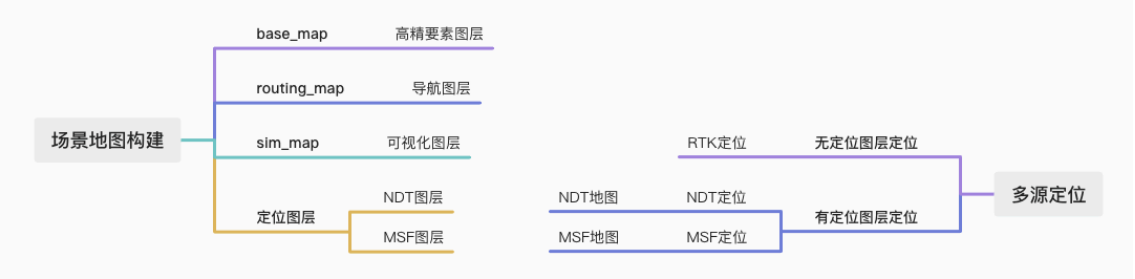
组11-Week7-操作指导

1. 采集真实场景的数据，构建base_map\routing_map\sim_map，用于无定位图层的定位；
2. 在1的基础上，新增构建NDT定位图层和MSF定位图层，用于有定位图层的定位；
3. 基于所建地图，完成RTK定位，NDT定位和MSF定位的算法验证。



1. Apollo中的地图总结

2. 新场景地图构建

由于正规车道线地图构建的原理较为复杂，因此我们采用虚拟车道线的方式进行新场景地图的构建。

2.2. 现场录制数据集

根据*传感器驱动配置章节*中所述，开启`Transform`，`Lidar`，`GPS`，`Localization`模块，并使用`cyber_monitor`监控各个信息通道，确保所有模块开启正常。

在遥控器控制模式下，开启`cyber_recorder`记录数据，并驱动车辆绕较大的0字轨迹或较长的直线。开启记录的命令如下：

```
cyber_recorder record -a -i 600 -o localization.record
```

按`ctrl c`结束记录后，会在apollo根目录下生成`localization.record.0000*`文件。将该文件保存在`/apollo/data/bag/localization/`文件夹下（确保该目录下不存在其他文件）。

2.3. 构建虚拟车道地图

虚拟车道线的核心思想非常简单，即记录车辆行驶的轨迹，以此为中心向左右各扩展若干距离。

0. 使用如下指令从录制的数据包中提取位置路径文件，会在apollo根目录下生成`path.txt`文件：

```
# 参数依次为：提取轨迹程序 输出轨迹文件 输入数据文件路径
./bazel-bin/modules/tools/map_gen/extract_path \
./path.txt \
data/bag/localization/*
```

1. 修正`modules/tools/map_gen/map_gen_single_lane.py`脚本中的`LANE_WIDTH`参数可以调整车道线宽度。本次实践中，推荐设置宽度为5。

```
#第27行
LANE_WIDTH = 5.0
```

2. 使用如下指令生成地图文件，其中1表示冗余区域大小为1，会在apollo根目录下生成`base_map.txt`文件：

```
# 参数依次为：沿轨迹生成单根车道线程序 输入轨迹文件 输出basemap 冗余区域大小
./bazel-bin/modules/tools/map_gen/map_gen_single_lane \
./path.txt \
./base_map.txt \
1
```

3. 为`base_map.txt`文件增加header用于可视化，举例如下：

```
# 添加在文件开头
header {
  version: "0326"
  date: "20220326"
  projection {
    proj: "+proj=tmerc +lat_0={39.52} +lon_0={116.28} +k={-48.9} +ellps=WGS84
+no_defs"
  }
}
```

4. 建立地图文件夹`/apollo/modules/map/data/map_test`（可以修改为自己地图名称），确保该目录下不存在其他文件，将`base_map.txt`移动到该目录下，并生成.bin文件

```
mkdir modules/map/data/map_test

rm -rf path.txt

mv base_map.txt modules/map/data/map_test/base_map.txt

# base_map.bin
./bazel-bin/modules/tools/create_map/convert_map_txt2bin \
-i /apollo/modules/map/data/map_test/base_map.txt \
-o /apollo/modules/map/data/map_test/base_map.bin
```

5. 建立`routing_map`

```
bash scripts/generate_routing_topo_graph.sh \
--map_dir /apollo/modules/map/data/map_test
```

> 注意：第一次运行可能会提示如下报错，属于正常现象，继续即可：

```
E0406 15:11:07.321321 10341 hdmap_util.cc:40] [map]No existing file found in
/apollo/modules/map/data/map_test/routing_map.bin|routing_map.txt. Fallback
to first candidate as default result
```

6. 建立`sim_map`

```
./bazel-bin/modules/map/tools/sim_map_generator \
--map_dir=/apollo/modules/map/data/map_test \
--output_dir=/apollo/modules/map/data/map_test
```

7. 可视化车道线

- 修复软件源(docker内执行):

```
sudo vim /etc/apt/sources.list
```

在文件中将`https`修改为`http`修改:

```
deb http://mirrors.tuna.tsinghua.edu.cn/ubuntu/ bionic main restricted
universe multiverse
# deb-src https://mirrors.tuna.tsinghua.edu.cn/ubuntu/ bionic main restricted
universe multiverse
deb http://mirrors.tuna.tsinghua.edu.cn/ubuntu/ bionic-updates main
restricted universe multiverse
# deb-src https://mirrors.tuna.tsinghua.edu.cn/ubuntu/ bionic-updates main
restricted universe multiverse
deb http://mirrors.tuna.tsinghua.edu.cn/ubuntu/ bionic-backports main
restricted universe multiverse
# deb-src https://mirrors.tuna.tsinghua.edu.cn/ubuntu/ bionic-backports main
restricted universe multiverse
deb http://mirrors.tuna.tsinghua.edu.cn/ubuntu/ bionic-security main
restricted universe multiverse
# deb-src https://mirrors.tuna.tsinghua.edu.cn/ubuntu/ bionic-security main
restricted universe multiverse
```

- 更新并安装缺少的依赖库

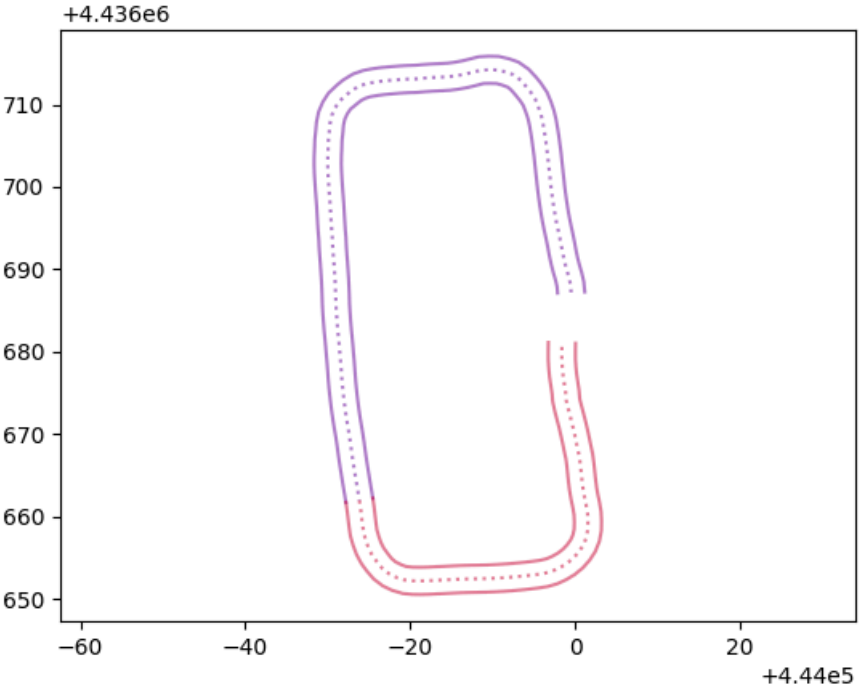
```
sudo apt update

sudo apt-get install tcl-dev tk-dev python3-tk
```

> 注意：上述修改涉及Apollo系统，因此使用`dev_start.sh -l`时会重建一个docker容器，此时对系统的修改会全部失效，需要重新换源操作；但是`docker start + 容器id/tag`的方式并不会重建容器，而是会继续使用之前容器，因此可以不用重新换源。

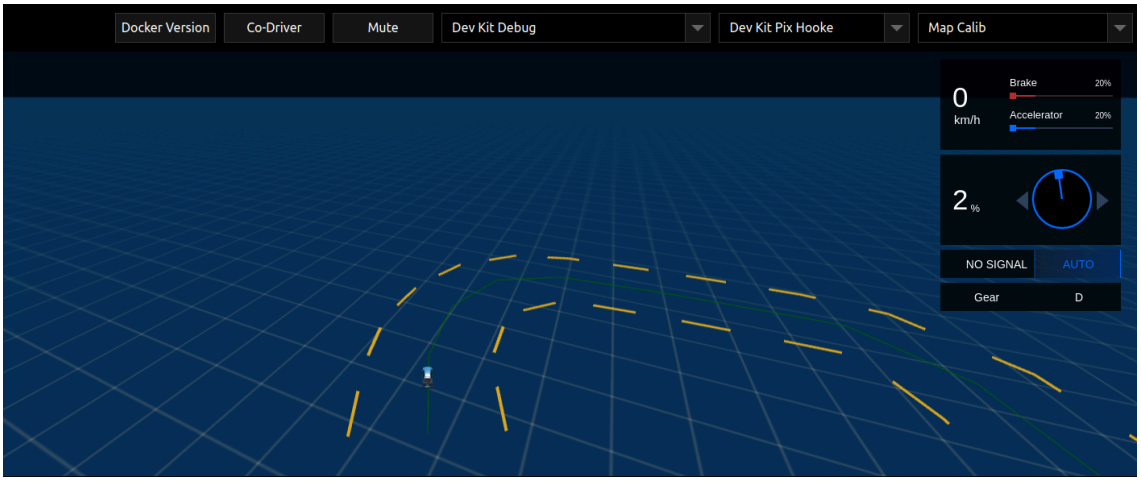
- Python可视化

```
./bazel-bin/modules/tools/mapshow/mapshow \
-m /apollo/modules/map/data/map_test/base_map.txt
```

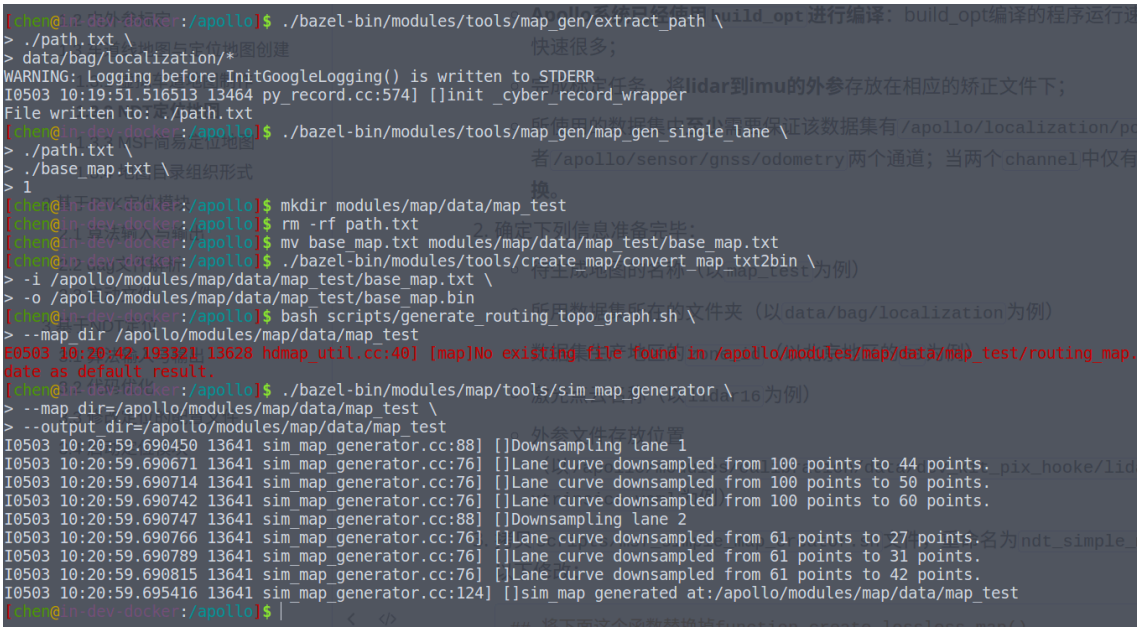


9. 使用如下命令重启Dreamview，在Dreamview的地图菜单中选择map_test即可显示地图：

```
bash scripts/bootstrap.sh restart
```



10. 过程截图显示（仅供参考）：



3. 基于RTK算法的定位

3.4. 修改启动文件

修改或创建启动文件：`/apollo/modules/calibration/data/车辆底盘名称/localization_dag/dag_streaming_rtk_localization.dag`

```
# 路径：/apollo/modules/calibration/data/底盘名称/localization_dag/dag_streaming_rtk_localization.dag
module_config {
    module_library : "/apollo/bazel-bin/modules/localization/rtk/librtk_localization_component.so"

    components {
      class_name : "RTKLocalizationComponent"
      config {
        name : "rtk_localization"
        config_file_path : "/apollo/modules/localization/conf/rtk_localization.pb.txt"
        readers: [
          {
            channel: "/apollo/sensor/gnss/odometry"
            qos_profile: {
              depth : 10
            }
            pending_queue_size: 50
          }
        ]
      }
    }
}
```

- `module_library`：启动文件对应的动态链接库
- `components.class_name`：实例所属的类名（class name）
- `components.config.name`：配置的名称定义

- `components.config.config_file_path`：对应的参数配置文件，以`gflags`形式进行处理
- `components.config.readers.channel`：组件读取的channel名称。`RTKLocalizationComponent`类会继承`cyber::Component<localization::Gps>`（即通道所读取的channel对应的类别）。每次通道中有数据传入时，会调用一次`Proc`函数。
- `components.config.readers.qos_profile`：处理后的消息被保留的数量
- `components.config.readers.pending_queue_size`：未及时处理消息的缓存队列长度

3.5. 修改配置文件

将`/apollo/modules/localization/conf/localization.conf`复制粘贴至新建文件夹`/apollo/modules/calibration/data/地盘名称/localization_conf/`下，并确保其内容如下：

```
# 路径：/apollo/modules/calibration/data/地盘名称/localization_conf/localization.conf

# 5行
--map_dir=/apollo/modules/map/data/map_test # 指定地图位置

# 115行
--local_utm_zone_id=50 # 北京地区zone id为50
自行搜索海南zoneid
```

3.6. 启动RTK定位

启动Transform、GNSS/IMU模块后，输入如下指令：

```
mainboard -d modules/localization/dag/dag_streaming_rtk_localization.dag
```

- **注意**：受限于法律法规等相关问题，部分数据包**不提供**`/apollo/sensor/gnss/odometry`、`/apollo/sensor/gnss/ins_stat`**这两个**`channel`，而直接提供`/apollo/localization/pose`数据。此时需要借助`/apollo/modules/tools/sensor_calibration/`下的两个脚本工具（本质上时py脚本，但是在Apollo 6.0后也被统一编译成了可执行文件）。

  开启两个不同终端进入docker后在/apollo根目录下分别执行：

  ```
  ./bazel-bin/modules/tools/sensor_calibration/ins_stat_publisher

  ./bazel-bin/modules/tools/sensor_calibration/odom_publisher
  ```

  这两个脚本便可以产生`/apollo/sensor/gnss/ins_stat`、`/apollo/sensor/gnss/odometry`这两个`channel`，之后用`cyber_recorder`工具重新生成一个数据包。如果上述任一脚本找不到，请执行`./apollo.sh build_opt tools`来生成它们。

4. 基于NDT算法的定位

4.3. 代码优化

1. 修改源码中因为Eigen内存没对齐导致的相关错误，

   位置：文件`/apollo/modules/localization/ndt/ndt_localization.h`第136行

   ```
   //std::list<TimeStampPose> odometry_buffer_;
   std::list<TimeStampPose,Eigen::aligned_allocator<TimeStampPose>>
   odometry_buffer_;
   ```

2. 由于NDT定位没有用到点云强度信息，为了便于观察，可以在建图时候提高点云的强度。具体为修改`/apollo/modules/localization/msf/local_tool/data_extraction/pcd_exporter.cc`第81行左右：

   ```
   cloud.points[i].intensity = static_cast<unsigned
   char(msg.point(i).intensity()) *10;
   ```

修改完并保存后，重新编译文件：

```
bash apollo.sh build_opt localization
```

4.4. 构建NDT定位地图

1. 进行定位地图前需要准备以下工作：
   - **Apollo系统已经使用`build_opt`进行编译**：build_opt编译的程序运行速度比使用build进行编译的程序要快速很多；
   - 完成标定任务，将**lidar到imu的外参**存放在相应的矫正文件下；
   - 所使用的数据集中**至少**需要保证该数据集有`/apollo/localization/pose`或者`/apollo/sensor/gnss/odometry`两个通道；当两个`channel`中仅有一个存在时，**两者可以相互替换。**
2. 确定下列信息准备完毕：
   - 待生成地图的名称（以`map_test`为例）
   - 所用数据集所在的文件夹（以`data/bag/localization`为例）
   - 数据集生产地区的`zone_id`（北京为`50`）
   - 激光点云名称（以`lidar`为例）
   - 外参文件存放位置（以`/apollo/modules/calibration/data/地盘名称/lidar_params/lidar_novatel_extrinsics.yaml`为例）
3. 运行代码生成：新的地图将在`/apollo/modules/map/data/map_test`下存储

```
bash supplement/ndt_map_creator.sh \
data/bag/localization \
/apollo/modules/calibration/data/地盘名
称/lidar_params/lidar_novatel_extrinsics.yaml \
51 \
/apollo/modules/map/data/map_test/ndt_map \
lidar
```

4. 创建msf地图，用于可视化显示

```
bash supplement/msf_map_creator.sh \
data/bag/localization \
/apollo/modules/calibration/data/底盘名
称/lidar_params/lidar_novatel_extrinsics.yaml \
51 \
/apollo/modules/map/data/map_test \
lidar
```

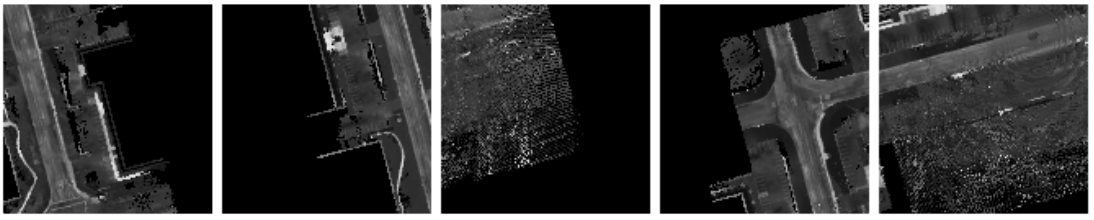
5. 在utm区域边界建图时，有可能出现拉取地图过小，导致建图时出现非法节点。报错为：

```
MapNodeIndex::get_map_node_index illegal n: xxx
```

应修改文件
`/apollo/modules/localization/msf/local_pyramid_map/base_map/base_map_config.cc`
中第 35 行左右：

```
// map_range_ = Rect2D<double>(0, 0, 1000448.0, 10000384.0);  // in meters
map_range_ = Rect2D<double>(0, 0, 9000448.0, 90000384.0);  // in meters
```

6. 代码分析：核心思路包括以下几个步骤
   - 数据包解压生成pcd文件以及对应的位姿（`cyber_record_parser`）
   - 位姿插值（`poses_interpolator`）
   - 创建`ndt mapping`（`ndt_map_creator`）
7. 验证：查看`/apollo/modules/map/data/map_test/ndt_map/image/`中的图像



4.5. 修改启动文件

修改或创建启动文件：`/apollo/modules/calibration/data/底盘名
称/localization_dag/dag_streaming_ndt_localization.dag`，确保内容如下：

```
# 路径：/apollo/modules/calibration/data/底盘名
称/localization_dag/dag_streaming_ndt_localization.dag
module_config {
```

```
  module_library : "/apollo/bazel-
bin/modules/localization/ndt/libndt_localization_component.so"

  components {
    class_name : "NDTLocalizationComponent"
    config {
      name : "ndt_localization"
      flag_file_path : "/apollo/modules/localization/conf/localization.conf"
      readers: [
        {
          channel: "/apollo/sensor/gnss/odometry"
          qos_profile: {
            depth : 10
          }
          pending_queue_size: 50
        }
      ]
    }
  }
}
```

4.6. 修改配置文件

确保`/apollo/modules/calibration/data/底盘名称/localization_conf/localization.conf`内容如下:

```
# 路径: /apollo/modules/calibration/data/底盘名
称/localization_conf/localization.conf

# 5行
--map_dir=/apollo/modules/map/data/map_test # 指定地图位置

# 40行
--lidar_height_default=1.70 #设定激光雷达坐标系原点到地面的高度

# 115行
--local_utm_zone_id=50 # 北京地区zone id为50

# 130行
--lidar_topic=/apollo/sensor/lidar/compensator/PointCloud2 # 点云话题的名称

# 135行
--lidar_extrinsics_file=/apollo/modules/calibration/data/底盘名
称/lidar_params/lidar_novatel_extrinsics.yaml # 外参文件，确保已经完成外参校正
```

4.7. 启动NDT定位

1. 启动Transform、Lidar、GNSS/IMU模块，输入如下指令:

```
mainboard -d modules/localization/dag/dag_streaming_ndt_localization.dag
```

2. 将`/apollo/modules/localization/dag/dag_streaming_msf_visualizer.dag`文件复制粘贴至`/apollo/modules/calibration/data/底盘名称/localization_dag/`文件夹下，并确保内容如下：

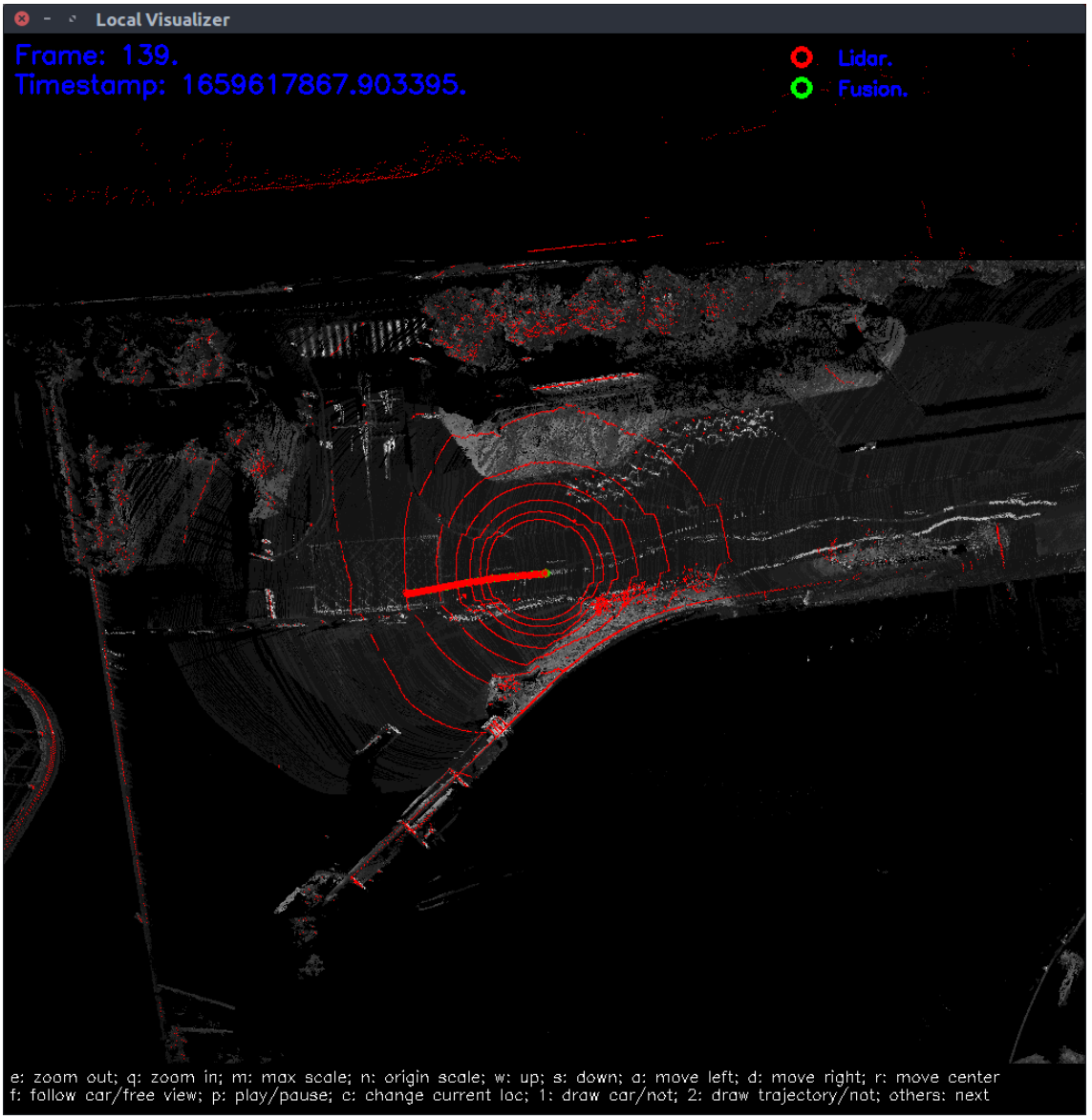
```
# 路径：/apollo/modules/calibration/data/底盘名
称/localization_dag/dag_streaming_msf_visualizer.dag
module_config {
    module_library : "/apollo/bazel-
bin/modules/localization/msf/local_tool/local_visualization/online_visual/onl
ine_visualizer_compenont.so"

    components {
      class_name : "OnlineVisualizerComponent"
      config {
        name : "msf_visualizer"
        flag_file_path :
"/apollo/modules/localization/conf/localization.conf"
        readers: [
          {
            channel: "/apollo/sensor/lidar/compensator/PointCloud2"
          }
        ]
      }
    }
}
```

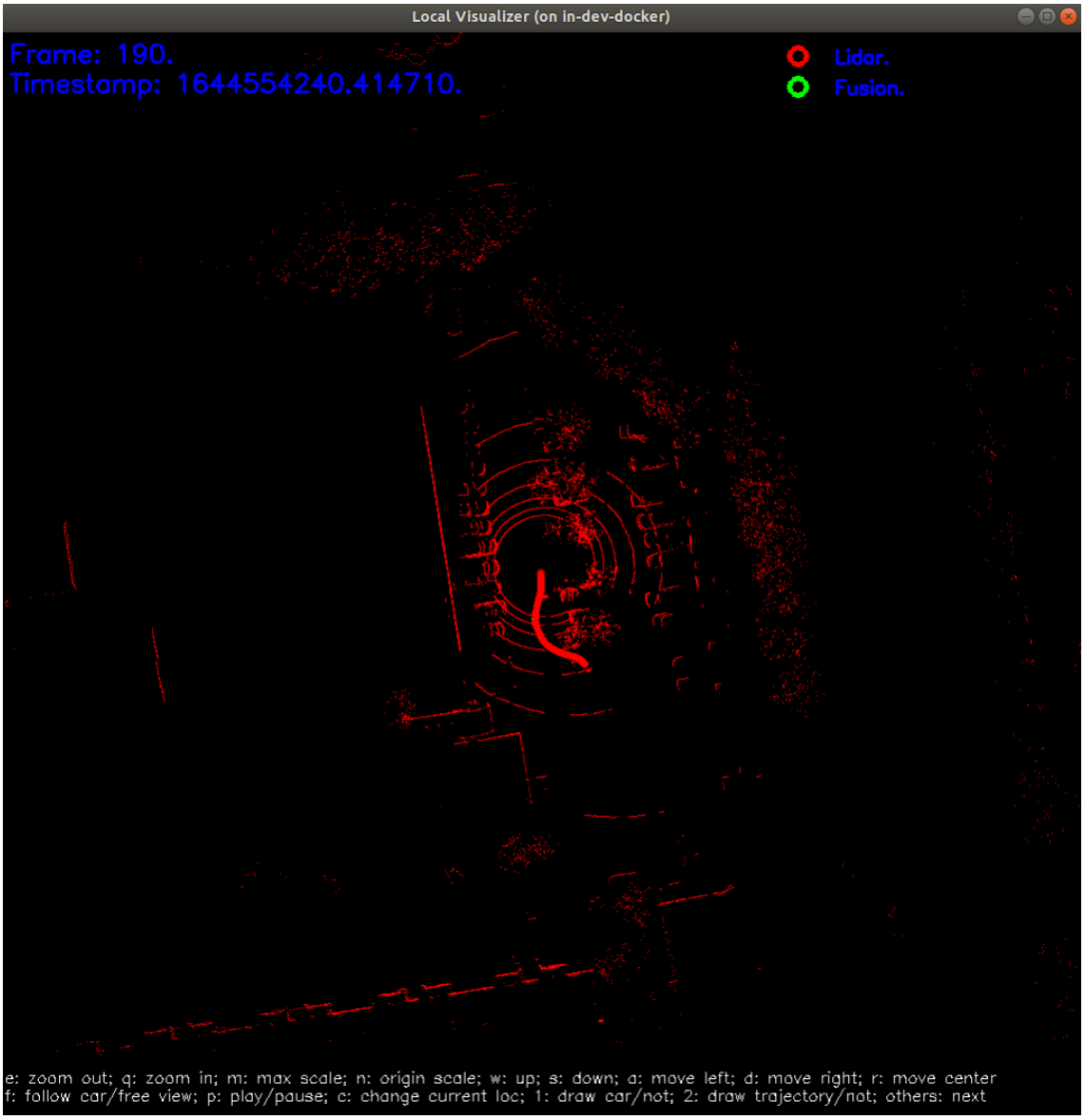
3. 启动可视化程序
   尽管可视化程序在名称上归属于msf，但是它在所有定位方式中均可以使用。使用时需要确认：
   - 定位方式的地图依赖于msf地图，**需要预先建立msf的`local map`地图**，详见本章5.3 构建MSF定位地图；
   - 定位策略依赖于`localization.conf`文件的配置，特别是地图所在位置，需要仔细审查。

```
mainboard -d modules/localization/dag/dag_streaming_msf_visualizer.dag
```

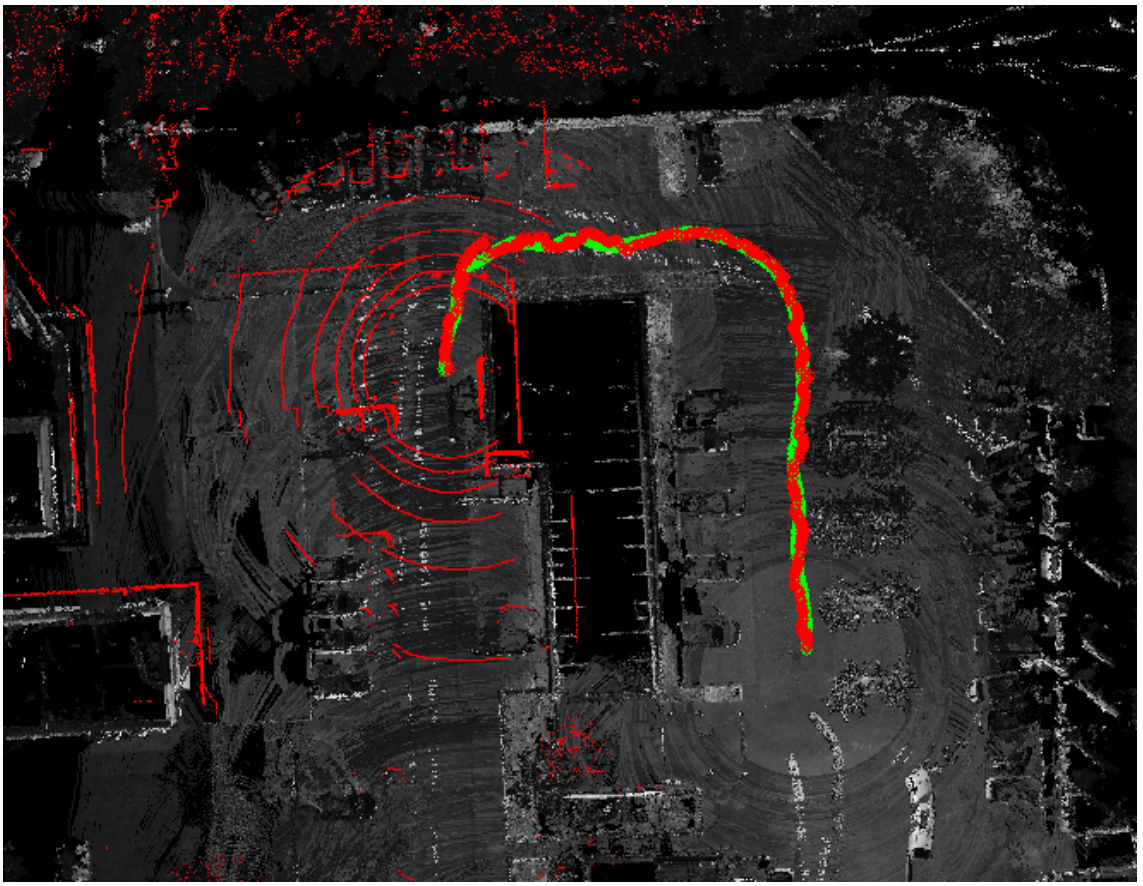
启动`cyber_monitor`，当出现`/apollo/localization/ndt_lidar`时认为ndt启动成功。在可视化结果展示界面中，按c键可以在Lidar和Fusion定位中进行切换：



> 注意：当地图不可显示并且monitor显示定位正常时，删除缓存文件：`rm -rf cyber/data/map_visual`后重新启动，下图为不正常显示的一个例子：



4. 遥控车辆行驶一段距离，观察可视化界面，分析雷达里程计轨迹和融合轨迹是否一致，并观察点云与地图匹配程度，当点云和实际匹配较差时，认为定位失败：



5. 定量评价定位效果：
    - 与之前一样，这里的Apollo代码没有针对Eigen进行内存对齐，导致我们运行出错，因此需要首先修正代码。在 `/apollo/modules/localization/msf/local_tool/data_extraction/compare_poses.cc` 中：

```
// 第58行：std::map<unsigned int, Eigen::Affine3d> *out_poses,
std::map<unsigned int, Eigen::Affine3d, std::less<unsigned int>,
Eigen::aligned_allocator<std::pair<unsigned int, Eigen::Affine3d>>>
*out_poses,

// 第171行：std::map<unsigned int, Eigen::Affine3d> out_poses_a;
std::map<unsigned int, Eigen::Affine3d, std::less<unsigned int>,
Eigen::aligned_allocator<std::pair<unsigned int, Eigen::Affine3d>>>
out_poses_a;

// 第173行：std::map<unsigned int, Eigen::Affine3d> out_poses_b;
std::map<unsigned int, Eigen::Affine3d, std::less<unsigned int>,
Eigen::aligned_allocator<std::pair<unsigned int, Eigen::Affine3d>>>
out_poses_b;
```

    - 修改脚本文件 `/apollo/scripts/msf_local_evaluation.sh`：

```
# 路径：/apollo/scripts/msf_local_evaluation.sh

# 第22行：LIDAR_LOC_TOPIC="/apollo/localization/msf_lidar"
LIDAR_LOC_TOPIC="/apollo/localization/ndt_lidar"

# 第25行：CLOUD_TOPIC="/apollo/sensor/velodyne64/compensator/PointCloud2"
CLOUD_TOPIC="/apollo/sensor/lidar/compensator/PointCloud2"
```

```
# 第47行：注释掉
#$APOLLO_BIN_PREFIX/modules/localization/msf/local_tool/data_extraction/
compare_poses \
#   --in_folder $IN_FOLDER \
#   --loc_file_a $GNSS_LOC_FILE \
#   --loc_file_b $ODOMETRY_LOC_FILE \
#   --imu_to_ant_offset_file "$ANT_IMU_FILE" \
#   --compare_file "compare_gnss_odometry.txt"
```

- 编译 `Localization` 模块：

```
bash apollo.sh build_opt localization
```

- 启动Transform、Lidar、GNSS/IMU模块，启动NDT定位，录制使用NDT算法进行定位的record包，并放置于 `/apollo/data/bag/ndt` 文件夹下：

```
cyber_recorder record -a -i 600 -o ndt.record
```

- 运行脚本文件：

```
bash scripts/msf_local_evaluation.sh data/bag/ndt
```

- 结果如下：

```
Fusion localization result:
9765 frames
criteria : mean     std      max      < 30cm   < 20cm   < 10cm  con_frames(>30cm)
error    : 0.032434 0.021282 0.117411 1.000000 1.000000 0.981567 000000
error lon: 0.020713 0.015008 0.070142 1.000000 1.000000 1.000000 000000
error lat: 0.020675 0.020571 0.104969 1.000000 1.000000 0.992729 000000
error alt: 0.016919 0.017489 0.082650 1.000000 1.000000 1.000000 000000
criteria : mean     std      max      < 1.0d   < 0.6d   < 0.3d  con_frames(>1.0d)
error rol: 0.000033 0.000027 0.000142 1.000000 1.000000 1.000000 000000
error pit: 0.000035 0.000028 0.000148 1.000000 1.000000 1.000000 000000
error yaw: 0.022564 0.013742 0.073510 1.000000 1.000000 1.000000 000000

Lidar localization result:
976 frames
criteria : mean     std      max      < 30cm   < 20cm   < 10cm  con_frames(>30cm)
error    : 0.037171 0.025449 0.201953 1.000000 0.998975 0.971311 000000
error lon: 0.023682 0.018310 0.136131 1.000000 1.000000 0.995902 000000
error lat: 0.023309 0.024289 0.201769 1.000000 0.998975 0.977459 000000
error alt: 0.018768 0.020661 0.158064 1.000000 1.000000 0.991803 000000
criteria : mean     std      max      < 1.0d   < 0.6d   < 0.3d  con_frames(>1.0d)
error rol: 0.032969 0.054758 1.043162 0.998975 0.997951 0.992828 000001
error pit: 0.038421 0.077384 2.134161 0.998975 0.998975 0.995902 000000
error yaw: 0.029062 0.026158 0.302041 1.000000 1.000000 0.998975 000000
```

定位结果分为横向精度与纵向精度，可以用 10 cm 位置精度来衡量。同时，在自动驾驶中一般认为 30 cm 为最大允许误差，因此小于 30 cm 精度的占比用于衡量定位算法的鲁棒性。

> 注意：在进行 NDT 定位时，有时会出现GNSS时间戳落后与点云时间戳，造成无法定位的情形。此时可以采用外插法或者等待若干时间，可修改 `/apollo/modules/localization/ndt/ndt_localization.cc` 实现。具体内容在这里不再展开，感兴趣的可以自行尝试修改代码。