log	Time	HQ(High Quality)
interval = 1 frames	0.897	3.786	0.993	232.654	109
interval = 3 frames	**0.869**	3.691	**0.991**	225.237	**111**
interval = 5 frames	0.904	**3.689**	0.998	221.765	104
interval = 10 frames	0.923	3.732	1.027	**220.128**	102

- SIFT 筛选出高质量匹配对指标结果

	Avg Abs	Avg RMSE	Avg RMSE log
interval = 1 frames	0.346	0.873	0.481
interval = 3 frames	**0.321**	**0.839**	**0.477**
interval = 5 frames	0.335	0.854	0.485
interval = 10 frames	0.378	0.897	0.498

- **设置最大图像对为 3000 pairs**
  - 总体实验指标结果

Max 3000 pairs	Avg Abs	Avg RMSE	Avg RMSE log	Time	HQ(High Quality)
interval = 1 frames	0.949	4.002	1.082	1074.652	307
interval = 3 frames	**0.932**	**3.988**	**1.068**	1071.856	324
interval = 5 frames	0.953	4.015	1.087	**1064.365**	**326**
interval = 10 frames	0.987	4.034	1.105	1080.827	312

  - SIFT 筛选出高质量匹配对指标结果

	Avg Abs	Avg RMSE	Avg RMSE log
interval = 1 frames	0.377	0.912	0.492
interval = 3 frames	**0.365**	**0.896**	**0.483**
interval = 5 frames	0.371	0.904	0.496
interval = 10 frames	0.402	0.948	0.509

和 ORB 算法相同的，我们做了 16 组关于 SIFT 算法的对比实验，从两个关键因素展开分析，一是图像对的数据规模，二是图像间的帧间隔。我们此处在统一算法 SIFT 框架下进行内部对比，三种算法的对比将在 3.4 节进行分析。

1. 首先是图像对数量对误差与稳定性的影响。从整体误差角度看，随着图像对数量从 200 对扩展到 3000 对，各项误差指标呈现出较小幅度的增长。

2. 然后是帧间隔对重建误差的影响。帧间隔决定了视差的大小，帧间隔不是越大越好也不是越小越好。`interval = 3` 几乎都在总体误差与高质量匹配精度两个维度上取得了最优结果。如在 200 对数据中，`interval = 3` 获得最低的 Avg Abs 和 Avg RMSE；

SIFT 输出了稳定数量的高质量匹配点，并维持较低的误差水平，特别是在中等间隔下效果最佳，显示出其良好的尺度不变性与鲁棒性。

3.3. SURF 算法

3.3.1. SURF 算法原理

SURF（Speeded Up Robust Features）是一种用于图像特征提取和描述的局部特征算法，由 Herbert Bay 等人于 2006 年提出。SURF 是对 SIFT 的加速和优化版本，该算子在保持 SIFT 算子优良性能特点的基础上，同时解决了 SIFT 计算复杂度高、耗时长的缺点，对兴趣点提取及其特征向量描述方面进行了改进，且计算速度得到提高，适用于图像匹配、物体识别、三维重建等任务。

- **构造 Hessian 矩阵并计算行列式近似值**

SURF 使用 Hessian 矩阵的行列式来检测图像中的特征点。对于图像中的每个像素点 x，在尺度 σ 下，其 Hessian 矩阵定义为：

$$H(x,\sigma) = \begin{bmatrix} L_{xx}(x,\sigma) & L_{xy}(x,\sigma) \\ L_{xy}(x,\sigma) & L_{yy}(x,\sigma) \end{bmatrix}$$

其中，L_{xx}、L_{yy}、L_{xy} 分别表示图像在 x 处的二阶高斯导数。为了加速计算，SURF 使用 Box Filter 近似高斯导数，并利用积分图像快速计算滤波响应。

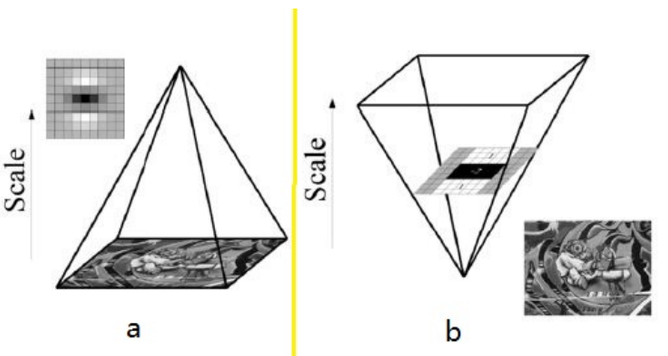
行列式的近似计算公式为：

$$\mathrm{Det}(H_{\mathrm{approx}}) = D_{xx}D_{yy} - (w \cdot D_{xy})^2$$

其中，D_{xx}、D_{yy}、D_{xy} 是 Box Filter 的响应，w 是加权系数，通常取 0.9。

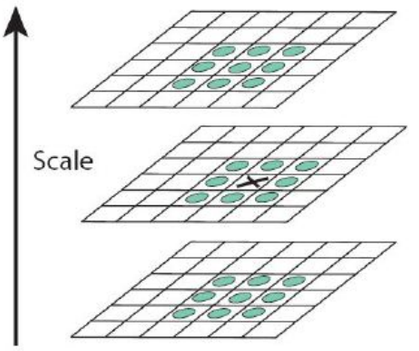
- **构建尺度空间金字塔**

  相比于 SIFT 算法的高斯金字塔构造过程，SURF 算法速度有所提高。在 SURF 算法中，每一组（octave）的图像大小是不一样的，下一组是上一组图像的降采样（1/4大小）；在每一组里面的几幅图像中，他们的大小是一样的，不同的是他们采用的尺度 σ 不同。而且在模糊的过程中，他们的高斯模板大小总是不变的，只是尺度 σ 改变。对于 SURF 算法，图像的大小总是不变的，改变的只是高斯模糊模板的尺寸，当然，尺度 σ 也是在改变的。



- **特征点检测与精确定位**

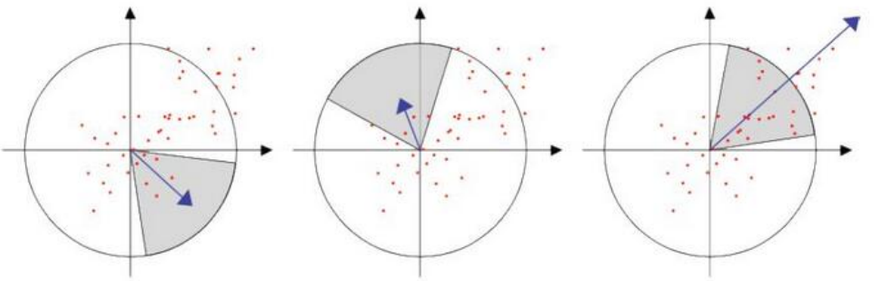
  首先初步定为特征点，如下图，将经过 Hessian 矩阵处理过的每个像素点与其 3 维领域的 26 个点进行大小比较，如果它是这26个点中的最大值或者最小值，则保留下来，当做初步的特征点。然后，跟 SIFT 算法类似，采用3维线性插值法得到亚像素级的特征点，同时也去掉那些值小于一定阈值的点，增加极值使检测到的特征点数量减少，最终只有几个特征最强点会被检测出来。



- **主方向分配（实现旋转不变性）**

  为了赋予特征点旋转不变性，SURF 和 SIFT 不同，不统计其梯度直方图，而是在特征点邻域内计算 Haar 小波响应，确定主方向。具体步骤如下：

  - 以特征点为中心，半径为 6s（ s 为特征点的尺度）构建邻域；
  - 在该邻域内，计算每个像素点在 x 和 y 方向的 Haar 小波响应；
  - 将响应值加权（使用高斯权重），并在 60° 的扇形窗口内求和；
  - 扫描整个邻域，找到响应和最大的方向，作为特征点的主方向。



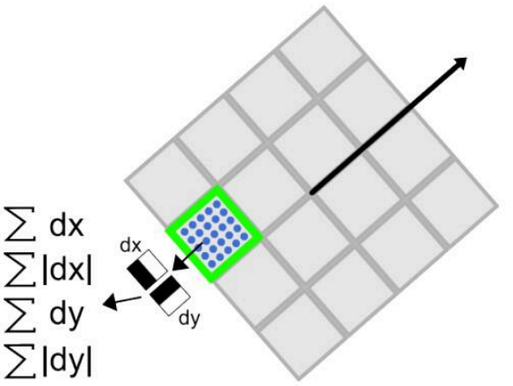
- **构建特征描述子**

  SURF 在特征点的主方向上，构建描述子以实现旋转不变性。具体步骤如下：

  - 在特征点主方向上，取边长为 20s 的正方形区域；
  - 将该区域划分为 4×4 的子区域，共 16 个；
  - 在每个子区域内，计算 Haar 小波在 x 和 y 方向的响应值之和及其绝对值之和，得到四个特征量：

$$\sum dx, \quad \sum dy, \quad \sum |dx|, \quad \sum |dy|$$

  - 将所有子区域的特征量串联，构成一个 64 维的描述子向量。



- **特征匹配**

SURF 使用欧氏距离（L2 范数）来衡量描述子之间的相似性。对于两个描述子 d_1 和 d_2，其距离计算公式为：

$$\text{distance}(d_1, d_2) = \sqrt{\sum_{i=1}^{n} (d_{1i} - d_{2i})^2}$$

3.3.2. 实验代码实现过程

接下来基于 SURT 特征检测器的三角化估计深度的具体实验代码的实现过程。

首先读取我们上面所示的数据集中的指定文件夹下所有图像路径，并按文件名排序，保证图像对顺序一致。然后进行 SURF 过程，核心匹配函数是`find_SURF_feature_matches`，它包含了SURF特征提取和匹配的完整流程：初始化SURF检测器、检测关键点、计算特征描述子、特征匹配和筛选高质量匹配点。

```
void find_SURF_feature_matches(const Mat &img_1, const Mat &img_2,
                               std::vector<KeyPoint> &keypoints_1,
                               std::vector<KeyPoint> &keypoints_2,
                               std::vector<DMatch> &matches)
{
  //-- 初始化
  Mat descriptors_1, descriptors_2;
  Ptr<SURF> surf = SURF::create();

  //-- 第一步:检测 SURF 关键点位置
  surf->detect(img_1, keypoints_1);
  surf->detect(img_2, keypoints_2);

  //-- 第二步:根据关键点位置计算 SURF 描述子
  surf->compute(img_1, keypoints_1, descriptors_1);
  surf->compute(img_2, keypoints_2, descriptors_2);

  //-- 第三步:对两幅图像中的SURF描述子进行匹配，使用 L2 距离
  Ptr<DescriptorMatcher> matcher = DescriptorMatcher::create("BruteForce");
  vector<DMatch> match;
  matcher->match(descriptors_1, descriptors_2, match);

  //-- 第四步:匹配点对筛选
  double min_dist = 10000, max_dist = 0;

  // 找出所有匹配之间的最小距离和最大距离，即最相似的和最不相似的两组点之间的距离
  for (int i = 0; i < descriptors_1.rows; i++)
  {
    double dist = match[i].distance;
    if (dist < min_dist)
      min_dist = dist;
    if (dist > max_dist)
      max_dist = dist;
  }

  // 当描述子之间的距离大于两倍的最小距离时，即认为匹配有误。但有时候最小距离会非常小，设置一个经验值30作为下限
  for (int i = 0; i < descriptors_1.rows; i++)
  {
    if (match[i].distance <= max(2 * min_dist, 30.0))
    {
      matches.push_back(match[i]);
    }
  }
}
```

在使用 SURF 匹配完成后，需要使用匹配点对计算两个相机之间的相对位姿（旋转 R 和平移 t）：将匹配点转换为标准格式，计算本质矩阵和从本质矩阵恢复相机的旋转和平移。

```
void pose_estimation_2d2d(
    const std::vector<KeyPoint> &keypoints_1,
    const std::vector<KeyPoint> &keypoints_2,
    const std::vector<DMatch> &matches,
    Mat &R, Mat &t)
{
  // 提供的内参矩阵
  Mat K = (Mat_<double>(3, 3) << 570.342204741529712919145822525024414062 5, 0, 320,
```

```
            0, 570.342204741529712919145822525024414062 5, 240,
            0, 0, 1);
  //-- 把匹配点转换为vector<Point2f>的形式
  vector<Point2f> points1;
  vector<Point2f> points2;
  for (int i = 0; i < (int)matches.size(); i++)
  {
    points1.push_back(keypoints_1[matches[i].queryIdx].pt);
    points2.push_back(keypoints_2[matches[i].trainIdx].pt);
  }
  //-- 计算本质矩阵
  Mat essential_matrix;
  essential_matrix = findEssentialMat(points1, points2, K);
  //-- 从本质矩阵中恢复旋转和平移信息.
  recoverPose(essential_matrix, points1, points2, K, R, t);
}
```

最后，三角测量函数使用匹配点和估计的相机位姿，计算特征点的 3D 坐标：将像素坐标转换为相机坐标系下的归一化坐标，使用两帧相机位姿和对应点进行三角测量和将结果转换为 3D 点坐标。

```
void triangulation(
    const vector<KeyPoint> &keypoint_1,
    const vector<KeyPoint> &keypoint_2,
    const std::vector<DMatch> &matches,
    const Mat &R, const Mat &t,
    vector<Point3d> &points,
    vector<int> &query_indices) // 添加参数来保存第一帧特征点索引
{
  Mat T1 = (Mat_<float>(3, 4) << 1, 0, 0, 0,
            0, 1, 0, 0,
            0, 0, 1, 0);
  Mat T2 = (Mat_<float>(3, 4) << R.at<double>(0, 0), R.at<double>(0, 1), R.at<double>
(0, 2), t.at<double>(0, 0),
            R.at<double>(1, 0), R.at<double>(1, 1), R.at<double>(1, 2), t.at<double>
(1, 0),
            R.at<double>(2, 0), R.at<double>(2, 1), R.at<double>(2, 2), t.at<double>
(2, 0));
  // 提供的内参矩阵
  Mat K = (Mat_<double>(3, 3) << 570.342204741529712919145822525024414062 5, 0, 320,
            0, 570.342204741529712919145822525024414062 5, 240,
            0, 0, 1);
  vector<Point2f> pts_1, pts_2;
  query_indices.clear(); // 清空索引向量
  for (DMatch m : matches)
  {
    // 将像素坐标转换至相机坐标
    pts_1.push_back(pixel2cam(keypoint_1[m.queryIdx].pt, K));
    pts_2.push_back(pixel2cam(keypoint_2[m.trainIdx].pt, K));
    query_indices.push_back(m.queryIdx); // 记录第一帧中的特征点索引
  }
  Mat pts_4d;
  cv::triangulatePoints(T1, T2, pts_1, pts_2, pts_4d);
  // 转换成非齐次坐标
  points.clear(); // 确保点集是空的
  for (int i = 0; i < pts_4d.cols; i++)
  {
    Mat x = pts_4d.col(i);
    x /= x.at<float>(3, 0); // 归一化
    Point3d p(
        x.at<float>(0, 0),
        x.at<float>(1, 0),
        x.at<float>(2, 0));
    points.push_back(p);
  }
}
```

在此之后，我们就得到了 3D 空间中的点云坐标，便可以从三角化结果中得到的 Z 与深度图中真实 Z 值做差，记录每个匹配点的深度误差，计算误差指标计算与输出了，得到下面小节中的实验结果了。

3.3.3. 实验结果

三角化（Triangulation）需要两帧图像来估计深度，具体来说，是需要两帧之间的对应点和这两帧的相对位姿，才能进行深度估计。在实验过程中，我们划分了四种数据规模，因为在这么多数据集中，并不是每一个图像对都是有效的，有些可能因为模糊、角度突然变化导致深度误差太过于不合理，因此在这里我们设置最大图像对分别为 200, 500, 1000, 3000 pairs，目的是观察在数据量增长的情况下，SURF 特征在三角化深度估计中的稳定性和误差表现；同时，帧间隔（interval）影响三角化中视差的大小，从而直接关系到重建精度。我们从 `interval = 1` 到 `interval = 10` 逐步增大帧间隔，探索视差变化对深度估计误差的影响，看看是否存在一个较优帧间隔，使得误差小、效率优。以下是我们的实验结果：

- **设置最大图像对为 200 pairs**
  - 总体实验指标结果

Max 200 pairs	Avg Abs	Avg RMSE	Avg RMSE log	Time	HQ(High Quality)
interval = 1 frames	0.881	3.621	0.988	36.354	24
interval = 3 frames	**0.863**	3.454	**0.972**	37.541	**26**
interval = 5 frames	0.897	**3.437**	1.005	**34.182**	25
interval = 10 frames	0.924	3.741	1.053	35.573	25

  - SURF 筛选出高质量匹配对指标结果

	Avg Abs	Avg RMSE	Avg RMSE log
interval = 1 frames	0.303	0.801	0.451
interval = 3 frames	**0.292**	**0.796**	**0.439**
interval = 5 frames	0.301	0.808	0.465
interval = 10 frames	0.336	0.825	0.482

- **设置最大图像对为 500 pairs**
  - 总体实验指标结果

Max 500 pairs	Avg Abs	Avg RMSE	Avg RMSE log	Time	HQ(High Quality)
interval = 1 frames	**0.917**	3.848	1.069	86.123	**53**
interval = 3 frames	0.928	**3.775**	**1.063**	83.645	50
interval = 5 frames	0.935	3.809	1.098	**79.761**	47
interval = 10 frames	0.922	3.797	1.102	87.173	45

  - SURF 筛选出高质量匹配对指标结果

	Avg Abs	Avg RMSE	Avg RMSE log
interval = 1 frames	0.352	0.892	0.479
interval = 3 frames	**0.350**	**0.883**	**0.475**
interval = 5 frames	0.361	0.898	0.493
interval = 10 frames	0.379	0.911	0.499

- **设置最大图像对为 1000 pairs**
  - 总体实验指标结果

Max 1000 pairs	Avg Abs	Avg RMSE	Avg RMSE log	Time	HQ(High Quality)
interval = 1 frames	**0.962**	4.016	**1.083**	192.653	92
interval = 3 frames	0.969	**3.971**	1.092	195.541	**99**
interval = 5 frames	0.981	3.994	1.112	**191.765**	94
interval = 10 frames	0.993	4.043	1.128	200.467	96

  - SURF 筛选出高质量匹配对指标结果

	Avg Abs	Avg RMSE	Avg RMSE log
interval = 1 frames	0.374	0.913	0.496
interval = 3 frames	**0.368**	**0.906**	**0.502**
interval = 5 frames	0.385	0.924	0.516
interval = 10 frames	0.399	0.934	0.537

- **设置最大图像对为 3000 pairs**
  - 总体实验指标结果

Max 3000 pairs	Avg Abs	Avg RMSE	Avg RMSE log	Time	HQ(High Quality)
interval = 1 frames	0.971	4.134	1.108	639.651	278
interval = 3 frames	**0.969**	**4.075**	**1.127**	683.854	302
interval = 5 frames	0.984	4.106	1.129	654.466	291
interval = 10 frames	0.975	4.193	1.145	**635.562**	297

  - SURF 筛选出高质量匹配对指标结果

	Avg Abs	Avg RMSE	Avg RMSE log
interval = 1 frames	0.418	0.996	**0.518**
interval = 3 frames	**0.407**	0.998	0.521

	Avg Abs	Avg RMSE	Avg RMSE log
interval = 5 frames	0.411	**0.987**	0.529
interval = 10 frames	0.443	1.015	0.556

和上面的算法相同的，我们做了 16 组关于 SURF 算法的对比实验，从两个关键因素展开分析，一是图像对的数据规模，二是图像间的帧间隔。我们此处在统一算法 SURF 框架下进行内部对比，三种算法的对比将在 3.4 节进行分析。

1. 首先是图像对数量对误差与稳定性的影响。整体误差在不同规模下较为稳定，当图像对数量从200增加到3000时，Avg Abs 的增幅控制在 10% 左右，Avg RMSE 也从约 3.6 增长至 4.1 上下，表明 SURF 匹配的质量并未因图像数量增加而显著退化。
2. 然后是帧间隔对重建误差的影响。帧间隔决定了视差的大小，帧间隔不是越大越好也不是越小越好。在大多数设置下，`interval = 3` 在整体误差和高质量匹配误差之间取得了最优平衡；`interval = 1` 对于小数据集略有优势，在 200 对与 500 对设置中，`interval = 1`帧表现略优，可能是因为图像间变化较小，SURF 特征点更易匹配。

总体而言，SURF 在不同大小的数据集下的三角化深度估计中展现出良好的稳健性与可扩展性，中等帧间隔综合表现最佳，对于小数据集来说小帧间隔表现也不错。

3.4. 三种传统算法对比

如上面的实验结果所示，我们对每一种算法都做了 4 种规模数据集的实验（ 200/500/1000/3000 个帧对），每种规模的数据集我们也探究了选取 4 种不同帧间隔（1/3/5/10帧间隔）的两帧进行了不同的实验评估平均绝对误差（Abs）、均方根误差（RMSE）、对数尺度RMSE（RMSE log）、运行时间（Time）和高质量图像对数量（HQ(High Quality)）五个指标。

为评估三种传统特征提取与匹配算法在不同数据规模与帧间隔条件下的表现，我们构造了四种评估场景，分别为小数据集小帧间隔、小数据集大帧间隔、大数据集小帧间隔和大数据集大帧间隔，并记录了每种算法在平均绝对误差（Abs）、均方根误差（RMSE）、对数尺度RMSE（RMSE log）、运行时间（Time）以及高质量图像对数量（HQ(High Quality)）五个指标下的结果。以下为实验结果与分析。

- **小数据集小帧间隔（数据量大小：pairs = 200, interval = 1）**

  在该设置下，图像帧间差异较小，更考验算法在细微变化中的匹配鲁棒性，三种算法的对比结果如下：

	Abs	RMSE	RMSE log	Time	HQ(High Quality)
ORB	0.926	4.024	1.154	**9.348**	23
SIFT	0.872	3.536	0.964	45.523	32
SURF	0.881	3.621	**0.988**	36.354	24

  - **SIFT** 算法在 RMSE 指标上表现最优，表明其在消除极端误差方面更强；
  - **SURF** 算法在 Abs 和 RMSE log 的指标上领先，说明其在整体稳定性和小误差控制方面表现更平衡；
  - **ORB** 误差相对较大，但在运行速度上具有显著优势，是该场景下的最快算法。

  结论：在小数据集小帧间差异场景中，若优先考虑精度，推荐 SURF 算法；若对运行速度要求较高，可选择 ORB 算法。

- **小数据集大帧间隔（数据量大小：pairs = 200, interval = 10）**

  帧间差异增大，考验算法对较大视差或变化的适应能力，三种算法的对比结果如下：

	Abs	RMSE	RMSE log	Time	HQ(High Quality)
ORB	0.966	4.192	1.198	**9.917**	21
SIFT	0.904	3.546	1.007	45.635	31
SURF	**0.924**	3.741	1.053	35.573	25

  - **SIFT** 算法以最低的 RMSE 和 RMSE log 值表现出对大视差的良好适应；
  - **SURF** 算法具有最低的平均绝对误差 Abs，整体性能也非常接近 SIFT 算法；

- **ORB** 算法误差相对略高，但是运行时间依然最短，适合对实时性要求高的场景。

结论：SIFT 算法与 SURF 算法在精度上不相上下，SIFT 算法略胜一筹；ORB 算法依然适合速度敏感型应用。

- **大数据集小帧间隔（数据量大小：pairs = 3000, interval = 1）**

  数据量大，对算法的稳定性和效率提出更高要求，三种算法的对比结果如下：

	Abs	RMSE	RMSE log	Time	HQ(High Quality)
ORB	1.028	4.311	1.278	**203.142**	255
SIFT	0.949	4.002	1.082	974.652	307
SURF	0.971	4.134	1.108	1039.651	278

  - **SIFT** 算法在 Abs 和 RMSE 的指标上表现最优，说明其在大样本量条件下保持了较高精度；
  - **SURF** 算法在 RMSE log 的指标最优，显示其对小误差项控制较好；
  - **ORB** 算法的误差整体较大，但是在时间效率上远胜其他两者。

  结论：若以精度为目标，推荐 SIFT 算法；若关注效率，ORB 算法是可以接受的实时选项。

- **大数据集大帧间隔（数据量大小：pairs = 3000, interval = 10）**

  数据量大，在前面的基础上对算法的稳定性和效率提出更高要求，三种算法的对比结果如下：

	Abs	RMSE	RMSE log	Time	HQ(High Quality)
ORB	1.039	4.483	1.314	215.645	241
SIFT	0.987	**4.034**	**1.105**	1080.827	312
SURF	0.975	4.193	1.145	635.562	297

  - **SIFT** 算法再一次在 RMSE 和 RMSE log 的指标上展现出最佳性能，误差压制能力强；
  - **SURF** 算法在 Abs 指标上最优，表明其预测偏差最小，综合表现略逊于 SIFT 算法；
  - **ORB** 算法虽然误差最大，但仍以超高效率运行，完成任务时间远低于其他算法。

  结论：SIFT 算法依然为比较稳健算法，SURF 算法也不差，具有一定平衡性；ORB 算法适合极限场景下的快速估计。

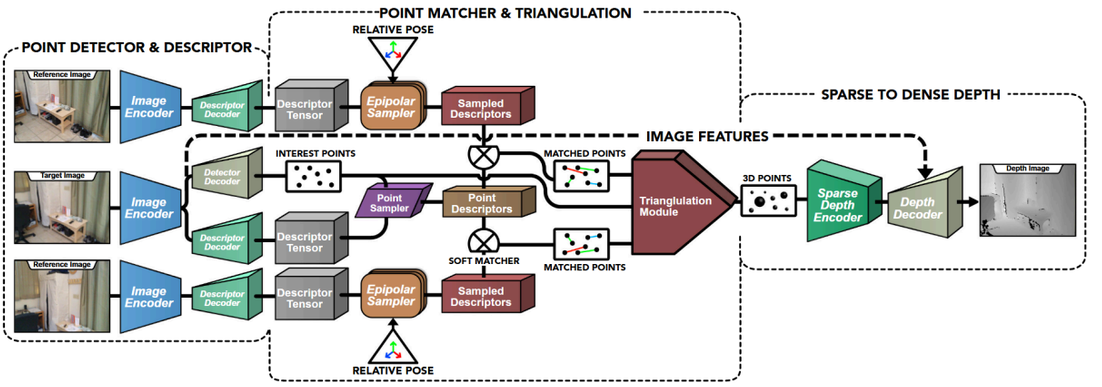
根据实验的最后结果来看，SIFT 算法在几乎所有场景中均展现出极高的精度，但是运行时间最长、运行效率不高；SURF 算法在精度上紧随其后，且在运行效率方面略优于 SIFT 算法，是一个性能均衡的选择；ORB 算法虽然在误差控制方面较弱，但其运行效率远高于其他两者，是资源受限、实时需求强的首选方案。综上所述，传统特征匹配算法在不同场景下表现有所差异，选择合适的方法需权衡精度与效率。SIFT 算法适合高精度需求场景，ORB 算法适合实时系统，SURF 算法适用于精度与速度均衡的应用环境。

4. 拓展部分 - DELTAS

4.1. 论文阅读与总结

DELTAS（Depth Estimation by Learning Triangulation And Sparse-to-dense）是一种端到端深度估计方法，旨在利用图像间几何关系和稀疏监督信号，生成高质量的稠密深度图。其核心思路是将传统的多视图三角化思想引入神经网络框架中，通过一系列可微模块，将从兴趣点出发的稀疏几何重建，逐步转换为完整的稠密深度估计。该方法主要包括以下三个关键步骤：

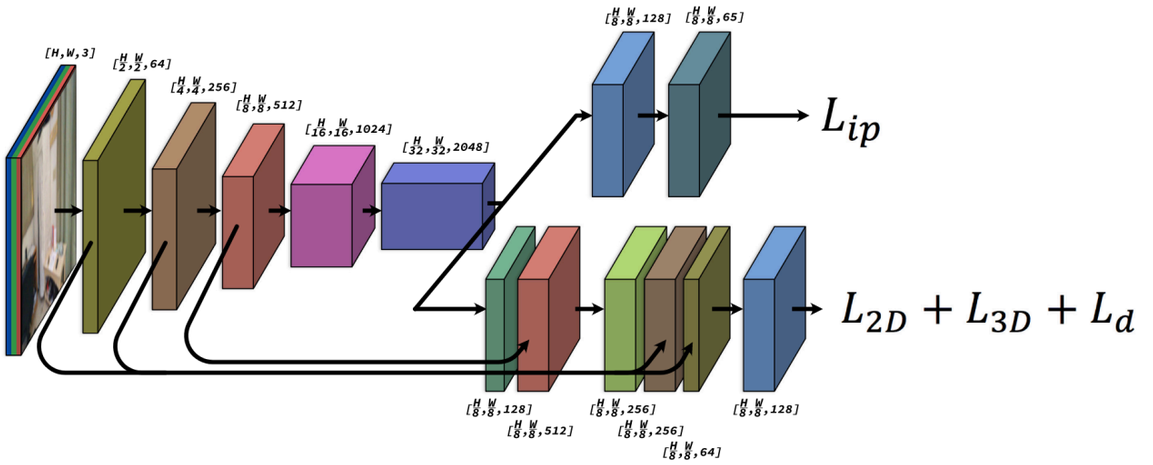
- **特征提取与兴趣点检测**
- **基于几何约束的点匹配与三角化**
- **稀疏深度向稠密深度的转换**



首先，在特征编码与兴趣点检测阶段，将目标图像与多个辅助视角图像输入共享的 RGB 编码器和描述子解码器，提取每张图像的描述子场，同时在目标图像中检测一组兴趣点。接着进入匹配与三角化阶段，利用目标图像兴趣点与其他视角图像之间的相对位姿，确定其在参考图像中的对极线搜索范围，通过采样匹配描述子获取多个视角下的对应点，并使用奇异值分解（SVD）进行三角化，得到稀疏三维点，构建稀疏深度图。最后，在稀疏到稠密深度估计阶段，将稀疏深度图编码结果与 RGB 图像的中间特征图融合，输入至深度解码器中，生成完整的稠密深度图，实现从局部几何恢复到全图重建的端到端深度估计流程。

4.1.1. 特征提取与兴趣点检测

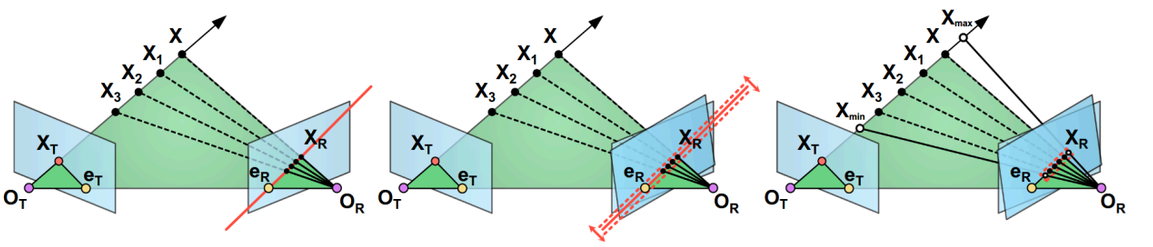
这部分采用了改进的 SuperPoint 架构，使用共享编码器处理全分辨率 RGB 图像，同时输出兴趣点检测和固定长度描述符。与原始 SuperPoint 不同，这里用 ResNet-50 替换了浅层编码器，以获得更强的特征表示能力。编码器输出馈入两个专门的解码器头部：检测头输出兴趣点位置，描述符头输出特征描述。



如图，在共享编码器主干中，输入是 [H,W,3] 的RGB图像。编码器通过逐层下采样，依次输出不同分辨率的特征图。在上方的兴趣点检测分支里，从编码器的最后两层特征 ([H/8, W/8, 128] 和 [H/8, W/8, 65]) 出发，通过简单的卷积层输出兴趣点检测结果。在下方的描述符提取分支里，U-Net风格的解码器架构，通过跳跃连接融合编码器的多尺度特征，从最深层特征 [H/32, W/32, 2048] 开始上采样，逐步融合 [H/16, W/16, 1024]、[H/8, W/8, 512] 等中间特征，通过跳跃连接保留细节信息，最终输出 [H/8, W/8, 128] 的描述符特征图。

4.1.2. 基于几何约束的点匹配与三角化

第二部分采用了基于多视几何理论的高效匹配策略，将传统的暴力搜索转化为几何约束下的精确定位。核心创新在于将极线几何约束与深度学习的软匹配机制相结合，实现了计算效率和匹配精度的双重优化。



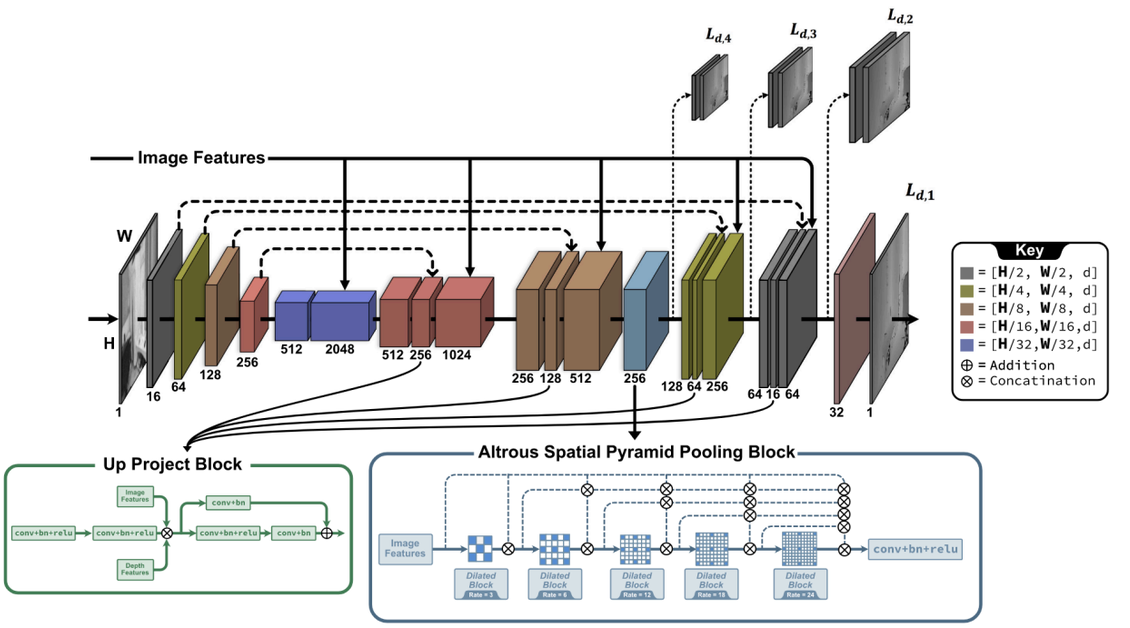
采用多视几何理论，利用基础矩阵 F 确定极线约束关系 $x^T Fx = 0$，将传统的全图搜索转化为沿极线的一维搜索。考虑到实际位姿估计误差，在极线两侧设置小幅偏移；同时将无限延伸的极线限制在可行深度范围内，显著提高计算效率。

通过卷积操作计算锚点图像描述符与辅助图像极线上描述符的互相关图 $C_{j,k}$。使用 softmax 归一化和软 argmax 操作提取匹配位置的质心，实现亚像素级精度的可微分匹配，避免了硬匹配的不连续问题。

采用线性代数方法求解齐次坐标系下的超定方程组 $Az = 0$。创新性地引入基于匹配置信度的权重机制，权重设为各互相关图的最大值，使高质量匹配对三角测量贡献更大。通过可微分 SVD 分解求解，最终获得非齐次 3D 坐标。

4.1.3. 稀疏深度向稠密深度的转换

采用稀疏到密集的网络架构，将前一步骤三角测量得到的稀疏 3D 点转化为稠密深度图。采用图像编码器和深度编码器的并行设计。图像编码器使用完整的 ResNet-50 架构提取丰富的视觉特征，而深度编码器采用通道宽度为图像编码器 1/4 的窄版本。然后将稀疏深度编码器与图像编码器输出特征图拼接；U-Net 风格解码器解码拼接特征图生成稠密深度图像；使用多尺度（4个分辨率）监督训练；加入空洞空间金字塔池化 (ASPP) 模块增强不同感受野的信息融合。



4.2. 实验复现和结果

4.3. 与传统算法对比

5. 遇到的问题和解决方案

5.1. 模块 xfeatures2d.hpp 安装使用问题

在我安装好 OpenCV 然后 make 项目的时候，出现报错：

```
[ 16%] Building CXX object CMakeFiles/ORB.dir/ORB.cpp.o
/root/Final_SLAM/Required_Section/C++/ORB.cpp:6:10: fatal error: opencv2/xfeatures2d.hpp: No such file or directo
ry
    6 | #include <opencv2/xfeatures2d.hpp>
      |          ^~~~~~~~~~~~~~~~~~~~~~~~~~
compilation terminated.
make[2]: *** [CMakeFiles/ORB.dir/build.make:79: CMakeFiles/ORB.dir/ORB.cpp.o] Error 1
make[1]: *** [CMakeFiles/Makefile2:93: CMakeFiles/ORB.dir/all] Error 2
make: *** [Makefile:91: all] Error 2
```

报错说明代码中尝试 `#include <opencv2/xfeatures2d.hpp>`，但编译器在我的系统中找不到这个头文件。之后上网查询并且询问 AI 后了解，这个文件是 OpenCV 的 contrib 模块（也是 `opencv_contrib`）中的一部分，默认情况下 OpenCV 不会包含这些扩展模块，我需要手动安装。初步解决方案如下：

```
cd ~
git clone https://github.com/opencv/opencv_contrib.git

cd opencv
mkdir build && cd build

cmake -D CMAKE_BUILD_TYPE=Release \
      -D CMAKE_INSTALL_PREFIX=/usr/local \
      -D OPENCV_EXTRA_MODULES_PATH=~/opencv_contrib/modules \
      -D OPENCV_ENABLE_NONFREE=ON \
      -D BUILD_EXAMPLES=ON ..
```

```
make -j$(nproc)
sudo make install
sudo ldconfig
```

编译 OpenCV 时要加 `opencv_contrib` 和 `OPENCV_ENABLE_NONFREE=ON`，从源码重新编译 OpenCV 并包含 `opencv_contrib` 模块。这里同时也要注意到，`xfeatures2d.hpp`，是 SIFT、SURF 等算法所在的位置，由于 SURF 算法有专利的限制，所以在某些版本的 OpenCV 是不支持的，因此这里建议安装 OpenCV 的版本为 4.2.0。

但是我这个时候编译完成后再次 make 项目，还是出现了和编译之前一样的错误，这就非常奇怪，所以我再一次上网查找解决方案同时询问 AI 获取求助，然后发现是我需要确保我的项目用的是我刚安装的 OpenCV，因此，我在我的 `CMakeLists.txt` 加上了如下代码（确保找到 `/usr/local/include/opencv4` 和 `/usr/local/lib` 中的 OpenCV）：

```
find_package(OpenCV REQUIRED)
include_directories(${OpenCV_INCLUDE_DIRS})
target_link_libraries(SURF ${OpenCV_LIBS})
```

在做了这个之后有时候系统有可能还是会找默认的 OpenCV（没带 contrib），这时我手动指定 OpenCV 的路径：

```
cmake -D OpenCV_DIR=/usr/local/lib/cmake/opencv4 ..
```

做完以上操作之后，再次 make 项目，问题解决，可以正常运行了。

6. 总结与思考