OpenCV使用手册

目录

- 分水岭算法

第三部分：特征提取与匹配

7. 特征检测

   - Harris角点检测
   - SIFT特征
   - SURF特征
   - ORB特征

8. 特征匹配

   - 特征描述子
   - 特征匹配算法
   - 单应性矩阵
   - 图像拼接

第四部分：机器学习与深度学习

9. 机器学习基础

   - 支持向量机(SVM)
   - 决策树
   - 随机森林
   - K近邻算法

10. 深度学习集成

    - DNN模块
    - 模型加载与推理
    - 图像分类
    - 目标检测

第五部分：视频处理

11. 视频基础

    - 视频读取
    - 视频保存
    - 视频流处理
    - 实时视频处理

12. 运动检测

    - 背景建模
    - 光流法
    - 运动跟踪
    - 目标检测

第六部分：实际应用

13. 图像识别
    - 人脸检测
    - 人脸识别
    - 物体识别
    - 场景识别
14. 计算机视觉应用
    - 图像增强
    - 图像修复
    - 图像拼接
    - 3D重建
15. 性能优化
    - 并行计算
    - GPU加速
    - 内存优化
    - 算法优化

第七部分：项目实战

16. 智能车视觉系统
    - 车道线检测
    - 交通标志识别
    - 障碍物检测
    - 路径规划
17. 工业视觉应用
    - 缺陷检测
    - 尺寸测量
    - 物体分类
    - 质量检测
18. 移动端应用
    - Android集成
    - iOS集成
    - 性能优化
    - 实际部署

第一部分：OpenCV基础

1. OpenCV简介

1.1 OpenCV的历史与发展

OpenCV（Open Source Computer Vision Library）是一个开源的计算机视觉库，由Intel公司于1999年发起，现在由非营利组织OpenCV.org维护。它提供了丰富的图像处理和计算机视觉算法，支持多种编程语言，包括C++、Python、Java等。

1.2 OpenCV的主要特点

- 跨平台：支持Windows、Linux、MacOS、Android、iOS等
- 高性能：优化的C++实现，支持GPU加速
- 丰富的算法：包含2500多个优化算法
- 活跃的社区：持续更新和维护
- 完善的文档：详细的API文档和示例

1.3 OpenCV的应用领域

- 图像处理
- 视频分析
- 人脸识别
- 物体检测
- 机器学习
- 深度学习
- 3D重建
- 增强现实

1.4 OpenCV的安装与配置

Linux系统安装

```
# 安装依赖
sudo apt-get update
sudo apt-get install build-essential cmake pkg-config
sudo apt-get install libjpeg-dev libtiff-dev libjasper-dev libpng-dev
sudo apt-get install libavcodec-dev libavformat-dev libswscale-dev libv4l-
dev
sudo apt-get install libxvidcore-dev libx264-dev
sudo apt-get install libgtk-3-dev
sudo apt-get install libatlas-base-dev gfortran
sudo apt-get install python3-dev

# 下载OpenCV源码
wget -O opencv.zip https://github.com/opencv/opencv/archive/4.5.1.zip
unzip opencv.zip

# 编译安装
cd opencv-4.5.1
mkdir build && cd build
cmake -D CMAKE_BUILD_TYPE=RELEASE \
      -D CMAKE_INSTALL_PREFIX=/usr/local \
      -D OPENCV_EXTRA_MODULES_PATH=../../opencv_contrib-4.5.1/modules \
      -D WITH_TBB=ON \
```

```
      -D WITH_V4L=ON \
      -D WITH_QT=OFF \
      -D WITH_OPENGL=ON ..
make -j4
sudo make install
```

CMake配置

```
cmake_minimum_required(VERSION 3.10)
project(OpenCV_Project)

# 查找OpenCV包
find_package(OpenCV REQUIRED)

# 添加可执行文件
add_executable(${PROJECT_NAME} main.cpp)

# 链接OpenCV库
target_link_libraries(${PROJECT_NAME} ${OpenCV_LIBS})
```

2. OpenCV核心概念

2.1 Mat类详解

Mat类是OpenCV中最核心的类，用于存储图像数据。它就像是一个二维数组，但比普通数组更强大。

基本概念

1. **图像的本质**：
   - 图像本质上是一个二维矩阵
   - 每个像素点都是一个数值（灰度图）或一组数值（彩色图）
   - 比如一张800x600的彩色图像，就是一个800行600列的矩阵，每个元素包含3个值（BGR）
2. **Mat类的特点**：
   - 自动内存管理：不需要手动分配和释放内存
   - 引用计数：多个Mat对象可以共享同一块内存
   - 支持多种数据类型：可以存储不同位深度的图像

基本结构

```
class Mat {
public:
    // 构造函数
    Mat();  // 默认构造函数
    Mat(int rows, int cols, int type);  // 指定大小和类型
```

```
    Mat(Size size, int type);  // 使用Size指定大小

    // 常用方法
    void create(int rows, int cols, int type);  // 创建矩阵
    Mat clone() const;  // 深拷贝
    void copyTo(Mat& m) const;  // 复制到目标矩阵
    Mat& setTo(const Scalar& s);  // 设置所有元素为指定值

    // 属性
    int rows;  // 行数
    int cols;  // 列数
    int type;  // 数据类型
    uchar* data;  // 数据指针
};
```

创建Mat对象

```
// 创建空矩阵
Mat empty;  // 创建一个空的Mat对象

// 创建指定大小的矩阵
Mat img(480, 640, CV_8UC3);  // 创建一个480x640的彩色图像
// CV_8UC3表示：8位无符号整数，3通道（BGR）

// 创建并初始化
Mat ones = Mat::ones(3, 3, CV_32F);  // 3x3的全1矩阵
Mat zeros = Mat::zeros(3, 3, CV_32F);  // 3x3的全0矩阵
Mat eye = Mat::eye(3, 3, CV_32F);  // 3x3的单位矩阵
```

2.2 图像的基本操作

图像读取

```
// 读取图像
Mat img = imread("image.jpg", IMREAD_COLOR);
// IMREAD_COLOR：以彩色模式读取
// IMREAD_GRAYSCALE：以灰度模式读取
// IMREAD_UNCHANGED：按原样读取（包括alpha通道）

if(img.empty()) {
    std::cerr << "Error: Could not read the image." << std::endl;
    return -1;
}
```

图像显示

```
// 显示图像
namedWindow("Display window", WINDOW_AUTOSIZE);  // 创建窗口
imshow("Display window", img);  // 显示图像
waitKey(0);  // 等待按键，0表示无限等待
```

图像保存

```
// 保存图像
imwrite("output.jpg", img);  // 保存为JPEG格式
imwrite("output.png", img);  // 保存为PNG格式
```

2.3 数据类型与内存管理

常用数据类型

- CV_8U：8位无符号整数（0..255）
    - 最常用的图像数据类型
    - 每个像素值范围是0-255
    - 适合存储普通图像
- CV_8S：8位有符号整数（-128..127）
    - 用于存储有符号的图像数据
    - 比如某些滤波器的输出
- CV_16U：16位无符号整数（0..65535）
    - 用于高动态范围图像
    - 比如医学图像
- CV_16S：16位有符号整数（-32768..32767）
    - 用于存储有符号的高精度数据
    - 比如某些算法的中间结果
- CV_32F：32位浮点数
    - 用于存储浮点数图像
    - 比如某些滤波器的输出
- CV_64F：64位浮点数
    - 用于存储高精度浮点数
    - 用于科学计算

通道数表示

- CV_8UC1：单通道
  - 灰度图像
  - 每个像素只有一个值
- CV_8UC3：三通道
  - 彩色图像（BGR）
  - 每个像素有三个值
- CV_8UC4：四通道
  - 带透明通道的图像（BGRA）
  - 每个像素有四个值

内存管理

```
// 引用计数
Mat A = imread("image.jpg");
Mat B = A;  // 浅拷贝，共享数据
// 此时A和B指向同一块内存
// 修改B会影响A

Mat C = A.clone();  // 深拷贝，独立数据
// 此时C有自己独立的内存
// 修改C不会影响A

// 子矩阵
Mat roi = img(Rect(10, 10, 100, 100));  // 提取感兴趣区域
// 创建了一个100x100的子图像
// 从原图像的(10,10)位置开始
```

2.4 图像格式与色彩空间

色彩空间转换

```
// BGR转灰度
Mat gray;
cvtColor(img, gray, COLOR_BGR2GRAY);
// 将彩色图像转换为灰度图像
// 使用公式：Gray = 0.299*R + 0.587*G + 0.114*B

// BGR转HSV
Mat hsv;
cvtColor(img, hsv, COLOR_BGR2HSV);
// HSV色彩空间更适合颜色分割
// H：色调（0-180）
// S：饱和度（0-255）
// V：亮度（0-255）
```

```
// BGR转RGB
Mat rgb;
cvtColor(img, rgb, COLOR_BGR2RGB);
// OpenCV默认使用BGR顺序
// 某些其他库使用RGB顺序
```

常用色彩空间

- BGR：OpenCV默认色彩空间
  - 蓝色(Blue)、绿色(Green)、红色(Red)
  - 每个通道范围0-255
  - 适合图像处理和显示
- HSV：色调、饱和度、亮度
  - 色调(Hue)：表示颜色类型（0-180）
  - 饱和度(Saturation)：表示颜色纯度（0-255）
  - 亮度(Value)：表示颜色亮度（0-255）
  - 适合颜色分割和识别
- LAB：亮度、a通道、b通道
  - L：亮度（0-100）
  - a：从绿色到红色（-128到+127）
  - b：从蓝色到黄色（-128到+127）
  - 适合颜色差异分析
- YCrCb：亮度、红色差、蓝色差
  - Y：亮度分量
  - Cr：红色差分量
  - Cb：蓝色差分量
  - 适合视频压缩

3. 图像处理基础

3.1 图像读取与保存

```
// 读取图像
Mat img = imread("input.jpg", IMREAD_COLOR);

// 保存图像
imwrite("output.jpg", img);

// 读取视频
VideoCapture cap("video.mp4");
Mat frame;
while(cap.read(frame)) {
```

```
    // 处理帧
}

// 保存视频
VideoWriter writer("output.avi",
                   VideoWriter::fourcc('M','J','P','G'),
                   30, Size(640, 480));
writer.write(frame);
```

3.2 图像显示

```
// 创建窗口
namedWindow("Window", WINDOW_NORMAL);

// 显示图像
imshow("Window", img);

// 等待按键
int key = waitKey(0);
if(key == 27)  // ESC键
    break;

// 调整窗口大小
resizeWindow("Window", 800, 600);
```

3.3 图像转换

```
// 调整大小
Mat resized;
resize(img, resized, Size(320, 240));

// 旋转
Mat rotated;
Point2f center(img.cols/2.0, img.rows/2.0);
Mat rot = getRotationMatrix2D(center, 45, 1.0);
warpAffine(img, rotated, rot, img.size());

// 翻转
Mat flipped;
flip(img, flipped, 1);  // 1:水平翻转, 0:垂直翻转, -1:同时翻转
```

3.4 图像缩放与旋转

```
// 缩放
Mat scaled;
resize(img, scaled, Size(), 0.5, 0.5, INTER_LINEAR);
```

```
// 旋转
Mat rotated;
double angle = 45;
Point2f center(img.cols/2.0, img.rows/2.0);
Mat rot = getRotationMatrix2D(center, angle, 1.0);
warpAffine(img, rotated, rot, img.size());

// 透视变换
Point2f src[] = {Point2f(0,0), Point2f(640,0),
                 Point2f(0,480), Point2f(640,480)};
Point2f dst[] = {Point2f(100,100), Point2f(540,100),
                 Point2f(0,480), Point2f(640,480)};
Mat persp = getPerspectiveTransform(src, dst);
warpPerspective(img, warped, persp, img.size());
```

第二部分：图像处理进阶

4. 图像滤波

4.1 线性滤波

线性滤波是最基本的图像滤波方法，通过卷积运算实现。它的原理类似于用一个"窗口"在图像上滑动，对窗口内的像素进行加权平均。

基本原理

1. **卷积运算**：
    - 将一个小矩阵（称为卷积核或滤波器）在图像上滑动
    - 对每个位置，计算卷积核与图像对应区域的乘积和
    - 结果作为该位置的输出值
2. **滤波效果**：
    - 平滑图像：减少噪声
    - 锐化图像：增强边缘
    - 边缘检测：突出图像边缘

均值滤波

```
// 均值滤波
Mat blur_img;
blur(img, blur_img, Size(5, 5));  // 5x5的均值滤波核
// 原理：用5x5窗口内的像素平均值替换中心像素
// 效果：去除噪点，但会使图像变模糊

// 自定义核的均值滤波
Mat kernel = Mat::ones(3, 3, CV_32F) / 9.0;  // 3x3的均值核
Mat custom_blur;
```

```
filter2D(img, custom_blur, -1, kernel);
// 可以自定义不同的权重，实现不同的滤波效果
```

高斯滤波

```
// 高斯滤波
Mat gaussian_blur;
GaussianBlur(img, gaussian_blur, Size(5, 5), 1.5);  // 核大小5x5，标准差1.5
// 原理：使用高斯函数作为权重进行加权平均
// 特点：中心像素权重最大，周围像素权重随距离减小
// 效果：比均值滤波更好地保持边缘信息

// 分离的高斯滤波（更高效）
Mat gaussian_blur_sep;
GaussianBlur(img, gaussian_blur_sep, Size(5, 5), 1.5, 1.5, BORDER_DEFAULT);
// 原理：将二维高斯核分解为两个一维高斯核
// 优点：计算量从O(n²)降低到O(2n)，大大提高效率
```

4.2 非线性滤波

中值滤波

```
// 中值滤波（对椒盐噪声特别有效）
Mat median_blur;
medianBlur(img, median_blur, 5);  // 核大小5
// 原理：用窗口内像素的中值替换中心像素
// 特点：对椒盐噪声特别有效，能很好地保持边缘
// 应用：去除图像中的噪点，特别是随机分布的噪点
```

双边滤波

```
// 双边滤波（保持边缘的同时进行平滑）
Mat bilateral_blur;
bilateralFilter(img, bilateral_blur, 9, 75, 75);  // 直径9，颜色空间标准差75，
坐标空间标准差75
// 原理：同时考虑空间距离和像素值差异
// 特点：在平滑图像的同时保持边缘清晰
// 应用：图像降噪、美颜、HDR等
```

4.3 形态学操作

基本原理

1. **结构元素**：

- 一个小的二值图像（通常是矩形、圆形或十字形）
- 用于定义操作的邻域形状和大小

2. **操作类型**：

- 膨胀：扩大亮区域
- 腐蚀：缩小亮区域
- 开运算：先腐蚀后膨胀
- 闭运算：先膨胀后腐蚀

膨胀与腐蚀

```
// 创建结构元素
Mat kernel = getStructuringElement(MORPH_RECT, Size(3, 3));
// MORPH_RECT：矩形结构元素
// Size(3, 3)：3x3大小

// 膨胀操作
Mat dilated;
dilate(img, dilated, kernel);
// 原理：用结构元素扫描图像，如果结构元素与图像有重叠，则输出像素设为1
// 效果：扩大亮区域，填充小洞

// 腐蚀操作
Mat eroded;
erode(img, eroded, kernel);
// 原理：用结构元素扫描图像，只有当结构元素完全在图像内时，输出像素才设为1
// 效果：缩小亮区域，去除小物体
```

开运算与闭运算

```
// 开运算（先腐蚀后膨胀）
Mat opened;
morphologyEx(img, opened, MORPH_OPEN, kernel);
// 原理：先腐蚀后膨胀
// 效果：去除小的干扰区域，保持主要形状

// 闭运算（先膨胀后腐蚀）
Mat closed;
morphologyEx(img, closed, MORPH_CLOSE, kernel);
// 原理：先膨胀后腐蚀
// 效果：填充小的空洞，连接临近区域
```

5. 图像变换

5.1 仿射变换

仿射变换是一种保持直线平行性的变换，包括平移、旋转、缩放和剪切。

基本原理

1. **变换矩阵**：
   - 2x3的矩阵，包含6个参数
   - 可以表示任何仿射变换
2. **变换类型**：
   - 平移：改变位置
   - 旋转：改变角度
   - 缩放：改变大小
   - 剪切：改变形状

```
// 仿射变换
Point2f srcTri[3];
Point2f dstTri[3];

// 设置源点和目标点
srcTri[0] = Point2f(0, 0);
srcTri[1] = Point2f(img.cols - 1, 0);
srcTri[2] = Point2f(0, img.rows - 1);

dstTri[0] = Point2f(img.cols * 0.0, img.rows * 0.33);
dstTri[1] = Point2f(img.cols * 0.85, img.rows * 0.25);
dstTri[2] = Point2f(img.cols * 0.15, img.rows * 0.7);

// 计算仿射变换矩阵
Mat warp_mat = getAffineTransform(srcTri, dstTri);
// 原理：通过三对对应点计算变换矩阵
// 应用：图像校正、图像对齐等

// 应用变换
Mat warp_dst = Mat::zeros(img.rows, img.cols, img.type());
warpAffine(img, warp_dst, warp_mat, warp_dst.size());
```

5.2 透视变换

透视变换是一种更复杂的变换，可以模拟视角变化，常用于图像校正和全景拼接。

基本原理

1. **变换特点**：
   - 可以改变图像的视角
   - 保持直线的直线性
   - 可以模拟3D效果

2. **应用场景**：
    - 文档扫描
    - 车牌识别
    - 图像拼接

```
// 透视变换
Point2f src_points[4];
Point2f dst_points[4];

// 设置源点和目标点
src_points[0] = Point2f(0, 0);
src_points[1] = Point2f(img.cols - 1, 0);
src_points[2] = Point2f(0, img.rows - 1);
src_points[3] = Point2f(img.cols - 1, img.rows - 1);

dst_points[0] = Point2f(50, 50);
dst_points[1] = Point2f(img.cols - 50, 50);
dst_points[2] = Point2f(50, img.rows - 50);
dst_points[3] = Point2f(img.cols - 50, img.rows - 50);

// 计算透视变换矩阵
Mat perspective_mat = getPerspectiveTransform(src_points, dst_points);
// 原理：通过四对对应点计算变换矩阵
// 特点：可以模拟视角变化

// 应用变换
Mat perspective_dst;
warpPerspective(img, perspective_dst, perspective_mat, img.size());
```

5.3 图像金字塔

图像金字塔是一种多分辨率表示方法，用于图像缩放、图像融合等。

基本原理

1. **高斯金字塔**：
    - 自顶向下：图像尺寸逐层减半
    - 每层图像都是上一层的高斯模糊结果
    - 用于图像缩放和特征提取

2. **拉普拉斯金字塔**：
    - 记录每层高斯金字塔的细节信息
    - 用于图像重建和图像融合

```
// 高斯金字塔
vector<Mat> gaussian_pyramid;
```

```
Mat current = img.clone();
gaussian_pyramid.push_back(current);

for(int i = 0; i < 4; i++) {
    Mat down;
    pyrDown(current, down);  // 降采样
    gaussian_pyramid.push_back(down);
    current = down;
}
// 原理：每层图像是上一层的高斯模糊和降采样结果
// 应用：多尺度特征提取、图像融合

// 拉普拉斯金字塔
vector<Mat> laplacian_pyramid;
for(int i = 0; i < gaussian_pyramid.size() - 1; i++) {
    Mat up;
    pyrUp(gaussian_pyramid[i + 1], up, gaussian_pyramid[i].size());  // 上采样
    Mat laplacian;
    subtract(gaussian_pyramid[i], up, laplacian);  // 计算差值
    laplacian_pyramid.push_back(laplacian);
}
// 原理：记录每层高斯金字塔的细节信息
// 应用：图像重建、图像融合
```

5.4 图像梯度

图像梯度用于检测图像中的边缘和轮廓，是许多图像处理算法的基础。

基本原理

1. **梯度概念**：
   - 表示图像亮度变化的方向和大小
   - 梯度大的地方通常是边缘
   - 梯度方向垂直于边缘方向
2. **计算方法**：
   - Sobel算子：计算x和y方向的梯度
   - Scharr算子：对边缘更敏感
   - 合并梯度：计算梯度幅值和方向

```
// Sobel算子
Mat grad_x, grad_y;
Mat abs_grad_x, abs_grad_y;
Mat grad;

// 计算x方向梯度
Sobel(img, grad_x, CV_16S, 1, 0, 3);
```

```
convertScaleAbs(grad_x, abs_grad_x);
// 原理：使用Sobel算子计算x方向梯度
// 效果：检测垂直边缘

// 计算y方向梯度
Sobel(img, grad_y, CV_16S, 0, 1, 3);
convertScaleAbs(grad_y, abs_grad_y);
// 原理：使用Sobel算子计算y方向梯度
// 效果：检测水平边缘

// 合并梯度
addWeighted(abs_grad_x, 0.5, abs_grad_y, 0.5, 0, grad);
// 原理：将x和y方向梯度加权合并
// 效果：得到完整的边缘信息

// Scharr算子（对边缘更敏感）
Mat scharr_x, scharr_y;
Scharr(img, scharr_x, CV_16S, 1, 0);
Scharr(img, scharr_y, CV_16S, 0, 1);
// 原理：使用Scharr算子计算梯度
// 特点：对边缘更敏感，但计算量更大
```

6. 图像分割

6.1 阈值分割

```
// 简单阈值分割
Mat binary;
threshold(img, binary, 127, 255, THRESH_BINARY);

// 自适应阈值分割
Mat adaptive_binary;
adaptiveThreshold(img, adaptive_binary, 255,
                  ADAPTIVE_THRESH_GAUSSIAN_C,
                  THRESH_BINARY, 11, 2);

// Otsu阈值分割
Mat otsu_binary;
threshold(img, otsu_binary, 0, 255, THRESH_BINARY | THRESH_OTSU);
```

6.2 边缘检测

```
// Canny边缘检测
Mat edges;
Canny(img, edges, 100, 200);  // 低阈值100，高阈值200

// 使用高斯模糊预处理
Mat blurred;
```

```
GaussianBlur(img, blurred, Size(5, 5), 1.5);
Canny(blurred, edges, 100, 200);
```

6.3 轮廓检测

```
// 查找轮廓
vector<vector<Point>> contours;
vector<Vec4i> hierarchy;
findContours(edges, contours, hierarchy,
             RETR_TREE, CHAIN_APPROX_SIMPLE);

// 绘制轮廓
Mat contour_img = Mat::zeros(edges.size(), CV_8UC3);
for(int i = 0; i < contours.size(); i++) {
    Scalar color = Scalar(rand()&255, rand()&255, rand()&255);
    drawContours(contour_img, contours, i, color, 2, 8, hierarchy);
}
```

6.4 分水岭算法

```
// 分水岭算法
Mat gray;
cvtColor(img, gray, COLOR_BGR2GRAY);

// 阈值处理
Mat thresh;
threshold(gray, thresh, 0, 255, THRESH_BINARY_INV + THRESH_OTSU);

// 形态学操作
Mat kernel = getStructuringElement(MORPH_RECT, Size(3, 3));
Mat opening;
morphologyEx(thresh, opening, MORPH_OPEN, kernel, Point(-1,-1), 2);

// 确定背景区域
Mat sure_bg;
dilate(opening, sure_bg, kernel, Point(-1,-1), 3);

// 距离变换
Mat dist_transform;
distanceTransform(opening, dist_transform, DIST_L2, 5);

// 确定前景区域
Mat sure_fg;
threshold(dist_transform, sure_fg, 0.7*dist_transform.max(), 255, 0);

// 标记
Mat markers;
sure_fg.convertTo(markers, CV_32S);
```

```
markers = markers + 1;
markers.setTo(0, sure_bg);

// 应用分水岭算法
watershed(img, markers);
```

第三部分：特征提取与匹配

7. 特征检测

7.1 基本原理

1. **什么是特征点**：
   - 图像中具有独特性的点
   - 在不同视角、光照下都能被稳定检测
   - 通常包括角点、边缘点、斑点等
2. **特征点的重要性**：
   - 用于图像匹配和拼接
   - 用于目标跟踪
   - 用于3D重建
   - 用于图像识别

7.2 Harris角点检测

Harris角点检测是一种经典的角点检测算法，通过计算图像局部区域的灰度变化来检测角点。

基本原理

1. **角点特征**：
   - 在任意方向移动窗口，图像灰度都会发生显著变化
   - 比边缘点更稳定，比平面点更具独特性
2. **检测步骤**：
   - 计算图像在x和y方向的梯度
   - 计算梯度矩阵的特征值
   - 根据特征值判断是否为角点

```
// Harris角点检测
Mat gray;
cvtColor(img, gray, COLOR_BGR2GRAY);

// 计算Harris角点响应
Mat harris_response;
cornerHarris(gray, harris_response, 2, 3, 0.04);
// 参数说明：
```

```
// 2：邻域大小
// 3：Sobel算子大小
// 0.04：Harris检测器自由参数
// 原理：计算每个像素点的角点响应值

// 归一化响应值
Mat harris_norm;
normalize(harris_response, harris_norm, 0, 255, NORM_MINMAX, CV_32FC1,
Mat());
// 将响应值归一化到0-255范围，便于显示

// 阈值处理
Mat harris_thresh;
threshold(harris_norm, harris_thresh, 150, 255, THRESH_BINARY);
// 设置阈值，筛选出强角点

// 绘制角点
Mat harris_corners = img.clone();
for(int i = 0; i < harris_thresh.rows; i++) {
    for(int j = 0; j < harris_thresh.cols; j++) {
        if((int)harris_thresh.at<float>(i,j) > 150) {
            circle(harris_corners, Point(j,i), 5, Scalar(0,255,0), 2);
        }
    }
}
// 在检测到的角点位置画圆
```

7.3 SIFT特征

SIFT（尺度不变特征变换）是一种强大的特征检测和描述算法，具有尺度、旋转、光照不变性。

基本原理

1. **尺度空间构建**：
    - 使用高斯金字塔构建尺度空间
    - 在不同尺度下检测特征点
2. **关键点定位**：
    - 在尺度空间中寻找极值点
    - 通过插值确定精确位置
    - 去除低对比度和边缘响应点
3. **方向分配**：
    - 计算特征点周围区域的梯度方向直方图
    - 确定主方向，实现旋转不变性
4. **特征描述**：
    - 在特征点周围区域计算梯度

- 生成128维的特征向量

```
// 创建SIFT检测器
Ptr<SIFT> sift = SIFT::create();
// 可以设置参数：
// nfeatures：特征点数量
// nOctaveLayers：每组金字塔的层数
// contrastThreshold：对比度阈值
// edgeThreshold：边缘阈值
// sigma：高斯模糊参数

// 检测关键点和计算描述子
vector<KeyPoint> keypoints;
Mat descriptors;
sift->detectAndCompute(img, Mat(), keypoints, descriptors);
// keypoints：存储检测到的关键点信息
// descriptors：存储每个关键点的特征描述子

// 绘制关键点
Mat sift_img;
drawKeypoints(img, keypoints, sift_img, Scalar::all(-1),
              DrawMatchesFlags::DRAW_RICH_KEYPOINTS);
// 绘制关键点位置、尺度和方向
```

7.4 SURF特征

SURF（加速稳健特征）是SIFT的改进版本，计算速度更快，但精度略低。

基本原理

1. **积分图像**：
   - 使用积分图像加速卷积计算
   - 大大提高了特征检测速度
2. **特征检测**：
   - 使用Hessian矩阵检测特征点
   - 在尺度空间中寻找极值点
3. **特征描述**：
   - 计算特征点周围区域的Haar小波响应
   - 生成64维或128维的特征向量

```
// 创建SURF检测器
Ptr<SURF> surf = SURF::create(400);  // 阈值400
// 参数说明：
// hessianThreshold：Hessian矩阵阈值
// nOctaves：金字塔组数
```

```
// nOctaveLayers：每组金字塔的层数
// extended：是否使用扩展描述子（128维）

// 检测关键点和计算描述子
vector<KeyPoint> keypoints;
Mat descriptors;
surf->detectAndCompute(img, Mat(), keypoints, descriptors);
// 原理：使用积分图像加速特征检测和描述

// 绘制关键点
Mat surf_img;
drawKeypoints(img, keypoints, surf_img, Scalar::all(-1),
              DrawMatchesFlags::DRAW_RICH_KEYPOINTS);
```

7.5 ORB特征

ORB（定向FAST和旋转BRIEF）是一种快速的特征检测和描述算法，适合实时应用。

基本原理

1. **特征检测**：
   - 使用FAST算法检测角点
   - 计算角点的主方向
2. **特征描述**：
   - 使用改进的BRIEF描述子
   - 考虑特征点方向，实现旋转不变性
3. **特点**：
   - 计算速度快
   - 内存占用小
   - 适合实时应用

```
// 创建ORB检测器
Ptr<ORB> orb = ORB::create();
// 参数说明：
// nfeatures：特征点数量
// scaleFactor：金字塔缩放因子
// nlevels：金字塔层数
// edgeThreshold：边缘阈值
// firstLevel：第一层金字塔的索引
// WTA_K：BRIEF描述子的采样点数

// 检测关键点和计算描述子
vector<KeyPoint> keypoints;
Mat descriptors;
orb->detectAndCompute(img, Mat(), keypoints, descriptors);
// 原理：结合FAST角点检测和BRIEF描述子
```

```
// 绘制关键点
Mat orb_img;
drawKeypoints(img, keypoints, orb_img, Scalar::all(-1),
              DrawMatchesFlags::DRAW_RICH_KEYPOINTS);
```

8. 特征匹配

8.1 基本原理

1. **匹配目的**：
   - 找到两幅图像中对应的特征点
   - 用于图像配准、拼接、跟踪等
2. **匹配方法**：
   - 暴力匹配：计算所有特征点对之间的距离
   - FLANN匹配：使用近似最近邻搜索，速度更快
3. **匹配策略**：
   - 最近邻匹配
   - 比率测试
   - 交叉验证

8.2 特征描述子

```
// 计算两个图像的特征描述子
Mat img1 = imread("image1.jpg");
Mat img2 = imread("image2.jpg");

// 使用SIFT检测器
Ptr<SIFT> sift = SIFT::create();
vector<KeyPoint> keypoints1, keypoints2;
Mat descriptors1, descriptors2;

sift->detectAndCompute(img1, Mat(), keypoints1, descriptors1);
sift->detectAndCompute(img2, Mat(), keypoints2, descriptors2);
// 原理：计算每个特征点的描述子
// 描述子：表示特征点周围区域的特征
```

8.3 特征匹配算法

```
// 创建特征匹配器
BFMatcher matcher(NORM_L2);  // 暴力匹配器
// 或者使用FLANN匹配器
// FlannBasedMatcher matcher;
```

```
// 原理：计算描述子之间的距离
// NORM_L2：欧氏距离
// NORM_HAMMING：汉明距离（用于二进制描述子）

// 进行特征匹配
vector<DMatch> matches;
matcher.match(descriptors1, descriptors2, matches);
// 原理：找到每个特征点的最近邻匹配

// 筛选好的匹配点
double min_dist = 100;
for(int i = 0; i < descriptors1.rows; i++) {
    double dist = matches[i].distance;
    if(dist < min_dist) min_dist = dist;
}

vector<DMatch> good_matches;
for(int i = 0; i < descriptors1.rows; i++) {
    if(matches[i].distance < max(2*min_dist, 0.02)) {
        good_matches.push_back(matches[i]);
    }
}
// 原理：去除距离太大的匹配点
// 保留距离小于2倍最小距离的匹配点
```

8.4 单应性矩阵

```
// 提取匹配点的坐标
vector<Point2f> points1, points2;
for(int i = 0; i < good_matches.size(); i++) {
    points1.push_back(keypoints1[good_matches[i].queryIdx].pt);
    points2.push_back(keypoints2[good_matches[i].trainIdx].pt);
}

// 计算单应性矩阵
Mat H = findHomography(points1, points2, RANSAC);
// 原理：通过匹配点对计算变换矩阵
// RANSAC：随机采样一致性，去除错误匹配

// 使用单应性矩阵进行图像变换
Mat warped;
warpPerspective(img1, warped, H, img1.size());
// 原理：将第一幅图像变换到第二幅图像的视角
```

8.5 图像拼接

```
// 创建拼接器
Ptr<Stitcher> stitcher = Stitcher::create(Stitcher::PANORAMA);
```

```
// 原理：使用特征匹配和单应性矩阵进行图像拼接

// 准备输入图像
vector<Mat> images;
images.push_back(img1);
images.push_back(img2);

// 执行拼接
Mat panorama;
Stitcher::Status status = stitcher->stitch(images, panorama);
// 步骤：
// 1. 检测特征点
// 2. 特征匹配
// 3. 计算变换矩阵
// 4. 图像变换
// 5. 图像融合

if(status == Stitcher::OK) {
    imwrite("panorama.jpg", panorama);
} else {
    cout << "拼接失败！" << endl;
}
```

9. 特征点跟踪

9.1 基本原理

1. **跟踪目的**：
   - 在视频序列中跟踪目标
   - 用于目标跟踪、运动分析等
2. **跟踪方法**：
   - 光流法：基于像素灰度变化
   - 特征点跟踪：基于特征点匹配
3. **应用场景**：
   - 目标跟踪
   - 运动分析
   - 视频稳定

9.2 光流法

```
// 计算光流
vector<Point2f> prev_points, curr_points;
vector<uchar> status;
vector<float> err;

// 检测初始特征点
```

```
goodFeaturesToTrack(prev_gray, prev_points, 100, 0.01, 10);
// 参数说明：
// 100：最大特征点数量
// 0.01：最小特征值
// 10：最小距离

// 计算光流
calcOpticalFlowPyrLK(prev_gray, curr_gray,
                     prev_points, curr_points,
                     status, err);
// 原理：基于像素灰度变化计算运动
// 使用金字塔LK算法提高精度

// 筛选好的跟踪点
vector<Point2f> good_points;
for(int i = 0; i < status.size(); i++) {
    if(status[i]) {
        good_points.push_back(curr_points[i]);
    }
}
// 原理：去除跟踪失败的点
```

9.3 特征点跟踪

```
// 创建特征跟踪器
Ptr<Tracker> tracker = TrackerKCF::create();
// 原理：使用核相关滤波器进行跟踪
// 特点：速度快，精度高

// 初始化跟踪器
Rect2d bbox(100, 100, 100, 100);  // 初始边界框
tracker->init(img, bbox);
// 原理：在第一帧初始化跟踪器

// 跟踪目标
while(true) {
    // 读取新帧
    cap >> frame;
    if(frame.empty()) break;

    // 更新跟踪器
    bool ok = tracker->update(frame, bbox);
    // 原理：在新帧中更新目标位置

    // 绘制跟踪结果
    if(ok) {
        rectangle(frame, bbox, Scalar(0,255,0), 2);
    }
}
// 原理：逐帧跟踪目标
// 应用：目标跟踪、运动分析
```