作业目标

熟悉单目SLAM的初始化过程；

整体框架

1. 参数的初始化；
2. 确定第一帧图像，后续以此帧的相机坐标系当作世界坐标系；
3. 依次对后续的每一帧 进行以下操作，直到找到可以初始化的那一帧后结束；（特征点提取，匹配，筛选，三角化等）

   3.1 对每一帧进行特征点的提取，并进行暴力匹配；

   3.2 使用GMS 方法 对暴力匹配的结果 进行筛选；

   3.3 使用通过GMS筛选的特征点匹配对对进行位姿估计；

   3.4 根据估计的位姿，以及匹配点对关系，三角化空间点；

   3.5 对三角化的空间点进行重投影检验，剔除误差比较大的匹配对；

   3.6 对3.5 检验后的点 进行观察角度检验；剔除观测角度小于设定角度的点；

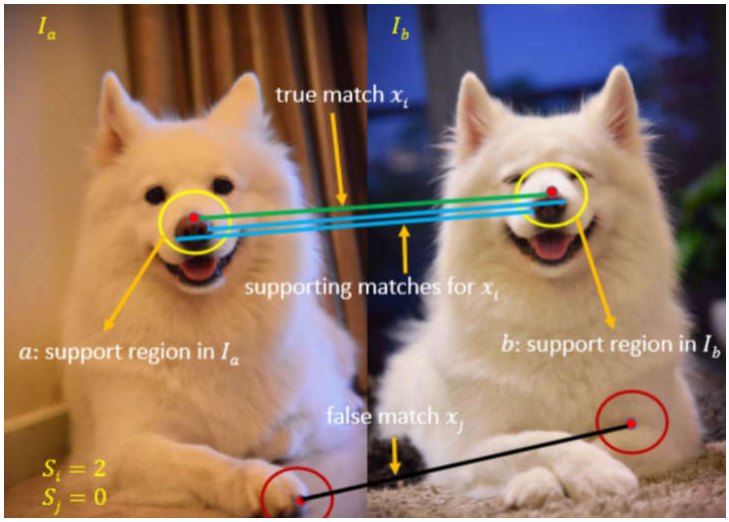
   3.7 检查3.6 之后剩余的点数 以及角度关系是否满足要求，满足则初始化完成，不满足则使用下一张进行初始化；
4. 绘制 内点外点匹配点对；

   绘制 内点 外点 的重投影；

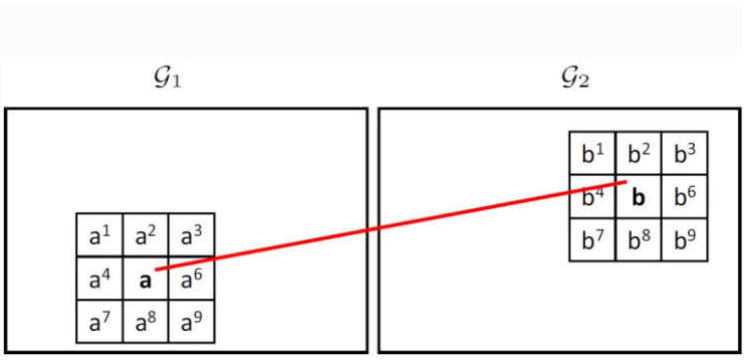
作业代码讲解

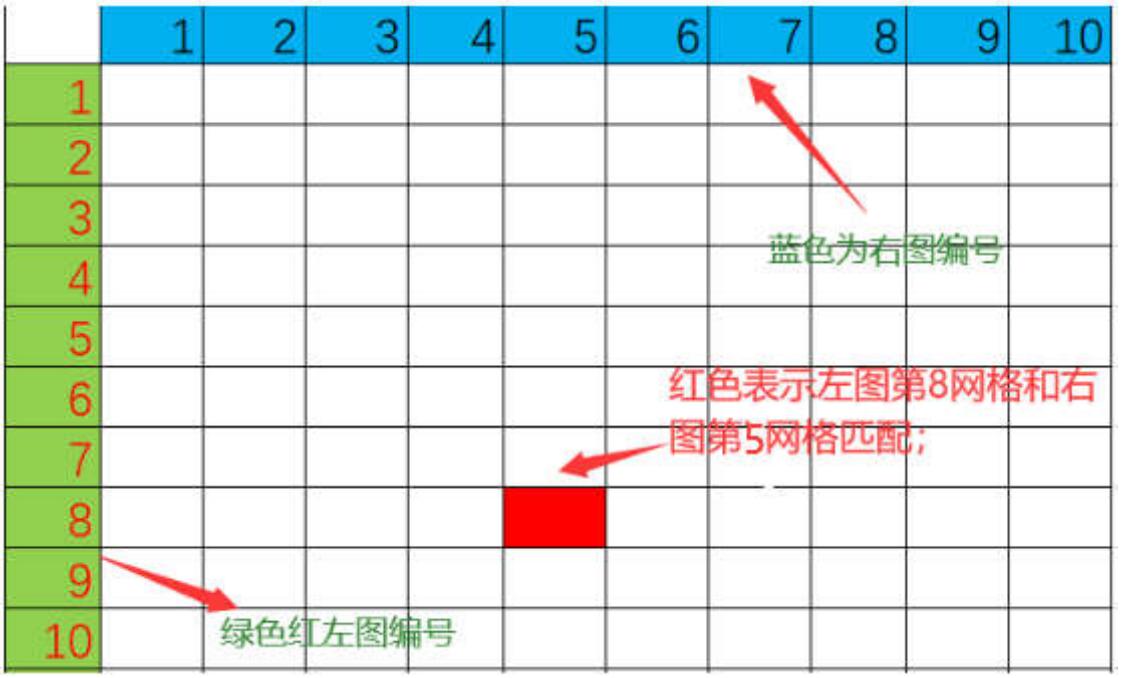
GMS(Grid-based Motion Statistics) 筛选外点

GMS 原理



特征点匹配不仅要看描述子是否匹配，而且要看当前特征点邻域内的其他特征点 是否 与 对应特征点 邻域内的特征点相互匹配；如果两个邻域相互匹配，则说明这一对特征点的匹配概率是比较大的。





实现步骤：

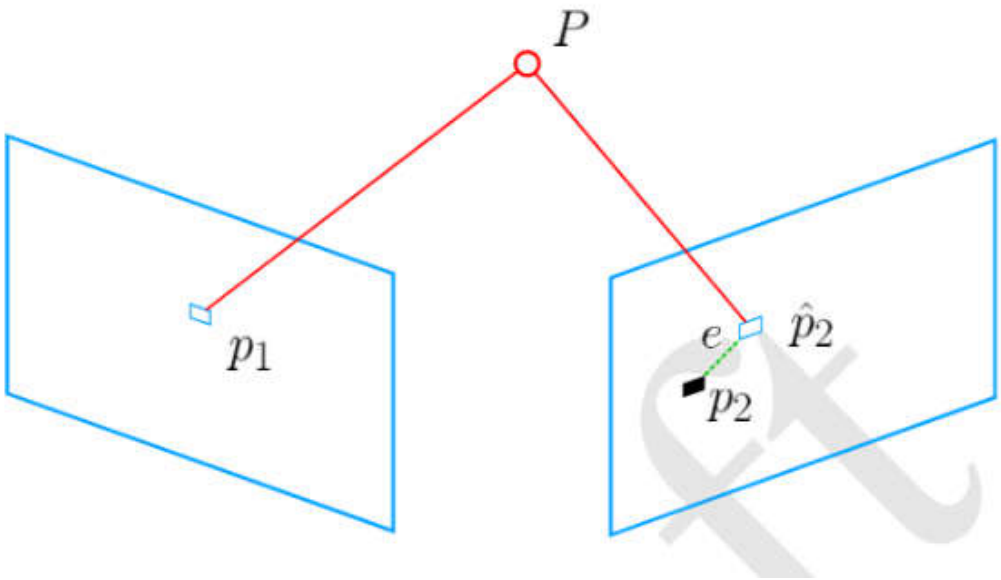
- step1：输入特征点的归一化、匹配点对的转换、 左图 网格数量mGridNumberLeft的计算，左图每个网格 各自的 9个邻居网格编号的计算；
- step2：设定尺度，根据尺度计算右图的网格数mGridNumberRight以及每个网格对应的9个邻居节点的编号；
- step3：设定初始化一个mGridNumberLeft* mGridNumberRight 的 匹配统计矩阵 mMotionStatistics ；
- step4：遍历每个特征点匹配对，把他们分配到各个网格中，计算他们所在的网格编号，统计左侧每个网格的特征点数 以及 匹配统计矩阵mMotionStatistics ；
- step5: 挑选出 匹配统计矩阵mMotionStatistics 每行中值最大的网格编号（右图中的网格编号），由此可以知道左图中哪个网格 与右图中哪个网格可能相匹配； 对这样的 网格匹配对，分别取各自的 9个邻域，计算对应网格的匹配度，并通过阈值筛选后，认为这两个网格是匹配的。
- step6: 对于每个匹配点对，只有点对匹配，并且每个点对应的网格相匹配，才能算作匹配成功，否则，标记为这对特征点匹配失败；

代码实现；

```
    // ---------- homework1：用GMS方法筛选暴力匹配结果中正确的匹配对 --------------//
    // ----------------  代码开始 -----------------//
    std::vector<bool> vbInliers;
    int num_inliers;


   gms_matcher  my_gms = gms_matcher(keypoints_1, img_1.size(), keypoints_2,
img_2.size(), Matches_bf) ;
    num_inliers = my_gms.GetInlierMask(vbInliers, false, false);

    matches_gms.clear();
    for(int i = 0; i < Matches_bf.size(); i++)
    {
        if(vbInliers[i])
        {
            //  DMatch
now_match(Matches_bf[i].queryIdx,Matches_bf[i].trainIdx,Matches_bf[i].imgIdx,Mat
ches_bf[i].distance);
            // matches_gms.push_back(now_match);
            matches_gms.push_back(Matches_bf[i]);
        }

    }
    // ----------------  代码结束-----------------//
```

根据重投影误差筛选匹配点：

(目标：滤除重投影误差比较大的点)



版本1；

```
    // 根据重投影误差筛选内外点
    convertPointsFromHomogeneous(pts_4d.t(), pts_3d);   //齐次坐标转化到非齐次坐标；
```

```
    pts_3d  = pts_3d.reshape(1);                                //这里的类型是CV_FC3，要把三通道变为一通道，否则后面无法进行；


    int radius_threshold2 = (MAX_REPROJECT_ERROR)*(MAX_REPROJECT_ERROR);  //设定误差阈值；
    vector<int> inlier3dPoints;                                           //初始化记录内点的vector 都为0;
    inlier3dPoints.resize(pts_3d.rows, 0);


    for(int i = 0; i < pts_3d.rows; i++){

        if(inlierPts.at<int>(i) == 0)                                     //只对内点进行投影;
        {
            continue;
        }
        cv::Mat p3Dw = pts_3d.row(i).t();            //转化为列向量;
        cv::Mat  p3Dc2  = (R * p3Dw  + t );          //将第一帧相机坐标系下的3D点坐标转换到第二帧相机坐标系下。
        const double invz = 1/p3Dc2.at<double>(2);  //注意这里的类型，如果使用float，会出现错误的结果;
        if(invz <=0)                                 //剔除深度为负或者 在无穷远出的点；//注意等于号;
        {
            inlierPts.at<int>(i) = 0;
            continue;
        }
        //第二帧相机坐标系下的3D点坐标归一化
        const double x = p3Dc2.at<double>(0) * invz;
        const double y = p3Dc2.at<double>(1) * invz;

        //使用内参矩阵，将归一化的3D坐标转化为 图像坐标系下的坐标;
        const double u = K.at<double>(0,0) * x  + K.at<double>(0,2);
        const double v = K.at<double>(1,1) * y  + K.at<double>(1,2);

        //计算投影点到 匹配的特征点之间的距离,小于距离阈值的标记为1;
        const double distance2 =  (u - pts_2[i].x )  * (u - pts_2[i].x ) +   (v - pts_2[i].y)  *   (v - pts_2[i].y);

        if(  distance2 < radius_threshold2  )
        {
            inlier3dPoints[i] = 1;
        }
    }
     int sum = std::accumulate(inlier3dPoints.begin(), inlier3dPoints.end(), 0);
    printf("经过重投影误差筛选后，有效3D点数为：%d / %d \n", sum, inlier3dPoints.size());  //每次出现结果可能不一样，为什么？
```

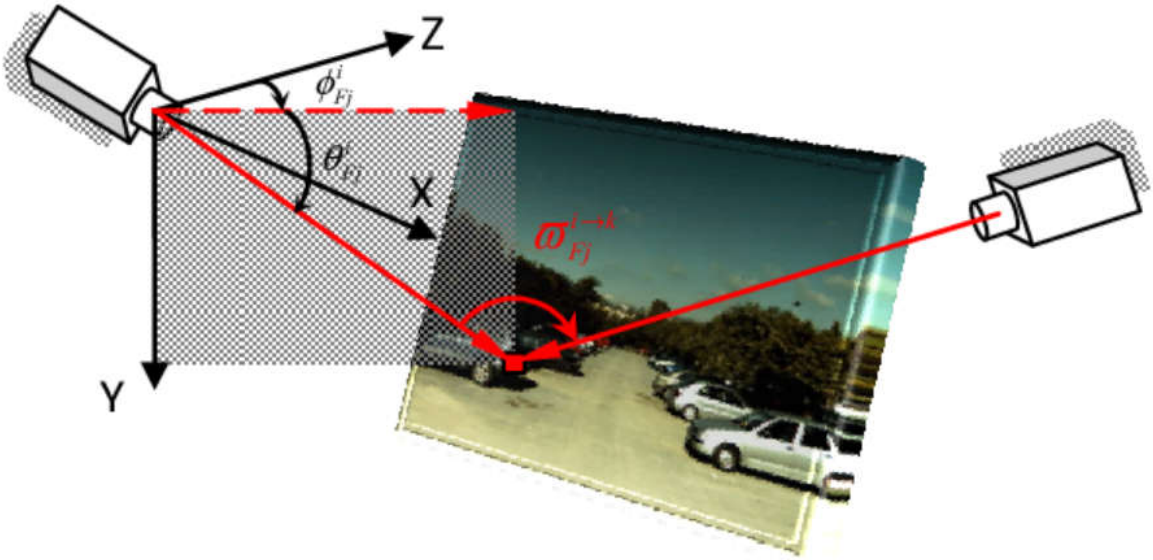
位姿估计每次选取的点是随机的，所以，估计的位姿可能不完全一样；

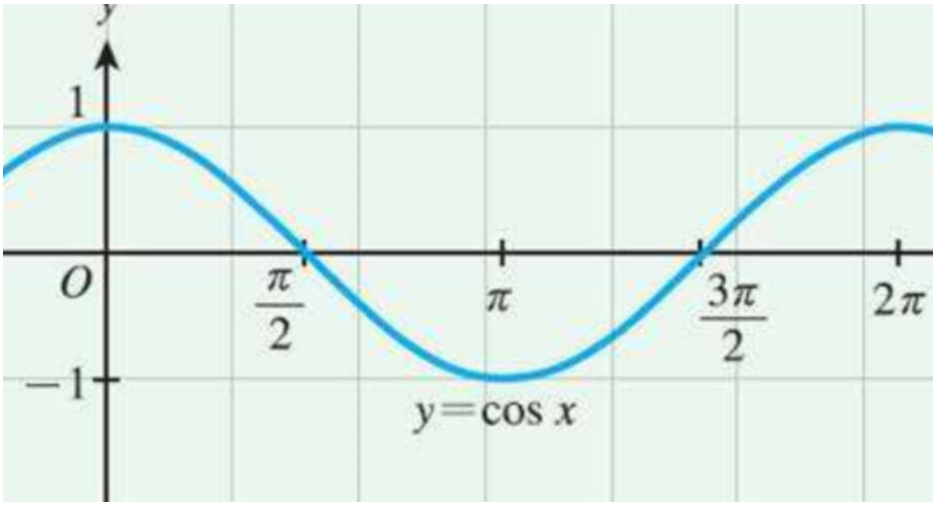
版本2；

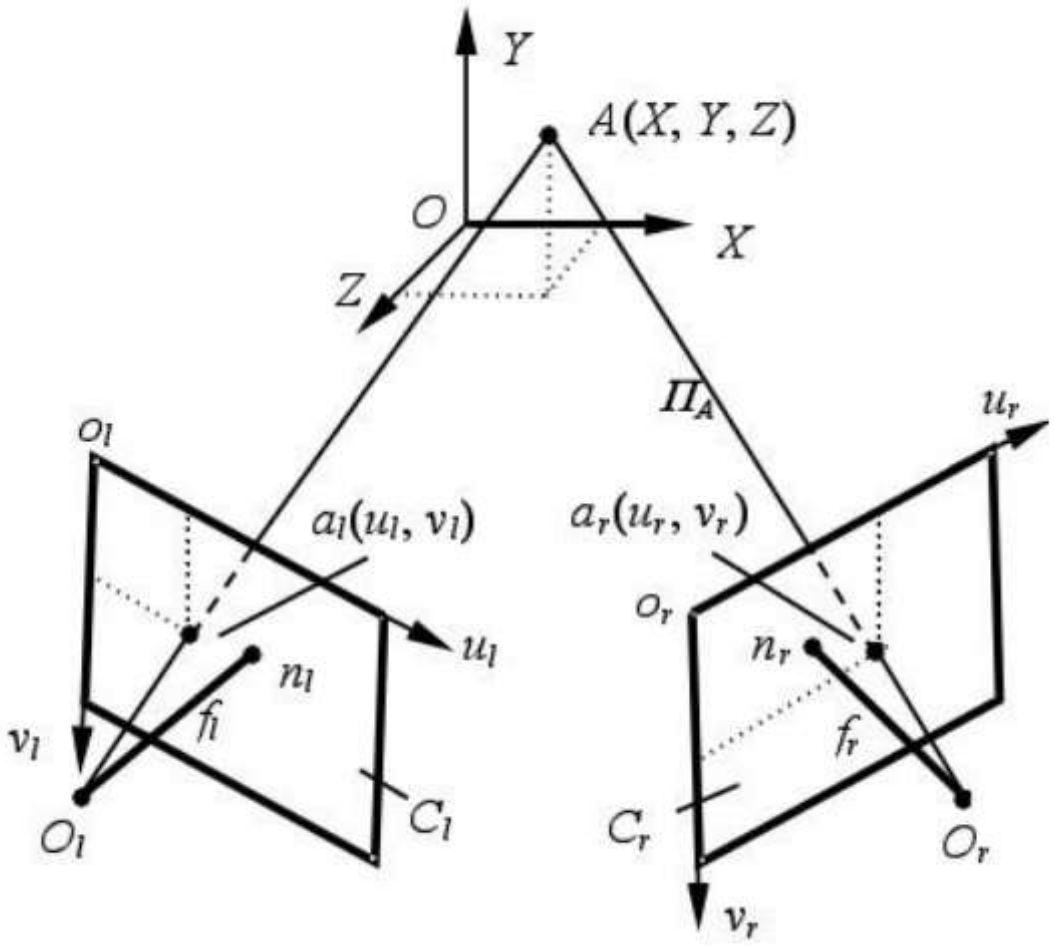
```
 // 转换成非齐次坐标
    convertPointsFromHomogeneous(pts_4d.t(), pts_3d);

    // 矩阵转化为向量
    Mat vecR_1, vecR_2;
    Rodrigues(I, vecR_1);
    Rodrigues(R, vecR_2);
    Mat vect_1 = T1.rowRange(0,3).col(3).t();
    Mat vect_2 = t.t();

    // 重投影
    vector<Point2d> projectedLeft, projectedRight;
    projectPoints(pts_3d, vecR_1, vect_1, K, Mat(), projectedLeft);
    projectPoints(pts_3d, vecR_2, vect_2, K, Mat(), projectedRight);

    // 根据重投影误差判断内外点
    vector<int> inlier3dPoints;
    inlier3dPoints.resize(pts_3d.rows, 0);
    for (int i = 0; i < pts_3d.rows; ++i) {
        if(inlierPts.at<int>(i) == 0){     // outliners
            continue;
        }
        if (pts_3d.at<double>(i, 2) < 0){
            inlierPts.at<int>(i) = 0;
//            printf("不满足3D点深度为正！ \n");
            continue;
        }
        Point2d p1 = projectedLeft[i];
        Point2d p2 = projectedRight[i];
        double error1 = sqrt(pow(p1.x - pts_1[i].x, 2) + pow(p1.y - pts_1[i].y,
2));
        double error2 = sqrt(pow(p2.x - pts_2[i].x, 2) + pow(p2.y - pts_2[i].y,
2));

        if (error1 > MAX_REPROJECT_ERROR || (error2 > MAX_REPROJECT_ERROR)){
            inlierPts.at<int>(i) = 0;
//            printf("不满足3D点重投影误差限制！ \n");
            continue;
        }
        inlier3dPoints[i] = 1;
    }

    int sum = std::accumulate(inlier3dPoints.begin(), inlier3dPoints.end(), 0);
    printf("经过重投影误差筛选后，有效3D点数为：%d / %d \n", sum,
inlier3dPoints.size());
```

根据观测角度来筛选匹配点

（目标：滤除 观测角度比较小的关键点，稳定性比较差，不适宜初始化时使用；）







$$(1)\vec{a}\cdot\vec{b}=\left|\vec{a}\right|\cdot\left|\vec{b}\right|\cos\theta$$

```
    // 根据三角化夹角来筛选合格的三维点
    vector<double> cosinPts;
    //计算最小的观测角度的cos值，角度越趋向于零，cos越大；最小角度观测意味着最大cos值；夹角过小，说明视差过小，这种匹配点并不可信；
    double max_cos = cos(MIN_TRIANGLE_ANGLE * 3.1415926 / 180);

    Mat O1 = Mat::zeros(3,1,CV_64F);       //相机1 的光心；

    for(int i  =0 ; i < inlier3dPoints.size(); i++)
    {
        if(0 == inlier3dPoints[i])
        {
            continue;
        }
        //相机2 的 光心；
        Mat O2 = -R.t()*t;
        O2.convertTo(O2,CV_64F);
        // 计算空间三维点到光心的连线；
        Mat PO1 = pts_3d.at<double>(i) - O1;
        Mat PO2 = pts_3d.at<double>(i) - O2;
```

```
        double observe_cos = PO1.dot(PO2) /(norm(PO1) * norm(PO2));
        if(observe_cos > max_cos )
        {
            continue;
        }
        cosinPts.push_back(observe_cos);


    }
    printf("经过角度筛选后，有效3D点数为： %d / %d \n",cosinPts.size() ,
inlier3dPoints.size());
```