****

机器视觉创新实践报告书

**题目：静脉识别算法实验研究**

**学 院 吴贤铭智能工程学院**

**专 业 21机器人珠峰班**

**学生姓名 张至睿**

**学生学号 202130462158**

**指导教师 赵俊红**

**课程编号**

**课程学分**

**起始日期**

|  |  |
| --- | --- |
| 教  师  评  语 | 教师签名：  日期： |
| 成  绩  评  定 |  |
| 备  注 |  |

1. **选题背景**

本课题旨在研究手指静脉、手掌静脉和手背静脉的生物特征识别技术，并比较分析不同方法的识别技术及其优缺点。最终目标是在生物特征匹配上实现高效、准确的识别效果。随着社会的发展，身份认证方式也在不断演变，我们始终追求更加便捷、安全可靠的身份认证方法。生物特征识别技术利用了个体特有的生理或行为特征，具有安全、可靠、方便等独特优势。有效的生物特征需要满足普遍性、唯一性、可采集性和稳定性四个条件。

**1.1 手指静脉识别技术**

手指静脉识别技术利用近红外光透射手指，获取手指静脉图像进行身份识别。手指静脉的分布具有唯一性和稳定性，且由于静脉位于皮肤下层，难以伪造。

手指静脉识别技术具有高安全性，手指静脉位于体内，难以通过外部手段伪造或复制，安全性极高。高稳定性，静脉图像不会因年龄、外部环境变化而改变，长期保持稳定。高识别率，手指静脉识别具有较低的错误率和拒绝率，识别精度较高。非接触式识别，可以通过非接触方式进行识别，减少交叉感染风险，提升用户体验。

手指静脉识别技术在金融、安防、医疗等领域有广泛应用。例如，银行的ATM机使用手指静脉识别验证用户身份，确保交易安全；在高安全性要求的区域，通过手指静脉识别控制访问权限；在医疗领域，通过手指静脉识别进行患者身份验证，确保医疗信息的准确性和安全性。

**1.2 手掌静脉识别技术**

手掌静脉识别技术通过近红外光照射手掌，检测血液中的氧气水平以获取静脉图像进行身份识别。手掌静脉覆盖面积大，信息量丰富，识别准确率高。

手掌静脉识别技术具有高安全性，手掌静脉位于皮肤下，难以伪造，安全性极高。大面积图像，手掌静脉覆盖面积大，信息量丰富，识别准确率更高。高稳定性， 静脉图像不会因外部环境和年龄变化而发生显著变化，具有较高的稳定性。舒适性，手掌识别过程简单、快速，用户体验良好。

**1.3 手背静脉识别技术**

手背静脉识别技术通过近红外光照射手背，获取手背静脉图像进行身份识别。手背静脉识别技术与手指静脉和手掌静脉识别技术相比，识别过程更为隐蔽，用户无需直接接触设备。

手背静脉识别具有高安全性，手背静脉位于皮肤下层，不易被伪造，安全性高。隐蔽性好，手背识别过程隐蔽，适合在需要保护隐私的场景中使用。高稳定性，静脉图像不易受外部环境影响，具有较高的稳定性。舒适性，用户无需直接接触识别设备，识别过程舒适。

**1.4 研究目标**

本课题聚焦于指静脉和手掌静脉识别领域，通过设计图像处理和比对算法，对原始图像进行处理和比对，验证这些算法的正确性和可靠性。生物特征识别的技术难点在于每次认证需要专门的设备检测，检测到的信息需经过复杂的数字处理才能用于数据库比对。这涉及到识别的正确率、成本、易用性和部署便捷性等问题。虽然科技进步已使生物特征识别技术得到广泛应用，但处理和识别算法的改进依然是该领域持续发展的关键。

1. **方案论证（设计理念）**

本项目需要对已经采集的手掌和手指图像进行图像处理和算法匹配。首先对图像带有重要信息的部分进行截取，然后经过增强和滤波等一系列操作，凸显手掌静脉和手指静脉的纹路。随后把所有的静脉纹路进行相互间的匹配，以检验类内和雷剑的匹配度，观察其是否能够进行明显的区分，并以此计算匹配正确率。

本项目算法主要流程如下：识别图像边缘轮廓，并且找出中线位置；调整图像选准，方便后续截取；直方图均衡化增加图像对比度，截取ROI区域；采用CLAHE自适应直方图均衡化、二值化图像、Frangi滤波等算法提取和增强静脉纹路；最后次啊用而治文禄特征匹配进行图像匹配。

本项目选择成熟的Opencv库作为主要开发工具，算法成功通过Cmake部署到Windows 11系统中。

1. **过程论述**

**3.1 手指静脉识别**

**3.1.1 手指图像预处理**

图像预处理的主要目的是增强图像的对比度，突出有用的信息，同时去除干扰噪声，并且截取用于识别的ROI区域。

图像预处理部分主要包括对图像进行滤波、裁剪、边缘检测、旋转调整，最后嫩巩固截取有效的ROI区域。

首先对图像进行高斯滤波处理，高斯模糊是一种图像平滑技术，用于减少图像中的噪声和细节。使用高斯滤波的目的在于对图像进行轻度去噪。这对于高斯滤波的参数调试有一定要求，如果平滑效果过于明显，可能会使有价值的静脉纹路信息也被平滑掉。高斯滤波的对比效果如下。

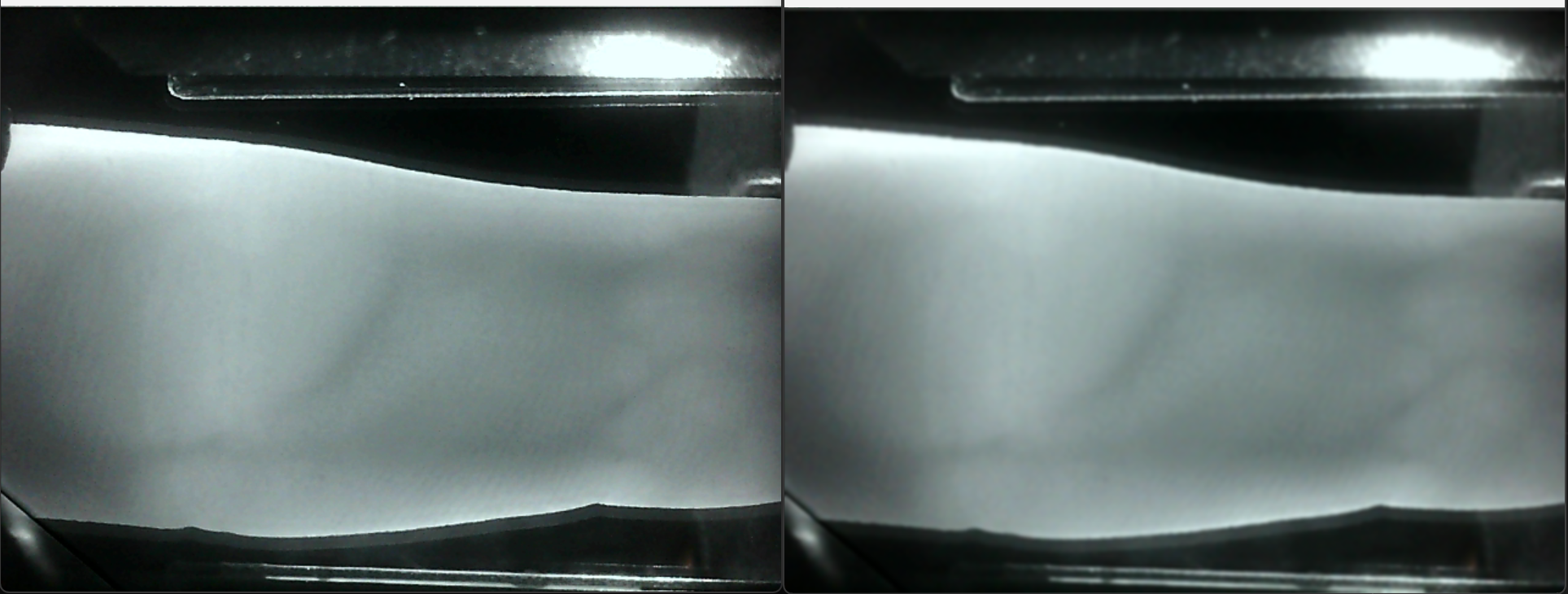


图1. 高斯滤波处理

随后就是对图像进行旋转调整，以便于后续进行ROI区域截取。而进行旋转调整首先需要通过手指边缘检测确定手指中线，从而获取旋转角度。选择使用Sobel算子对图像的竖直方向进行梯度检测，提取梯度图像。随后进行二值化处理、开运算、腐蚀三个步骤，进一步去除图像噪声，获得更理想的边缘灰度图。



图2. 边缘灰度图

随后通过对图像进行从上到下的像素遍历，以及从下到上的像素遍历，能够确定边缘线。对上下边缘线取平均值能够得到图像的中线。通过对中线的最小二乘法拟合，获取拟合直线。从而实现图像旋转调整。

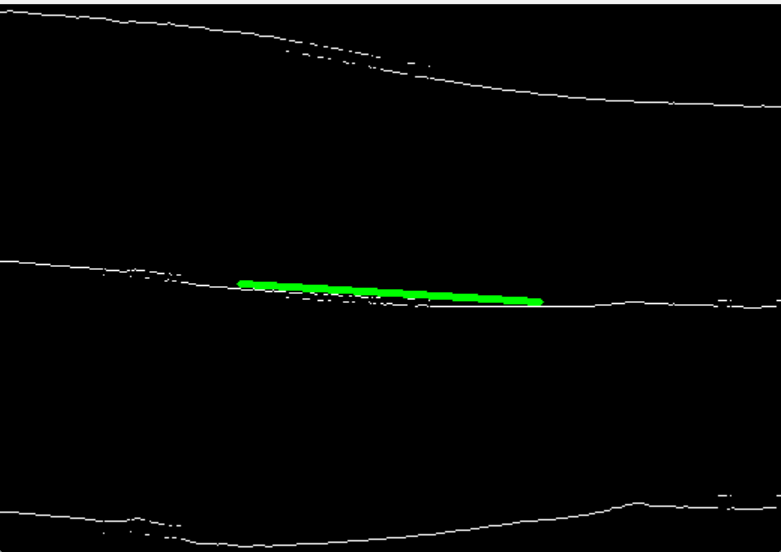


图3. 图像边缘以及中线拟合



图4. 图像旋转前后

图像预处理的最后一步就是进行ROI区域截取。这一步主要通过手指的上下边缘的极值确定上下截取范围，同时通过对指关节最亮的部分进行识别，确定左右截取范围。由于在本次数据集中右侧关节的亮度没有明显增强，因此在确定左右边界时有时只能检测到左边的点。为了尝试改善这个问题，在进行左右截取区域识别之前对图像进行一次直方图均衡化处理由于数据集手指的位置区域统一，因此这种情况可以直接向右选取一定区域作为ROI截取，最终也能够完成任务。

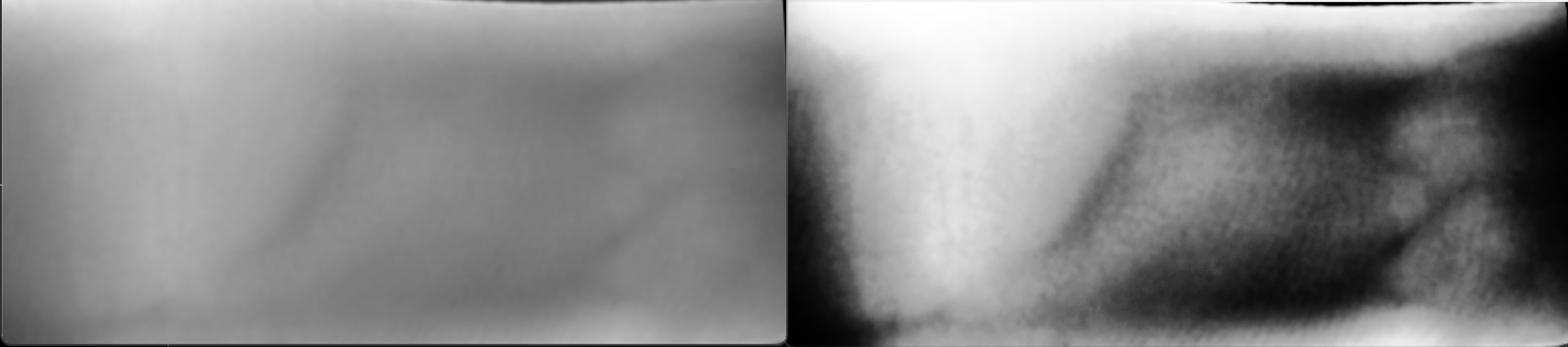


图5. 直方图均衡化处理

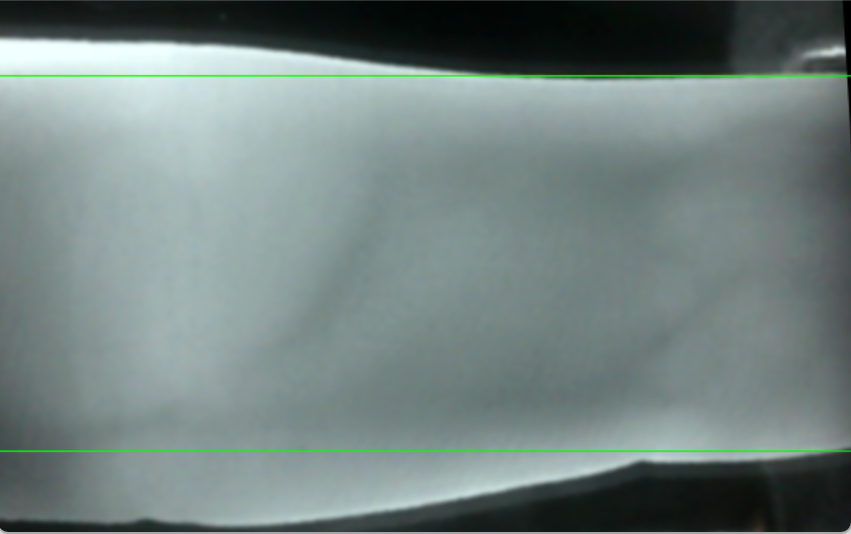


图6. 上下ROI区域截取



图7. 左右ROI区域截取

至此完成ROI区域截取，完成图像预处理阶段。

**3.1.2 手指图像静脉纹路增强**

Frangi滤波（Frangi Filter）是一种用于增强血管状结构（或线状结构）的图像处理技术，广泛应用于医学图像分析、血管分割和其他需要提取细小线状特征的图像处理中。该方法基于多尺度Hessian矩阵分析，能够在不同尺度上识别和增强细长结构，如血管、纤维等。

首先进行Hessian矩阵的计算。对于每个像素位置，Hessian矩阵是图像的二阶偏导数矩阵，定义如下：

其中，是图像的像素值，是尺度参数，用于控制Hessian矩阵计算的卷积核的大小。

随后对矩阵的特征值进行分析。通过特征值分解计算Hessian矩阵的特征值和（假设）。这些特征值提供了局部曲率的信息。

然后利用特征值来计算线性结构的响应函数。Frangi滤波器的响应函数如下：

其中， 用于度量线性结构的强度，用于度量特征强度的总量。

在线性结构的基础上对不同尺度上的响应进行组合，定义最终的Frangi响应函数为：

其中， 和 是常数参数，用于调节对线性结构和特征强度的响应。

最后进行多尺度增强。对不同的尺度进行组合，得到最终的Frangi增强结果：

其中， 是多个尺度上的标准差，取其响应的最大值。

结合论文和公式，最终实现了Frangi滤波。

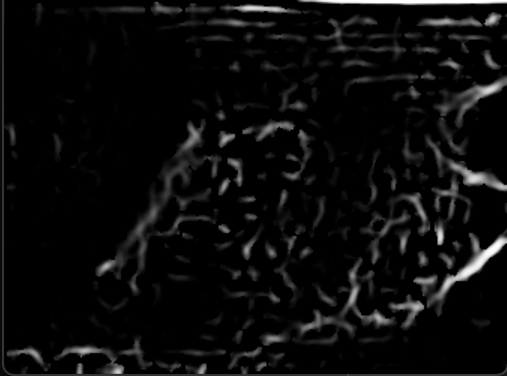


图8. Frangi滤波进行血管增强

Frangi滤波能够增强图像中的血管信息，但过多的血管结构也会影响识别和匹配的结果。因此要提取显著的特征，只保留血管较大的结构，去除图像中的小面积区域。

CLAHE算法只对图像中的小区域进行操作，而不是对整个图像进行操作。该算法将单独计算每个切片的对比度变换函数。每个切片对比度都得到增强，以便输出区域的直方图与分布值指定的直方图大致匹配。然后，使用双线性插值组合相邻切片，以消除人为诱导的边界。对比度（尤其是在同质区域）可以限制以避免放大图像中可能存在的任何噪声。

最终的实现中静脉图像经过两次增强，一次对比度限制自适应直方图均衡 （CLAHE），再使用一次frangi滤波处理一次得到最终的特征提取效果图。值得注意的是，图像增强的过程中都将原本8bit的图像提升精度为16bit 以取的更精细的静脉特征。 上述滤波处理后的图像再经过二值化和去噪就能进行匹配运算了。

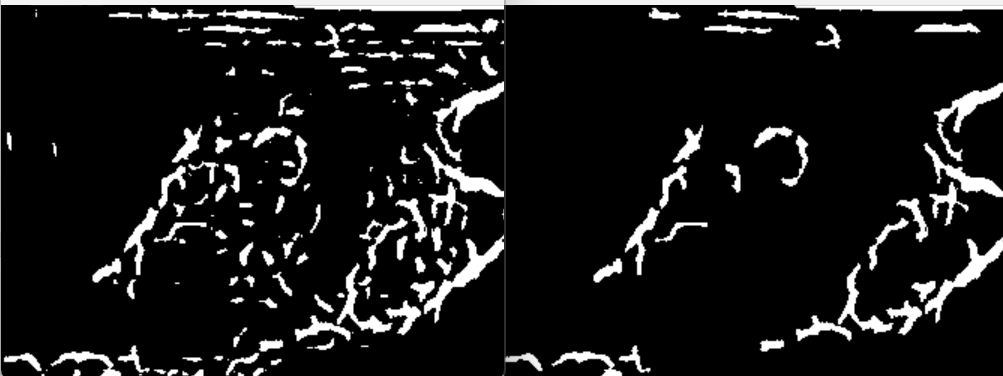


图9. 最终的手指血管结构

**3.1.3 手指图像匹配**

图像匹配是进行静脉识别的最后一步，不同的提取特征类型往往有其对应的匹配方法，而合适的匹配方法能够极大提高识别精度。现有静脉图像匹配算法常采用某种距离去度量类内样本匹配分数和类间样本匹配分数。 针对图像匹配需与提取特征，本文选取了二值纹路特征匹配进行研究。匹配的精确标准是：计算两张纹理前景图像的交集（重叠）像素点数与两张纹理前景图像的并集像素点数之比，这个比值即为两张图像的匹配分数。

在课设提供的数据集中，同一个文件夹内的10张图片进行两两匹配，按照课设指导书中提供的类内样本和类间样本的距离分布图所示，选取0.15作为是否匹配成功的阈值，最终各类的类内匹配率如下图所示：

|  |  |
| --- | --- |
| 图10. 样本类内与类间特征的距离分布 | 图 11. 类内匹配率 |

类内匹配率基本在40%以上，而在程序测试中得出的类间匹配率最大也不超过30%，绝大部分在0-10%这个区间内，类内和类间的匹配准确率有一定差距，然而有些类内匹配率以及类间匹配率差距不大，分析原因是因为某几组数据的多个数据采集不稳定，导致其特征识别不够准确，其类内匹配率过低。但总的来说，通过选择合适的匹配率阈值，仍然能够胜任匹配任务。

**3.2 手掌静脉识别**

**3.2.1 手掌图像预处理**

手掌静脉识别与手指静脉识别最大的不同在图像预处理上，特别是截取ROI的方法与手指有很大不同。手掌图像预处理主要的流程为：手掌的前后背景分割、舍弃手腕部分图像、识别掌心的某一特定区域、识别出手掌轮廓的特征点、确定手指间缝靠近掌心的特征点、计算掌心区域。

首先使用二值化分割出亮度更高的手掌区域，接着使用轮廓查找其中面积最大的区域即是掌心区域。接着计算手掌轮廓点集的最大内切圆。如图所示：

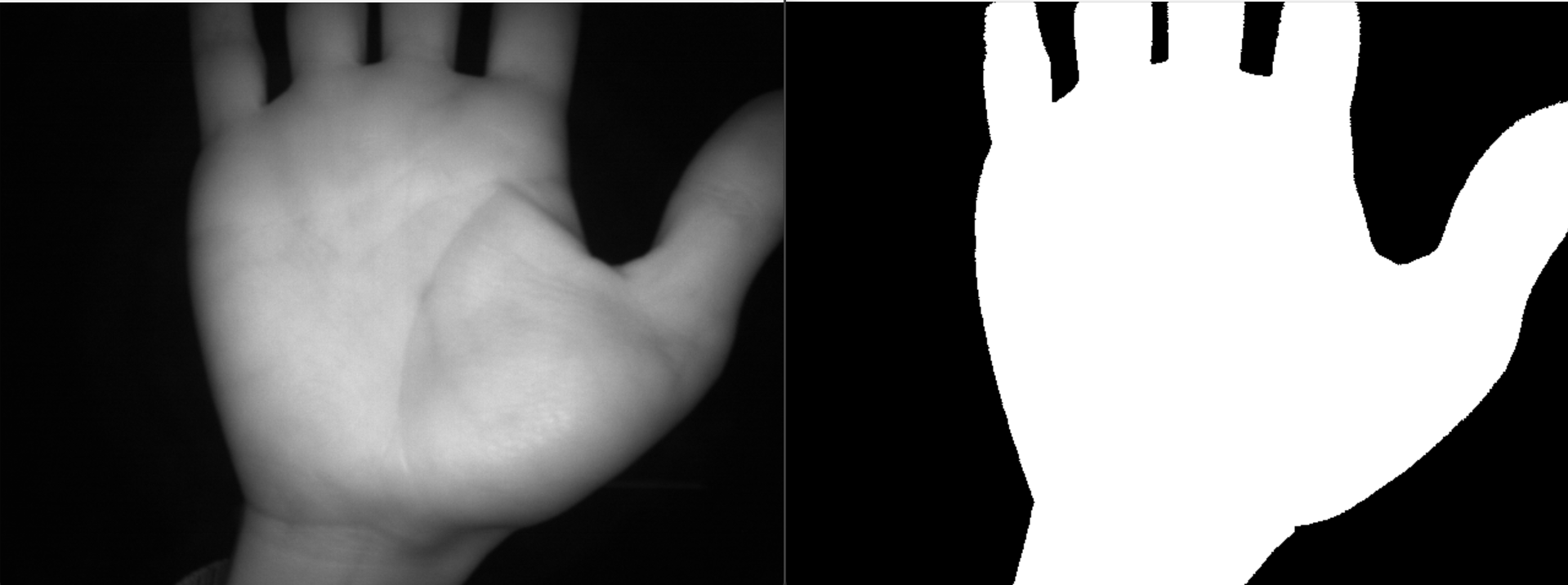


图12. 手掌前后背景分割

|  |  |
| --- | --- |
| 图13. 最大内切圆 | 图14. 去除手腕 |

计算得出圆心后根据圆心和图像比例算出手腕的大致位置，截取图片。除去手腕的目的是后续计算特征点更方便。剔除手腕后，将手掌底部的中点作为参考点，计算参考点与手掌轮廓每个点的距离构成以为数组。该一维数组的极大值点即为指尖，极小值点即为指根。

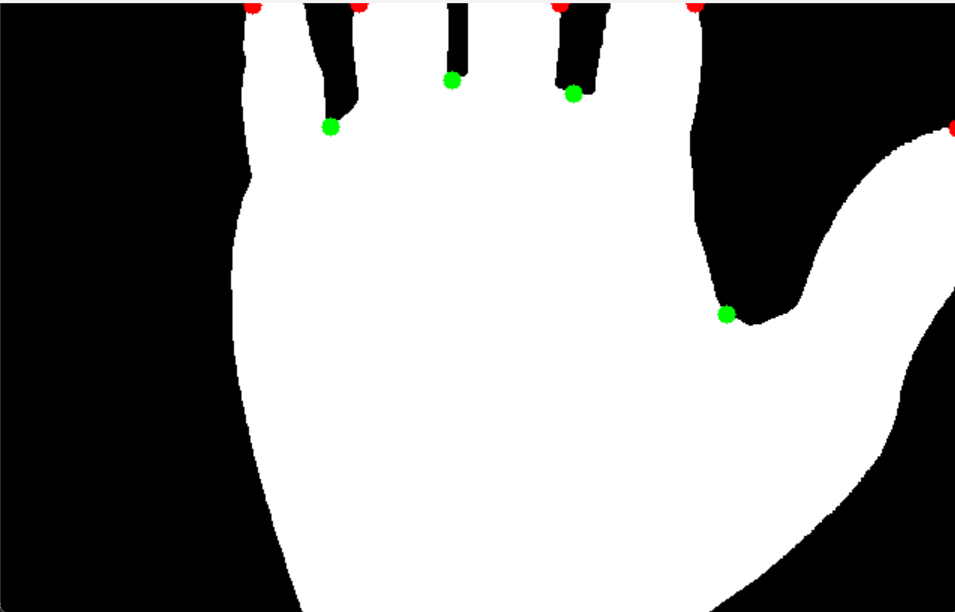


图15. 指尖与指根

最后是确定掌心区域，我选择了食指与中指之间的指根和无名指与小拇之间的指根连线，提取正方形ROI区域。为了保证ROI区域落在掌心中添加了平移和旋转功能。最后能够成功提取有效的ROI区域。



图16. ROI截取效果

**3.2.2 手掌图像静脉纹路增强**

根据前面的工作，对手掌图像增强时采用的策略和指静脉处理相同。CLAHE增强，再利用Frangi滤波提取特征，最后二值化并去除噪声。因为手掌的增强方法和手指几乎没有差别，在实现的过程中只需要重新修改合适的参数即可。

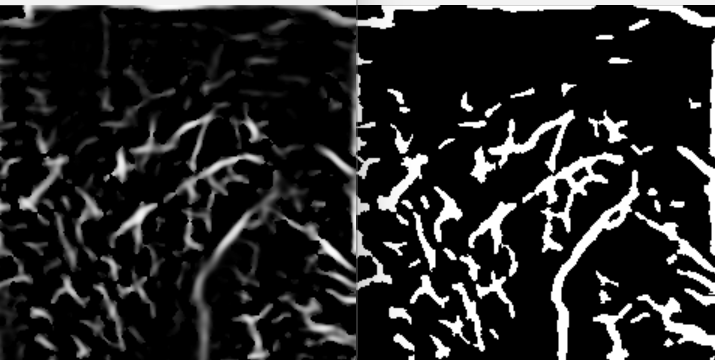
****

图17. 手掌静脉增强效果

**3.2.3 手掌图像匹配**

手掌图像匹配依然采用与手指相同的二值纹路匹配方法。由于手掌中血管的分布更加丰富，因此提取的信息也比手指要更多。最终的结果类内匹配率比手指更加稳定，能够保持在较高的类内匹配率。同时类间匹配率与类内匹配率的差距也更大，能够更加稳定的进行手掌匹配识别。当把静脉图中面积小于80的部分去除，并且设置通过匹配分数为0.18的时候，得到结果如下。

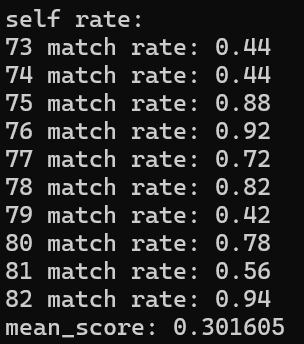


图18. 手掌类内匹配率

手掌类内匹配率在40%以上，同时类间匹配率不超过30%，绝大部分在20%以内能够符合匹配识别要求。

1. **程序运行流程以及核心代码**

**4.1 程序运行流程**

整个工程的代码可以通过Cmake或者vscode配置编译生成可执行文件。可执行文件可以在jsonDebug文件夹中找到，分别为手指和手掌的识别程序。在程序运行过程中，每进行一组识别，就会输出一张曲线图，其中包含一个类内任意两张图片的匹配分数（红线显示），以及成功配对的基准分数（蓝线显示），同时右上角还会打印出类内的匹配率。此时程序会停止运行，可以观察这一类图像配对的整体情况，如果想要继续运行下去，只需要按下任意按键就会开启下一组图片的识别。在所有组别识别完毕之后，会进行类间匹配识别，所有的识别率都会打印在终端上，此时不会输出任何图片，不需要操作。完成类间识别后，程序就结束运行。

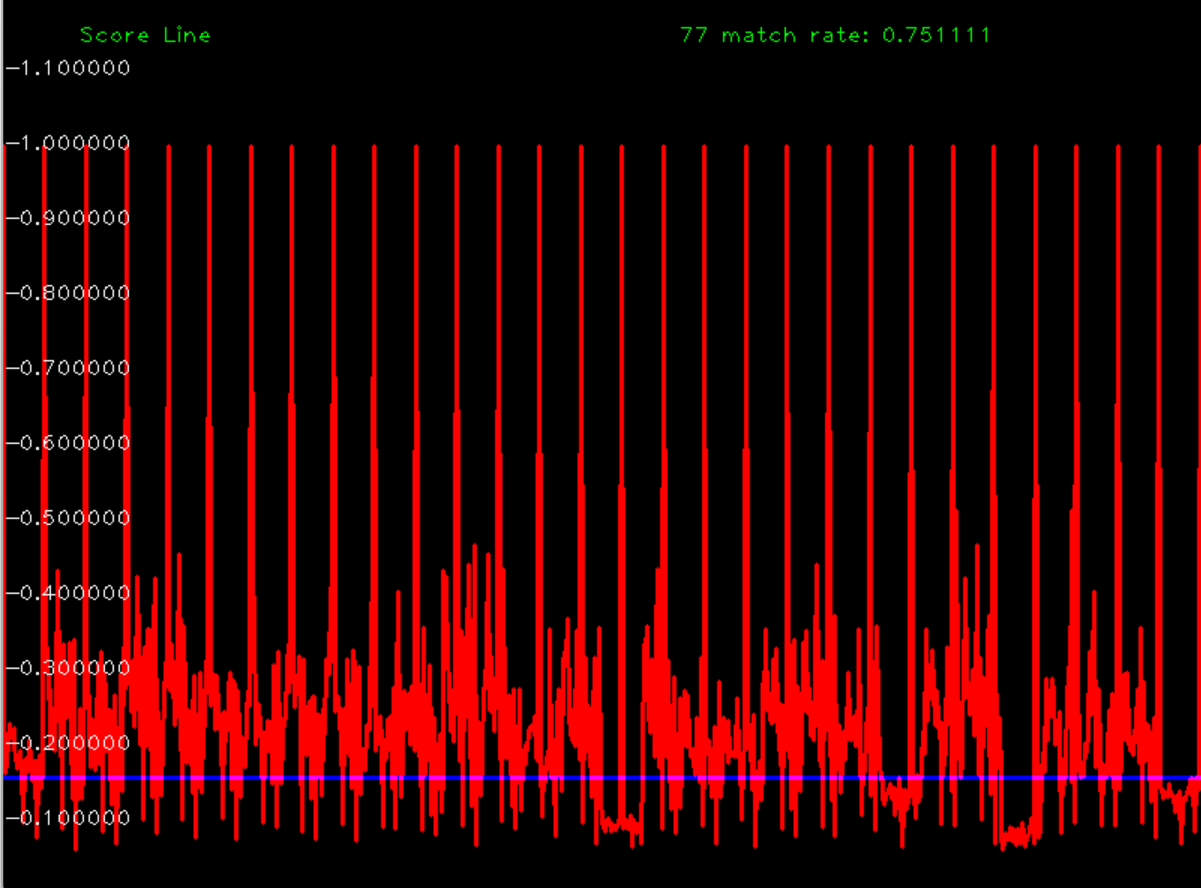


图18. 匹配分数图

**4.2 核心代码**

**4.2.1 手指图像预处理**

高斯滤波

void Finger::preprocessing()

{

  //高斯滤波处理

  cv::GaussianBlur(this->finger\_src, this->finger\_preprocessing, cv::Size(7, 7), 1.5);

}

寻找图像边缘和中线

void Finger::find\_finger\_edge(InputArray src, float cutScaleUp, float cutScaleDown, float cutScaleLeft, float cutScaleRight)

{

  edge\_up = new vector<cv::Point2i>;

  edge\_down = new vector<cv::Point2i>;

  edge\_middle = new vector<cv::Point2i>;

  Mat edge\_y;

  // 使用sobel算子边缘提取且只运算x方向上的梯度

  cv::Sobel(src, edge\_y, CV\_16S, 0, 1, 1, 30);

  convertScaleAbs(edge\_y, edge\_y); // 得到梯度图

  // 二值化处理

  cv::threshold(edge\_y, edge\_y, 200, 255, THRESH\_BINARY);

  // 截取图片中间的位置取得更好的边缘

  Mat temp;

  temp = edge\_y.rowRange(static\_cast<int>(edge\_y.rows \* cutScaleUp), static\_cast<int>(edge\_y.rows \* cutScaleDown)).clone();

  temp = temp.colRange(static\_cast<int>(temp.cols \* cutScaleLeft), static\_cast<int>(temp.cols \* cutScaleRight)).clone();

  // 私有的预处理图也同样截取图片

  // this->finger\_preprocessing =finger\_src.rowRange(static\_cast<int>(edge\_y.rows \* 0.15), static\_cast<int>(edge\_y.rows \* 0.85)).clone();

  // 将两条边界线 变粗 开运算

  Mat open\_kernel = cv::getStructuringElement(MORPH\_RECT, Size(3, 3));

  cv::morphologyEx(temp, temp, MORPH\_OPEN, open\_kernel);

  // 将两条边界线 变细 腐蚀

  Mat close\_kernel = cv::getStructuringElement(MORPH\_RECT, Size(3, 3));

  cv::morphologyEx(temp, temp, MORPH\_ERODE, close\_kernel);

  // 转为灰度图

  Mat temp\_binary;

  cv::cvtColor(temp, temp\_binary, COLOR\_BGR2GRAY, 1);

  // std::cout<<temp\_binary.cols<<" "<<temp\_binary.rows<<std::endl;

  // imshow("edges", temp\_binary);

  // waitKey(0);

  // 寻找上边缘线

  for (int i = 0; i < temp\_binary.cols - 1; ++i)

  {

    bool is\_check = false; // 是否能够检测到边缘

    for (int j = 0; j < temp\_binary.rows / 2; ++j)

    {

      // 如果是白色的则检测为边缘

      uchar \*point = temp\_binary.ptr(j);

      if (static\_cast<int>(\*(point + i)) > 0)

      {

        // i 表示列 对应横坐标  ； j对应纵坐标

        edge\_up->emplace\_back(Point2i(i, j));

        is\_check = true;

        break; // 退出本行的循环

      }

    }

    if (is\_check == false)

    {

      // 如果没有检测到边缘，则认为边缘在整个图像的边缘

      edge\_up->emplace\_back(Point2i(i, 0));

    }

  }

  // 下边的边缘线 从最下边向中间遍历

  for (int i = 0; i < temp\_binary.cols - 1; ++i)

  {

    bool is\_check = false; // 是否能够检测到边缘

    for (int j = temp\_binary.rows - 1; j > temp\_binary.rows / 2; --j)

    {

      uchar \*point = temp\_binary.ptr(j);

      if (static\_cast<int>(\*(point + i)) > 0)

      {

        // i 表示列 对应横坐标  ； j对应纵坐标

        edge\_down->emplace\_back(Point2i(i, j));

        is\_check = true;

        break; // 退出本行的循环

      }

    }

    if (is\_check == false)

    {

      // 如果没有检测到边缘，则认为边缘在整个图像的边缘

      edge\_down->emplace\_back(Point2i(i, temp\_binary.rows - 1));

    }

  }

  // std::cout<<edge\_up.size()<<"  "<<edge\_down.size()<<std::endl;

  for (int i = 0; i < temp\_binary.cols - 1; ++i)

  {

    Point2i middle\_point = (this->edge\_up->at(i) + this->edge\_down->at(i)) / 2;

    edge\_middle->emplace\_back(middle\_point);

  }

}

旋转调整图像

void Finger::rotate\_finger()

{

  find\_finger\_edge(finger\_preprocessing,0.18,0.945,0.1,0.92);

  Mat temp = Mat::zeros(finger\_preprocessing.size(), CV\_8UC3);

  temp = temp.rowRange(static\_cast<int>(temp.rows \* 0.18), static\_cast<int>(temp.rows \* 0.945)).clone();

  temp = temp.colRange(static\_cast<int>(temp.cols \* 0.1), static\_cast<int>(temp.cols \* 0.92)).clone();

  Vec<float, 4> middle\_fit\_line;

  // 别的方法 拟合出来的直线 角度是一样的， 距离有些不同罢了 没关系 所以选择L2计算快一点;

  cv::fitLine(\*this->edge\_middle, middle\_fit\_line, DIST\_L2, 0, 0.01, 0.01);

  double middle\_line\_k = middle\_fit\_line[1] / middle\_fit\_line[0];

  // std::cout<<middle\_line\_k<<std::endl;

  // std::cout<<middle\_fit\_line[2]<<"  "<<middle\_fit\_line[3]<<std::endl;

  // middle\_line\_k =(left\_line\_k+right\_line\_k)/2;

  double step\_middle = 100;

  cv::Point2i middle\_line\_point\_1(middle\_fit\_line[2] - step\_middle, middle\_fit\_line[3] - middle\_line\_k \* step\_middle);

  cv::Point2i middle\_line\_point\_2(middle\_fit\_line[2] + step\_middle, middle\_line\_k \* step\_middle + middle\_fit\_line[3]);

  cv::line(temp, middle\_line\_point\_1, middle\_line\_point\_2, Scalar(0, 255, 0), 4);

  // // 展示拟合中线

  // imshow("mid\_line&fit\_line", temp);

  // waitKey(0);

  // 画出真正的边缘线

  for (auto point : \*this->edge\_up)

  {

    temp.at<Vec3b>(point.y, point.x)[0] = 255;

    temp.at<Vec3b>(point.y, point.x)[1] = 255;

    temp.at<Vec3b>(point.y, point.x)[2] = 255;

  }

  for (auto point : \*this->edge\_down)

  {

    temp.at<Vec3b>(point.y, point.x)[0] = 255;

    temp.at<Vec3b>(point.y, point.x)[1] = 255;

    temp.at<Vec3b>(point.y, point.x)[2] = 255;

  }

  for (auto point : \*this->edge\_middle)

  {

    temp.at<Vec3b>(point.y, point.x)[0] = 255;

    temp.at<Vec3b>(point.y, point.x)[1] = 255;

    temp.at<Vec3b>(point.y, point.x)[2] = 255;

  }

  // 展示拟合中线

  imshow("mid\_line&fit\_line", temp);

  waitKey(1);

  double degree = atan(middle\_line\_k) \* 180 / CV\_PI; // 获取旋转角度

  Point2f center = Point2f(finger\_preprocessing.rows / 2, finger\_preprocessing.cols / 2);

  Mat martix = getRotationMatrix2D(center, degree, 1); // 得到仿射矩阵  k\_line2/CV\_PI\*180 弧度转角度

  warpAffine(this->finger\_preprocessing, this->finger\_rotate, martix, finger\_preprocessing.size());

}

截取ROI区域

void Finger::find\_knuckle()

{

    this->finger\_roi = this->finger\_rotate.clone();

    // 左右收缩

    // Mat temp = finger\_roi.rowRange(static\_cast<int>(finger\_roi.rows \* 0.05), static\_cast<int>(finger\_roi.rows \* 0.95)).clone();

    Mat temp;

    temp = finger\_roi.rowRange(static\_cast<int>(finger\_roi.rows \* 0.18), static\_cast<int>(finger\_roi.rows \* 0.92)).clone();

    temp = temp.colRange(static\_cast<int>(temp.cols \* 0.1),static\_cast<int>(temp.cols \* 0.99)).clone();

    find\_finger\_edge(this->finger\_roi,0.18,0.91,0.1,0.99);

    // imshow("ori",finger\_roi);

    // waitKey(1);

    // imshow("temp",temp);

    // waitKey(0);

    // 上下收缩

    int cod\_y\_down;

    {

        vector<int> edge\_down\_y;

        for (auto &point : \*this->edge\_down)

        {

            edge\_down\_y.emplace\_back(point.y);

        }

        cod\_y\_down = \*(std::min\_element(edge\_down\_y.begin(), edge\_down\_y.end()));

    }

    int cod\_y\_up;

    {

        vector<int> edge\_up\_y;

        for (auto &point : \*this->edge\_up)

        {

            edge\_up\_y.emplace\_back(point.y);

        }

        cod\_y\_up = \*(std::max\_element(edge\_up\_y.begin(), edge\_up\_y.end()));

    }

    // // 由于手指边缘干扰较多，多往内截取一部分

    // cod\_y\_up += 15;

    // cod\_y\_down -= 15;

    // 创建保存一个不画线的

    Mat temp1 = temp.clone();

    line(temp, Point2i(0, cod\_y\_up), Point2i(temp.cols - 1,cod\_y\_up), Scalar(0, 255, 0));

    line(temp, Point2i(0, cod\_y\_down), Point2i(temp.cols - 1, cod\_y\_down), Scalar(0, 255, 0));

    imshow("finger\_roi\_get", temp);

    waitKey(1);

    // 截取出最终选定区域

    this->finger\_roi = temp1.rowRange(static\_cast<int>(cod\_y\_up), static\_cast<int>(cod\_y\_down)).clone();

    // this->finger\_roi = this->finger\_roi.colRange(static\_cast<int>(finger\_roi.cols \* 0.1),static\_cast<int>(finger\_roi.cols \* 0.92)).clone();

    // imshow("finger\_roi\_true", this->finger\_roi);

    // waitKey(1);

    // 接下来在 手指图中寻找到两个亮的地方 找到关节

    vector<uint64\_t> windows\_value;

    vector<int> coordinate;

    Mat finger\_roi\_grey\_tmp;

    Mat finger\_roi\_gray;

    // 转为灰度图（由于某些图右侧指关节没有明显亮度增强，因此需要增加灰度图像的对比度）

    cvtColor(this->finger\_roi, finger\_roi\_grey\_tmp, COLOR\_BGR2GRAY);

    {

        // imshow("finger\_roi\_gray\_tmp", finger\_roi\_grey\_tmp);

        // waitKey(1);

        // 直方图均衡化

        cv::equalizeHist(finger\_roi\_grey\_tmp, finger\_roi\_gray);

        // imshow("finger\_roi\_gray", finger\_roi\_gray);

        // waitKey(1);

    }

    for (int col = 10; col < finger\_roi\_gray.cols - 11; col++)

    {

        uint64\_t pixel\_sum\_value = 0;                          // 初始化像素值为 0

        for (int col\_w = col - 10; col\_w <= col + 10; col\_w++) // 扫描每一列上下10个像素的领域

        {

            for (int row = 0; row < finger\_roi\_gray.rows - 1; row++)

            {

                pixel\_sum\_value += finger\_roi\_gray.ptr<uchar>(row)[col\_w]; // 记录下整个区域内的像素和

            }

        }

        windows\_value.emplace\_back(pixel\_sum\_value); //  ##将找到的像素值存起来

        coordinate.emplace\_back(col);                //  ##对应的中心行也存起来

    }

    vector<uint64\_t> peak; // 存放极大值点

    vector<int> pin;       // 用于存放极大值点的下标

    int field = 100; // 用于判断极大值点的领域大小

    // 寻找极大值点，当极大值点的数量>2时，扩大领域大小，重新寻找

    while(1)

    {

        for (int i = 0; i < windows\_value.size(); ++i)

        {

            // 检查当前点是否为极大值点

            bool isMaxima = true;

            for (int j = i - field; j <= i + field; ++j)

            {

                int temp\_j = j;

                // 处理边缘邻域溢出情况

                if (temp\_j < 0)

                    temp\_j += windows\_value.size();

                if (temp\_j >= windows\_value.size())

                    temp\_j -= windows\_value.size();

                if (temp\_j != i && windows\_value[i] < windows\_value[temp\_j])

                {

                    isMaxima = false;

                    break;

                }

            }

            // 如果是极大值点，将其索引添加到输出数组

            if (isMaxima)

            {

                peak.push\_back(windows\_value[i]);

                pin.push\_back(coordinate[i]);

            }

        }

        if(peak.size() <= 2)

            break;

        else if(peak.size() > 2)

        {

            peak.clear();

            pin.clear();

            field += 20;

            continue;

        }

    };

    // 截取到两个极大值，然后分割出最后的区域

    if(peak.size() == 2){

        Mat temp2 = this->finger\_roi.clone();

        for (auto col : pin)

        {

            line(temp2, Point2i(col, 0), Point2i(col, temp2.rows - 1), Scalar(255, 0, 0));

        }

        imshow("finger\_roi\_get2", temp2);

        waitKey(1);

        this->finger\_roi = this->finger\_roi.colRange(std::min(pin[0], pin[1]), std::max(pin[0], pin[1])).clone();

        // imshow("finger\_roi", this->finger\_roi);

        // waitKey(1);

    }

    else{

        Mat temp2 = this->finger\_roi.clone();

        for (auto col : pin)

        {

            line(temp2, Point2i(col, 0), Point2i(col, temp2.rows - 1), Scalar(255, 0, 0));

            line(temp2, Point2i(temp2.cols - 1 - static\_cast<int>(0.5\*col), 0), Point2i(temp2.cols - 1 - static\_cast<int>(0.5\*col), temp2.rows - 1), Scalar(255, 0, 0));

        }

        imshow("finger\_roi\_get2", temp2);

        waitKey(1);

        this->finger\_roi = this->finger\_roi.colRange(std::min(pin[0], temp2.cols - 1 - static\_cast<int>(0.5\*pin[0])), std::max(pin[0], temp2.cols - 1 - static\_cast<int>(0.5\*pin[0]))).clone();

        // imshow("finger\_roi", this->finger\_roi);

        // waitKey(1);

    }

}

**4.2.2 手掌图像预处理**

void Palm::palm\_roi\_process()

{

    // 转灰度图

    Mat temp\_roi;

    cvtColor(this->palm\_src, temp\_roi, COLOR\_BGR2GRAY);

    cv::threshold(temp\_roi, temp\_roi, 39, 255, THRESH\_BINARY);

    // 轮廓查找

    vector<vector<cv::Point2i>> contours;

    vector<Vec4i> hierarchy;

    findContours(temp\_roi, contours, hierarchy, RETR\_EXTERNAL, CHAIN\_APPROX\_NONE);

    // 找出面积最大的轮廓

    int max\_contour\_index = 0;

    {

        double max\_area = 0;

        for (int i = 0; i < contours.size(); ++i)

        {

            auto contour = contours[i];

            double temp\_area = contourArea(contour);

            if (temp\_area > max\_area)

            {

                max\_area = temp\_area;

                max\_contour\_index = i;

            }

        }

        temp\_roi = 0;

        drawContours(temp\_roi, contours, max\_contour\_index, Scalar(255), -1);

    }

    // imshow("src",palm\_src);

    // waitKey(1);

    // imshow("temp\_roi",temp\_roi);

    // waitKey(0);

    Point palm\_center;

    int maxdist = 0;

    // 寻找掌心，最大内切圆的中心，也就是距离圆轮廓最远的为掌心位置

    {

        int dist = 0;

        for (int i = 0; i < temp\_roi.cols; i++)

        {

            for (int j = 0; j < temp\_roi.rows; j++)

            {

                if (temp\_roi.at<uint8\_t>(j, i) == 0)

                    continue;

                dist = pointPolygonTest(contours[max\_contour\_index], Point(i, j), true);

                if (dist > maxdist)

                {

                    maxdist = dist;

                    palm\_center = Point(i, j);

                }

            }

        }

        Mat temp = temp\_roi.clone();

        circle(temp, palm\_center, maxdist, Scalar(0));

        imshow("CIRCLE", temp);

        waitKey(1);

    }

    // 手腕剔除

    {

        double cut\_scale = 0.8;

        int row\_end = maxdist \* cut\_scale + palm\_center.y;

        temp\_roi = temp\_roi.rowRange(0, row\_end).clone();

        // 定义结构元素

        cv::Mat kernel = cv::getStructuringElement(cv::MORPH\_RECT, cv::Size(7, 7));

        // 开运算

        cv::morphologyEx(temp\_roi, temp\_roi, cv::MORPH\_OPEN, kernel);

    }

    vector<cv::Point2i> target\_contour;

    {

        findContours(temp\_roi, contours, hierarchy, RETR\_EXTERNAL, CHAIN\_APPROX\_NONE);

        double max\_area = 0;

        for (int i = 0; i < contours.size(); ++i)

        {

            auto contour = contours[i];

            double temp\_area = contourArea(contour);

            if (temp\_area > max\_area)

            {

                max\_area = temp\_area;

                max\_contour\_index = i;

            }

        }

        target\_contour = contours[max\_contour\_index];

    }

    // 找出参考点

    Point2i point\_ref(0, temp\_roi.rows - 1);

    {

        Point2i point\_L(0, temp\_roi.rows - 1), point\_R(temp\_roi.cols - 1, temp\_roi.rows - 1);

        for (int i = 1; i < temp\_roi.cols - 1; i++)

        {

            if (temp\_roi.at<uint8\_t>(temp\_roi.rows - 1, i - 1) == 0 && temp\_roi.at<uint8\_t>(temp\_roi.rows - 1, i) == 255)

            {

                point\_L.x = i;

                break;

            }

        }

        for (int i = temp\_roi.cols - 2; i > 0; i--)

        {

            if (temp\_roi.at<uint8\_t>(temp\_roi.rows - 1, i) == 255 && temp\_roi.at<uint8\_t>(temp\_roi.rows - 1, i + 1) == 0)

            {

                point\_R.x = i;

                break;

            }

        }

        point\_ref.x = (point\_L.x + point\_R.x) / 2;

        Mat temp\_draw\_defects = temp\_roi.clone();

        cvtColor(temp\_draw\_defects, temp\_draw\_defects, COLOR\_GRAY2BGR);

        circle(temp\_draw\_defects, point\_ref, 6, Scalar(0, 0, 255), -1);

        circle(temp\_draw\_defects, point\_L, 6, Scalar(255, 255, 0), -1);

        circle(temp\_draw\_defects, point\_R, 6, Scalar(255, 255, 0), -1);

        imshow("point\_ref", temp\_draw\_defects);

        waitKey(1);

    }

    // 找出指尖与指根

    vector<cv::Point2i> fingerTip, fingerRoot;

    {

        Mat temp\_draw\_defects = temp\_roi.clone();

        cvtColor(temp\_draw\_defects, temp\_draw\_defects, COLOR\_GRAY2BGR);

        // 算出所有轮廓点与参考点的直线距离

        vector<float> distances(target\_contour.size());

        for (int i = 0; i < target\_contour.size(); i++)

        {

            distances[i] = calculateDistance(target\_contour[i], point\_ref);

        }

        // 找出distances中的极值点，提取出指尖与指根

        const int field = 50; // 用于判断极值点的领域长度

        for (int i = 0; i < distances.size(); ++i)

        {

            if (target\_contour[i].y >= 0.9 \* temp\_roi.rows)

                continue;

            // 检查当前点是否为极大值点

            bool isMaxima = true;

            bool isMinima = true;

            for (int j = i - field; j <= i + field; ++j)

            {

                int temp\_j = j;

                if (temp\_j < 0)

                    temp\_j += distances.size();

                if (temp\_j >= distances.size())

                    temp\_j -= distances.size();

                if (temp\_j != i && distances[i] < distances[temp\_j])

                {

                    isMaxima = false;

                }

                else if (temp\_j != i && distances[i] > distances[temp\_j])

                {

                    isMinima = false;

                }

                if (isMaxima == false && isMinima == false)

                    break;

            }

            // 如果是极大值点，将其索引添加到输出数组

            if (isMaxima)

            {

                fingerTip.push\_back(target\_contour[i]);

                circle(temp\_draw\_defects, target\_contour[i], 6, Scalar(0, 0, 255), -1);

            }

            else if (isMinima)

            {

                fingerRoot.push\_back(target\_contour[i]);

                circle(temp\_draw\_defects, target\_contour[i], 6, Scalar(0, 255, 0), -1);

            }

        }

        imshow("point\_ref", temp\_draw\_defects);

        waitKey(1);

    }

    // 检查指根是否成功提取，并排除虎口的点

    {

        sort(fingerRoot.begin(), fingerRoot.end(),

             [](Point2i point\_1, Point2i point\_2) -> bool

             {

                 int x1 = point\_1.x;

                 int x2 = point\_2.x;

                 return x1 < x2;

             });

        int i = 0;

        int size = fingerRoot.size();

        while (i < size - 1)

        {

            if (calculateDistance(fingerRoot[i], fingerRoot[i + 1]) < 35)

            {

                fingerRoot.erase(fingerRoot.begin() + i + 1);

                i--;

            }

            i++;

            size = fingerRoot.size();

        }

        if (fingerRoot.size() < 3 || fingerRoot.size() >= 5)

        {

            cout << "detect fingerRoot fail " << endl;

            while (1)

            {

            }

        }

        else

        {

            // cout << "detect fingerRoot success " << endl;

            if (fingerRoot.size() == 4)

            {

                if (calculateDistance(fingerRoot[0], fingerRoot[1]) > calculateDistance(fingerRoot[2], fingerRoot[3]))

                {

                    fingerRoot.erase(fingerRoot.begin());

                }

                else

                {

                    fingerRoot.erase(fingerRoot.end());

                }

            }

        }

    }

    // 从原图中提取ROI

    {

        // 计算旋转角度

        double angle = std::atan2(fingerRoot[2].y - fingerRoot[0].y, fingerRoot[2].x - fingerRoot[0].x) \* 180.0 / CV\_PI;

        // 定义旋转矩阵

        cv::Mat rotationMatrix = cv::getRotationMatrix2D(fingerRoot[0], angle, 1.0);

        // 进行仿射变换

        Mat temp\_img;

        cv::warpAffine(palm\_src, temp\_img, rotationMatrix, palm\_src.size());

        // imshow("temp\_img", temp\_img);

        // waitKey(0);

        int width = calculateDistance(fingerRoot[0], fingerRoot[2]) \* 1.4;

        int height = width;

        Point2i rect\_org\_point = fingerRoot[0];

        rect\_org\_point.x -= 0.15 \* calculateDistance(fingerRoot[0], fingerRoot[2]);

        if (rect\_org\_point.x + width > temp\_img.cols)

            width = height = temp\_img.cols - rect\_org\_point.x;

        if (rect\_org\_point.y + height > temp\_img.rows)

            width = height = temp\_img.rows - rect\_org\_point.y;

        Rect palm\_roi\_rect(rect\_org\_point.x, rect\_org\_point.y, width, height);

        this->palm\_roi = temp\_img(palm\_roi\_rect).clone();

        resize(palm\_roi, palm\_roi, Size(240, 240));

        imshow("palm\_roi\_rect", this->palm\_roi);

        waitKey(1);

    }

}

**4.2.3 图像静脉增强**

整体静脉增强代码

void Finger::enhance\_finger\_vein()

{

    this->finger\_vein\_enhance = this->finger\_roi.clone(); // 复制

    Mat temp16bit;

    cv::cvtColor(this->finger\_vein\_enhance, temp16bit, COLOR\_BGR2GRAY);

    temp16bit.convertTo(temp16bit, CV\_16U, 256, 0); // 将roi图转换成16位 提高精度

    temp16bit\_backup = temp16bit.clone();

    // CLAHE算法进行直方图均衡化

    auto cla = cv::createCLAHE(9000, Size(3, 28));

    cla->apply(temp16bit, temp16bit);

    // imshow("CLA-TRACKER", temp16bit);

    // waitKey(1);

    // 转32F归一化

    cv::Mat temp32bit\_fl;

    temp16bit.convertTo(temp32bit\_fl, CV\_32F, 1.0 / (256 \* 256 - 1));

    // 利用滤波函数处理32bit图像，并最后生成8bit图 存在finger\_vein\_enhance中

    {

        Mat vesselness, scale, angles;

        frangi2d\_opts\_t opts;

        frangi2d\_createopts\_custom(&opts, 3, 5, 3, 2.0, 0.06, true); // frangi滤波参数设置

        frangi2d(temp32bit\_fl, vesselness, scale, angles, opts);     // frangi滤波

        // 显示结果

        vesselness.convertTo(this->finger\_vein\_enhance, CV\_8UC1, 255);

        //  scale.convertTo(scale, CV\_8UC1, 255);

        //  angles.convertTo(angles, CV\_8UC1, 255);

        // imshow("finger\_vein\_enhance", finger\_vein\_enhance);

        // waitKey(1);

    }

}

Frangi滤波

//Frangi主要处理过程.

//Vesselness is saved in J, scale is saved to scale, vessel angle is saved to directions.

void frangi2d(const Mat &src, Mat &maxVals, Mat &whatScale, Mat &outAngles, frangi2d\_opts\_t opts){

    vector<Mat> ALLfiltered;

    vector<Mat> ALLangles;

    float beta = 2\*powf(opts.BetaOne,2);

    float c = 2\*powf(opts.BetaTwo,2);

    for (float sigma = opts.sigma\_start; sigma <= opts.sigma\_end; sigma += opts.sigma\_step){

        //创建2维 hessians 矩阵，并且得到Dxx Dxy Dyy

        Mat Dxx, Dyy, Dxy;

        frangi2d\_hessian(src, Dxx, Dxy, Dyy, sigma);

        //根据论文定义，对结果进行修正

        Dxx = Dxx\*powf(sigma,2);

        Dyy = Dyy\*powf(sigma,2);

        Dxy = Dxy\*powf(sigma,2);

        //calculate (abs sorted) eigenvalues and vectors

        Mat lambda1, lambda2, Ix, Iy;

        frangi2\_eig2image(Dxx, Dxy, Dyy, lambda1, lambda2, Ix, Iy);

        //计算方向

        Mat angles;

        phase(Ix, Iy, angles);

        ALLangles.push\_back(angles);

        //根据定义，计算“血管增强响应函数”

        lambda2.setTo(nextafterf(0, 1), lambda2 == 0);

        Mat Rb = lambda1.mul(1.0/lambda2);

        Rb = Rb.mul(Rb);

        Mat S2 = lambda1.mul(lambda1) + lambda2.mul(lambda2);

        Mat tmp1, tmp2;

        exp(-Rb/beta, tmp1);

        exp(-S2/c, tmp2);

        //保存单尺度结果

        Mat Ifiltered = tmp1.mul(Mat::ones(src.rows, src.cols, src.type()) - tmp2);

        if (opts.BlackWhite){

            Ifiltered.setTo(0, lambda2 < 0);

        } else {

            Ifiltered.setTo(0, lambda2 > 0);

        }

        //store results

        ALLfiltered.push\_back(Ifiltered);

    }

    float sigma = opts.sigma\_start;

    ALLfiltered[0].copyTo(maxVals);

    ALLfiltered[0].copyTo(whatScale);

    ALLfiltered[0].copyTo(outAngles);

    whatScale.setTo(sigma);

    //多尺度叠加

    for (int i=1; i < ALLfiltered.size(); i++){

        maxVals = max(maxVals, ALLfiltered[i]);

        whatScale.setTo(sigma, ALLfiltered[i] == maxVals);

        ALLangles[i].copyTo(outAngles, ALLfiltered[i] == maxVals);

        sigma += opts.sigma\_step;

    }

}

**4.2.4 图像匹配**

double palm\_match(Mat &img1, Mat &img2)

{

    // 计算交集

    cv::Mat intersection;

    cv::bitwise\_and(img1, img2, intersection);

    // 计算并集

    cv::Mat union\_;

    cv::bitwise\_or(img1, img2, union\_);

    // 计数非零像素点（255值像素点）

    int count\_intersection = cv::countNonZero(intersection);

    int count\_union = cv::countNonZero(union\_);

    double score = static\_cast<double>(count\_intersection) / count\_union;

    return score;

}