Turtlebot3 SLAM与导航虚拟仿真实验-拓展实验一

- 本次实验课适用ROS系统版本为`melodic`和`kinetic`。如果有同学使用其他版本ROS系统而在实验过程中出现问题，恐怕无法给予帮助。
- 仿真实验和实际环境实验有较大不同，本实验仿真运行效果也许不错，但实际环境中要差很多。
- 从PDF文件复制终端命令的时候会将行末的回车换行符一并复制，可先将复制内容粘贴到一文本文件中，删除行尾换行符后再复制、粘贴到终端中执行。

目录

二、拓展实验一：高级导航

在基本实验部分，我们通过rviz提供的GUI工具，完成了设定机器人初始位姿、指定机器人导航目标点的工作。但在实际应用中，更多情况下我们希望自己边写程序控制机器人完成导航任务。拓展实验一分为三个部分：设定机器人初始位姿、设定单个导航目标点、设置一系列机器人运行路径点，同学们通过补充程序代码实现功能。

2.1 源代码文件介绍

为了降低实验难度，避免在非ROS的内容上浪费时间，在进行实验之前，请大家将功能包`advanced_navgation`的源代码放在自己的工作空间下完成编译。功能包中的文件及其作用如下：

```
├── CMakeLists.txt                  指示 catkin_make 如何编译功能包的源代码
├── data                            保存数据的文件夹
│   └── landmarks.txt               路径点数据
├── include                         头文件文件夹
│   └── ExperNodeBase.hpp           ROS 节点基类ExperNodeBase定义实现
├── launch                          launch文件
│   ├── navi_expr.launch            启动文件，可在启动导航时自动设置机器人初始位姿
│   └── sequence_goals.launch       启动文件，实现控制机器人沿路径点运行
├── package.xml                     ROS功能包定义文件
└── src                             源代码文件夹
```

```
    ├── InitPose                    设置初始位姿
    │   └── InitPose.cc
    ├── SeqGoal                     设置导航点序列
    │   └── SeqGoal.cc
    └── SetGoal                     设置单个导航点
        └── SetGoal.cc
```

进行实验之前请大家先阅读一下相关代码的注释，对代码比较好的理解有助于快速完成实验。确认编译没有错误后，可以先执行下列指令保存最初版本的代码：

```
$ roscd advanced_navi     # 如果出现找不到功能包的错误，就 souce 一下
$ git init
$ git add --all && git commit -m "init repo"
```

此时功能包`advanced_navigation`下的所有代码均以`init repo`为名被记录在版本数据库中，数据库保存在`$(find advanced_navigation)/advanced_navigation/.git/`下，同学们不要删除。

> 上述路径中`$(find advanced_navigation)`表示功能包`advanced_navigation`的路径，这是借用了launch文件中的写法；也可以在终端中执行`rospack find advanced_navi`，两者指代的是相同的东西。下文中将会使用相同的描述指代功能包的具体路径。

2.2 编写程序设定机器人初始位姿

本节实验完成编写程序设定机器人初始位姿的任务，主要分四个阶段进行，分别完成设置机器人初始位姿、清空Cost Map、从Gazebo直接获取机器人初始位姿和launch文件编写实现一“键”启动四个步骤。

2.2.1 第一阶段：发布机器人初始位姿

开展实验之前需要明确机器人的初始位姿的格式和发送到的topic。首先运行导航实验中需要启动的launch文件，完成后打开新的终端窗口，执行`rostopic list`命令查看当前所有的topic，找到名为`/initialpose`的topic，执行`rostopic type`查看该topic中发送的消息类型：

```
guoqing@GQLU18:~$ rostopic type /initialpose
geometry_msgs/PoseWithCovarianceStamped
```

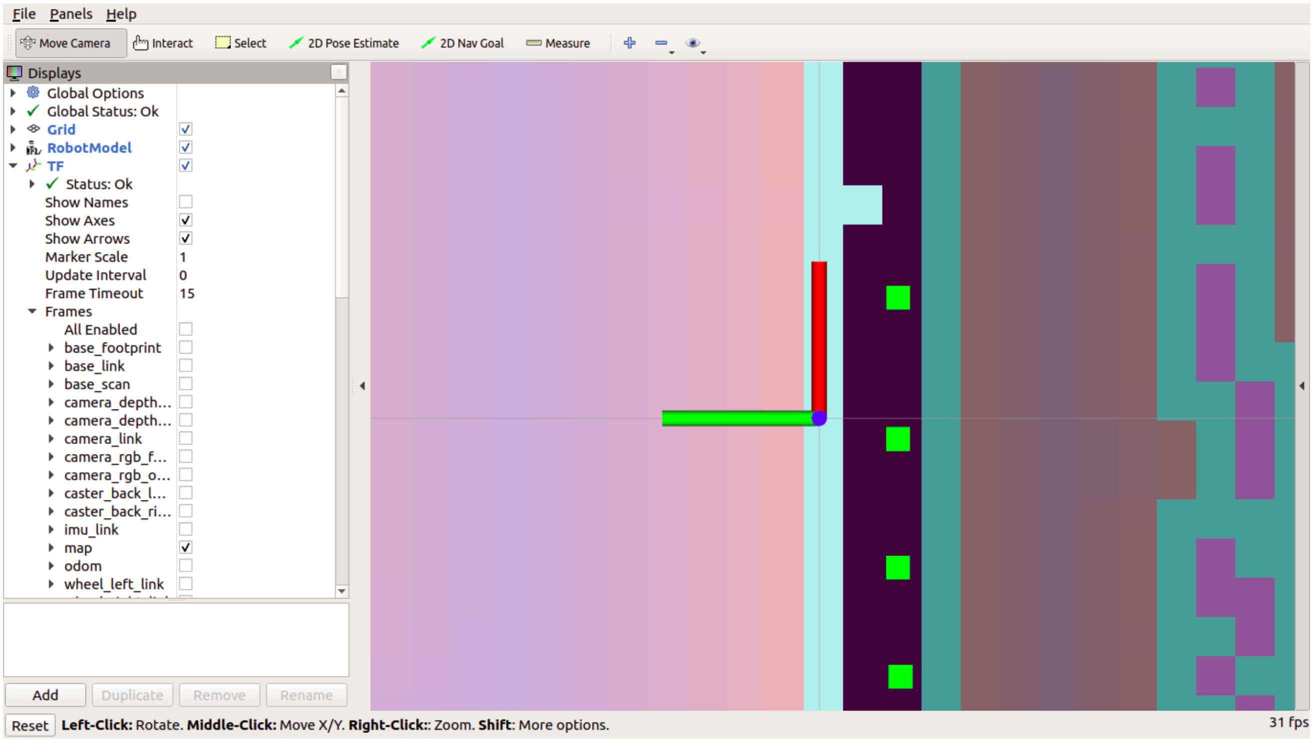
对于消息类型`geometry_msgs/PoseWithCovarianceStamped`的具体组成格式，则使用命令`rosmsg show geometry_msgs/PoseWithCovarianceStamped`查看：

```
guoqing@GQLU18:~$ rosmsg show geometry_msgs/PoseWithCovarianceStamped
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
geometry_msgs/PoseWithCovariance pose
  geometry_msgs/Pose pose
    geometry_msgs/Point position
      float64 x
      float64 y
      float64 z
    geometry_msgs/Quaternion orientation
      float64 x
      float64 y
      float64 z
      float64 w
  float64[36] covariance
```

观察输出，我们得知该消息类型主要由消息头（header）、位姿（position）、四元数表示的旋转（orientation）和协方差矩阵（covariance）组成。为了验证我们的想法，可以执行`rostopic echo /initialpose`捕捉该topic中传输的数据。在rviz中手动设置一次机器人初始位姿，可以捕捉到以下输出：

```
guoqing@GQLU18:~$ rostopic echo /initialpose
WARNING: no messages received and simulated time is active.
Is /clock being published?
header:
  seq: 0
  stamp:
    secs: 614
    nsecs: 247000000
  frame_id: "map"
pose:
  pose:
    position:
      x: -3.00999975204
      y: 0.980000197887
      z: 0.0
    orientation:
      x: 0.0
      y: 0.0
      z: 0.00399980512294
      w: 0.999992000747
  covariance: [0.25, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.25, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.
0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.06853892326654787]
---
```

通过数据我们可以发现我们的设想是正确的。但在我们尝试确定坐标时，有个新问题：坐标系 `map` 是如何定义的？在rviz中我们打开TF变换树的显示，并且只显示 `map` 的坐标系：



坐标系图样有点小，需要放大才能够看清。其中红轴表示X轴正方向，绿轴表示Y轴正方向，蓝轴表示Z轴正方向（上图无法体现）。知道坐标系定义后，我们就可以通过数格子的方式确定机器人初始位姿的坐标了；消息类型和发布的topic也已知，接下来请同学们自行补充完善 `$(find advanced_navigation)/src/InitPose/InitPose.cc` 中的内容。

> - 机器人只能在平面上运行，因此Z轴坐标设置为0；
> - 机器人的头部为安装有轮子的一端，可通过rviz中发大机器人模型查看；
> - 偏航角（机器人头部朝向角度）大家可以通过实验确定；程序中提供了将该角度转换成为四元数的函数，具体数学原理同学们在本次实验课中不必深究；
> - 协方差矩阵的含义参考程序中的注释；本次实验中可以参考rviz消息中的数据设置协方差矩阵。

在导航的launch文件启动后，通过启动 `init_pose` 节点，可以设置机器人的初始位姿：

```
rosrun advanced_navi init_pose
```

如果在rviz中发现机器人的位姿改变到设置的位置，则阶段一成功。

为了方便以后复现本阶段实验结果，**确保所有代码文件保存后**，在终端中执行下面的命令：

```
$ roscd advanced_navi && git add --all && git commit -m "expr 2.2.1"
```

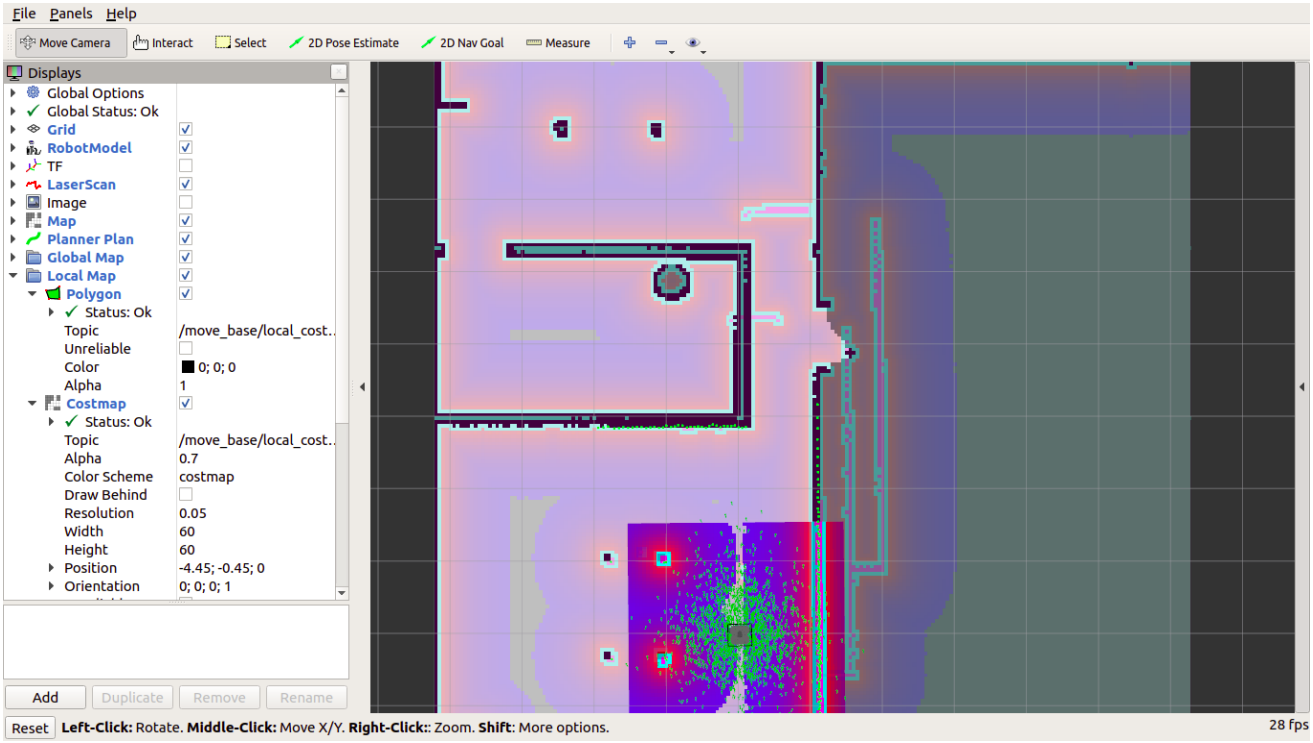
其中字符串"expr 2.2.1"可以换成任何你觉得合适的版本说明字符。如果在写实验报告时需要复现这一阶段的实验、摘抄关键程序代码，则执行下面的命令即可切换不同的代码版本：

```
$ roscd advanced_navi && git reset --soft "要切换到的代码的版本说明字符"
```

> git中对这里的称呼是某次“代码提交”，这里称之为“版本”是为了方便不熟悉git的同学理解。

2.2.2 第二阶段：清除Costmap

在第一阶段完成之后，我们通过rviz可以发现，在最初时刻根据激光雷达数据创建的CostMap依旧存在，这些障碍在有些特定的环境结构中会将可通行路径堵死，使得全局路径规划器无法计算出可行路径：



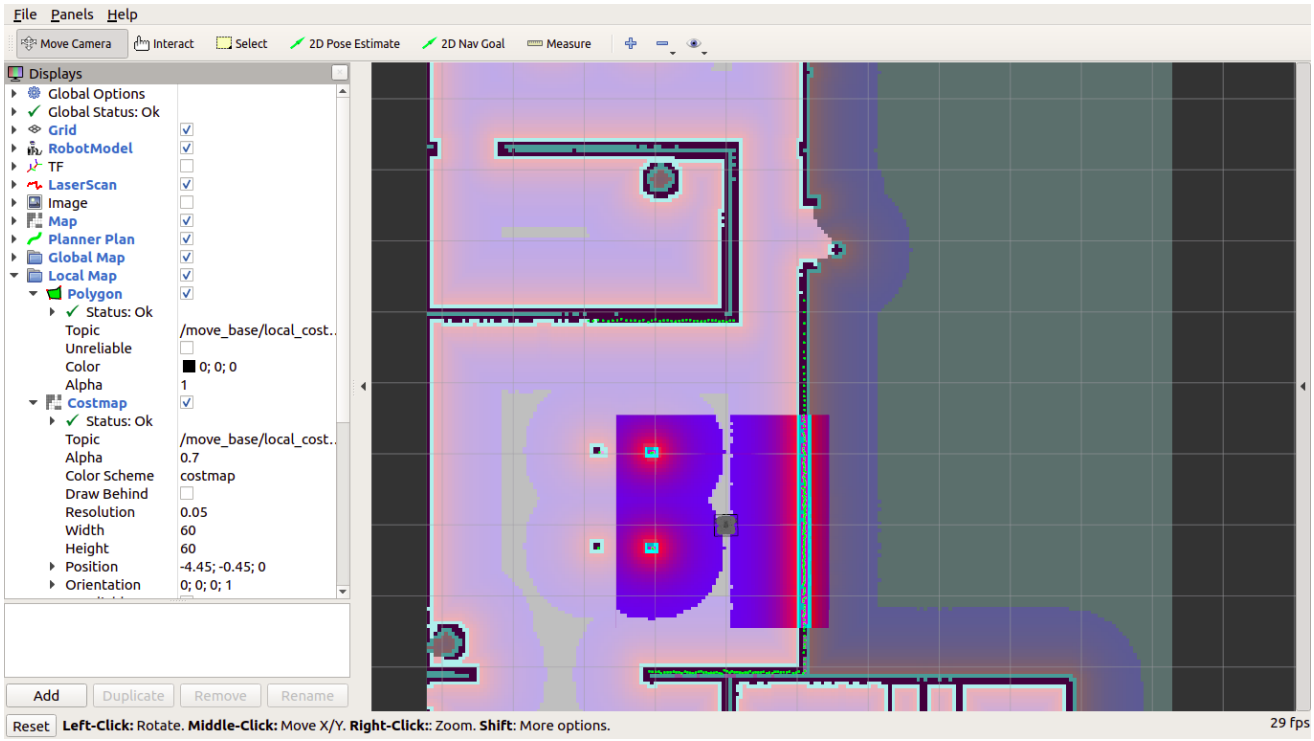
在实验第二阶段，我们研究一下如何清除建立的Costmap。执行`rosservice list`查看当前所有可用的服务，找到`/move_base/clear_costmaps`。通过调用此服务，可以实现清空Costmap的目的。利用`rosservice type`可以查看服务类型：

```
guoqing@GQLU18:~$ rosservice type /move_base/clear_costmaps
std_srvs/Empty
```

对于服务类型`std_srvs/Empty`，可以执行`rossrv show std_srvs/Empty`查看：

```
guoqing@GQLU18:~$ rossrv show std_srvs/Empty
---
```

可以发现`std_srvs/Empty`是比较特殊的服务类型，它的请求和响应均为空；因此我们只需要调用此服务，`move_base`收到后就会执行清除Costmap的操作了，请同学们补充完善程序，实现运行`init_pose`节点后不仅能够设置机器人的初始位姿，还可以清除之前创建的Costmap。理想的实验结果如下图所示：



完成实验后，记得保存所有编辑的代码文件，记录当前代码版本：

```
$ roscd advanced_navi && git add --all && git commit -m "expr 2.2.2"
```

2.2.3 第三阶段：从gazebo获取机器人初始位姿

在第一阶段的实验中，我们是通过数格子的方式确定机器人的位置坐标的；但是这并不适用实际应用场景。考虑到这是个虚拟仿真的实验，我们可以在最开始运行的时候，调用gazebo的服务`/gazebo/get_model_state`获取机器人的当前位姿。请同学们使用第一阶段实验中用到的方法，自行确定服务的数据类型，实现机器人初始位姿的自动获取和设置。

完成实验后，记得保存所有编辑的代码文件，记录当前代码版本：

```
$ roscd advanced_navi && git add --all && git commit -m "expr 2.2.3"
```

2.2.4 第四阶段：编写launch文件实现一“键”启动运行

经过上述三个阶段实验，我们基本完成了节点`init_pose`设置机器人初始位姿的功能。但在上述实验过程中，每次调试程序我们都需要重新启动两个launch文件，然后再启动`init_pose`节点，十分不便。我们希望在执行`turtlebot3_navigation`（第二个launch文件）时可以自动执行`init_pose`节点，一种实现方式是编写launch文件。请同学们在文件`$(find advanced_navigation)/launch/navi_expr.launch`现有内容的基础上，添加运行`init_pose`节点的描述标签，实现执行下述命令时，能够同时执行`turtlebot3_navigation`和`init_pose`的目的：

```
$ roslaunch advanced_navi navi_expr.launch
```

launch文件中各个节点的标签虽然有先后，但实际运行时几乎是并发执行的，这可能会导致`ros::Publisher`在topic尚未被订阅时就发布数据、`ros::ServiceClient`在服务端尚未运行时就调用服务。为了解决上述问题，`ros::Publisher`和`ros::ServiceClient`均提供了相关函数：

- `ros::Publisher::getNumSubscribers()`
- `ros::ServiceClient::exists()`
- `ros::ServiceClient::waitForExistence()`
- ....

请同学们查阅有关资料，灵活选择使用相关函数，避免出现“导航功能包未准备好而`init_pose`节点已经发送初始数据”的情况。

完成实验后，记得保存所有编辑的代码文件，记录当前代码版本：

```
$ roscd advanced_navi && git add --all && git commit -m "expr 2.2.4"
```

2.3 编写程序设定机器人导航目标点

本节实验完成编写程序设定机器人导航目标点的任务，主要分两个阶段进行，分别完成设置导航目标点和获取导航执行结果两个步骤。

2.3.1 第一阶段：发布导航目标点

已知节点程序通过向topic `/move_base/goal` 发布消息可以设置导航目标点。请同学们综合使用`rostopic`、`rosmsg`等工具确定消息数据类型，修改`$(find advanced_navigation)/src/SetGoal/SetGoal.cc`，实现设定机器人导航目标点；要求可以在终端中手动输入导航目标点的坐标和朝向（x,y,yaw），yaw角转换为四元数的函数程序中已有提供。

> 进行此实验时同学们请注意，如果程序在主循环中等待输入时，对`Ctrl+C`这样的请求可能无法即时响应，需要随便输入数据让程序执行过去后，才会轮到`ros::ok()`函数检测是否有`Ctrl+C`按下过。所以同学们如果发现按下`Ctrl+C`无法终止程序时，并非是程序失去响应，输入伪数据即可；不要尝试使用`kill`指令、`Ctrl+Z`或者是直接发送其他进程信号等方式关闭节点程序，否则可能会出现ROS主节点中相关信息未更新，下次运行该节点时可能会不正常（比如明明发布了数据但是用`rostopic echo`就是看不到）；一旦这样的情况发生，只有重启ROS主节点才能解决。

完成实验后，记得保存所有编辑的代码文件，记录当前代码版本：

```
$ roscd advanced_navi && git add --all && git commit -m "expr 2.3.1"
```

2.3.2 第二阶段：获取导航执行结果

已知通过topic `/move_base/result` 和 `/move_base/status` 均可以获取当前导航的执行状态。请同学们使用rviz设置导航点，查看这两个topic中信息的异同，选择任一topic编写程序，实现只有在上一次的导航成功进行或失败后才能继续输入下一次的导航目标点。

完成实验后，记得保存所有编辑的代码文件，记录当前代码版本：

```
$ roscd advanced_navi && git add --all && git commit -m "expr 2.3.2"
```

2.4 编写程序设置机器人导航路径点序列

本次实验比较简单，要求大家补充程序`$(find advanced_navigation)/src/SeqGoal/SeqGoal.cc`，实现从外部文件`$(find advanced_navigation)/data/landmarks.txt`中读取路径点（其实就是每次导航的导航目标点），机器人按照一定的次序先后到达不同的地点；要求机器人抵达一个地点后停留2秒钟后再动身前往下一个路标点。程序中关于文件数据读取的函数已经实现，同学们可以根据自己的需要改动数据文件中的坐标点进行测试。

注意使用rosrun运行节点`sequence_goal`时需要在命令行中指定数据文件的路径：

```
$ rosrun advanced_navi sequence_goal $(find advanced_navigation)/data/landmarks.txt
```

由于在目前的实验中，这个数据文件的路径是固定的，因此请同学们搜索相关资料，编写`$(find advanced_navigation)/launch/sequence_goals.launch`文件，实现通过运行此launch文件达到自动给定节点命令行参数的目的。

> 本实验中的launch文件编写所需要知识大多超过了课本介绍的范畴，但是这些用法在实际的ROS工程中又经常出现。除了搜索资料外，查看本次实验用到的turtlebot3各功能包的launch文件也很有参考意义，例如查看导航实验中多次执行的`turtlebot3_navigation.launch`：
>
> ```
> $ roscd turtlebot3_navigation/launch
> $ <你的编辑器命令，如code gedit vim等>  turtlebot3_navigation.launch
> ```

刘国庆

2020.06.07