2 ORB 特征点(4 分，约 2.5 小时)

1 自行书写 ORB 的提取、描述子的计算以及匹配的代码

计算角度

```
// START YOUR CODE HERE (~7 lines)
kp.angle = 0; // compute kp.angle
if (kp.pt.x >= (image.cols - 16) || kp.pt.y >= (image.rows - 16) || kp.pt.x < 16 || kp.pt.y < 16) { continue; }
float m01 = 0;
float m10 = 0;
for (int x_shift = -8; x_shift < 8; x_shift++) {
    for (int y_shift = -8; y_shift < 8; y_shift++) {
        uchar pixel = image.at<uchar>( i0: kp.pt.y + y_shift,  i1: kp.pt.x + x_shift);
        m01 += y_shift * pixel;
        m10 += x_shift * pixel;
    }
}
kp.angle = atan2(m01, m10);
// END YOUR CODE HERE
```

计算描述子

```
// START YOUR CODE HERE (~7 lines)
d[i] = 0; // if kp goes outside, set d.clear()
if (kp.pt.x >= (image.cols - 16) || kp.pt.y >= (image.rows - 16) || kp.pt.x < 16 || kp.pt.y < 16) {
    d.clear();
    continue;
}
cv::Point2f p( _x: ORB_pattern[i * 4],  _y: ORB_pattern[i * 4 + 1]);
cv::Point2f q( _x: ORB_pattern[i * 4 + 2],  _y: ORB_pattern[i * 4 + 3]);
float cos_theta = cos(kp.angle);
float sin_theta = sin(kp.angle);
cv::Point2f pp = cv::Point2f( _x: cos_theta * p.x - sin_theta * p.y,  _y: sin_theta * p.x + cos_theta * p.y) + kp.pt;
cv::Point2f qq = cv::Point2f( _x: cos_theta * q.x - sin_theta * q.y,  _y: sin_theta * q.x + cos_theta * q.y) + kp.pt;
if (image.at<uchar>(pp.y, pp.x) < image.at<uchar>(qq.y, qq.x)) {
    d[i] = true;
}
// END YOUR CODE HERE
```

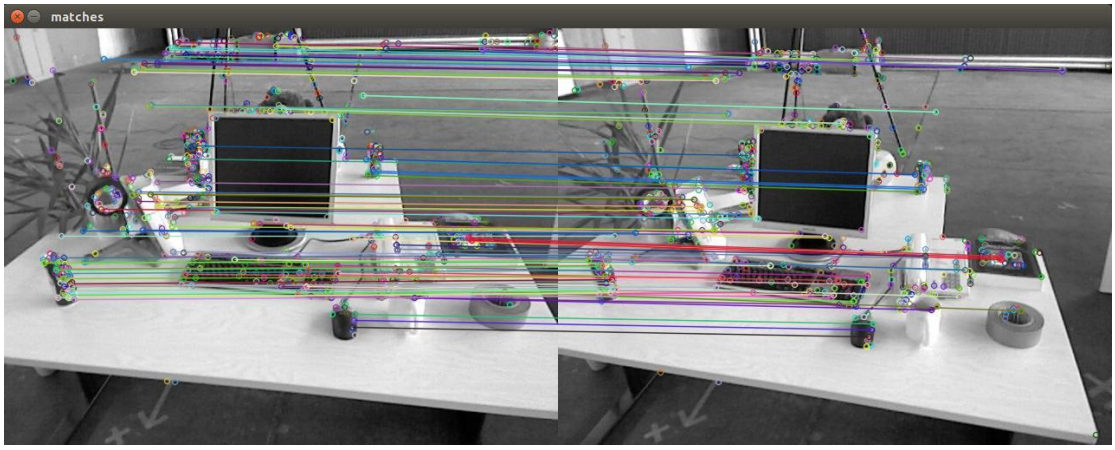
暴力匹配

```
// START YOUR CODE HERE (~12 lines)
// find matches between desc1 and desc2.
for (size_t index1 = 0; index1 < desc1.size(); index1++) {
    if (desc1[index1].empty())
        continue;
    cv::DMatch m{index1,  _trainIdx: 0,  _distance: 256};
    for (size_t index2 = 0; index2 < desc2.size(); index2++) {
        if (desc2[index2].empty())
            continue;
        int distance = 0;
        for (int k = 0; k < 256; k++) {
            if (desc1[index1][k] ^ desc2[index2][k])
                distance = distance + 1;
        }
        if (distance < d_max && distance < m.distance) {
            m.distance = distance;
            m.trainIdx = index2;
        }
    }
    if (m.distance < d_max) {
        matches.push_back(m);
    }
}
// END YOUR CODE HERE
```

特征点筛选结果

```
/home/xin/VSLAM-course/vSLAM-course/Ch5/SLAM第五讲作业资料/L5/code/cmake-build-debug/orb_calc
keypoints: 638
bad/total: 43/638
keypoints: 595
bad/total: 8/595
```

匹配结果



2 最后，请结合实验，回答下面几个问题：

1 为什么说 ORB 是一种二进制特征？

因为在 ORB 中我们一般用 128/256 位二进制数来描述特征点的特征。

2 为什么在匹配时使用 50 作为阈值，取更大或更小值会怎么样？

因为每一个点都会找到一个对应的匹配，如果不设置阈值即便是 256 这样的“远”的距离也可能被作为最佳匹配输出。取更大的阈值会找到更多的匹配则更有可能输出错误匹配，更小的阈值则获得的匹配对数量可能更少。

3. 暴力匹配在你的机器上表现如何？你能想到什么减少计算量的匹配方法吗？

暴力匹配每次都要独立的比较 P 某个点和 Q 中所有点的距离，明显要等一下才出现结果。

可以考虑用树来表示两组点，比较树结构中的上层节点，而不需要遍历每个叶子节点。亦或者利用机器学习的方法离线训练匹配算法。

也可以根据运动模型，预测每个特征点在下一帧图像的位置，在那个预测位置附近寻找匹配。

3 从 E 恢复 R;t (3 分，约 1 小时)

代码，包括：SVD 分解，奇异值重构成（sig1+sig2/2, sig1+sig2/2, 0）的形式。
从 UV，singular 和 R_z(90), R_z(-90)中恢复 t 和 R

```
// SVD and fix sigular values
// START YOUR CODE HERE
JacobiSVD<Eigen::MatrixXd> svd(E, computationOptions: ComputeFullU | ComputeFullV);
Vector3d oldsingular = svd.singularValues();
Matrix3d singular_matrix = Matrix3d::Identity();
singular_matrix( row: 0, col: 0) = (oldsingular( index: 0) + oldsingular( index: 1)) / 2;
singular_matrix( row: 1, col: 1) = singular_matrix( row: 0, col: 0);
singular_matrix( row: 2, col: 2) = 0;
cout << "new singular " << endl << singular_matrix << endl;
// END YOUR CODE HERE

// set t1, t2, R1, R2
// START YOUR CODE HERE
Matrix3d t_wedge1;
Matrix3d t_wedge2;
Matrix3d R_z90 = AngleAxisd( angle: 3.141592653 / 2, axis: Vector3d( x: 0, y: 0, z: 1)).toRotationMatrix();
Matrix3d R_z_90 = AngleAxisd( angle: 3.141592653 / -2, axis: Vector3d( x: 0, y: 0, z: 1)).toRotationMatrix();
t_wedge1 = svd.matrixU() * R_z90 * singular_matrix * svd.matrixU().transpose();
t_wedge2 = svd.matrixU() * R_z_90 * singular_matrix * svd.matrixU().transpose();
Matrix3d R1;
Matrix3d R2;
R1 = svd.matrixU() * R_z90.transpose() * svd.matrixV().transpose();
R2 = svd.matrixU() * R_z_90.transpose() * svd.matrixV().transpose();
// END YOUR CODE HERE
```

结果满足 t^R=E

```
/home/xin/VSLAM-course/vSLAM-course/Ch5/SLAM第五讲作业资料/L5/code/cmake-build-debug/E2Rt
new singular
0.707107        0        0
       0 0.707107        0
       0        0        0
R1 =
  -0.365887  -0.0584576    0.928822
-0.00287462    0.998092   0.0616848
   0.930655  -0.0198996    0.365356
R2 =
 -0.998596  0.0516992 -0.0115267
-0.0513961   -0.99836 -0.0252005
 0.0128107  0.0245727  -0.999616
t1 =  -0.581301
-0.0231206
  0.401938
t2 =  0.581301
0.0231206
-0.401938
t^R =
 -0.0203619   -0.400711  -0.0332407
   0.393927   -0.035064    0.585711
-0.00678849   -0.581543  -0.0143826
```

4 用 G-N 实现 Bundle Adjustment 中的位姿估计(3 分，约 2 小时)

在书写程序过程中，回答下列问题：

1. 如何定义重投影误差？

Error = 观测 – 投影。

```
// START YOUR CODE HERE
Vector3d thisPoint = p3d[i];
Vector3d projPoint = T_esti * thisPoint;
Vector2d projUV( x: fx * projPoint[0] / projPoint[2] + cx,  y: fy * projPoint[1] / projPoint[2] + cy);
Vector2d e = p2d[i] - projUV;
cost += e.squaredNorm()
// END YOUR CODE HERE
```

Matrix<double, 2, 1> projUV(fx * projPoint[0] / projPoint[2] + cx, fy * projPoint[1] / projPoint[2] + cy)

2. 该误差关于自变量的雅可比矩阵是什么？

```
// START YOUR CODE HERE
double X = projPoint[0];
double Y = projPoint[1];
double Z = projPoint[2];
J( row: 0,  col: 0) = fx / Z;
J( row: 0,  col: 1) = 0;
J( row: 0,  col: 2) = -fx * X / (Z * Z);
J( row: 0,  col: 3) = -fx * X * Y / (Z * Z);
J( row: 0,  col: 4) = fx + fx * X * X / (Z * Z);
J( row: 0,  col: 5) = -fx * Y / Z;
J( row: 1,  col: 0) = 0;
J( row: 1,  col: 1) = fy / Z;
J( row: 1,  col: 2) = -fy * Y / (Z * Z);
J( row: 1,  col: 3) = -fy - fy * Y * Y / (Z * Z);
J( row: 1,  col: 4) = fy * X * Y / (Z * Z);
J( row: 1,  col: 5) = fy * X / Z;
J = -J;
// END YOUR CODE HERE
```

3. 解出更新量之后，如何更新至之前的估计上？

```
// START YOUR CODE HERE
dx = H.ldlt().solve(b);
// END YOUR CODE HERE
```

```
// START YOUR CODE HERE
T_esti = Sophus::SE3::exp(dx) * T_esti;
// END YOUR CODE HERE
```

最后的结果跟题干的 T 接近

```
/home/xin/VSLAM-course/vSLAM-course/Ch5/SLAM第五讲作业资料/L5/code/cmake-build-debug/GN-BA
points: 76
iteration 0 cost=645538.2282513
iteration 1 cost=12413.208557065
iteration 2 cost=12301.351931575
iteration 3 cost=12301.350653801
iteration 4 cost=12301.3506538
iteration 5 cost=12301.3506538
cost: 301.3506538, last cost: 301.3506538
estimated pose:
   0.997866186837  -0.0516724392948   0.0399128072707   -0.127226620999
  0.0505959188721    0.998339770315   0.0275273682287 -0.00750679765283
  -0.041268949107  -0.0254492048094    0.998823914318   0.0613860848809
                0                 0                 0                 1
```

5 * 用 ICP 实现轨迹对齐 (2 分，约 2 小时)

代码如下，分为读取轨迹，去重心，构建 W，从 SVD 恢复 Rt，投影第一条轨迹，构建成 se3 的形式。调用之前的 drawjractory 函数显示（没修改此函数所以截图省略）

```
int main()
{
    //load trajectory
    vector<Point3f>points1;
    vector<Point3f>points2;
    ifstream fin(trajectory_file);
    vector<Quaterniond, Eigen::aligned_allocator<Quaterniond>> oritations1;
    vector<Quaterniond, Eigen::aligned_allocator<Quaterniond>> oritations2;
    while (!fin.eof()) {
        double timestamp1, tx1,ty1,tz1,qx1,qy1,qz1,qw1,timestamp2,tx2,ty2,tz2,qx2,qy2,qz2,qw2;
        fin >> timestamp1>>tx1>>ty1>>tz1>>qx1>>qy1>>qz1>>qw1>>timestamp2>>tx2>>ty2>>tz2>>qx2>>qy2>>qz2>>qw2;
        Point3f thisPoint1(tx1,ty1,tz1);
        Point3f thisPoint2(tx2,ty2,tz2);
        points1.push_back(thisPoint1);
        points2.push_back(thisPoint2);
        Quaterniond thisOritation1(qx1,qy1,qz1,qw1);
        Quaterniond thisOritation2(qx2,qy2,qz2,qw2);
        oritations1.push_back(thisOritation1);
        oritations2.push_back(thisOritation2);
    }
    assert(points1.size()==points2.size());
    cout<<"loaded "<<points1.size()<<" point pairs"<<endl;

    //-----ICP SVD Approach
    //remove centre
    int pairNum = points1.size();
    Point3f point_sum1;
    Point3f point_sum2;
    for(int i=0; i< pairNum;i++)
    {
        point_sum1 +=points1[i];
        point_sum2 +=points2[i];
    }
    Point3f point_mean1 = point_sum1 /pairNum;
    Point3f point_mean2 = point_sum2 /pairNum;
    vector<Point3f>points_uniform1;
    vector<Point3f>points_uniform2;
    for(int i=0; i< pairNum;i++)
    {
        points_uniform1.push_back(points1[i]-point_mean1);
        points_uniform2.push_back(points2[i]-point_mean2);
    }
    //define W
    Matrix3d W = Matrix3d::Zero();

    {
        Vector3d V1(points_uniform1[i].x, points_uniform1[i].y, points_uniform1[i].z);
        Vector3d V2(points_uniform2[i].x, points_uniform2[i].y, points_uniform2[i].z);
```

```
    {
        Vector3d V1(points_uniform1[i].x, points_uniform1[i].y, points_uniform1[i].z);
        Vector3d V2(points_uniform2[i].x, points_uniform2[i].y, points_uniform2[i].z);
        W +=V1 * V2.transpose();
    }
    cout<<"W="<<endl<<W<<endl;
    //SVD
    JacobiSVD<Matrix3d> svd_w(W, computationOptions: ComputeFullU | ComputeFullV);
    cout<<"U="<<endl<<svd_w.matrixU()<<endl;
    cout<<"V="<<endl<<svd_w.matrixV()<<endl;
    Matrix3d R = Matrix3d::Zero();
    R = svd_w.matrixU() * svd_w.matrixV().transpose();
    if(R.determinant()<0)
        R = -R;
    Vector3d t = Vector3d(point_mean1.x, point_mean1.y, point_mean1.z) - R * Vector3d(point_mean2.x, point_mean2.y, point_m
    cout<<"R "<<endl<<R<<endl;
    cout<<"t "<<endl<<t<<endl;
    //project all points
    //and construct SE3
    vector<Vector3d> projectedPoint2;
    TrajectoryType trajectory1;
    TrajectoryType trajectory2;
    for(int i =0;i<pairNum;i++)
    {
        Vector3d thisPoint2(points2[i].x, points2[i].y, points2[i].z);
        Vector3d thisProj =R*thisPoint2 + t;
        projectedPoint2.push_back(thisProj);
        Sophus::SE3 p1(oritations1[i], translation_: Vector3d(points1[i].x,points1[i].y,points1[i].z));
        Sophus::SE3 p2(oritations2[i], thisProj);
        trajectory1.push_back(p1);
        trajectory2.push_back(p2);
    }
    DrawTrajectory(trajectory1,trajectory2);
    return 0;
}
```

投影到一起显示