



激光里程计I



主讲人 雍川



- 第一部分：SVD_ICP思路分享
- 第二部分：高斯牛顿法

框架中的SVD_ICP函数只需要完成icp_svd_registration.cpp文件中的TODO部分即可。

在调参方面，需要注意的是，SVD_ICP的精度比较依赖于邻近点对的准确度，比如将邻近点对的距离阈值设置得尽可能小，比如0.5。

求出的旋转矩阵必须满足是正交阵并且行列式为1，因此，需要对求出的旋转矩阵进行正交化。

最后，可以适当的降低点云数据包播放速度，以防止点云数据的丢失。
补足代码如下图：

scanMatch()函数

```
// get correspondence:
std::vector<Eigen::Vector3f> xs;
std::vector<Eigen::Vector3f> ys;

// do not have enough correspondence -- break:
if (GetCorrespondence(curr_input_source, xs, ys) < 3)
    break;

// update current transform:
Eigen::Matrix4f delta_transformation;
GetTransform(xs, ys, delta_transformation);

// whether the transformation update is significant:
if (!IsSignificant(delta_transformation, trans_eps_))
    break;

transformation_ = delta_transformation * transformation_;

++curr_iter;
```

```
// set output:
result_pose = transformation_ * predict_pose;
//归一化
Eigen::Quaternionf qr(result_pose.block<3,3>(0,0));
qr.normalize();
Eigen::Vector3f t = result_pose.block<3,1>(0,3);
result_pose.setIdentity();
result_pose.block<3,3>(0,0) = qr.toRotationMatrix();
result_pose.block<3,1>(0,3) = t;
pcl::transformPointCloud(*input_source_, *result_cloud_
```

GetCorrespondence()函数 GetTransform()函数

```
for (size_t i = 0; i < input_source->points.size(); ++i) {
    std::vector<int> corr_ind;
    std::vector<float> corr_sq_dis;
    input_target_kdtree->nearestKSearch(
        input_source->at(i),
        1,
        corr_ind, corr_sq_dis
    );

    if (corr_sq_dis.at(0) > MAX_CORR_DIST_SQR)
        continue;

    // add correspondence:
    Eigen::Vector3f x(
        input_target->at(corr_ind.at(0)).x,
        input_target->at(corr_ind.at(0)).y,
        input_target->at(corr_ind.at(0)).z
    );
    Eigen::Vector3f y(
        input_source->at(i).x,
        input_source->at(i).y,
        input_source->at(i).z
    );

    xs.push_back(x);
    ys.push_back(y);

    ++num_corr;
}
```

```
// find centroids of mu_x and mu_y:
Eigen::Vector3f mu_x = Eigen::Vector3f::Zero();
Eigen::Vector3f mu_y = Eigen::Vector3f::Zero();
for (size_t i = 0; i < N; ++i) {
    mu_x += xs.at(i);
    mu_y += ys.at(i);
}
mu_x /= N;
mu_y /= N;
```

```
// build H:
Eigen::Matrix3f H = Eigen::Matrix3f::Zero();
for (size_t i = 0; i < N; ++i) {
    H += (ys.at(i) - mu_y) * (xs.at(i) - mu_x).transpose();
}
```

```
// solve R:
Eigen::JacobiSVD<Eigen::MatrixXf> svd(H, Eigen::ComputeThinU |
Eigen::Matrix3f R = svd.matrixV() * svd.matrixU().transpose();
```

```
transformation_.setIdentity();
transformation_.block<3, 3>(0, 0) = R;
transformation_.block<3, 1>(0, 3) = t;
```

高斯牛顿法

- 高斯牛顿原理见pdf。部分代码：

```
bool ICPGNRegistration::ScanMatch(const CloudData::CLOUD_PTR &input_source,
                                   const Eigen::Matrix4f& predict_pose,
                                   CloudData::CLOUD_PTR& result_cloud_ptr,
                                   Eigen::Matrix4f& result_pose) {
    Eigen::Matrix3f rotation_matrix = Eigen::Matrix3f::Identity();
    Eigen::Vector3f translation = Eigen::Vector3f::Identity();

    transformation_ = predict_pose;
    rotation_matrix = transformation_.block<3, 3>(0, 0) ;
    translation = transformation_.block<3, 1>(0, 3);
    //计算点云位姿运动量
    getTrans(input_source, rotation_matrix, translation);

    pcl::transformPointCloud(*input_source, *result_cloud_ptr, transformation_);
    result_pose = transformation_;

    return true;
}
```

高斯牛顿迭代过程

```
//高斯牛顿迭代过程
while(iterator_num < max_iterator_)
{
    pcl::transformPointCloud(*input_source, *transformed_cloud, transformation_);
    Eigen::Matrix<float,6,6> Hessian;
    Eigen::Matrix<float,6,1> Gx;
    Hessian.setZero();
    Gx.setZero();

    //遍历所有点对,更新Hessian矩阵,误差等值
    for(size_t i =0; i < transformed_cloud->size(); ++i)
    {
        pcl::PointXYZ source_point = input_source->at(i);
        //auto source_point = input_source->at(i);
        if(! pcl::isFinite(source_point))
        {
            continue;
        }
        pcl::PointXYZ transformed_point = transformed_cloud->at(i);
        //auto transformed_point = transformed_cloud->at(i);
        std::vector<float> distances;
        std::vector<int> indexes;
        //查找最近邻点
        kdtree_ptr->nearestKSearch(transformed_point,knn,indexs,distances);
        if(distances[0] > max_correspond_distance_)
        {
            continue;
        }
        Eigen::Vector3f closet_point = Eigen::Vector3f(target_cloud->at(indexs[0]).x, target_cloud->at(indexs[0]).y ,
                                                         target_cloud->at(indexs[0]).z );
        // 计算原始点与最近邻点的误差
        Eigen::Vector3f err_dis = Eigen::Vector3f(transformed_point.x,transformed_point.y,transformed_point.z) - closet_point;
```


高斯牛顿迭代过程

```
//计算雅可比,海森矩阵.右乘模型
Eigen::Matrix<float,3,6> Jacobian(Eigen::Matrix<float,3,6>::Zero());
Jacobian.leftCols<3>() = rotation_matrix;
Jacobian.rightCols<3>() =
    -rotation_matrix* Sophus::SO3f::hat(Eigen::Vector3f(source_point.x,source_point.y,source_point.z)) ;
Hessian += Jacobian.transpose()*Jacobian;
Gx += -Jacobian.transpose()*err_dis;
}

iterator_num++;
if(Hessian.determinant() == 0)
{
    continue;
}
//求解位姿运动量的李代数形式
Eigen::Matrix<float,6,1> delta_x = Hessian.inverse()*Gx;
//利用李代数更新位姿
Sophus::SE3f SE3_Rt(rotation_matrix,translation);
Sophus::SE3f update = SE3_Rt * Sophus::SE3f::exp(delta_x);

transformation_ = update.matrix();
rotation_matrix = transformation_.block<3,3>(0,0);
translation = transformation_.block<3,1>(0,3);
```




感谢各位聆听 !
Thanks for Listening

