

天津大学

《计算机视觉》

课程设计大作业



学 院 精仪学院

学 号 3022202283

专 业 测控技术与仪器

班 级 测控四班

姓 名 卢雨萌

2025 年 05 月 27 日

目 录

1 二维图像识别	1
1.1 任务陈述	1
1.2 算法流程	1
1.3 实验环境	1
1.4 算法设计	3
1.5 代码运行方法及参数设定	4
1.6 实验结果	5
1.7 实验分析	7
1.8 实验验证	7
2 三维视觉计算——相机外参及基础矩阵求解	11
2.1 任务陈述	11
2.2 算法流程	11
2.3 实验环境	11
2.4 算法设计	12
2.5 代码运行方法	13
2.6 实验结果	13
2.7 实验分析	15
2.8 实验验证	15
3 三维视觉计算——长度估计	19
3.1 任务陈述	19
3.2 算法流程	19
3.3 实验环境	19
3.4 算法设计	19
3.5 代码运行方法	19
3.6 实验结果	19
3.7 实验分析	20

1 二维图像识别

1.1 任务陈述

对于给定的两张工业图像 0.JPG, 1.JPG，图像及工业相机参数如表 1-1。

表 1-1 图像及相机参数

分辨率	7840x5184
水平/垂直分辨率	300dpi
位深度	24
色彩空间	sRGB
拍摄设备	LEICA M10 MONOCHROM
曝光时间	1/250 秒
焦距	21mm
最大光圈	3.52734375
测光模式	多点、闪光

已知：图像中含有 11 个编码块（每个由 8 个圆点组成）、2 个基准尺和若干外圆点作干扰；

求解：设计算法自动统计图像中的编码块个数。

1.2 算法流程

main.cpp 中使用的算法流程如下：

1. 图像加载与预处理
 - 读取图像
 - 转换为灰度图
 - Gauss 滤波
 - 二值化
2. Hough 变换
 - 霍夫圆检测
 - 霍夫圆标记
3. 基于距离的聚类
 - 广度优先搜索（BFS）
 - 判定是否为编码块
4. 可视化
 - 统计编码块个数
 - 可视化聚类结果

1.3 实验环境

实验环境配置如表 1-2，图 1-1 和图 1-2 所示。

表 1-2 实验环境配置

编程语言	C++
开发工具	Visual Studio 2022
cpp 库	iostream/vector/queue/cmath/string/ctime/algorithm
图像处理库	OpenCV 4.11.0
系统环境	Windows 11
路径设置	G:/cpp/cv_1/Data/
编译环境	Debug x64

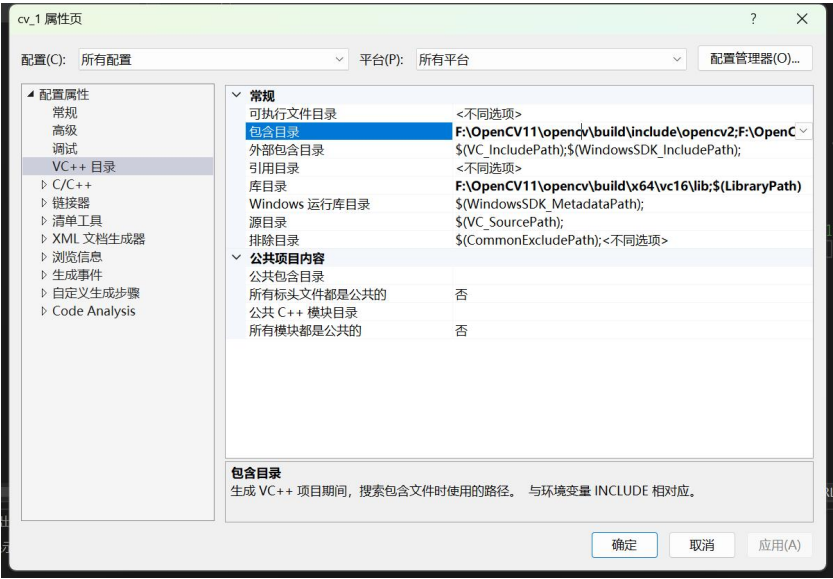


图 1-1 实验环境配置

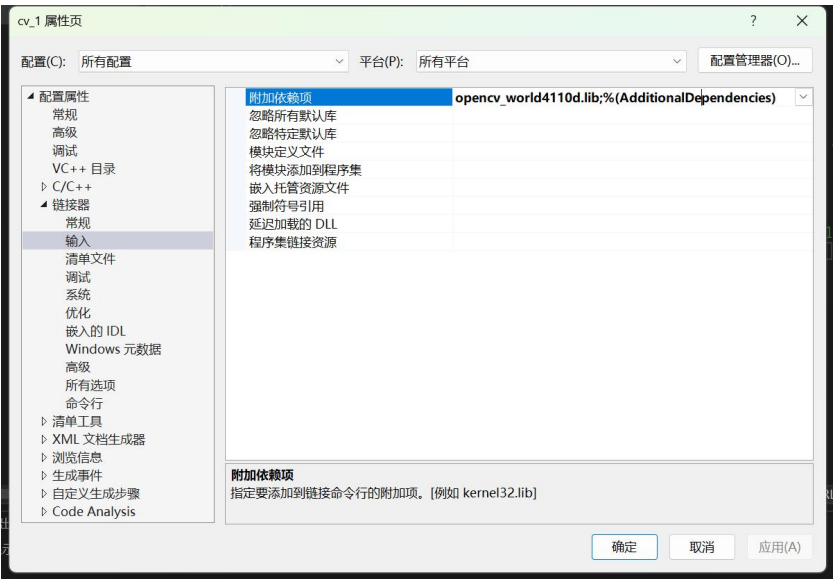


图 1-2 实验环境配置

1.4 算法设计

本节详细介绍 part1.cpp 中核心算法模块的功能与设计思路。

1. HoughTest 函数

该函数对 OpenCV 中的 HoughCircles 进行模拟。

传入的形参为原图 `Mat& image`，圆信息返回值 `vector<Vec3f>& circles`，Hough 空间与原像素空间比率 `dp`，圆心间的最小距离 `minDist`，Canny 高阈值 `param1`，投票阈值 `param2` 以及检测的圆半径范围 `minRadius ~ MaxRadius`。

`image` 在整体设计中，会传入预处理好的二值图，满足已经是灰度图的条件，因此可以直接进行 Canny 边缘检测。其中，高阈值为传入的形参 `param1`，低阈值设为其一半即可。定义一个三维度的 `acc` 投票箱，首先对灰度值为 255 的边缘提取出的白点做坐标变换。`r` 从最小值变到最大值会求出 `rRange` 个圆心坐标，对每一个变换出的圆心坐标及其半径进行投票。因为一个轮廓上的所有点都具有相同的圆心坐标和半径，因此 `acc` 中达到设定投票阈值的组可能为理想的圆心点和半径。随后进行局部极大值抑制，防止因为边缘检测产生的粗边对圆信息造成的干扰；同时进行距离抑制，距离小于 `minDist` 的坐标去除掉，以去除 Hough 变换时产生的误差。最后将满足要求的圆信息加入三通道 `circles` 中。

2. Hough 变换

该函数的功能是在图像中检测出所有小圆点的圆心及半径，并在图像中进行可视化的标记。

函数形参为原图像 `Mat img` 和输出图像 `Mat output`。前者为二值图；后者为绘制圆心与轮廓的输出图像。调用自行编写的 `HoughTest` 进行圆检测，形参的参数需自行设定。多轮调试以及检测到的部分圆半径输出结果，最终选定如表 1-3，解释如下：`dp` 此处设为 1，使得累加器与图像的分辨率保持一致；`minDist` 应选为大概一个圆点的直径大小；Canny 高阈值设为 100；累加器的投票阈值经投票值输出结果选定 20；半径大小经半径输出结果选定 8-16。

表 1-3 Hough 变换参数

<code>dp=1</code>	累加器与输入图像分辨率一致
<code>minDist=20</code>	圆心之间的最小间距
<code>param1=100</code>	Canny 高阈值
<code>param2=20</code>	累加器的投票阈值
<code>minRadius=8, maxRadius=16</code>	圆半径范围

对于检测好的圆，用绿色点绘制圆心，红色线绘制轮廓，用于可视化调试。函数返回值 `vector<Vec3f>` 为 3 通道 float 型 `vector` 的 `vector`，包含所有圆的信息。通道 0 和 1 分别存储霍夫圆的 `x` 和 `y` 坐标，通道 2 存储霍夫圆的半径，上述返回

值均作像素单位下的四舍五入，即返回的参数 `vector` 中，每个 `<Vec3f>` 为：(x, y, r)。

3. 基于距离的聚类

上述 Hough 圆检测的结果包括：编码块中的圆点、基准尺两端的圆点和若干作为干扰的外圆点。为区分出编码块中的圆点，该函数的功能是作基于距离的聚类，将彼此靠近的圆（即在一个编码块中的圆）划分到一个容器中。

函数的形参为 `const vector<Vec3f>& circles` 和 `float radius_thresh`。前者传入 Hough 变换的返回值 `vector<Vec3f>`，包含所有圆的圆心坐标及半径信息，同时设为 `const &` 型常引用访问，限制只读；后者为距离阈值，需自行设置，圆心距离小于此阈值的点才会被划分到一个容器中。函数的返回值 `vector<vector<Point>>` 为圆心 `vector` 的 `vector`，即返回一个容器的 `vector`，其中每个容器包含一个聚类下的所有圆心坐标。

函数首先进行所有检测到的 Hough 圆的圆心坐标提取，将传入的 `vector<Vec3f>` 中的圆心值提取到 `vector<Point> circle_points` 中。设定一 `bool` 型向量，用作记录每个圆是否被访问过，防止重复聚类，其初始值全部设为 `false`。

随后，在所有检测到的 Hough 圆，即 `circle_points` 中进行遍历。遍历方法通常有 BFS 和 DFS 两种。BFS 采用队列的数据结构对节点进行逐层访问，适用于在非加权图中进行搜索，不需要回溯，可以有效防止重复访问，常用于聚类、寻找最短路径等，但有较高的空间复杂度；DFS 采用栈或递归的数据结构对节点进行深层次的访问，适用于拓扑排序、迷宫问题等，但需要进行回溯以确保所有节点都被访问。基于上述，遍历每一个未访问的圆心，并以其为起点进行 BFS，将所有与当前点距离小于 `radius_thresh` 的圆心点加入队列中，即加入此聚类，同时最外层的 `while` 函数保证了扩展这些点的邻居，直到聚类不再增长。向量的二范数等价于欧式空间中的距离，因此距离的运算可通过调用 OpenCV 中的二范数运算实现。将每一个访问过的点的 `visited` 向量设为 `true`，避免重复访问，同时将聚类好的一个 `cluster` 压入待返回的向量中。

4. 可视化

该函数用于统计编码块个数并进行可视化输出。

函数的形参为 `Mat& image` 和 `const vector<vector<Point>>& cluster`。前者传入可视化的底图，后者传入聚类函数的返回值，即每个聚类的圆心坐标。

聚类函数的聚类下可能会有 8 个圆点（即编码块）、1 个圆点（外点干扰）或其他个数的圆点。当且仅当一个聚类包含 8 个圆点的时候，才能判定此聚类为编码块。计算 8 个圆心坐标下的中心坐标，用蓝色圆圈绘制出其位置，黄色字体标识不同编码块编号，并保存图像，同时在 `console` 中输出统计的总编码块个数和上述全部代码的运行时间。

1.5 代码运行方法及参数设定

main.cpp 和 part1.cpp 及头文件 part1.h 放在同一项目中；图像文件 0.JPG 和 1.JPG 置于环境配置中的目录下；在 Debug x64 下编译运行项目。

main.cpp 中需要进行一系列的参数设定。高斯滤波采用 5*5 大小滤波核，x 方向和 y 方向的标准差均设为 2；二值化的阈值经多轮调试设为 80 较为合适；Hough 变换的参数设定在 1.4 节中已经阐释过，在此不做赘述；聚类函数的半径大小经多轮调试，设置为 95.0 比较合适，即一个编码块中两个圆点的最大距离。

上述调试过程将在 1.8 节中进行具体解释。

1.6 实验结果

0.JPG 和 1.JPG 运行结果分别如图 1-3 和图 1-4 所示，console 打印出的编码块个数及运行时间分别如图 1-5 和图 1-6 所示。可以看出两张图片在同一算法下统计到的编码块个数均为 11，与人工验证识别值一致，可见算法具有较高的识别准确性与稳定性。受限于原始图像的高分辨率，单张图像的处理周期较长，平均为 45s 左右。其中 Hough 圆检测时间复杂度为 $O(W*H*R)$ ，其中 W、H 为图像宽高，R 为半径搜索范围大小；BFS 的时间复杂度为 $O(n^2)$ ，其中 n 为圆点个数；编码块个数统计及可视化的时间复杂度为 $O(m)$ ，其中 m 为总聚合类数。囿于知识容量的有限，尚未对时间复杂度进行进一步优化。未来工作中，可通过 KD-Tree 做聚类、GPU 加速 Hough、并行计算等手段做进一步的优化。

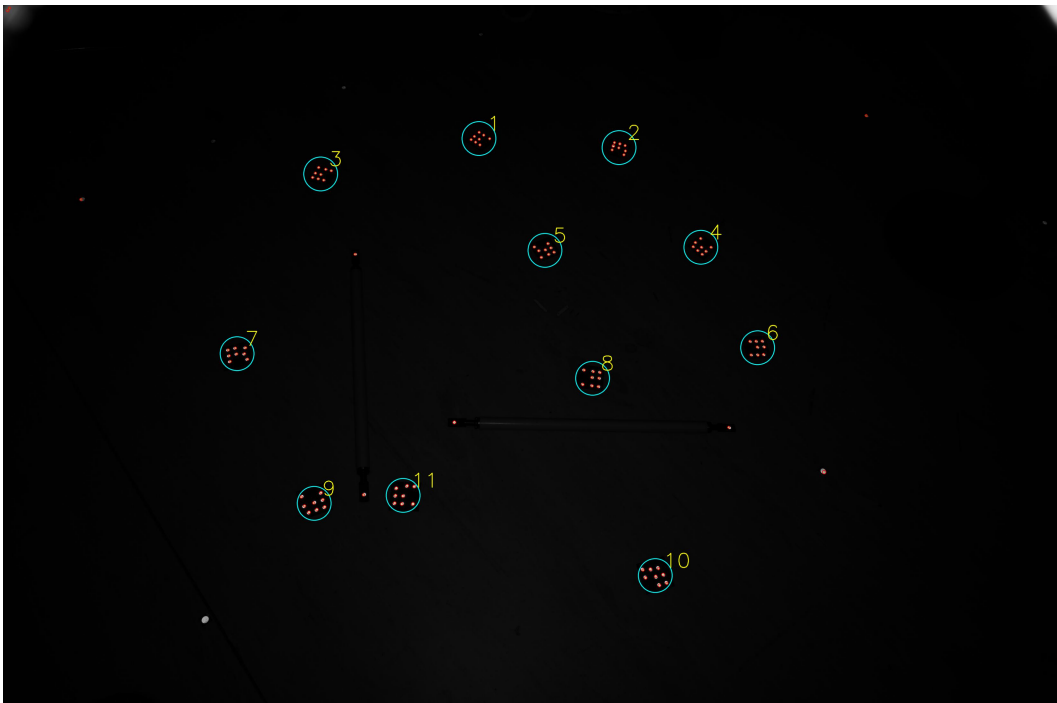


图 1-3 0.JPG 编码块输出图像



图 1-4 1.JPG 编码块输出图像

```
Microsoft Visual Studio 调试
[ INFO:000.715] global_registry_parallel.impl.hpp:96 cv::parallel::ParallelBackendRegistry::ParallelBackendRegistry core
(parallel): Enabled backends(3, sorted by priority): ONETBB(1000); TBB(990); OPENMP(980)
[ INFO:000.715] global_plugin_loader.impl.hpp:67 cv::plugin::impl::DynamicLib::LibraryLoad load C:\WINDOWS\SYSTEM32\open
cv_core_parallel_onetbb4110_64d.dll => FAILED
[ INFO:000.717] global_plugin_loader.impl.hpp:67 cv::plugin::impl::DynamicLib::LibraryLoad load opencv_core_parallel_one
tbb4110_64d.dll => FAILED
[ INFO:000.718] global_plugin_loader.impl.hpp:67 cv::plugin::impl::DynamicLib::LibraryLoad load C:\WINDOWS\SYSTEM32\open
cv_core_parallel_tbb4110_64d.dll => FAILED
[ INFO:000.720] global_plugin_loader.impl.hpp:67 cv::plugin::impl::DynamicLib::LibraryLoad load opencv_core_parallel_tbb
4110_64d.dll => FAILED
[ INFO:000.720] global_plugin_loader.impl.hpp:67 cv::plugin::impl::DynamicLib::LibraryLoad load C:\WINDOWS\SYSTEM32\open
cv_core_parallel_openmp4110_64d.dll => FAILED
[ INFO:000.722] global_plugin_loader.impl.hpp:67 cv::plugin::impl::DynamicLib::LibraryLoad load opencv_core_parallel_ope
nmp4110_64d.dll => FAILED

The number of coding blocks:11
Run time: 40.099 Second

G:\cpp\cv_1\x64\Debug\cv_1.exe (进程 5372)已退出, 代码为 0。
要在调试停止时自动关闭控制台, 请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”。
按任意键关闭此窗口...
```

图 1-4 0.JPG 编码块个数及运行时间 (console)

```
Microsoft Visual Studio 调试
[ INFO:000.757] global_registry_parallel.impl.hpp:96 cv::parallel::ParallelBackendRegistry::ParallelBackendRegistry core
(parallel): Enabled backends(3, sorted by priority): ONETBB(1000); TBB(990); OPENMP(980)
[ INFO:000.757] global_plugin_loader.impl.hpp:67 cv::plugin::impl::DynamicLib::LibraryLoad load C:\WINDOWS\SYSTEM32\open
cv_core_parallel_onetbb4110_64d.dll => FAILED
[ INFO:000.759] global_plugin_loader.impl.hpp:67 cv::plugin::impl::DynamicLib::LibraryLoad load opencv_core_parallel_one
tbb4110_64d.dll => FAILED
[ INFO:000.760] global_plugin_loader.impl.hpp:67 cv::plugin::impl::DynamicLib::LibraryLoad load C:\WINDOWS\SYSTEM32\open
cv_core_parallel_tbb4110_64d.dll => FAILED
[ INFO:000.761] global_plugin_loader.impl.hpp:67 cv::plugin::impl::DynamicLib::LibraryLoad load opencv_core_parallel_tbb
4110_64d.dll => FAILED
[ INFO:000.762] global_plugin_loader.impl.hpp:67 cv::plugin::impl::DynamicLib::LibraryLoad load C:\WINDOWS\SYSTEM32\open
cv_core_parallel_openmp4110_64d.dll => FAILED
[ INFO:000.763] global_plugin_loader.impl.hpp:67 cv::plugin::impl::DynamicLib::LibraryLoad load opencv_core_parallel_ope
nmp4110_64d.dll => FAILED

The number of coding blocks:11
Run time: 48.064 Second

G:\cpp\cv_1\x64\Debug\cv_1.exe (进程 28708)已退出, 代码为 0。
要在调试停止时自动关闭控制台, 请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”。
按任意键关闭此窗口...
```

图 1-5 1.JPG 编码块个数及运行时间 (console)

1.7 实验分析

优点：利用 Hough 圆检测有较高的鲁棒性；BFS 提高识别准确度，同时在搜索过程中避免了回溯。

缺点：main.cpp 中诸多参数需要多轮调试后进行手动设定，麻烦且有误差；对高反光较为敏感，将在 1.8 节中解释；速度慢。

可改进方向：使用第三方库，如 OpenCV；二值化用 Otsu 计算自适应阈值；用 KD-Tree 代替 BFS；并行计算加速等。囿于知识容量有限，上述想法也不一定正确，如有错误，望老师能进行批评指正。

1.8 实验验证

首先 Hough 圆检测用的 OpenCV 中的 HoughCircles，调试结束后重新编写的自定义的 HoughTest 做圆检测。采用 VS2022 自带的 debug 进行调试，遇到的一些问题、验证过程和改进方法如下：

1. 编码块中的圆点不能全部识别出

经 imshow，发现其导致原因是二值化后的圆点有畸变。为此尝试在滤波和二值化之间加入图像增强的算法，以获得更清晰的边界。设置增益值为 150 和 200 的增强图像分别如图 1-6 和图 1-7 所示。对增益 200 的二值化如图 1-8 所示。

不难发现图像增强后的曝光过高，且存在阴影和反光，甚至还不如增强前的效果。因此调试的方向改为二值化和 Hough 的阈值调整。最终发现二值化阈值 80，Hough 投票 20 的时候比较合适。Hough 投票阈值及半径结果均由检测到的半径输出值调试结果设定，console 中的半径输出大多为 7-18，因此最终设定为 8-16。

正确 Hough 圆检测结果如图 1-9 所示。

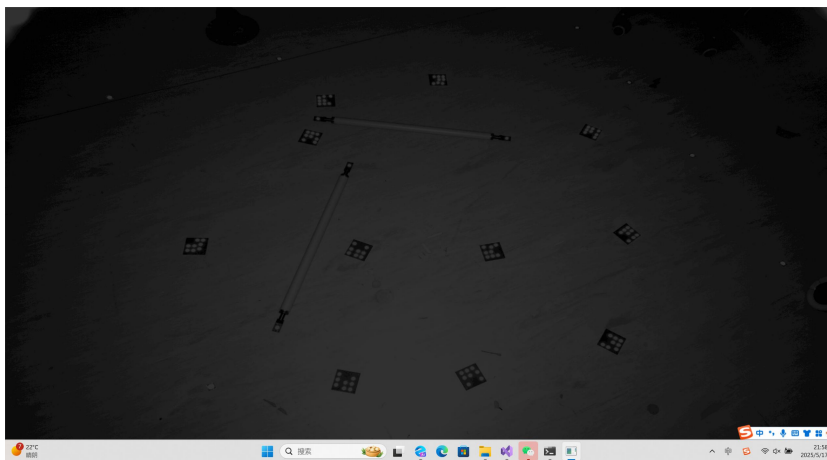


图 1-6 增益为 150 的图像增强

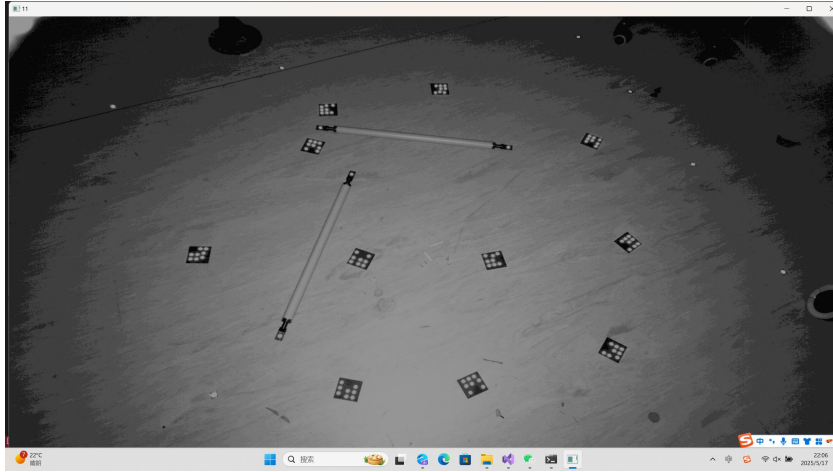


图 1-7 增益为 200 的图像增强

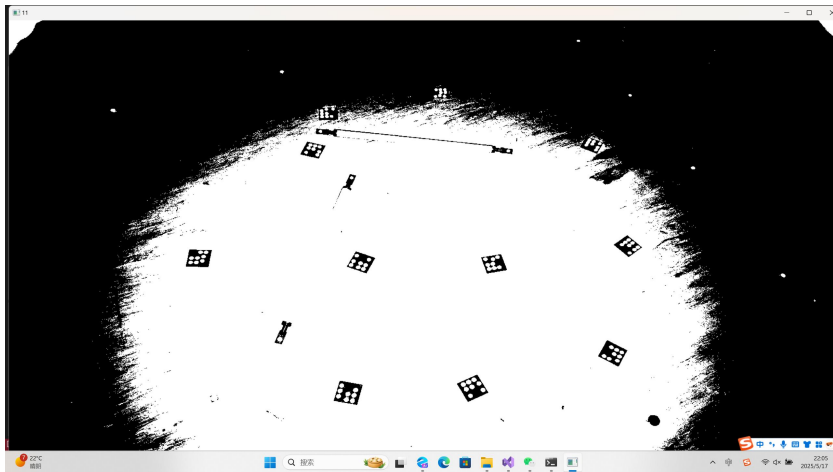


图 1-8 增益为 200 的二值图

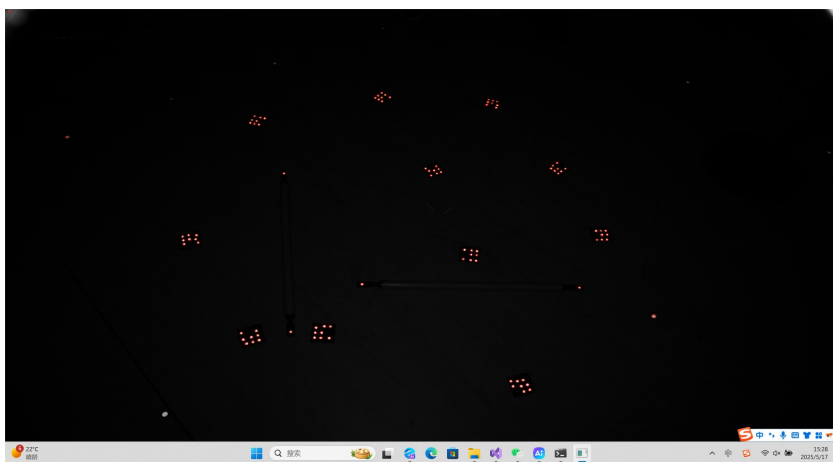


图 1-9 Hough 圆正确检测结果

2. BFS 错误

分步调试时，发现像素点每轮只计算了一次，norm 函数求距离时每轮只有

一个结果，即 BFS 节点未连通，如图 1-10。后增加一 for 循环遍历解决此问题。

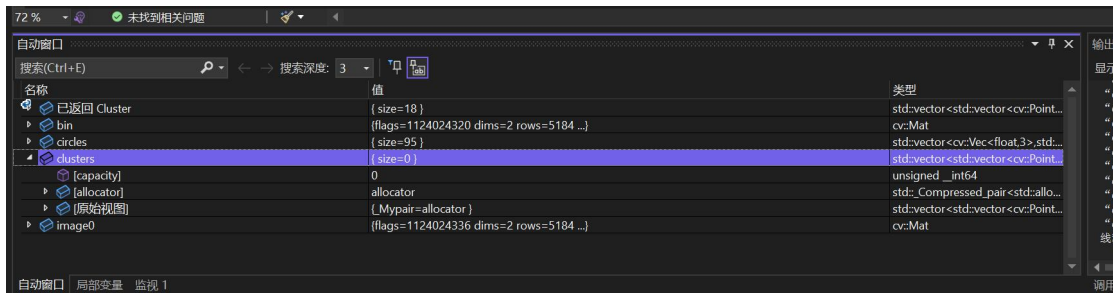


图 1-10 BFS 未连通

3. 错误聚类

Cluster 中的半径阈值设置不合理，起初设置为半径 max 值 20 导致输出值为 0。适当增大设置为 60.0 时，如图 1-11。经观察发现，radius_thresh 需设为最远两圆心之间的距离，因此设置为 $60 \times \sqrt{2}$ 比较合适（约为 84.85），从 85 开始设置，当其达到 95.0 时，可以正确聚类。

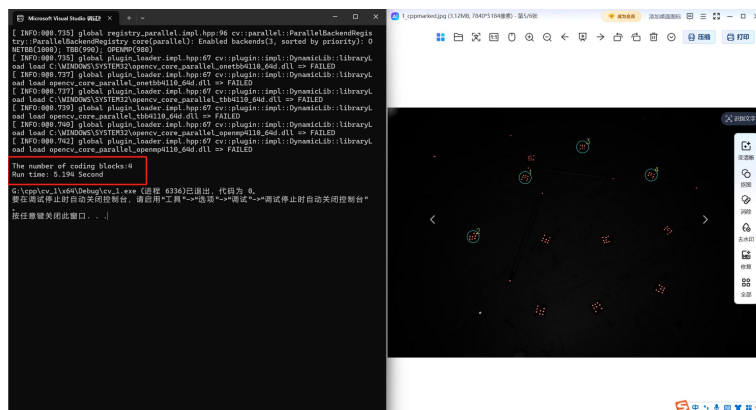


图 1-11 错误聚类

4. 速度过慢

该问题现在也没有得到很好的解决，同时也是最大的问题。上面提到的我是先用 OpenCV 库中的 HoughCircles 函数进行的圆检测调试，以 0.JPG 为例，圆提取的结果即运行时间如图 1-12 和图 1-13 所示，为 5s 左右。可以看出圆点的标记更与其实际位置重合，计算更准确；同时对聚类中的阈值大小更不敏感，鲁棒性都更高。即使计算时间也很长，但相较于自定义的 HoughTest 已经大大降低。

除去 HoughTest 函数外，其他的函数操作对圆信息、聚类信息就进行了多次遍历。这两个容器的值也很大，由 1.pix 文件可以看出前者大小约为三通道的 94，后者大小应至少为 15（11 个编码块+两个基准尺两端，还有一些外点作干扰）。因此，仅对这两个容器进行多次遍历就会导致很长的计算时间。而自定义的 HoughTest 在投票阶段和极大值抑制时，不仅要对整个图像进行遍历，即两个 for 循环，7840*5184 个像素点，还要在此基础上增加一轮半径大小 rRange 中的遍

历。如此大的计算量还要循环两次，这是导致计算时间增长的根本原因。

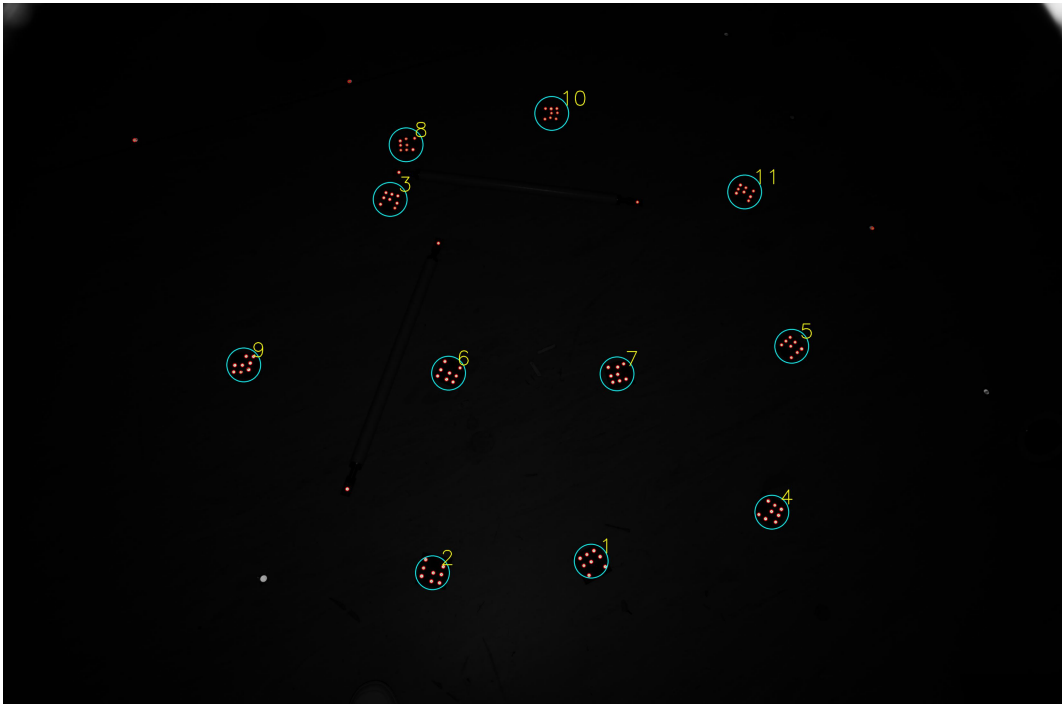


图 1-12 OpenCV 中 HoughCircles 计算结果

```
Microsoft Visual Studio 调试
[ INFO:000.730] global_registry_parallel.impl.hpp:96 cv::parallel::ParallelBackendRegistry::ParallelBackendRegistry core
(parallel): Enabled backends(3, sorted by priority): ONETBB(1000); TBB(990); OPENMP(980)
[ INFO:000.730] global_plugin_loader.impl.hpp:67 cv::plugin::impl::DynamicLib::LibraryLoad load C:\WINDOWS\SYSTEM32\open
cv_core_parallel_onetbb4110_64d.dll => FAILED
[ INFO:000.732] global_plugin_loader.impl.hpp:67 cv::plugin::impl::DynamicLib::LibraryLoad load opencv_core_parallel_one
tbb4110_64d.dll => FAILED
[ INFO:000.733] global_plugin_loader.impl.hpp:67 cv::plugin::impl::DynamicLib::LibraryLoad load C:\WINDOWS\SYSTEM32\open
cv_core_parallel_tbb4110_64d.dll => FAILED
[ INFO:000.734] global_plugin_loader.impl.hpp:67 cv::plugin::impl::DynamicLib::LibraryLoad load opencv_core_parallel_tbb
4110_64d.dll => FAILED
[ INFO:000.735] global_plugin_loader.impl.hpp:67 cv::plugin::impl::DynamicLib::LibraryLoad load C:\WINDOWS\SYSTEM32\open
cv_core_parallel_openmp4110_64d.dll => FAILED
[ INFO:000.737] global_plugin_loader.impl.hpp:67 cv::plugin::impl::DynamicLib::LibraryLoad load opencv_core_parallel_ope
nmp4110_64d.dll => FAILED
The number of coding blocks:11
Run time: 5.209 Second
G:\cpp\cv_1\64\Debug\cv_1.exe (进程 30752)已退出。代码为 0。
要在调试停止时自动关闭控制台，请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”。
按任意键关闭此窗口...
```

图 1-13 OpenCV 中 HoughCircles 计算时间

2 三维视觉计算——相机外参及基础矩阵求解

2.1 任务陈述

对于给定的两张无畸变图像：

已知：准确的特征点对应关系，如 1.pix 文件；相机的内参矩阵 K ；

求解：相机的外参数（3 个欧拉角，3 个平移向量）及基础矩阵。

2.2 算法流程

main.cpp 中使用的算法流程如下：

1. 估计基础矩阵 F
 - 计算两张图的坐标系变换矩阵 T
 - 坐标归一化
 - 八点算法计算矩阵 F_q
 - 逆归一化得 F
2. 验证 F 的准确度
 - 用全部点的极线约束做验证
3. 计算本质矩阵 E
 - SVD 分解，并令 $\Sigma = \text{diag}(1, 1, 0)$
 - 归一化
4. 分解 E 得到外参矩阵 $[R | t]$
 - SVD 分解矩阵 E
 - 分别计算两种可能的 R, t
 - R 行列式大于 0
 - 通过三角化重建选择最佳外参矩阵
5. 计算欧拉角和平移向量
 - R 计算欧拉角
 - t 得到平移向量
6. 通过 OpenCV 重复上述流程，对比两次结果

2.3 实验环境

实验环境配置大体与 Part1 相同，库如表 2-1 所示。

表 2-1 实验环境配置

编程语言	C++
开发工具	Visual Studio 2022
cpp 库	iostream/vector/map/cmath/string/fstream/ssstream
图像处理库	OpenCV 4.11.0

矩阵运算库	Eigen 3.4.0
系统环境	Windows 11
路径设置	G:/cpp/cv_2/Data/data.txt(1.pix 转换为 txt 文档)
编译环境	Debug x64

2.4 算法设计

本节详细介绍 part2.cpp 和 part2_opencv.cpp 中核心算法模块的功能与设计思路。因为 1.pix 中给出的点的位置是准确值，因此不做 RANSAC，可以直接进行所需问题的求解操作。

1. normalize 函数

该函数用作两张图坐标归一化和坐标系变换矩阵的计算。

传入的形参为可改变参数值的 `vector<Point2d>& fig`，返回值为 `Mat` 类型。首先，计算变换后的坐标系原点位置，即原图的重心；其次，使各个像点到坐标原点的均方根距离等于 $\sqrt{2}$ 。计算归一化后的各点坐标位置放回可修改的形参 `fig` 中，同时传回坐标系变换矩阵 T 。

2. calF 函数

该函数用作基础矩阵 F 的估计。

传入的形参为不可修改的两张图的 8 组点，返回值为 `Mat` 类型的 F 矩阵。首先，将点组分别压入两个 `vector` 容器中，方便后续操作；构建 W 矩阵，其中 $W \in R^{8 \times 9}$ ；其次，对 W 进行 SVD 分解，求出向量 f ，且满足 $\|f\|=1$ ， $f \in R^{9 \times 1}$ ；重排 f ，使其成为 3×3 的矩阵；接着，对 f 进行 SVD 分解，并执行秩 2 约束，求出 F_q ；最后，用两张图的坐标系变换矩阵进行逆归一化，求出基础矩阵 F ，并做类似 OpenCV 的 $a_{33}=1$ 的归一化。

3. verifyF 函数

该函数用作基于极线约束条件 $x_2^T F x_1 \approx 0$ 对 F 进行验证，其中 x_1 为 0.JPG 像点位置； x_2 为 1.JPG 像点位置。

函数中调用的 `readPoints` 目的是读取 txt 文件，并将坐标点放入 `Map` 容器中，其中，Key 为像点编号；`vector<Point2d>` 为两张图中该编号像点的位置。因为 txt 中 0.JPG 位置在上，1.JPG 位置在下，所以所有 `vector` 容器中的 0 号元素均为 0.JPG 中对应编号的像点坐标；1 号元素均为 1.JPG 中对应编号的像点坐标。

函数中调用的 `calValue` 目的是对极限约束进行计算，并输出结果做测试，看是否接近于 0。

4. calE 函数

该函数用作本质矩阵 E 的计算。

传入的形参为基础矩阵 F ，返回值为本质矩阵 E 。对 E 进行 SVD 分解，发现对角矩阵并非为严格的 $\text{diag}(1, 1, 0)$ ，但相差不大。经查询可能是由数值计算的偏差所导致，因此在此对对角阵 Σ 进行强制 $\text{diag}(1, 1, 0)$ 操作，并做类似 OpenCV 的归一化。

5. triangulatePoint 函数

该函数用作三角化。

采用线性的方法，分别传入 3×4 大小的两个相机的 M 矩阵（对于 0 相机来说是 $[I|0]$ ，对于 1 相机来说是 $[R|t]$ ，变量类型为 `const Matrix<double, 3, 4>`），和两张图的二维归一化像素坐标；返回值为 Eigen 中的 `Vector4d` 三维齐次坐标。

构建矩阵 A ，用作线性的 DLT 三角化。因为 $Ax = 0$ ，对 A 做 SVD 分解，求出 X ，并做齐次坐标转化。

6. calRT 函数

该函数用作计算外参矩阵 $[R|t]$ ，从而在后续中进一步分解得到 3 个欧拉角和 3 个平移向量。

传入的形参为本质矩阵 `const Mat& E`，两张图的 8 组像点位置，以 `const vector<Point2d>&` 类型传入和 Eigen 中 `Matrix3d` 类型的内参 K 。首先，对 E 进行 SVD 分解；其次，计算出 2 种可能的 R 和 2 种可能的 t ；接着，验证 R 的行列式是否大于 0，如果不是，对 R 进行取反操作；最后，调用 `triangulatePoint` 函数进行三角化，通过重建的方式寻找出正确解。当所有点或尽可能多的点的 $z > 0$ 时，即满足三维点在两个相机之前，满足此要求的外参矩阵 $[R|t]$ 为最佳解。

7. opencvTest 函数

该函数位于 `part2_opencv.h` 中，用 OpenCV 库中的函数进行上述所有操作，和自行编写的函数做对比验证。

2.5 代码运行方法

`main.cpp`，`part2.cpp`，`part2_opencv.cpp` 及头文件 `part2.h`，`part2_opencv.h` 放在同一项目中；两张图的像点编号对应位置坐标转化为 `data.txt` 置于环境配置中的目录下；在 Debug x64 下编译运行项目。

`main.cpp` 中先使用自编写的函数做求解，并输出 3 个欧拉角和 3 个平移向量；随后使用 OpenCV 库函数重复上述过程，对比两个结果做验证。

2.6 实验结果

自编写和 OpenCV 求得的外参数，包括 3 个欧拉角和 3 个平移向量如表 2-2，图 2-1 所示。

表 2-2 欧拉角和平移向量

	自行编写函数	OpenCV 计算结果
偏航角 yaw: Z	105.4080°	106.4210°

俯仰角 pitch: Y	26.0907°	27.6166°
滚转角 row: X	26.2073°	31.1370°
t _x	-0.549876	-0.600493
t _y	-0.649141	-0.740490
t _z	0.525597	0.301800

自编写函数和 OpenCV 得到的基础矩阵 F 如表 2-3 和图 2-1 所示。

表 2-3 基础矩阵 F

	自行编写函数	OpenCV 计算结果
基础矩阵 F	$\begin{bmatrix} -1.237473 \times 10^{-8} & -1.586247 \times 10^{-8} & -0.000135 \\ -2.898190 \times 10^{-8} & -2.177157 \times 10^{-8} & 0.000354 \\ -0.000105 & -8.517207 \times 10^{-5} & 1 \end{bmatrix}$	$\begin{bmatrix} -1.240302 \times 10^{-8} & -1.586195 \times 10^{-8} & -0.000135 \\ -2.900228 \times 10^{-8} & -2.180961 \times 10^{-8} & 0.000354 \\ -0.000105 & -8.517379 \times 10^{-5} & 1 \end{bmatrix}$

```

Microsoft Visual Studio 调试
F:
[-1.237473438127319e-08, -1.586247074529271e-08, -0.0001347202205736666;
-2.898189686749201e-08, -2.177156536695216e-08, 0.0003538280415279251;
-0.0001048234585977487, -8.517206998179815e-05, 1]
E:
[-0.1199147322335027, -0.1630496618463822, -0.5548455823790034;
-0.2596816843552761, -0.1864278725044709, 0.4325711591208011;
-0.4461751746153205, -0.4008301070433239, -0.04622604011840088]
R =
-0.238612 -0.916558 0.320914
0.865821 -0.0511322 0.497734
-0.439793 0.396619 0.805776
t =
-0.549876
-0.649141
0.525597
Euler angles (ZYX: degree) :
Z = 105.408
Y = 26.0907
X = 26.2073
OpenCV计算基础矩阵F
-----OpenCV-----
F =
[-1.240302381094439e-08, -1.586195231089436e-08, -0.000134716480191469;
-2.900227579785362e-08, -2.180960570903511e-08, 0.000354098560537227;
-0.0001047811975993213, -8.517378902528848e-05, 1]
E =
[0.06134033088757655, -0.2373304770038155, -0.5095249027353363;
-0.2502864444786961, 0.00487685203049214, 0.4039436815992276;
-0.4920480007701226, -0.4602520889246416, -0.02269748187500585]
R =
[-0.2504919814924043, -0.8887804005779902, 0.3838270010778764;
0.8499252556955181, -0.0120532365037006, 0.5267653929603707;
-0.463552399345443, 0.4581747690982264, 0.7584161483136223]
Euler angles (ZYX: degree) :
Z = 106.421
Y = 27.6166
X = 31.137
t =
[-0.600493100683324;
-0.7404895868551918;
0.3017999466380912]
OpenCV计算欧拉角和平移向量
G:\cpp\cv_2\x64\Debug\cv_2.exe (进程 34244)已退出, 代码为 0。
要在调试停止时自动关闭控制台, 请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”。
  
```

图 2-1 欧拉角和平移向量计算结果

2.7 实验分析

本设计选择的 8 点编号为：2606, 3308, 3401, 2903, 1, 2, 3, 4；选取原则为分散选取，避免在同一编码块中取多点。

优点：实现了自编写的相机外参数、归一化 8 点算法基础矩阵计算；速度快；极限约束更接近于 0。

缺点：8 点算法的点组选择不同，可能会对结果有一定影响；存在一定的数值计算误差。

2.8 实验验证

采用 VS2022 自带的 debug 进行调试，遇到的一些问题、验证过程和改进方法如下：

1. 极限约束

使用极线约束条件 $x_2^T F x_1 \approx 0$ 对 F 进行验证，其中 x_1 为 0.JPG 像点位置； x_2 为 1.JPG 像点位置。自编写函数求得基础矩阵 F 的验证结果和 OpenCV 求得基础矩阵 F 的验证结果分别如图 2-2，图 2-3 所示。不难看出二者的验证结果大致相同，且自编写函数较于 OpenCV 更接近 0。

```
Point 3103 test result (x2^T * F * x1): 0.00185926
Point 3104 test result (x2^T * F * x1): 0.00180225
Point 3105 test result (x2^T * F * x1): 0.00179831
Point 3106 test result (x2^T * F * x1): 0.00174251
Point 3107 test result (x2^T * F * x1): 0.00174454
Point 3108 test result (x2^T * F * x1): 0.00185341
Point 3201 test result (x2^T * F * x1): 9.73884e-05
Point 3202 test result (x2^T * F * x1): 7.93254e-07
Point 3203 test result (x2^T * F * x1): 1.22891e-05
Point 3204 test result (x2^T * F * x1): 2.67418e-06
Point 3205 test result (x2^T * F * x1): 3.04724e-05
Point 3206 test result (x2^T * F * x1): 3.33417e-05
Point 3207 test result (x2^T * F * x1): 5.39801e-05
Point 3208 test result (x2^T * F * x1): 5.11072e-05
Point 3301 test result (x2^T * F * x1): 0.000106477
Point 3302 test result (x2^T * F * x1): 3.74997e-05
Point 3303 test result (x2^T * F * x1): 6.8361e-05
Point 3304 test result (x2^T * F * x1): 0.000110467
Point 3305 test result (x2^T * F * x1): 8.44412e-05
Point 3306 test result (x2^T * F * x1): 0.000168115
Point 3307 test result (x2^T * F * x1): 0.000145798
Point 3308 test result (x2^T * F * x1): 5.5636e-07
Point 3401 test result (x2^T * F * x1): 8.3629e-05
Point 3402 test result (x2^T * F * x1): 1.04546e-05
Point 3403 test result (x2^T * F * x1): 8.72962e-05
Point 3404 test result (x2^T * F * x1): 0.000159374
Point 3405 test result (x2^T * F * x1): 8.45218e-05
Point 3406 test result (x2^T * F * x1): 0.00022141
Point 3407 test result (x2^T * F * x1): 0.000151255
Point 3408 test result (x2^T * F * x1): 8.01775e-05
```

图 2-2 自编写函数极线约束

```

Point 3103 test result (x2^T * F * x1): 0.00186225
Point 3104 test result (x2^T * F * x1): 0.00180515
Point 3105 test result (x2^T * F * x1): 0.00180119
Point 3106 test result (x2^T * F * x1): 0.00174531
Point 3107 test result (x2^T * F * x1): 0.00174733
Point 3108 test result (x2^T * F * x1): 0.00185636
Point 3201 test result (x2^T * F * x1): 9.6622e-05
Point 3202 test result (x2^T * F * x1): 1.57697e-06
Point 3203 test result (x2^T * F * x1): 1.30547e-05
Point 3204 test result (x2^T * F * x1): 1.91164e-06
Point 3205 test result (x2^T * F * x1): 2.97006e-05
Point 3206 test result (x2^T * F * x1): 3.25945e-05
Point 3207 test result (x2^T * F * x1): 5.32207e-05
Point 3208 test result (x2^T * F * x1): 5.03199e-05
Point 3301 test result (x2^T * F * x1): 0.000106579
Point 3302 test result (x2^T * F * x1): 3.7503e-05
Point 3303 test result (x2^T * F * x1): 6.84059e-05
Point 3304 test result (x2^T * F * x1): 0.000110575
Point 3305 test result (x2^T * F * x1): 8.451e-05
Point 3306 test result (x2^T * F * x1): 0.000168317
Point 3307 test result (x2^T * F * x1): 0.000145959
Point 3308 test result (x2^T * F * x1): 5.07409e-07
Point 3401 test result (x2^T * F * x1): 8.37117e-05
Point 3402 test result (x2^T * F * x1): 1.04597e-05
Point 3403 test result (x2^T * F * x1): 8.73992e-05
Point 3404 test result (x2^T * F * x1): 0.000159563
Point 3405 test result (x2^T * F * x1): 8.4614e-05
Point 3406 test result (x2^T * F * x1): 0.000221679
Point 3407 test result (x2^T * F * x1): 0.000151429
Point 3408 test result (x2^T * F * x1): 8.03008e-05

```

图 2-3 OpenCV 极线约束

2. 外参矩阵的选择

进行三角化的三维重建，求得点在世界坐标系下的三维坐标。当且仅当某组外参矩阵 $[R|t]$ 三角化的全部结果在空间中的 z 坐标均大于 0 时，即所有像点的空间坐标都位于两个相机之前，则该组结果为正确的外参矩阵 $[R|t]$ 。三角化的点为整个 1.pix 中的 92 个点，计算结果如图 2-4 所示。

```

front_count:
0
front_count:
92
front_count:
0
front_count:
0
Points in front of both cameras: 92 / 92

```

图 2-4 四组 $[R|t]$ 三维重建结果

可以发现，仅第二组 $[R|t]$ 满足所有点均在两个相机之前，因此该组 $[R|t_2]$

为最佳外参矩阵，即所需要的结果。该组 $[R|t]$ 为：

$$\begin{cases} R = U W V^T \\ t = -U_3 \end{cases}$$

3. 旋转矩阵 R 和平移向量 t 的合理性验证

合理的旋转矩阵 R 应满足 $R^T R \approx I$ ，且有 R 的行列式， t 的范数均为 1。

上述验证结果如图 2-5 所示。

```
R^T * R =
      1 -8.32667e-17  1.66533e-16
-8.32667e-17      1  1.66533e-16
 1.66533e-16  1.66533e-16      1
det(R) = 1
||t|| = 1
```

图 2-5 旋转矩阵和平移向量的合理性验证

4. SVD 方法的选择

OpenCV 和 Eigen 都提供矩阵的 SVD 选择，需要注意的是，在对 W 进行 SVD 时应选择 ComputeFULLV，否则函数为节约算力，加快速度将不能得到所需要 9×9 矩阵的全部运算结果，会导致求解出现问题，如极线约束结果过高，所有点三维重建不能全部位于两个相机之前等。如图 2-6 为修正之前， V 仅能计算到 9×8 。

```
Microsoft Visual Studio 调试
nmp490_64d.dll => FAILED
U:
-0.522642  0.802576  0.0385721  0.247966  0.129014  0.00360672  0.0465263  0.0303228
-0.0310288 -0.0247886 -0.0197511  0.14109 -0.0867897  0.822679 -0.138035 -0.524156
-0.141269  0.0336488  0.438716 -0.179894 -0.35352  0.0292808 -0.753856  0.244836
-0.561228 -0.548482  0.190675  0.307744  0.479074  0.0404998 -0.060987  0.135116
-0.362852 -0.003407 -0.41812 -0.78383  0.236627  0.0776829 -0.115973 -0.0603056
-0.222943 -0.0528947  0.635563 -0.329717 -0.167589 -0.175722  0.4124 -0.453659
-0.0506966 -0.00869029  0.151766 -0.181236 -0.183769  0.510551  0.460821  0.659307
-0.454988 -0.224515 -0.409262  0.188651 -0.709567 -0.152031  0.104328 -0.0448478
V:
 0.348781  0.0487431  0.705475 -0.0997265  0.319265  0.126498 -0.383992  0.320836
 0.582386 -0.78574 -0.105062  0.00197985  0.00649217 -0.033478  0.132862 -0.116422
-0.018916  0.0838945 -0.0444297 -0.0614149  0.663727 -0.182687  0.0235427 -0.480821
-0.731809 -0.60407  0.225492 -0.106121  0.131525  0.029323 -0.103254  0.0923599
 0.0375694 -0.0320987 -0.655168 -0.255184  0.276237  0.144116 -0.498972  0.396346
-0.0283948 -0.0415996 -0.0529323  0.659582  0.1887 -0.328725 -0.426643 -0.204429
-0.0223841 -0.0549177 -0.0515686  0.683465 -0.0243155  0.375521  0.0754418  0.318004
 0.0214943 -0.0502221  0.0609633 -0.0844783 -0.57111 -0.24391 -0.579659 -0.211494
-0.00980986 -0.000136117  0.00478711 -0.0242573 -0.0390353  0.786959 -0.225809 -0.546857
A:
239.315
195.076
127.041
24.2907
16.8186
1.46964
0.198294
0.0244784
```

图 2-6 未完全计算的 SVD

修正为 ComputeFULLV 之后的计算结果如图 2-7，为 9×9 矩阵。

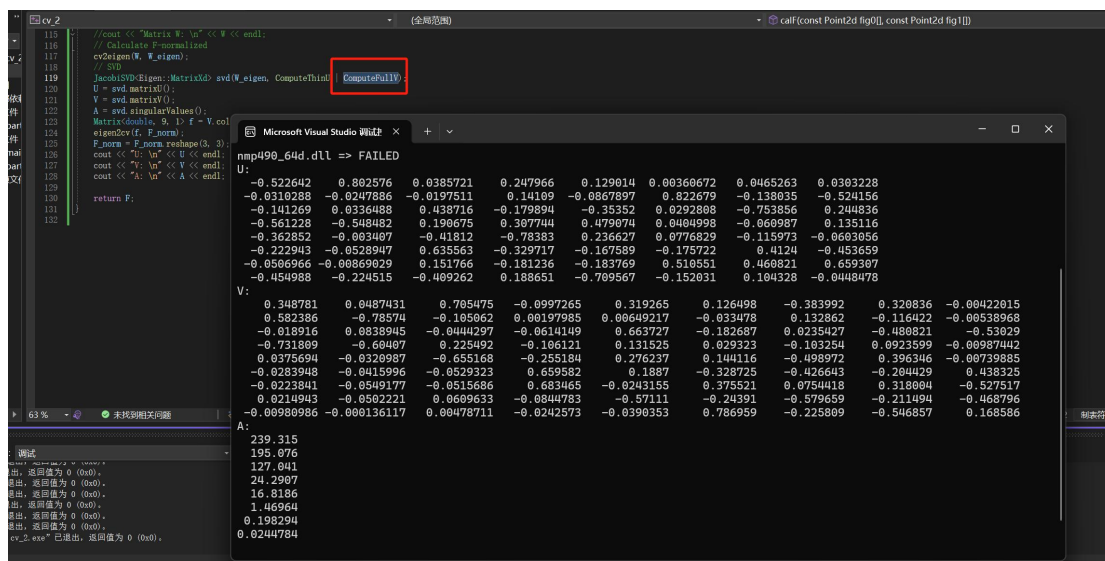


图 2-7 修正后的 SVD

最后出于算法时间复杂度考虑,采用 OpenCV 中的 `SVD::compute` 代替 Eigen 中的 `JacobiSVD` 做 SVD 分解,可以避免 Mat 和 Martrix 变量类型之间的来回转化。仅在后续不得不使用 Eigen 库进行一系列矩阵运算时,采用 `JacobiSVD` 做分解。

5. 本质矩阵 E 的数值误差

由表 2-2 和图 2-1 可以看出,自编写算法与 OpenCV 计算的本质矩阵 E 和欧拉角中的滚转角存在一定的偏差。当使用 OpenCV 计算 F, R, t , 仅替换求解 E 的函数为自编写的 `calE` 后,求解结果与完全使用自编写函数基本一致。因此可以确定造成偏差的原因是因为 `calE` 函数。

经调试多轮无果之后,我进行了一些调研。发现 OpenCV 计算本质矩阵的方式与计算基础矩阵类似,即直接采用对匹配点的归一化 8 点算法;而自编写的 `calE` 计算方式是用 $K^T F K$ 的方式计算,因此会导致一定的数值偏差。

自编写算法和 OpenCV 求得的所有参数结果(包括基础矩阵 F , 本质矩阵 E , 旋转矩阵 R , 平移向量 t , 3 个欧拉角)均可在图 2-1 中找到。中间任意过程的变量可以通过解注释掉代码中的 `cout` 得到。

3 三维视觉计算——长度估计

3.1 任务陈述

对于给定的工业图像，1、2 号点为第一把基准尺，3、4 号点为第二把尺的长度。

已知：第一把基准尺长度为 1000mm；

求解：第二把尺子长度。

3.2 算法流程

main.cpp 中使用的算法流程如下：

1. 将两张图各 4 点坐标置于 vector 容器中
2. 三维解算
 - 构建两个相机矩阵
 - 三角化
 - 计算长度

3.3 实验环境

实验环境在上一问的基础上进行，即与第 2 章相同。

3.4 算法设计

本节详细介绍 calLength.cpp 的功能与设计思路。

1. calLength 函数

该函数的功能是计算尺子长度。

传入的形参为上一问求得的相机外参矩阵 $[R | t]$ 和以 vector<Vector2d> 类型传入的两张图片中的 1、2、3、4 号点。为直接调用上一问编写的三角化函数，先对传入形参的类型进行转化，随后调用上一问编写的 triangulatePoint 函数进行三角化，求出三维空间中点的齐次坐标。最后计算 1、2 号点和 3、4 号点的距离，根据比例解算第二把尺子的长度。

3.5 代码运行方法

main.cpp, part2.cpp 和 calLength.cpp 及头文件 part2.h, calLength.h 放在同一项目中；在 Debug x64 下编译运行项目。

3.6 实验结果

使用 OpenCV 得到的外参矩阵和自编写函数得到的外参矩阵调用 calLength 计算结果如图 3-1 所示。可以发现存在约 45mm 的偏差，在第 2 章中也解释过了是由于本质矩阵 E 的计算方法不同，而导致的外参矩阵不同，从而影响的三维解算的最终结果。

```
OpenCV Length of the 2nd ruler: 1000.62 mm  
Length of the 2nd ruler: 955.569 mm
```

图 3-1 尺子长度估计

3.7 实验分析

优点：代码层次清晰，易于调试；可多组点批量测量。

缺点：对外参矩阵的准确度要求较高。