机器人学期中作业展示

肖振新

ROS Navigation GlobalPlanner全局路径规划

肖振新

改进源码的A*算法

问题在于，原本的A*算法一旦potential的值被置位（即，对应的g（n）函数），再次访问到这个点时，就不会再尝试更新这个点的potential值了，这对于二值栅格地图而言（即，只有有障碍物和没有障碍物两种区别，在本仿真环境中，可以近似是这样的地图），没有太大区别，但是如果在复杂权值的地图中，效果差异明显。

源代码：

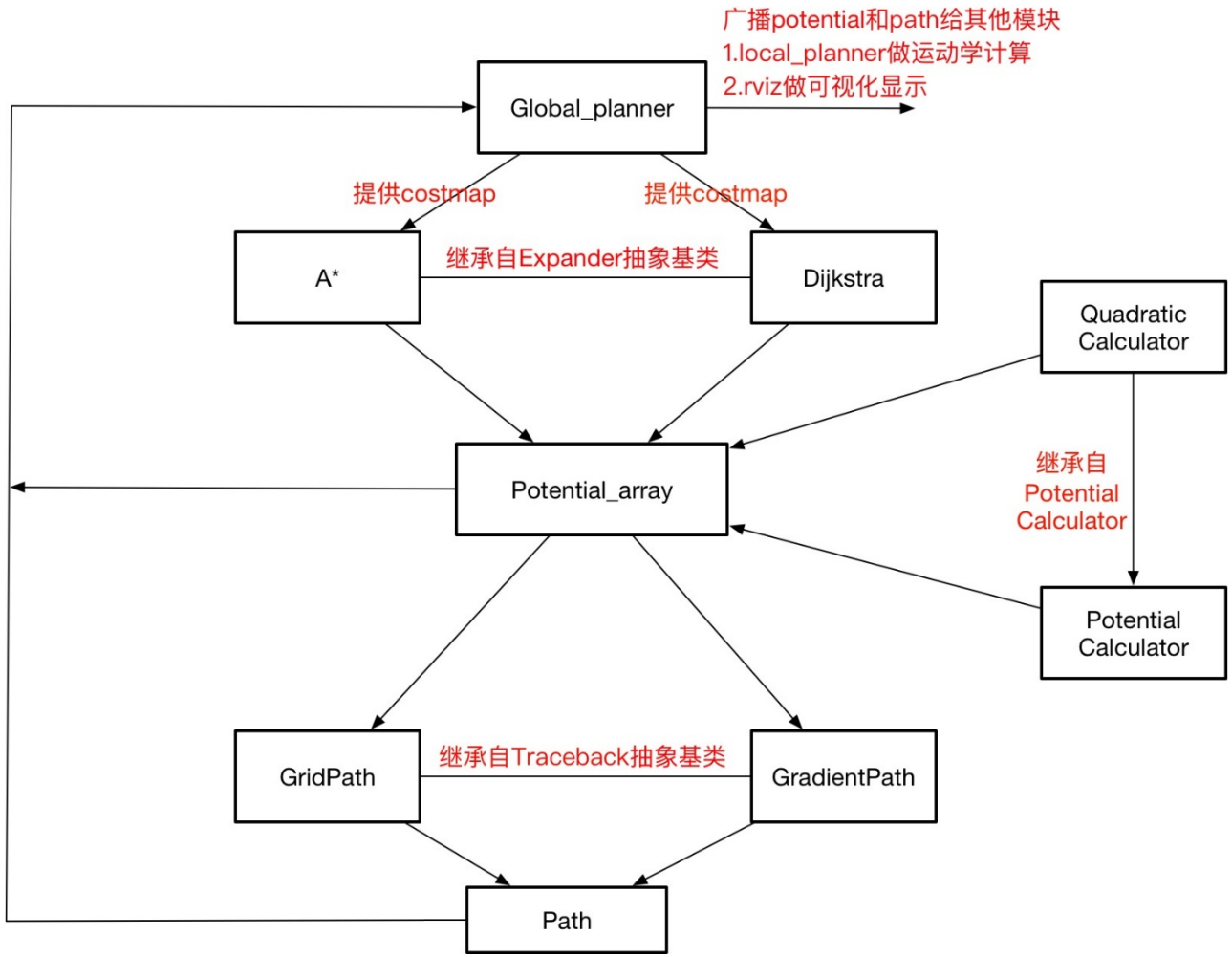
```
if (potential[next_i] < POT_HIGH)
    return;
```

改进后的代码：

```
if (potential[next_i] < POT_HIGH) {
    if(potential[next_i] > p_calc_->calculatePotential(potential,
                    costs[next_i] + neutral_cost_, next_i, prev_potential))
        potential[next_i] = p_calc_->calculatePotential(potential,
                    costs[next_i] + neutral_cost_, next_i, prev_potential);
    return;
}
```

目标导向的动态步长自适应RRT算法

ROS GlobalPlanner包计算图级



A*和dijkstra算法前向搜索计算各个点的potential_map，传给gridpath或gradientpath逆向求解梯度计算最佳路径。每下一个点的potential由quadratic_calculator和普通potential_calculator计算得出。

ROS GlobalPlanner包源码解读

*ROS global_planner*路径规划包源码解读

*ROS navigation*导航包概述

*global_planner*计算图级

动态*reconfigure*参数配置原理

*plan_node*顶层对外交互模块

*planner_core*主要逻辑实现解读

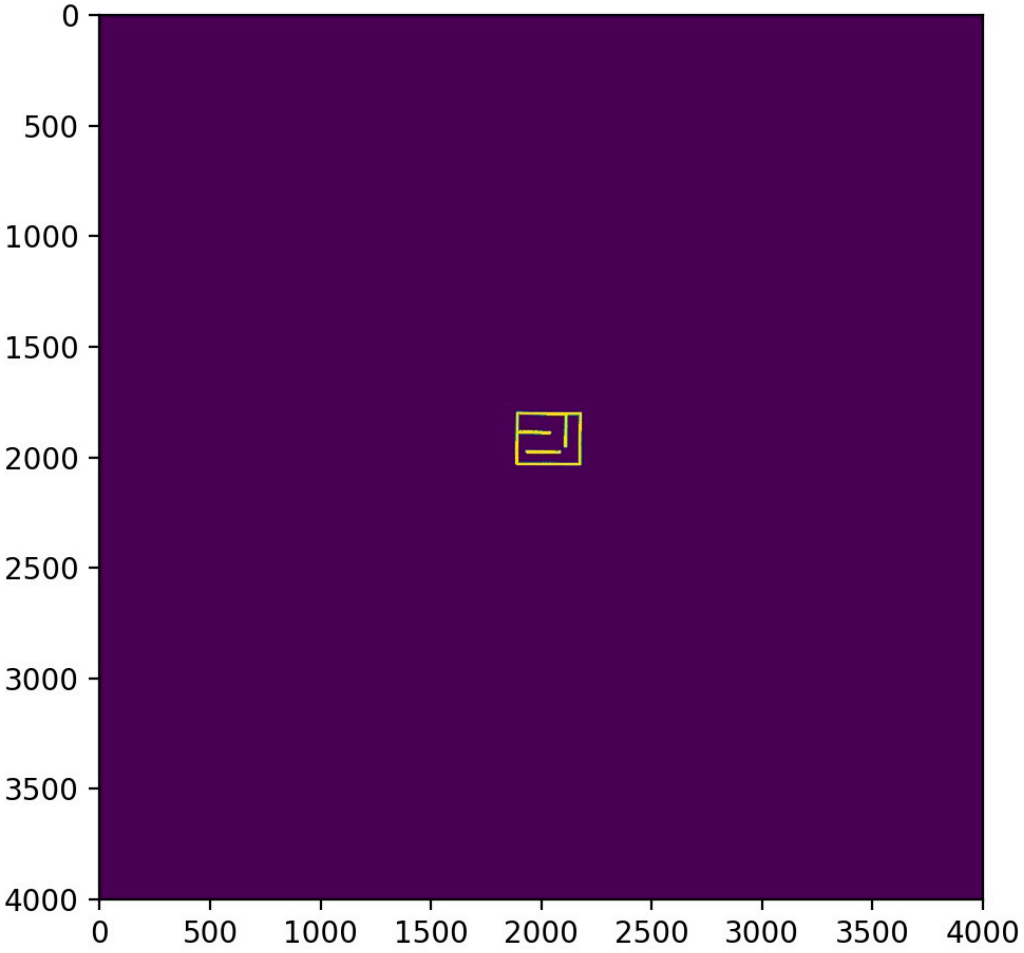
*A**与*dijistra*前向搜索算法

*grid_path*与*gradient_path*逆向搜索算法

*potential_calculator*与*quadratic_calculator*代价计算模块

本人将之前研究global_planner包所得写了一篇博客，因为考试周事情繁多一直没能写完，将在晚些时候放在群里共享给大家交流。左边为该blog的一些目录。

ROS rviz仿真环境（costmap地图解析）

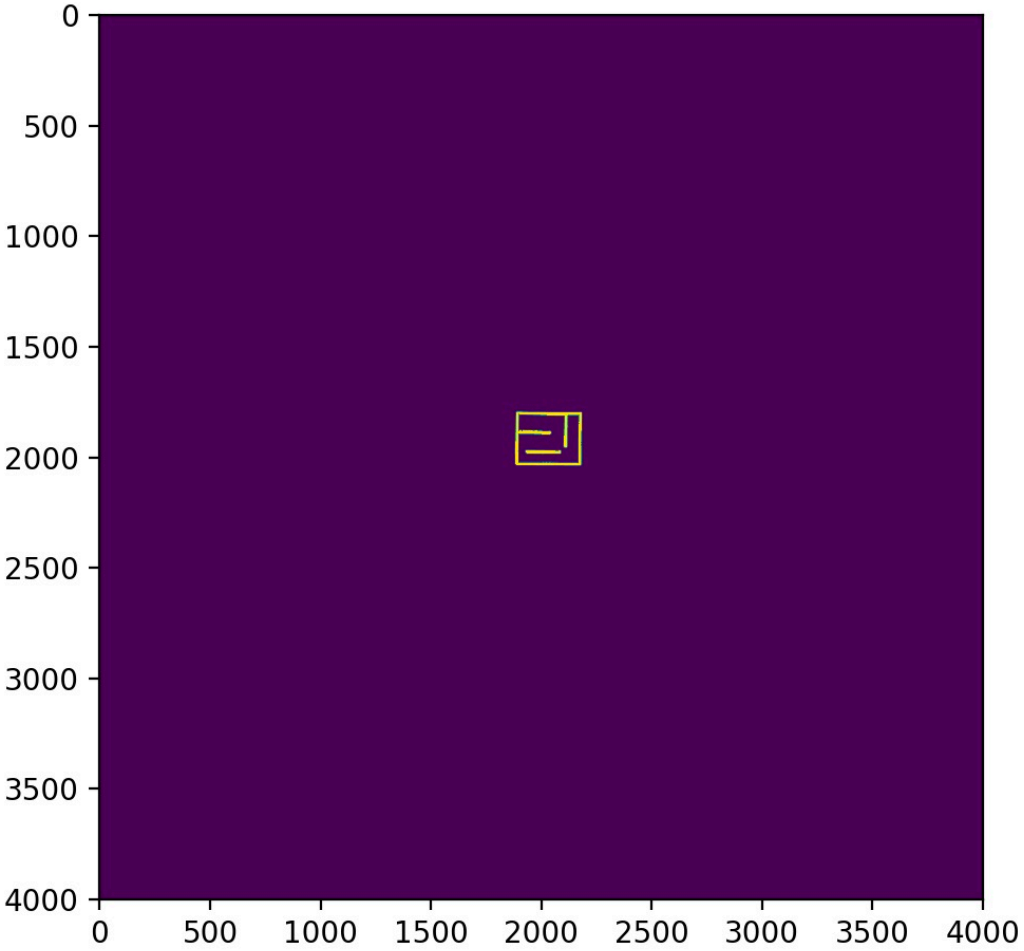


costmap_->getCharMap()

* @brief Will return a pointer to the underlying unsigned char array used as the costmap
* @return A pointer to the underlying unsigned char array storing cost values

costmap是一个unsigned char类型的一维c++数组，对应于二维地图的一维展开，每个点的值对应着经过该点的cost，在本地图中，没有障碍物的大部分位置cost都是0，所以在计算路径时，要人为加入普适cost（在源代码中，对应着neutral_cost参数），有障碍物的点大部分为253（在源码中，对应着一个叫做lethal_cost_的参数）

ROS rviz仿真环境（potential_map势能解析）

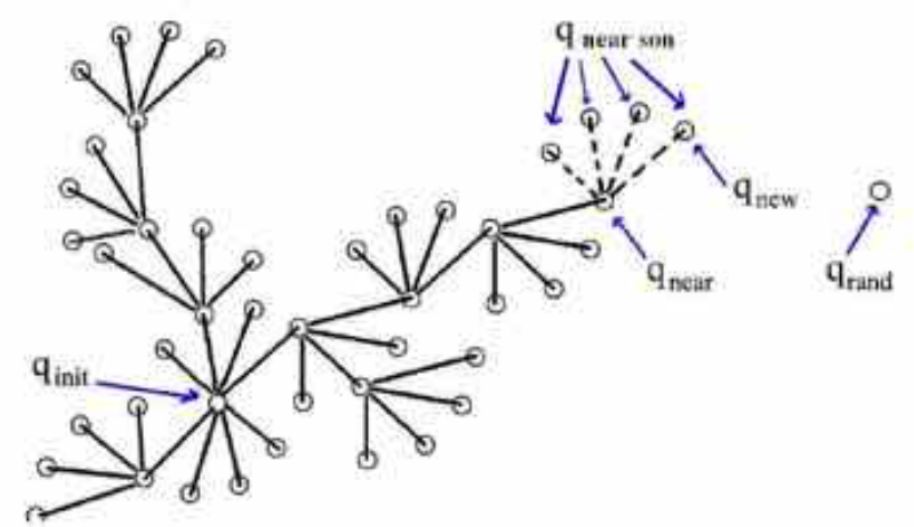


costmap是一个unsigned char类型的一维c++数组，对应于二维地图的一维展开，对于dijkstra算法，每个点的值对应着从起点到该点的总距离；对于A*算法，对应于从起点到该点的总cost（即g（n），注意不是f（n））

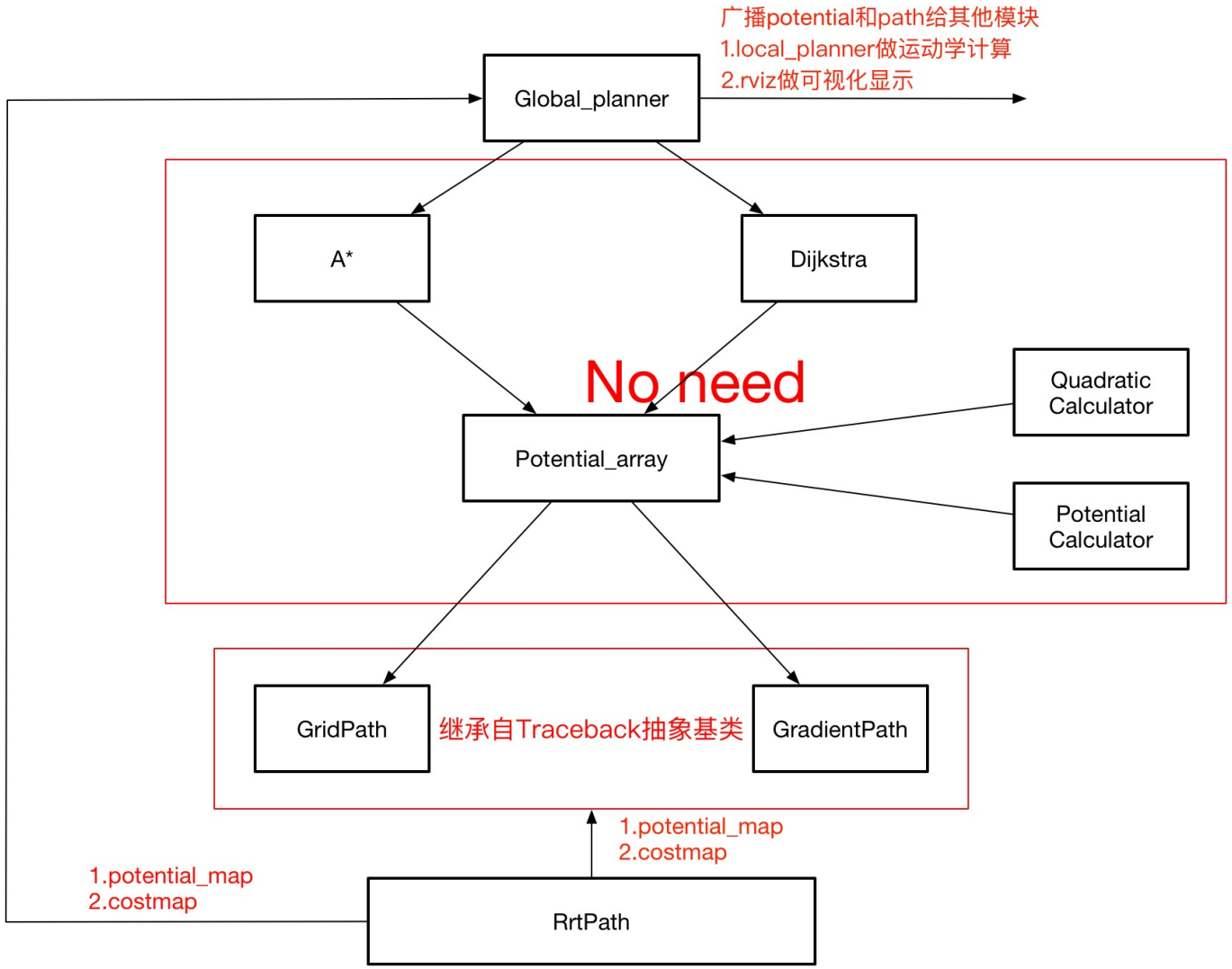
RRT算法简介

RRT(Rapid-Exploring Random Tree)

- 基本思想：
  - 在状态空间中，以运动规划初始状态 q_{init} 为根节点，建立搜索树
  - 循环以下步骤，完成树的扩张过程：
    - 在状态空间中，随机采样一个状态，用于引导搜索树的扩张，称为 q_{rand}；
    - 在现有的搜索树上查找与 q_{rand} 距离最近的节点 q_{near}，以 q_{near} 和 q_{rand} 构建新的输入u，以 q_{near} 作为当前状态x，根据系统状态方程 $\dot{x} = f(x,u)$，得到下一个状态即搜索树的扩张节点 q_{new}；

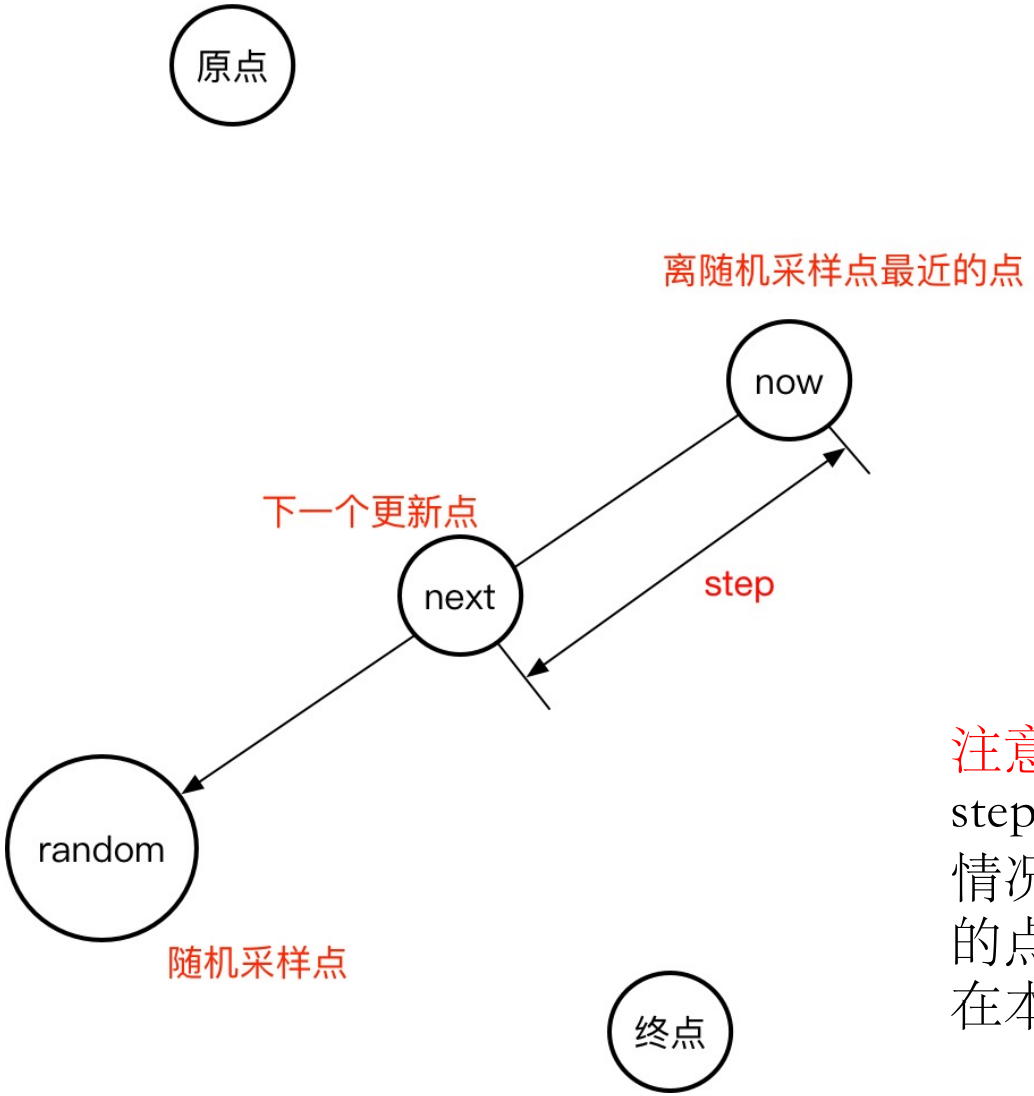


在ROS navigation包中嵌入RRT算法



重写Traceback抽象基类的getPath函数，将costmap也作为一个参数传进去，然后让我们自定义的RrtPath类继承Tracback抽象基类，自行实现getPath函数，这样在planner_core调用时，依靠c++的虚函数机制，可以根据yaml的内容动态决定使用的是哪一个算法。

基础RRT算法



基本步骤：

随机采样一个点，搜索出离这个点最近的点，计算出方向向量，下一个更新点就在往该方向一个步长的位置。如果计算出来的next点在障碍物内，则自动舍弃该点，进行下一轮迭代。

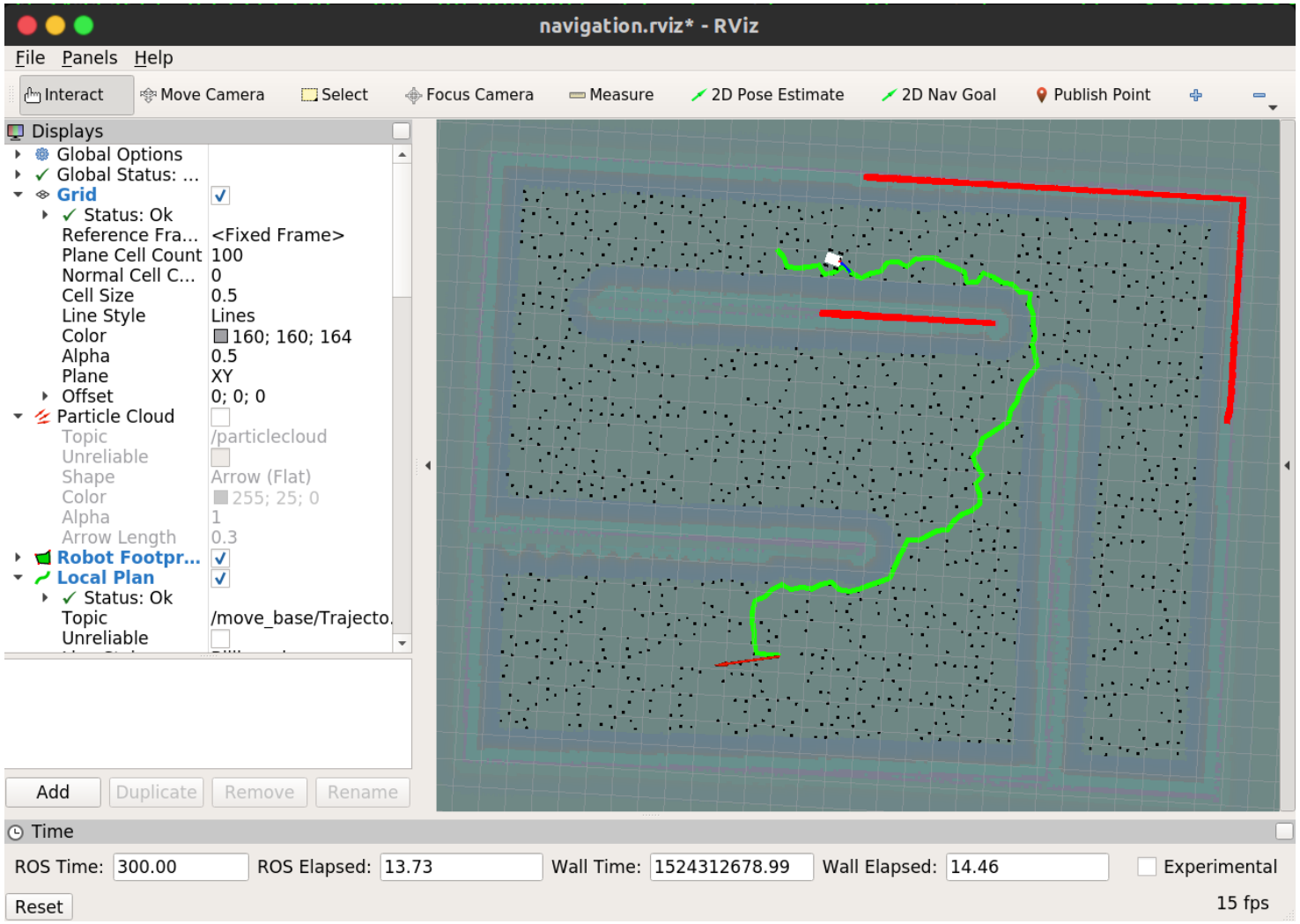
终止条件：

如果计算出来的点在终点范围一个step步长内，那么迭代结束，算法找到最优解并返回。

注意：

step是一个超参数，需要人为根据真实的物理情况进行设定（如果太小，那么需要采样更多的点，如果太大，则不能很好的规避障碍物），在本仿真环境中，step的默认值为4

基础RRT算法



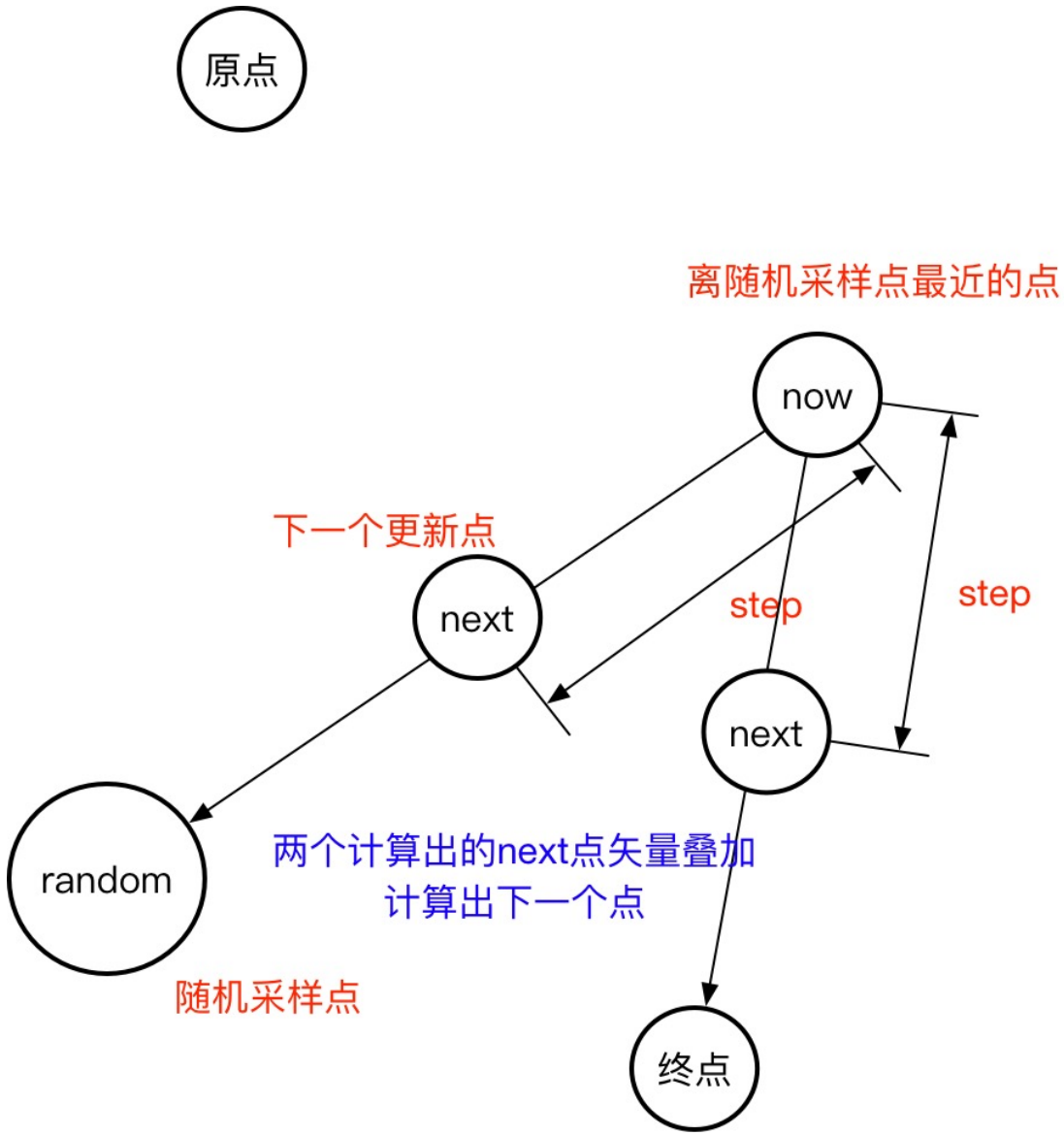
优点：

随机采点，可以应对有着很复杂障碍物的环境

缺点：

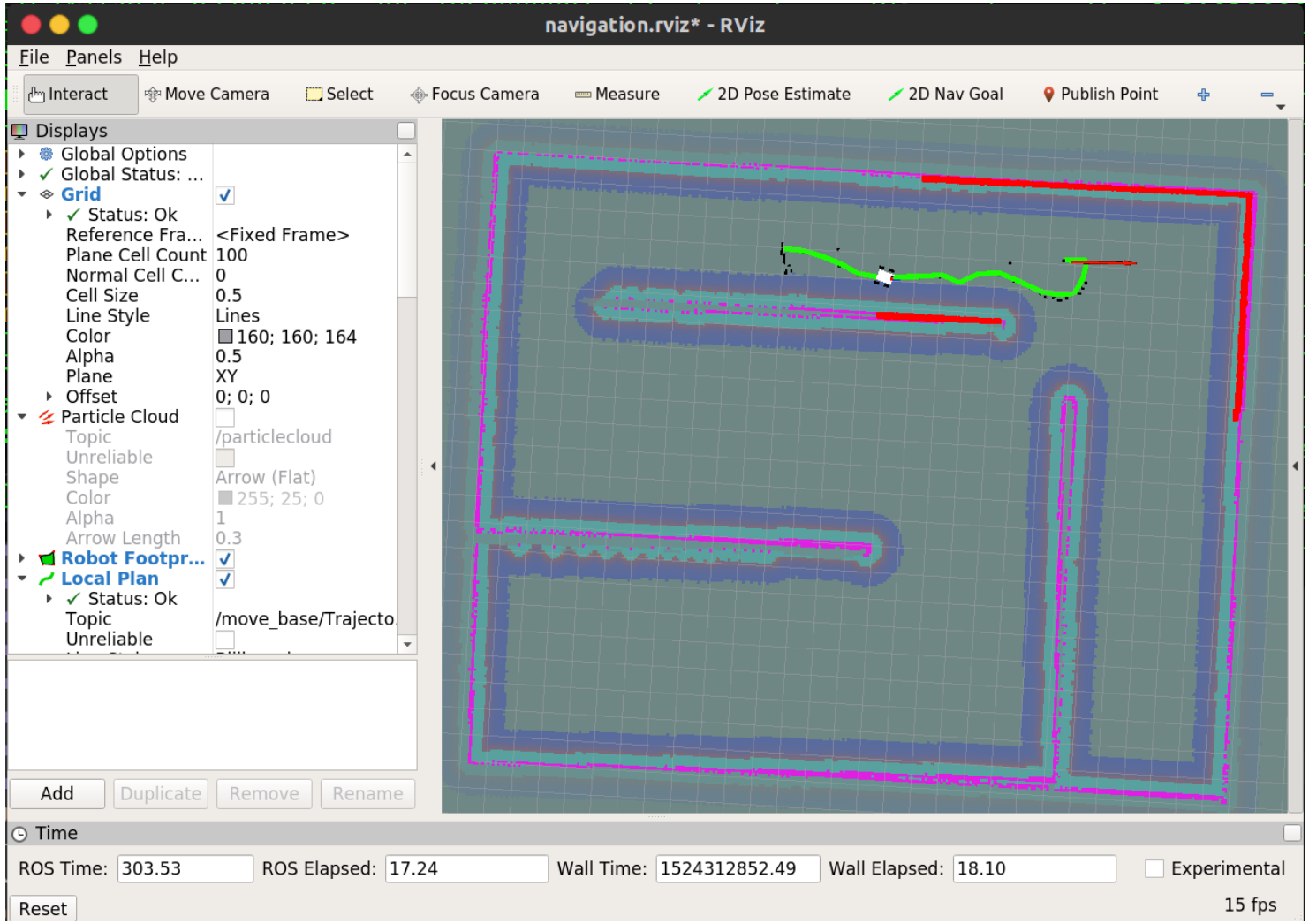
不管目标点在哪里，都是盲目的随机采样，搜索空间大，造成计算量的浪费

目标导向的RRT算法



在之前的基础上，计算下一个更新点的位置时，不单单是向随机采样点的位置移动一个步长，还加入了向终点一个步长的向量，两个向量矢量相加，计算出新的点。同样，如果计算出来的next点在障碍物内，则自动舍弃该点，进行下一轮迭代。

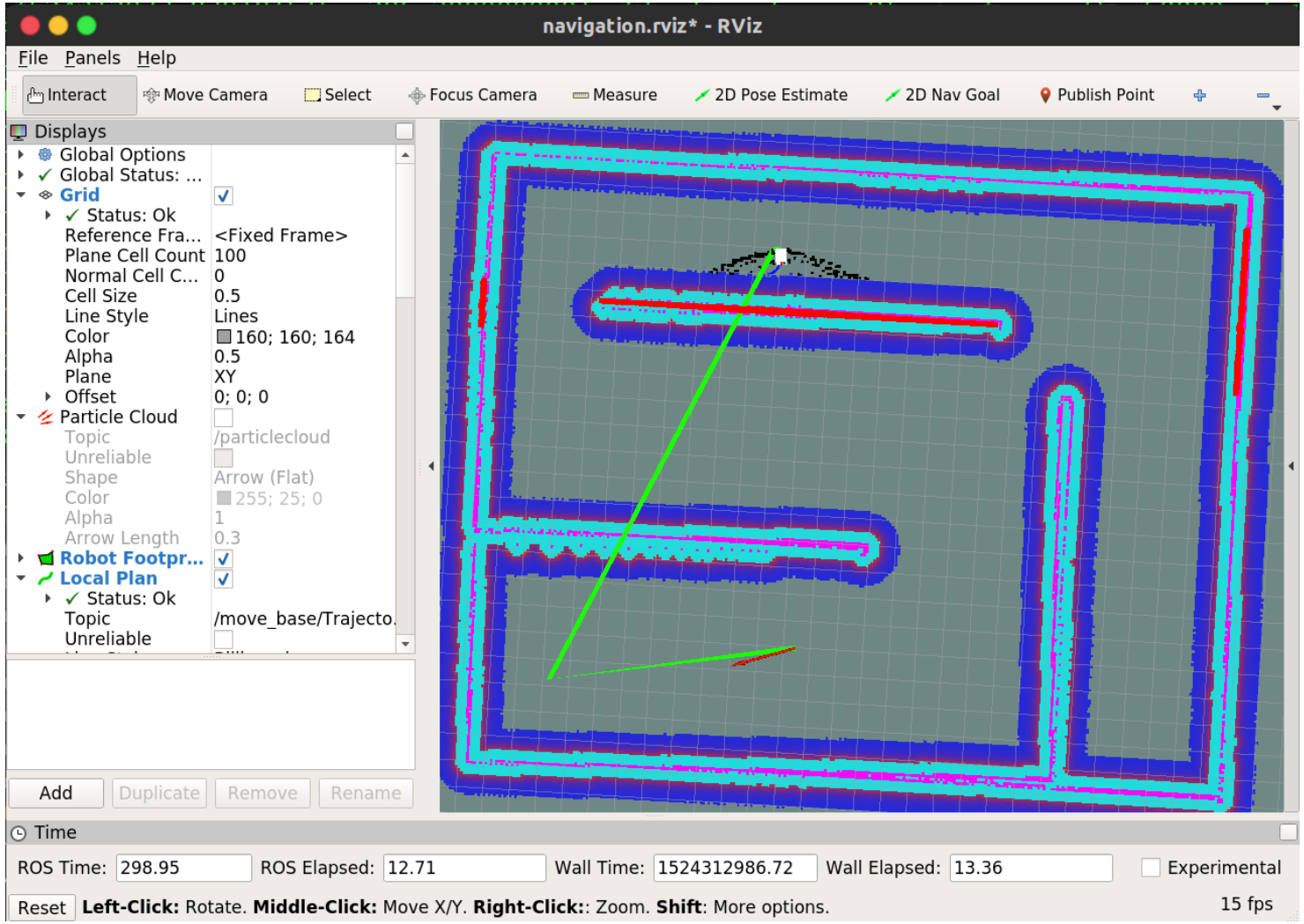
目标导向的RRT算法



优点：

加入目标导向后，搜索空间大大减小，在保证了随机性的同时，还兼顾了目标点的位置，使得采样点向目标点“靠拢”。

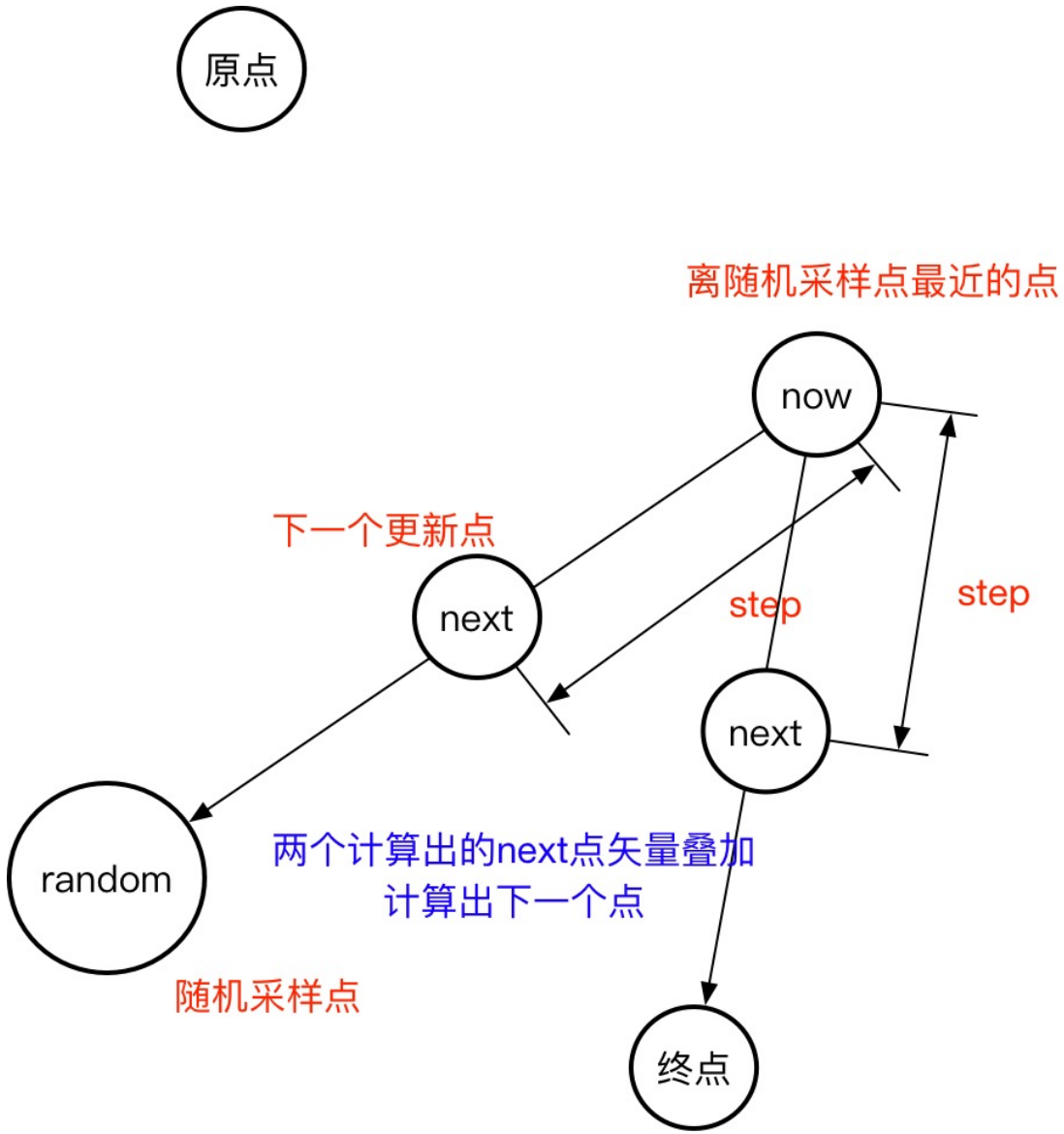
目标导向的RRT算法



缺点：

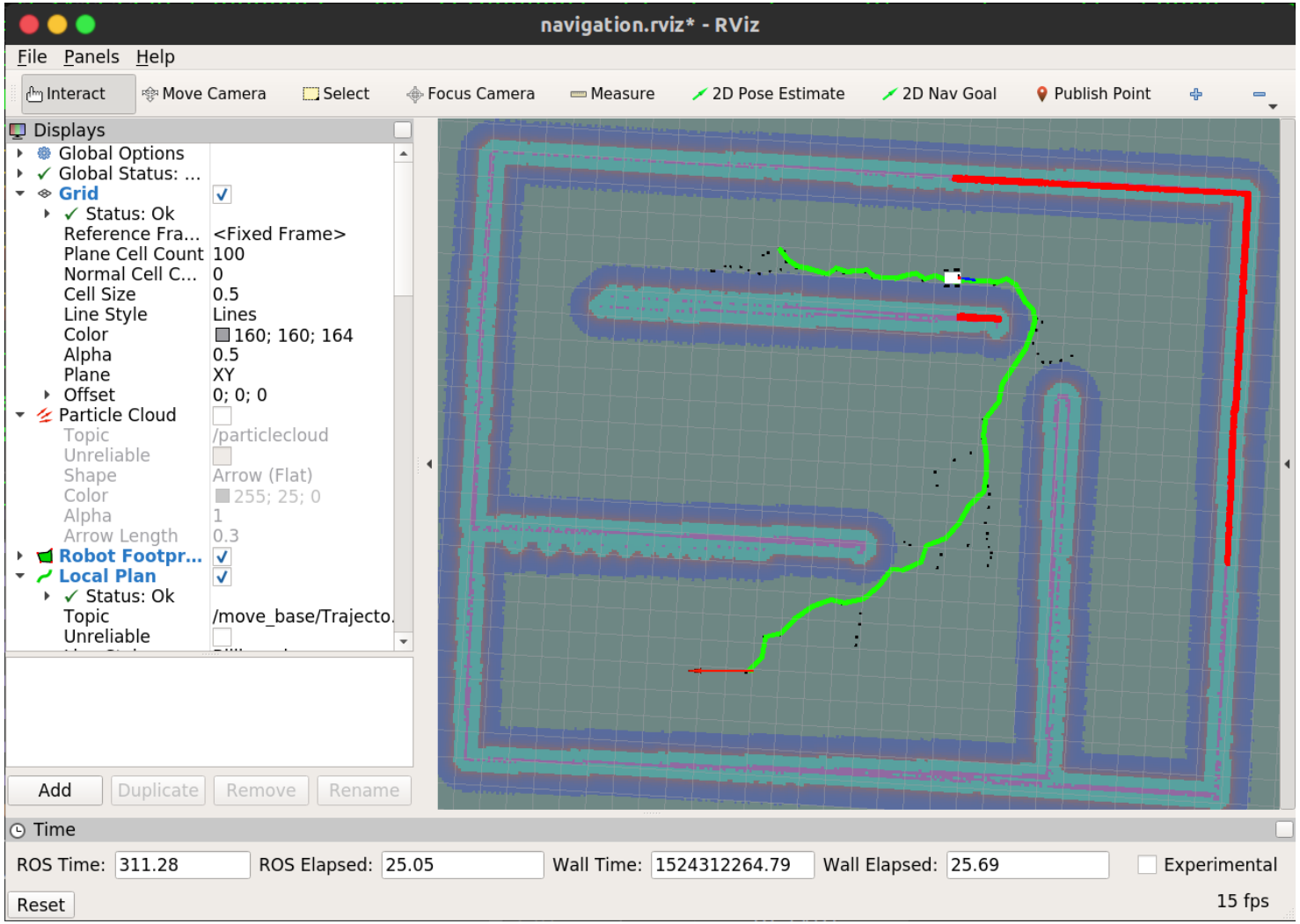
由于加入了无条件的目标导向，使得搜索不能很好的跨过障碍物，算法在障碍物较多的情况不能找到一个可行解（即使解存在）

自适应RRT算法



为了避免碰撞，先“尝试”引入目标导向计算出新的点，如果该点在障碍物内，那么舍弃该点，计算不带目标导向的next节点，如果计算出来的next点还在障碍物内，则自动舍弃该点，进行下一轮迭代。以此实现对障碍物的动态监测。

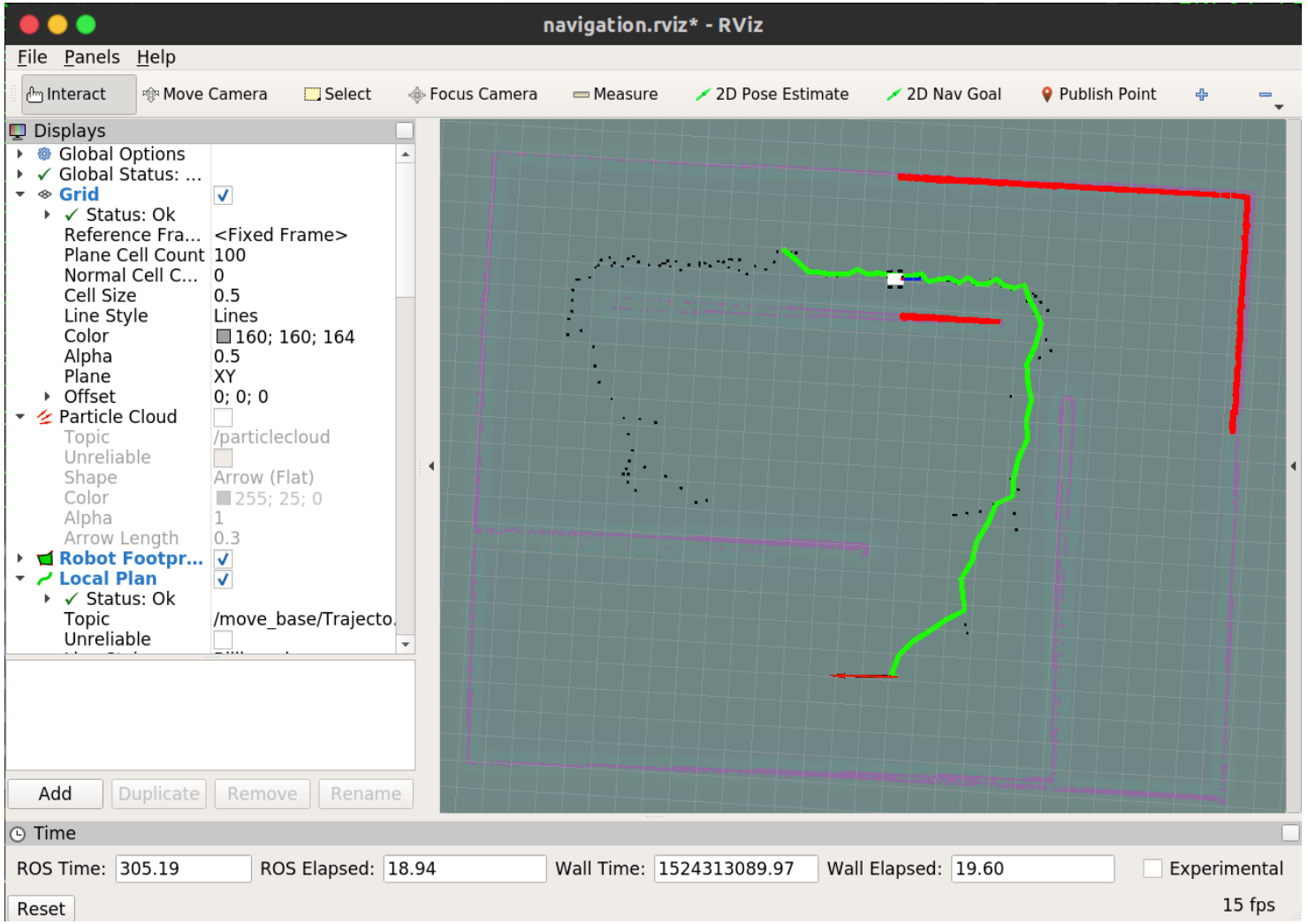
自适应RRT算法



优点：

加入自适应后，不仅搜索空间大大减小，而且算法会自动检测“碰撞”，变得更加智能，在遇到障碍物时，基本上是“贴着”障碍物行走，而在跨过障碍物时，由于目标导向的作用，算法可以迅速的靠近目标点

自适应RRT算法



效率衡量：

相比最原始的RRT随机采样算法，在本仿真环境下，进行多次计算，统计平均需要的采样次数，自适应RRT算法可以减少采样次数10倍以上。