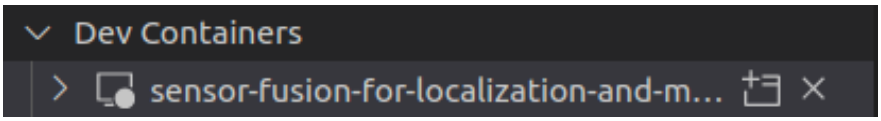
多传感器融合定位与建图第二章思路提示

VScode连接Docker

这里主要是使用`Remote – Containers`这里插件，大家可以在插件商店下载，下载之后左侧工具栏会出现下面这个图标

之后在你成功运行容器之后，容器列表中会出现课程运行的容器



之后点击`Attach to Container`就可以成功连接容器，之后的用法就与普通的VScode开发无异

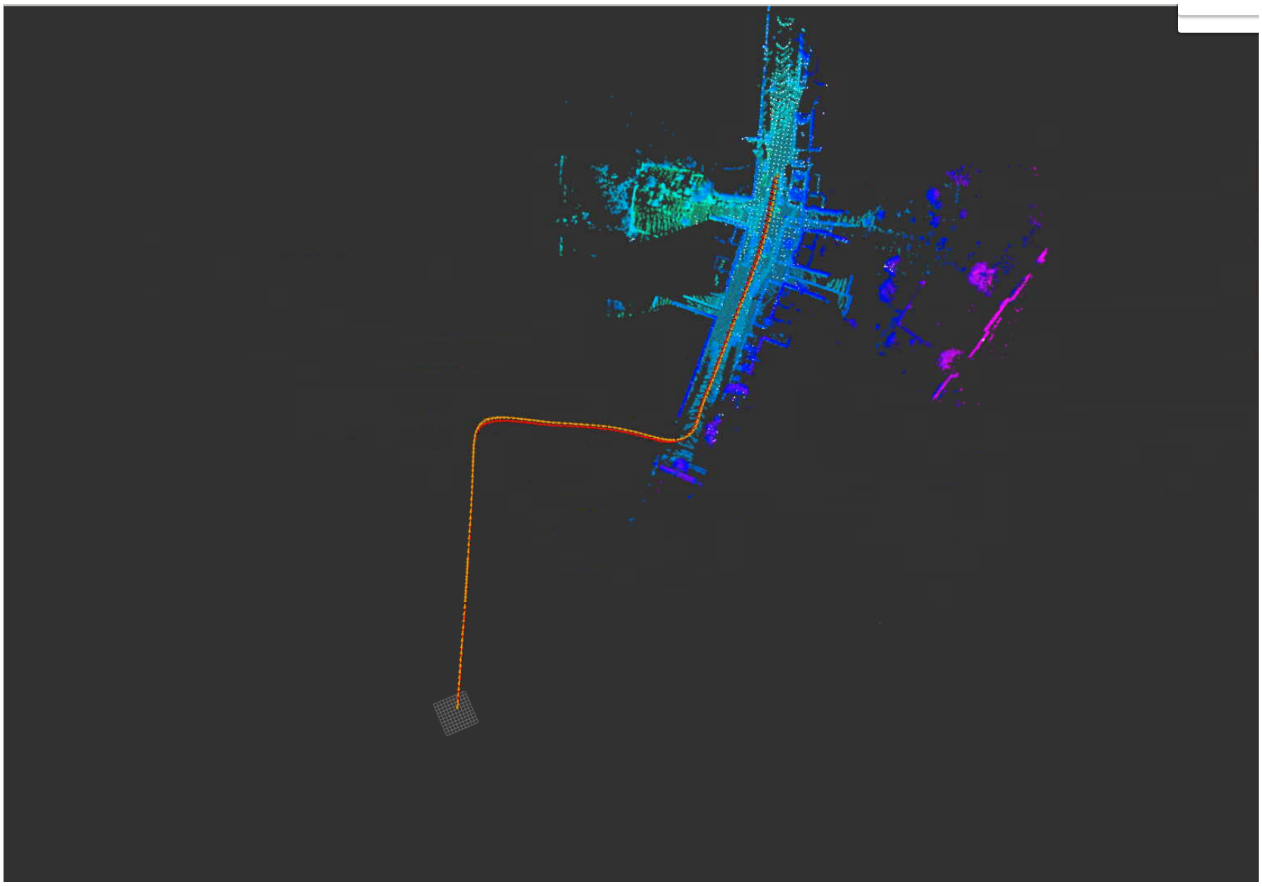
跑通框架代码

作业第一项为跑通作业中提供的代码，并展示结果，这一题的主要目的是为了让大家熟悉作业的代码框架，为了完成作业主要需要进行以下几步

- 修改配置文件，将默认的扫描匹配方法修改为NDT，配置文件路径为`/workspace/assignments/02-lidar-odometry-basic/src/lidar_localization/config/front_end/config.yaml`，修改后的结果如下

```
# 匹配
# TODO: implement your custom registration method and add it here
registration_method: NDT   # 选择点云匹配方法，目前支持：ICP, ICP_SVD, NDT, SICP，LOAM
```

- 根据课程文档的指引运行工程
- 正确运行的结果如下



使用evo进行评价

这里的做法比较简单，大家只需要执行课程中给定的两条evo指令并展示结果即可

自己实现匹配方法

这里主要以课程代码中的 `ICP-SVD` 为例，这里主要讲解关键的逻辑，具体的 `SVD` 分解过程和寻找对应关系的过程相信大家都没有问题

```
// 重载父类的虚函数，使用面向对象的三大特性之一的多态
bool ICPSVDRegistration::ScanMatch(
    const CloudData::CLOUD_PTR& input_source,
    const Eigen::Matrix4f& predict_pose,
    CloudData::CLOUD_PTR& result_cloud_ptr,
    Eigen::Matrix4f& result_pose
) {
    input_source_ = input_source;

    // 预处理点云:主要目的是根据预测的位姿将点云进行变换，后续的更新也是在预测位姿的基础上进行
    CloudData::CLOUD_PTR transformed_input_source(new CloudData::CLOUD());
```

```
    pcl::transformPointCloud(*input_source_, *transformed_input_source, predict_pose);

    // 初始化估计:我们后续计算的其实是变换后的点云和地图之间的位姿变换
    transformation_.setIdentity();

    // 进行求解
    int curr_iter = 0;
    while (curr_iter < max_iter_) {
        // 根据上一次迭代的结果将点云进行变换，迭代求解增量
        CloudData::CLOUD_PTR current_input_soucre(new CloudData::CLOUD());

 pcl::transformPointCloud(*transformed_input_source,*current_input_soucre,transformation_);

        // 寻找相关的点对
        std::vector<Eigen::Vector3f> xs;
        std::vector<Eigen::Vector3f> ys;

        // 如果相关点对数目小于一定阈值，则认为无法匹配：这里理论上只需要3点，所以作为系统错误的判断阈值取3就可以了，我看到有些同学取的几百这个没有很大的意义
        if(GetCorrespondence(current_input_soucre,xs,ys) < 3){
            break;
        }

        // 利用SVD分解获取R和T
        Eigen::Matrix4f delta;
        GetTransform(xs,ys,delta);

        // 判断delta是否符合更新要求：也就是判断两次迭代之间的位姿变换是否显著
        if(!IsSignificant(delta,trans_eps_)){
            break;
        }

        // 更新估计值：这里需要注意左乘和右乘的区别，直接使用*=是不行的
        transformation_ = delta * transformation_;

        ++curr_iter;
    }

    // 设置输出:这里需要注意我们上面求得的是变换后的点云和地图之间的位姿变换，这里还需要将这个位姿变换乘到预测位姿上才可以得到真正的全局坐标系下的位姿
    result_pose = transformation_ * predict_pose;
```

```
50      pcl::transformPointCloud(*input_source_, *result_cloud_ptr,
    result_pose);
51
52      return true;
53  }
```

这里我需要着重强调以下旋转归一化的方式，大家应该注意到上面的代码并没有作归一化，我这里提供三种归一化的方式

- 四元数法

```
1       Eigen::Quaternionf q(result_pose.block<3,3>(0,0));
2       q.normalize();
3       result_pose.block<3,3>(0,0) = q.toRotationMatrix();
```

- SVD分解法

```
1       Eigen::JacobiSVD<Eigen::MatrixXf> svd(result_pose.block<3,3>
    (0,0),Eigen::ComputeFullU | Eigen::ComputeFullV);
2       result_pose.block<3,3>(0,0) = svd.matrixU() *
    svd.matrixV().transpose();
```

- 流形投影

```
1       Eigen::Matrix3f  H = result_pose.block<3,3>(0,0) *
    result_pose.block<3,3>(0,0).transpose();
2       Eigen::Matrix3f L = H.llt().matrixL();
3       result_pose.block<3,3>(0,0) =
    L.inverse()*result_pose.block<3,3>(0,0);
```

经测试，在只对结果进行归一化的情况下四元数法的效果会更好一些，当然大家可以测试一下在每次迭代都进行归一化的结果。

> 不进行归一化是绝对不行的，长时间累计下尺度会出现大的漂移

最后这里提供个人对于参考文献的翻译，阅读有困难的同学可以借鉴一下，水平有限，有错误欢迎指出

- Least-Squares Fitting of Two 3-D Point Sets
- Least-Squares Rigid Motion Using SVD

- The normal-distributionstransform