计算机视觉基础实验报告（三）

1811361 物联网 郭宇

一、实验要求

自己实现SIFT算法，并测试特征点检测是否正常。

二、实验环境

opencv4.5.1

opencv contrib 4.5.1

visual studio 2019 x64

三、实验步骤

SIFT的工作过程可以分为以下几个步骤：预处理，构建高斯金字塔和高斯差值金字塔，检测特征点，提取特征点的特征向量。其中检测特征点中还包括：寻找极值点，筛选极值点，计算特征点主方向。

3.0 预处理

```
// 计算Octave 默认金字塔最上层 图像长宽较小值为8
    Octaves = round(log(float(min(img_org.cols, img_org.rows))) / log(2.f) - 2.f);

    // 图像灰度化 并且 像素转化为float类型
    Mat img_gray, img_gray_f;
    if (img_org.channels() == 3 || img_org.channels() == 4)
        cvtColor(img_org, img_gray, COLOR_BGR2GRAY);
    else
        img_org.copyTo(img_gray);
    img_gray.convertTo(img_gray_f, CV_32FC1);

    // INTER_LINEAR 双线性差值
    resize(img_gray_f, img, Size(img_gray_f.cols * 2, img_gray_f.rows * 2), 0, 0, INTER_LINEAR);
    // 插值后滤波
    double sigma_init = sqrt(max(Sigma * Sigma - 0.5 * 0.5 * 4, 0.01));
    GaussianBlur(img, img, Size(2 * cvCeil(2 * sigma_init) + 1, 2 * cvCeil(2 * sigma_init) + 1), sigma_init, sigma_init);
```

对于初始图像的预处理，为了后续便于操作和计算。

3.1 构建金字塔

```
void sift::buildPyramid()
{
    // 先计算好sigma
    vector<double> sigma_i(Layers + 1);
    sigma_i[0] = Sigma;
```

```
    for (int lyr_i = 1; lyr_i < Layers + 1; lyr_i++) // 0  1 2 3 4 5
    {
        double sigma_prev = pow(K, lyr_i - 1) * Sigma;
        double sigma_curr = K * sigma_prev;
        sigma_i[lyr_i] = sqrt(sigma_curr * sigma_curr - sigma_prev * sigma_prev);
    }

    // 金子塔初始化
    pyr_G.clear();
    pyr_DoG.clear();

    // 生成金字塔
    Mat img_i, img_DoG;
    img.copyTo(img_i);
    pyr_G.build(Octaves); // 确定金字塔层数
    pyr_DoG.build(Octaves);
    for (int oct_i = 0; oct_i < Octaves; oct_i++)
    {
        pyr_G.appendTo(oct_i, img_i); // 每层第一张图像不需要高斯模糊
        // 生成一个octave其他高斯模糊图像 同时生成这一个octaveDoG图像
        for (int lyr_i = 1; lyr_i < Layers + 1; lyr_i++) // 0    1 2 3 4 5
        {
            GaussianBlur(img_i, img_i, Size(2 * cvCeil(2 * sigma_i[lyr_i]) + 1, 2 * cvCeil(2 * sigma_i[lyr_i]) + 1), sigma_i[lyr_i], sigma_i[lyr_i]);
            pyr_G.appendTo(oct_i, img_i);
            subtract(img_i, pyr_G[oct_i][lyr_i - 1], img_DoG, noArray(), CV_32FC1);
            pyr_DoG.appendTo(oct_i, img_DoG);
        }

        // 降采样生成下一个octave的第一张图像
        resize(pyr_G[oct_i][Layers - 2], img_i, Size(img_i.cols / 2, img_i.rows / 2), 0, 0, INTER_NEAREST);
    }
}
```

通过高斯模糊和降采样并对两张图片之间做差值得到了高斯金字塔和高斯差值金字塔。

3.2 检测特征点

3.2.1 寻找极值点

```
void sift::findFeaturePoints(vector<keypoint>& kpts)
{
    // 清空kpts
    kpts.clear();

    // 迭代变量
    int oct_i, lyr_i, r, c, pr, pc, kpt_i, ang_i, k;

    // 27个像素点容器
    float pxs[27];

    // 极值点暂存序列
    vector<keypoint> kpts_temp;

```

```cpp
    // 寻找极值点
    int threshold = cvFloor(0.5 * 0.04 / S * 255);
    for (oct_i = 0; oct_i < Octaves; oct_i++)
    {
        for (lyr_i = 1; lyr_i < Layers - 1; lyr_i++) // 0  123  4
        {
            const Mat& img_curr = pyr_DoG[oct_i][lyr_i];
            // 遍历像素 排除最外层像素
            for (r = 1; r < img_curr.rows - 1; r++)
            {
                for (c = 1; c < img_curr.cols - 1; c++)
                {
                    // 取出比较像素块
                    const Mat& prev = pyr_DoG[oct_i][lyr_i - 1];
                    const Mat& curr = pyr_DoG[oct_i][lyr_i];
                    const Mat& next = pyr_DoG[oct_i][lyr_i + 1];
                    float px = curr.at<float>(r, c);

                    if (abs(px) >= threshold)
                    {
                        // 取出比较像素 -1 0 1
                        for (pr = -1, k = 0; pr < 2; pr++)
                            for (pc = -1; pc < 2; pc++)
                            {
                                pxs[k] = prev.at<float>(r + pr, c + pc);
                                pxs[k + 1] = curr.at<float>(r + pr, c + pc);
                                pxs[k + 2] = next.at<float>(r + pr, c + pc);
                                k += 3;
                            }

                        if ((px >= pxs[0] && px >= pxs[1] && px >= pxs[2] && px >= pxs[3] && px >= pxs[4] &&
                            px >= pxs[5] && px >= pxs[6] && px >= pxs[7] && px >= pxs[8] && px >= pxs[9] &&
                            px >= pxs[10] && px >= pxs[11] && px >= pxs[12] && px >= pxs[14] && px >= pxs[15] &&
                            px >= pxs[16] && px >= pxs[17] && px >= pxs[18] && px >= pxs[19] && px >= pxs[20] &&
                            px >= pxs[21] && px >= pxs[22] && px >= pxs[23] && px >= pxs[24] && px >= pxs[25] &&
                            px >= pxs[26]) ||
                            (px <= pxs[0] && px <= pxs[1] && px <= pxs[2] && px <= pxs[3] && px <= pxs[4] &&
                                px <= pxs[5] && px <= pxs[6] && px <= pxs[7] && px <= pxs[8] && px <= pxs[9] &&
                                px <= pxs[10] && px <= pxs[11] && px <= pxs[12] && px <= pxs[14] && px <= pxs[15] &&
                                px <= pxs[16] && px <= pxs[17] && px <= pxs[18] && px <= pxs[19] && px <= pxs[20] &&
                                px <= pxs[21] && px <= pxs[22] && px <= pxs[23] && px <= pxs[24] && px <= pxs[25] &&
                                px <= pxs[26]))
                        {
                            keypoint kpt(oct_i, lyr_i, Point(r, c));
                            kpts_temp.push_back(kpt);// 如果是极值点先保存
                        }
                    }
                }
```

```
            }
        }
    }

    for (kpt_i = 0; kpt_i < kpts_temp.size(); kpt_i++)
    {
        if (!filterExtrema(kpts_temp[kpt_i]))       //如果是合适的极值点，计算其主方向
            continue;

        vector<float> angs;
        calcMainOrientation(kpts_temp[kpt_i], angs);

        for (ang_i = 0; ang_i < angs.size(); ang_i++)
        {
            kpts_temp[kpt_i].angle = angs[ang_i];
            kpts.push_back(kpts_temp[kpt_i]);
        }
    }
}
```

对于每个像素，检测它本层周围8个，上层9个，下层9个共26个点，判断其是否是最大值或者极小值，如果是极值，先存储起来，等待进行筛选。

3.2.2 筛选极值点

```
bool sift::filterExtrema(keypoint& kpt)
{
    // 用到的阈值
    float biasThreshold = 0.5f;
    float contrastThreshold = 0.04f;
    float edgeThreshold = 10.f;

    float normalscl = 1.f / 255; // 将图像从 0~255 归一化到 0~1 的因子

    // 筛选步骤 1 去除与精确极值偏移较大的点
    bool isdrop = true; // 是否删除该极值点标识

    // 其他筛选步骤用得上的变量 预先声明
    // 取极值点坐标
    int kpt_r = kpt.pt.x;
    int kpt_c = kpt.pt.y;
    int kpt_lyr = kpt.layer;
    // 导数
    Vec3f dD;
    float dxx, dyy, dss, dxy, dxs, dys;
    // 偏差值
    Vec3f x_hat;

    // 5次插值逼近真实极值
    for (int try_i = 0; try_i < 5; try_i++)
    {
        // 取DoG图像 每次都要重新取 因为layer会变
        const Mat& DoG_prev = pyr_DoG[kpt.octave][kpt_lyr - 1];
        const Mat& DoG_curr = pyr_DoG[kpt.octave][kpt_lyr];
        const Mat& DoG_next = pyr_DoG[kpt.octave][kpt_lyr + 1];

```

```
        // 计算一阶导
        dD = Vec3f((DoG_curr.at<float>(kpt_r, kpt_c + 1) - DoG_curr.at<float>(kpt_r, kpt_c - 1)) * normalscl * 0.5f,
            (DoG_curr.at<float>(kpt_r + 1, kpt_c) - DoG_curr.at<float>(kpt_r - 1, kpt_c)) * normalscl * 0.5f,
            (DoG_prev.at<float>(kpt_r, kpt_c) - DoG_next.at<float>(kpt_r, kpt_c)) * normalscl * 0.5f);

        // 计算二阶导(Hessian矩阵)   注意分母为1
        dxx = (DoG_curr.at<float>(kpt_r, kpt_c + 1) + DoG_curr.at<float>(kpt_r, kpt_c - 1) - 2 * DoG_curr.at<float>(kpt_r, kpt_c)) * normalscl;
        dyy = (DoG_curr.at<float>(kpt_r + 1, kpt_c) + DoG_curr.at<float>(kpt_r - 1, kpt_c) - 2 * DoG_curr.at<float>(kpt_r, kpt_c)) * normalscl;
        dss = (DoG_next.at<float>(kpt_r, kpt_c) + DoG_prev.at<float>(kpt_r, kpt_c) - 2 * DoG_curr.at<float>(kpt_r, kpt_c)) * normalscl;

        // 混合导 注意分母为4
        dxy = (DoG_curr.at<float>(kpt_r + 1, kpt_c + 1) - DoG_curr.at<float>(kpt_r + 1, kpt_c - 1)
            - DoG_curr.at<float>(kpt_r - 1, kpt_c + 1) + DoG_curr.at<float>(kpt_r - 1, kpt_c - 1)) * normalscl * 0.25f;
        dxs = (DoG_next.at<float>(kpt_r, kpt_c + 1) - DoG_next.at<float>(kpt_r, kpt_c - 1)
            - DoG_prev.at<float>(kpt_r, kpt_c + 1) + DoG_prev.at<float>(kpt_r, kpt_c - 1)) * normalscl * 0.25f;
        dys = (DoG_next.at<float>(kpt_r + 1, kpt_c) - DoG_next.at<float>(kpt_r - 1, kpt_c)
            - DoG_prev.at<float>(kpt_r + 1, kpt_c) + DoG_prev.at<float>(kpt_r - 1, kpt_c)) * normalscl * 0.25f;

        // 由二阶导合成Hessian矩阵
        Matx33f H(dxx, dxy, dxs,
            dxy, dyy, dys,
            dxs, dys, dss);

        // 求解偏差值
        x_hat = Vec3f(H.solve(dD, DECOMP_LU));

        for (int x = 0; x < 3; x++)// 注意这里有个 - 号
            x_hat[x] *= -1;

        // 调整极值点坐标 以便进行下一次插值计算
        kpt_c += round(x_hat[0]);
        kpt_r += round(x_hat[1]);
        kpt_lyr += round(x_hat[2]);

        // 判断偏移程度  注意sigma的偏移也要考虑  满足直接通过筛选  通过筛选唯一出口
        if (abs(x_hat[0]) < biasThreshold && abs(x_hat[1]) < biasThreshold && abs(x_hat[2]) < biasThreshold) // 阈值为 0.5
        {
            isdrop = false;
            break;
        }

        // 判断下调整后的坐标是否超过边界(最外面一圈像素也不算)   超过跳出   否则再次求调整后的偏差
        if (kpt_r < 1 || kpt_r > DoG_curr.rows - 2 ||
            kpt_c < 1 || kpt_c > DoG_curr.cols - 2 ||
```

```
            kpt_lyr < 1 || kpt_lyr > Layers - 2) // 不能取第一张与最后一张 0 1 2 3   4
        {
            break;
        }
    }

    // 如果该点已删除 返回false
    if (isdrop)
        return false;

    // 筛选步骤 2 去除响应过小的极值点 阈值越大放过的点越多
    // 重新取DoG图像

    const Mat& DoG_curr = pyr_DoG[kpt.octave][kpt_lyr];

    float D_hat = DoG_curr.at<float>(kpt_r, kpt_c) * normalscl + dD.dot(x_hat) * 0.5f; // 待考???  vec3f转Matx31f
    if (abs(D_hat) * S < contrastThreshold) // 在opencv的SIFT实现里有 "* s" 但原因不明???
        return false;

    // 筛选步骤 3 去除边缘关键点
    // 计算Hessian矩阵的迹与行列式
    float trH = dxx + dyy;
    float detH = dxx * dyy - dxy * dxy;

    if (detH <= 0 || trH * trH * edgeThreshold >= (edgeThreshold + 1) * (edgeThreshold + 1) * detH)
        return false;

    // 到这里 极值点可以保留了 更新极值点信息
    kpt.pt = Point(kpt_r, kpt_c);
    kpt.layer = kpt_lyr;
    kpt.scale = pow(K, kpt_lyr) * Sigma; // 特征点所在尺度空间的尺度
    kpt.response = D_hat;
    kpt.oct_info = kpt.octave + (kpt.layer << 8) + (cvRound((x_hat[2] + 0.5) * 255) << 16);
    return true;
}
```

这一步共有三项工作，先通过计算Hessian矩阵来得到某个位置的理论极值，再与此处的真实极值做对比，如果在偏差值之内，既可作为极值点，这一步排除了与理论值偏差较大的点；然后去除响应过小的极值点，减少了极值点数量；最后通过计算Hessian矩阵的迹与行列式并与阈值进行比较即可排除边缘关键点，消除边缘效应。

3.2.3 计算特征点主方向

```
void sift::calcMainOrientation(keypoint& kpt, vector<float>& angs)
{
    // 取出特征点信息
    int kpt_oct = kpt.octave;
    int kpt_lyr = kpt.layer;
    int kpt_r = kpt.pt.x;
    int kpt_c = kpt.pt.y;
    double kpt_scl = kpt.scale;

```

```
    int radius = round(3 * 1.5 * kpt_scl); // 参与计算像素区域半径
    const Mat& pyr_G_i = pyr_G[kpt_oct][kpt_lyr]; // 取出高斯模糊图像

    int px_r, px_c; // 遍历到像素的绝对坐标(img_r, img_c)

    float histtemp[36 + 4] = { 0 }; // 两边各多出的2个空位便于插值运算    0  1  2~37  38  39

    // 遍历参与计算像素区域   (radius * 2 + 1)*(radius * 2 + 1)
    for (int i = -radius; i <= radius; i++) // 从上到下 i是行数
    {
        px_r = kpt_r + i;
        if (px_r <= 0 || px_r >= pyr_G_i.rows - 1) // 坐标超出图像 且不能是图像最边缘像素
            continue;

        for (int j = -radius; j <= radius; j++) // 从左到右 j是列数
        {
            px_c = kpt_c + j;
            if (px_c <= 0 || px_c >= pyr_G_i.cols - 1)  // 坐标超出图像 且不能是图像最边缘像素
                continue;

            // 计算梯度 注意dy的计算
            float dx = pyr_G_i.at<float>(px_r, px_c + 1) - pyr_G_i.at<float>(px_r, px_c - 1);
            float dy = pyr_G_i.at<float>(px_r - 1, px_c) - pyr_G_i.at<float>(px_r + 1, px_c);

            // 计算梯度 幅值 与 幅角
            float mag = sqrt(dx * dx + dy * dy);
            float ang = fastAtan2(dy, dx);

            // 计算落入的直方图柱数
            int bin = round(ang * 36.f / 360.f);

            if (bin >= 36) // bin取 0~35
                bin -= 36;
            else if (bin < 0)
                bin += 36;

            // 计算高斯加权项
            float w_G = exp(-(i * i + j * j) / (2 * (1.5 * kpt_scl) * (1.5 * kpt_scl)));

            // 存入histtemp   0  1  2~37  38  39
            histtemp[bin + 2] += mag * w_G;
        }
    }

    // 对直方图平滑处理 填充histtemp的空位      0  1  2~37  38  39
    float hist[36] = { 0 };// 两边各多出的2个空位便于插值运算    0  1  2~37  38  39

    histtemp[0] = histtemp[36];
    histtemp[1] = histtemp[37];
    histtemp[38] = histtemp[2];
    histtemp[39] = histtemp[3];
```

```
61
62      // 加权移动平均 求 最大幅值
63      float hist_max = 0;
64      for (int k = 2; k < 40 - 2; k++)
65      {
66          hist[k - 2] = (histtemp[k - 2] + histtemp[k + 2]) * (1.f / 16) +
67              (histtemp[k - 1] + histtemp[k + 1]) * (4.f / 16) +
68              histtemp[k] * (6.f / 16);
69          // 顺便求最大幅值
70          if (hist[k - 2] > hist_max)
71              hist_max = hist[k - 2];
72      }
73
74      // 遍历直方图 求 主方向 与 辅方向
75      float histThreshold = 0.8f * hist_max; // 计算幅度阈值
76
77      for (int k = 0; k < 36; k++) // 0 ~ 35
78      {
79          int kl = k > 0 ? k - 1 : 36 - 1;
80          int kr = k < 36 - 1 ? k + 1 : 0;
81
82          if (hist[k] > hist[kl] && hist[k] > hist[kr] && hist[k] >=
    histThreshold)
83          {
84              // 通过 抛物线插值 计算精确幅角公式  精确值范围  0 ~ 35...  |36
85              float bin = k + 0.5f * (hist[kl] - hist[kr]) / (hist[kl] - 2 *
    hist[k] + hist[kr]);
86
87              if (bin < 0) // bin越界处理
88                  bin += 36;
89              else if (bin >= 36)
90                  bin -= 36;
91
92              // 计算精确幅角  ang为角度制
93              float ang = bin * (360.f / 36);
94              if (abs(ang - 360.f) < FLT_EPSILON)
95                  ang = 0.f;
96
97              // 保存该极值点 主方向 与 辅方向
98              angs.push_back(ang);
99          }
100     }
101 }
102
```

通过计算像素点x和y方向的导数即可获得其对应的角度。对在特征点周围一定范围内的所有像素求其角度，并将其落入角度分布直方图中，直方图横坐标为0到360度，每10度为一个区间，直方图纵坐标上最大的值对应的角度即为此特征点的主方向，而对于其他角度，如果其纵坐标大于最大值的80%即作为此特征点的辅方向同样存储起来。

3.3 计算特征向量

```
1  void sift::calcFeatureVector(vector<keypoint>& kpts, Mat& fvec)
2  {
3      // fvec是特征向量组成的矩阵 每一行是一个特征点的128维向量 先申请内存空间
4      fvec.create(kpts.size(), 128, CV_32FC1);
```

```
    for (int kpt_i = 0; kpt_i < kpts.size(); kpt_i++)
    {
        // 迭代变量
        int i, j, k, ri, ci, oi;

        // 取出特征点信息
        int kpt_oct = kpts[kpt_i].octave;
        int kpt_lyr = kpts[kpt_i].layer;
        int kpt_r = kpts[kpt_i].pt.x;
        int kpt_c = kpts[kpt_i].pt.y;
        double kpt_scl = kpts[kpt_i].scale;
        float kpt_ang = kpts[kpt_i].angle;

        int d = 4; // 子区间数目 就是 4 * 4 * 8 中的4
        int n = 8; // 梯度直方图柱数目 8个幅角区间

        // 取出对应尺度的高斯模糊图像
        const Mat& pyr_G_i = pyr_G[kpt_oct][kpt_lyr];

        float hist_width = 3 * kpt_scl; // 子区域边长  方域

        // 计算旋转矩阵参数 cos接受的是弧度制
        // 并对旋转矩阵以子区域坐标尺度进行归一化 /hist_width  以便于之后计算出的旋转相对坐标(r_rot, c_rot)是子区域坐标尺度的
        float cos_t = cosf(kpt_ang * CV_PI / 180) / hist_width;
        float sin_t = sinf(kpt_ang * CV_PI / 180) / hist_width;

        int radius = round(hist_width * 1.4142135623730951f * (d + 1) * 0.5); // 计算所有参与计算像素区域半径   圆域

        // 判断下计算出的半径与图像对角线长 最后半径取小的
        // 当半径和图像对角线一样大时是一个极限 再大不过也是遍历图像所有像素 所以没有意义
        radius = min(radius, (int)sqrt(pyr_G_i.rows * pyr_G_i.rows + pyr_G_i.cols * pyr_G_i.cols));

        // 预分配空间
        int histlen = (d + 2) * (d + 2) * (n + 2);
        AutoBuffer<float> hist(histlen);
        // 初始化hist
        memset(hist, 0, sizeof(float) * histlen);

        // 依旧遍历区域内所有像素点 计算直方图
        for (i = -radius; i <= radius; i++) // 从上到下 i是行数
        {
            for (j = -radius; j <= radius; j++) // 从左到右 j是列数
            {
                // 计算旋转后的相对坐标(注意是子区域坐标尺度的)
                float c_rot = j * cos_t - i * sin_t;
                float r_rot = j * sin_t + i * cos_t;

                float rbin = r_rot + d / 2.f - 0.5f; // 计算子区域坐标尺度下的绝对坐标 并做了0.5的平移
                float cbin = c_rot + d / 2.f - 0.5f; // 取值范围  -1 ~ 3...|4

```

```
                // 0.5的平移使得子区域坐标轴交点都落在子区域左上角 便于之后插值 计算一个像素点对周围四个子区域的直方图贡献
                // 计算像素图像坐标绝对(px_r, px_c)
                int px_r = kpt_r + i;
                int px_c = kpt_c + j;

                // 如果旋转后坐标仍在图像内 参与直方图计算
                if (-1 < rbin && rbin < d && -1 < cbin && cbin < d &&
                    0 < px_r && px_r < pyr_G_i.rows - 1 && 0 < px_c && px_c < pyr_G_i.cols - 1)
                {
                    // 计算梯度 注意 dy的计算
                    float dx = pyr_G_i.at<float>(px_r, px_c + 1) - pyr_G_i.at<float>(px_r, px_c - 1);
                    float dy = pyr_G_i.at<float>(px_r - 1, px_c) - pyr_G_i.at<float>(px_r + 1, px_c);

                    // 计算梯度 幅值 与 幅角
                    float mag = sqrt(dx * dx + dy * dy);
                    float ang = fastAtan2(dy, dx);

                    // 判断幅角落在哪个直方图的柱内
                    float obin = (ang - kpt_ang) * (n / 360.f); // 取值 0 ~ 7... | 8

                    // 计算高斯加权后的幅值
                    float w_G = expf(-(r_rot * r_rot + c_rot * c_rot) / (0.5f * d * d)); // 计算高斯加权项 - 1 / ((d / 2) ^ 2 * 2)   -> - 1 / (d ^ 2 * 0.5)
                    mag *= w_G;

                    int r0 = cvFloor(rbin); // -1 0 1 2 3
                    int c0 = cvFloor(cbin);
                    int o0 = cvFloor(obin); // 0 ~ 7

                    // 相当于立方体内点坐标
                    rbin -= r0;
                    cbin -= c0;
                    obin -= o0;

                    // 柱数o0越界循环
                    if (o0 < 0)
                        o0 += n;
                    else if (o0 >= n)
                        o0 -= n;

                    // 三线性插值 填充hist的内容 将像素点幅值贡献到周围四个子区域的直方图中去
                    // 计算8个贡献值    0 < rbin cbin obin < 1
                    // 先计算贡献权值
                    float v_rco000 = rbin * cbin * obin;
                    float v_rco001 = rbin * cbin * (1 - obin);

                    float v_rco010 = rbin * (1 - cbin) * obin;
                    float v_rco011 = rbin * (1 - cbin) * (1 - obin);

                    float v_rco100 = (1 - rbin) * cbin * obin;
                    float v_rco101 = (1 - rbin) * cbin * (1 - obin);
```

```
106
107                     float v_rco110 = (1 - rbin) * (1 - cbin) * obin;
108                     float v_rco111 = (1 - rbin) * (1 - cbin) * (1 - obin);
109
110                     // rbin     0 ~ 5          r0  -1 ~ 3
111                     // cbin     0 ~ 5          c0  -1 ~ 3
112                     // obin     0 ~ 7 | 8 9  插值会到8     o0   0 ~ 7
113                     hist[60 * (r0 + 1) + 10 * (c0 + 1) + o0] += mag *
    v_rco000;
114                     hist[60 * (r0 + 1) + 10 * (c0 + 1) + (o0 + 1)] += mag *
    v_rco001;
115
116                     hist[60 * (r0 + 1) + 10 * (c0 + 2) + o0] += mag *
    v_rco010;
117                     hist[60 * (r0 + 1) + 10 * (c0 + 2) + (o0 + 1)] += mag *
    v_rco011;
118
119                     hist[60 * (r0 + 2) + 10 * (c0 + 1) + o0] += mag *
    v_rco100;
120                     hist[60 * (r0 + 2) + 10 * (c0 + 1) + (o0 + 1)] += mag *
    v_rco101;
121
122                     hist[60 * (r0 + 2) + 10 * (c0 + 2) + o0] += mag *
    v_rco110;
123                     hist[60 * (r0 + 2) + 10 * (c0 + 2) + (o0 + 1)] += mag *
    v_rco111;
124                 }
125             }
126         }
127
128         // 遍历 4 * 4 子区域  从hist直方图中导出特征向量
129         float fvec_i[128] = { 0 };
130         for (ri = 1, k = 0; ri <= 4; ri++)
131             for (ci = 1; ci <= 4; ci++)
132             {
133                 hist[60 * ri + 10 * ci + 0] += hist[60 * ri + 10 * ci + 8];
134                 hist[60 * ri + 10 * ci + 1] += hist[60 * ri + 10 * ci + 9];
135
136                 for (oi = 0; oi < 8; oi++)
137                     fvec_i[k++] = hist[60 * ri + 10 * ci + oi];
138             }
139
140         // 特征向量优化
141         float scl;
142         float fvec_norm = 0, fvecThreshold;
143         for (k = 0; k < 128; k++)
144             fvec_norm += fvec_i[k] * fvec_i[k];
145
146         fvecThreshold = 0.2f * sqrtf(fvec_norm);
147
148         // 对特征向量幅值大的分量进行处理  改善非线性光照改变影响
149         for (k = 0, fvec_norm = 0; k < 128; k++)
150         {
151             if (fvec_i[k] > fvecThreshold)
152                 fvec_i[k] = fvecThreshold;
153             fvec_norm += fvec_i[k] * fvec_i[k];
154         }
155
```

```
156             // 归一化 保存处理完的特征向量
157             scl = 1 / max(std::sqrt(fvec_norm), FLT_EPSILON);
158             float* fvec_temp = fvec.ptr<float>(kpt_i);
159             for (k = 0; k < 128; k++)
160                 fvec_temp[k] = fvec_i[k] * scl;
161         }
162     }
```

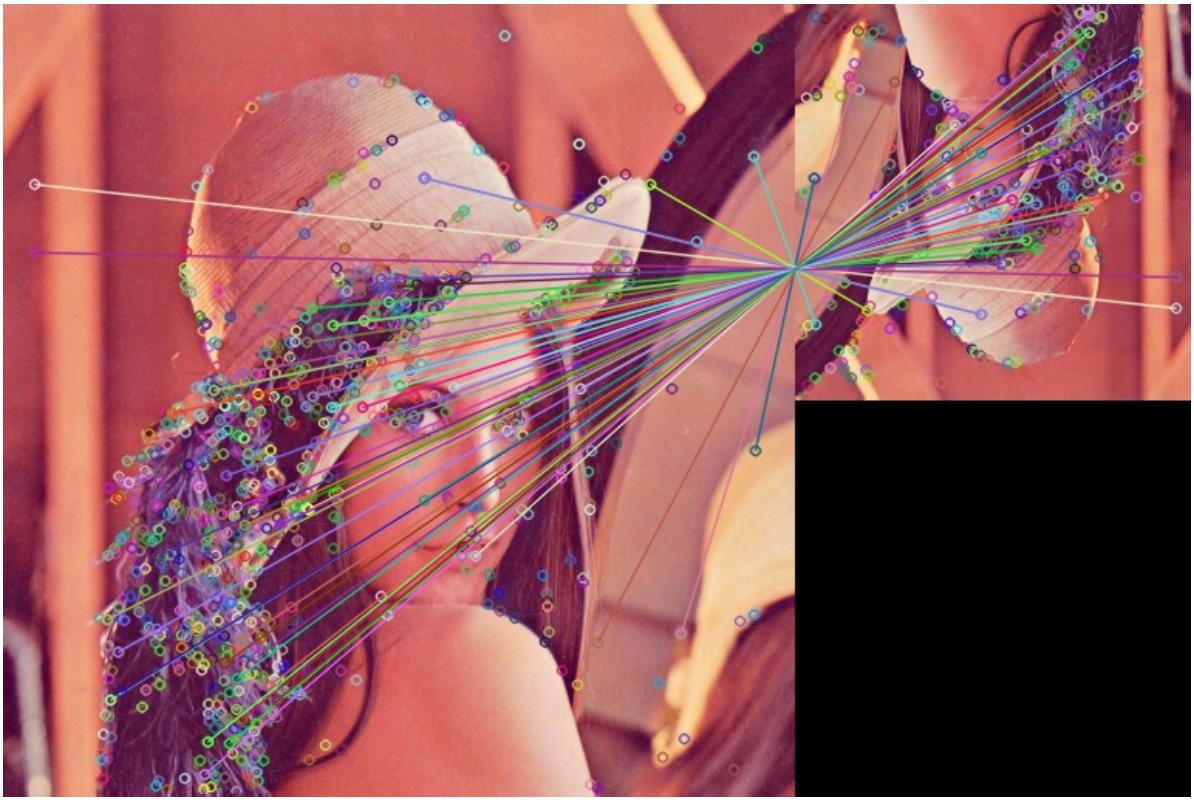
一个关键点点所包含的信息由特征描述子(Feature Descriptor)数值描述. 在SIFT中, 特征描述子是从关键点与其主方向确定的区域提取得到的一个128维特征向量.

特征向量的计算依旧需要用到统计梯度直方图. 具体来说, 统计4×4个子区间内像素(16个像素区间)的梯度(幅值与幅角), 与之前相同, 将幅角从360°每45°为一个区间划分作为横轴, 共8个区间, 纵轴为在对应幅角区间内的像素点幅值累加, 以此作出统计直方图.这样每个子区域都包含8个柱状图的信息, 所以对于一个关键点可提取4×4×8共128维的特征向量.

3.4 在主函数中调用进行检测

对于特征点的匹配使用了BF树进行计算匹配。

四、实验结果



五、实验感想

SIFT相较于SURF算法减少了计算量，从而使得得到特征点的速度点变得更快。并且SIFT算法在opencv4.5之后的版本已经开放了版权，未来可以直接免费使用SIFT算法进行特征点检测。