导航规划作业实践报告

张嘉浩（3190103683） 朱姜宇轩（3190103052） 蒋颜丞（3190102563）

1 路径规划（RRT*）

1.1 原理

RRT* Algorithm

Input : M, x_{init}, x_{goal}

Result : A path Γ from x_{init} to x_{goal}

$\Gamma.init()$	以x_{init}为根节点，建立搜索树
for $i = 1$ *to* n *do*	
$x_{rand} \leftarrow Sample(M)$	在可行空间中随机采样
$x_{near} \leftarrow Near(x_{rand}, \Gamma)$	找到树中离采样点最近的树节点
$x_{near} \leftarrow Steer(x_{rand}, x_{near}, StepSize)$	根据机器人的执行能力生成新的节点和新的路径
if $CollisionFree(x_{near})$**then**	无碰撞检测，根据成本进行树结构优化
$X_{near} \leftarrow NearC(\Gamma, x_{near})$	选择多个邻节点
$X_{min} \leftarrow ChooseParent(X_{near}, x_{near}, x_{new})$	评估各邻节点作为父节点下的总路径长度
$\Gamma.addNodEdge(x_{min}, x_{new})$	根据总最短路径选择父节点
$\Gamma.rewire()$	优化邻节点路径

1.2 代码说明

1.2.1 搜索树节点（`Node(object)`）的定义

每一个节点包含了自身的坐标`(x,y)`、从根节点到该节点的成本`cost`、该节点的父节点`parent`和该节点后续的节点列表`list`。搜索树的结构是一个多叉树，可以根据从最后的节点`x_goal`回溯到初始节点`x_start`得到可行路径。

1.2.2 路径规划的总体框架`plan`

实现路径规划的函数`plan`总体结构与上文提到的RRT*算法基本一致，函数`Treeinit()`用于对搜索树以及障碍物进行初始化，函数`Sample()`返回地图中的随机点`rnd_node`，函数`Nearest()`将返回离采样点`rnd_node`最近的树节点`neareast_node`，通过`Steer()`得到机器人能够达到的新节点`new_node`。在对路径成本进行树结构优化的部分中，函数`NearC()`将返回一个包含了N个树节点`node`的列表`list[]`，`ChooseParent()`，`addNodeEdge()`，`rewire()`函数的功能即为更新搜索树。

1.3 调试过程与问题解决

1.3.1 问题1：树结构的定义，如何读取搜索树的节点的问题

搜索树结构为多叉树，所以在定义节点的时候定义了父节点 Node `parent`和子节点列表 List `next`，计划用广度优先搜索来遍历每一个节点。但是，几乎每一个函数中都需要用到可行的搜索节点，每一次都便利过于费时费力，于是又单独创建了一个字典（Python dict）`node_dict`，以及用于字典索引和生成KD-Tree所用的列表的`path_x`，`path_y`。通过这样的定义，在进行最近节点搜索的时候使用字典数据结构，在找到`x_goal`之后，通过对`x_goal`进行回溯找到路径。

1.3.2 问题2：如何找到最后的目标节点

RRT*算法中的新节点`new_node`由函数`Sample()`返回的随机节点`rnd_node`并加以考虑机器人的运动能力`Step`得到，那么如何才可以让`new_node`与`goal_node`相匹配？ 我一开始的想法是增加路径规划中的循环次数，认为在多次随机取点之后，总能得到新的节点`new_node`与目标节点`goal_node`相等，那么这种方法当然是失败了。后来我通过研究PRM算法的例程得到，在每次随机取点之后，选取临近的N个节点，如果其中有一个节点与目标节点的距离小于步长，则将新的节点`new_node`设置为目标节点`goal_node`，解决了找不到目标节点的问题。

1.3.3 问题3：生成树的优化，pathsmoothing

RRT*算法是渐进优化的，随着迭代次数的增加，得出的路径是越来越优化的，而且永远不可能在有限的时间中得出最优的路径。因此换句话说，要想得出相对满意的优化路径，是需要一定的运算时间的。而在题目设定的情况下，机器人步长较大，可以优化的空间相对较少，于是我们在找到了目标节点之后就停止搜索了。虽然 `rewire()` 函数已经针对RRT随机小步拓展导致路径曲折，成本高的问题进行了优化，但是最后生成的路径仍有可以优化的部分，我们就仿照PRM算法中的路径优化方法，利用 `PathSmooth` 函数对生成的路径进行了优化，该方法的主要思想为连接节点与其间隔的节点，如果两者之间没有障碍物，则可将中间的节点去除，两个节点直接连接。

1.4 算法分析与比较

RRT算法是一个相对高效率，同时可以较好的处理带有非完整约束的路径规划问题的算法，并且在很多方面有很大的优势，但是RRT算法并不能保证所得出的可行路径是相对优化的。RRT*算法的主要特征是能快速的找出初始路径，之后随着采样点的增加，不断地进行优化直到找到目标点或者达到设定的最大循环次数。RRT*算法是渐进优化的，但不可否认的是，RRT*算法计算出的路径的代价相比RRT来说减小了不少。RRT* with PathSmoothing算法，相较于RRT*，在找到最后的节点之后就立即停止循环，通过回溯的方法减少路线的曲折度，在该题目下有较好的表现。

2 轨迹规划（反馈控制法）

2.1 原理

2.1.1 反馈控制法

设计线性控制律$v=k_\rho\rho, w=k_\alpha\alpha+k_\beta\beta$，使得$\begin{bmatrix}\rho'\\ \alpha'\\ \beta'\end{bmatrix}=\begin{bmatrix}-k_\rho\rho cos\alpha\\ k_\rho sin\alpha-k_\alpha\alpha-k_\beta\beta\\ k_\rho sin\alpha\end{bmatrix}$(奇异性消除)

2.1.2 反馈控制优化法

定义$k(\rho,\alpha,\beta)=\frac{1}{\rho}[k_2(\alpha-arctan(-k_1\beta))+(1+\frac{k_1}{1+(k_1\beta)^2})sin\alpha]$，选择速度为$v=\frac{v_{max}}{1+\mu|k(\rho,\alpha,\beta)|^\lambda},\mu>0,\lambda>1$，则角速度$w=vk(\rho,\alpha,\beta)$。

2.2 调试过程与问题解决

我们先是完成了反馈控制的基础算法，通过机器人当前位置与目标位置的比较，快速调整机器人速度与角速度，较好地实现了静态障碍物环境下的沿路径行驶，但是在到达中间目标点时会有冲过头又调头回到目标点，但其实并不需要完全到达中间目标点，因此我们调宽了到达中间节点的判定，使得机器人在行进时不会在中间节点徘徊太久。同时我们还发现在到达终点时，机器人的速度并不能很好地减小到零，以致于经过终点后会停留在离目标点较远处。因此我们在最后一段路径中添加了一个目标节点，使得机器人在靠近最终目标时的速度不会太快以无法停留在目标点处。

但是反馈控制法下实现的行进过程并不连续，在转换目标节点时会停顿。因此我们尝试了反馈控制优化法。在反馈控制优化算法的调试中，我们通过调节 k_1,k_2的大小，使得控制对象行进路线与规划路径贴合，在此基础上修改μ、λ的值，希望能使机器人以较快的速度按照规划路径行驶。但是我们一但加快机器人的行进速度，机器人在沿路径行驶时会产生较大的振荡。这四个参数的调节相互关联，而且每次观测效果时产生的路径都不相同，要想得到较好的结果比较依赖工程经验，经过多次尝试效果并没有未优化的好。

因此我们最后采用了未优化的反馈控制法，并使用if-else语句来保证(v,w)的可达性。

2.3 算法分析与比较

图形搜索法可以实现位置的连续平滑，并充分考虑了机器人的运动学约束和时间最优要求。但是构建单词和搜素最佳单词比较难做到，而且运动基元方式实际是对状态空间和/或抑制空间进行了离散化，直接导致空间分辨率低，能达到的边界状态只是预定义单词能够达到的状态。参数优化法能确保满足空间位置约束。在实际应用中通常预先构造初始猜测查找表，或者构建神经网络，这种方法在本情景下有些过于复杂，故不采用。反馈控制法相对简单、容易实现并且效果不错，在反馈控制中平移速度、转向速度相互独立，但机器人总能力是有限的且两个速度实际上是存在关联的。只是在这题下，通过反馈控制便能达到较好的效果。

3 避障规划（动态窗格法）

3.1 原理

动态窗格法的基本思想是在速度空间中搜索适当的平移速度和旋转速度指令(v,w)，算法的伪代码如下所示：

```
allowable_v = generateWindow(robotV, robotModel)
allowable_w  = generateWindow(robotW, robotModel) #首先在V_m∩V_d的范围内采样速度：
for each v in allowable_v:
    for each w in allowable_w:
       dist = find_dist(v,w,laserscan,robotModel)
       if (v in v_canStop and w in w_canStop) #如果能够及时刹车，该对速度可接受
          CalculateEvaluation(v,w) #利用评估函数对其评价
          FindBest(v,w) #找到最优的速度组
```

3.2 调试过程与问题解决

在计算评估函数时，理论上应当对各项进行归一化再取加权平均，但是在实际程序中，由于知道v和w的上下界，因此对$heading(v,w)$项和$velocity(v,w)$项进行MinMax归一化是容易的，但是对$dist(v,w)$项而言，由于并不知道其最大值和最小值，如要进行MinMax归一化，需要先遍历一遍速度空间得到其最大值和最小值，再重新遍历计算评估函数的值。由于速度空间的规模并不小，两次遍历导致计算所需要的时间几近翻倍，这就导致在实际运行中，机器人进行避障的实时性不是很好。为了解决这个问题，我们采用了比较简单粗暴的方式，即不对$dist(v,w)$项进行MinMax归一化，而是通过添加print语句，在运行过程中观察$dist(v,w)$项的大概数量级，根据这个数量级给$dist(v,w)$项乘以一个近似归一化系数，使其值大概保持在[0,1]之间。经过测试，这样的办法不仅大大提高了实时性，且避障表现十分不错。

但是，在动态窗格法介入后，机器人的轨迹非常容易出现扭动，即v,w的波动过大，为此，我们在原有的速度空间上加以限制，使得v的上界为v_{max}和v_f的加权平均，同时提高$heading(v,w)$项的权重，使得避障规划出的(v,w)更接近于反馈控制的结果。在解决上述问题后，我们又发现当机器人速度为负且绝对值较大时，机器人容易出现倒车失控的情况，故进一步限制速度空间中v的下界（限制为$max(v_{min},-800)$）。

此外，在反馈控制法和动态窗格法的共同作用下，在某些情况下（如机器人朝向目标点，但机器人和目标点的连线上有障碍物）容易出现震荡和死区，停在某一区域不动，往往需要很长时间才能跳出震荡或死区，甚至“卡死”在该处。我们的解决办法是给予避障规划更多的自主权，即在避障规划接手机器人的控制后，只有当机器人成功绕开障碍物后，才把控制权重新交回反馈控制法。

同时，动态窗格法舍弃了非前进路线上的障碍物信息，这就导致在hard模式下，移动的障碍物有可能从后方碰撞机器人，针对这一情况，我们加入了一个朴素的策略：当机器人后方一定距离内存在障碍物时，机器人就适当提速，以规避“追尾”的情况。

3.3 算法分析与比较

动态窗格法直接在(v,w)空间规划，适合于接续反馈控制法并发出相应的指令，而其他算法多在(x,y)空间规划或需要利用传感器信息，并不适合本次作业。而与单纯的动态窗格法控制机器人相比，我们的动态窗格法只会在反馈控制法得到的(v,w)会发生碰撞的情况下介入，而在反馈控制法得到的(v,w)不会发生碰撞的情况下不起作用，因此，我们的机器人运动在目标指向性、速度连续性和路径平滑性上都得到了一定的改善。遗憾的是，即便经过了种种改进和微调，我们的动态窗格法仍然难以适应各种情况（尤其是在$hard$模式下），例如，在障碍物“夹击”的情况下，机器人的反应速度不够快，可能会发生少许碰撞。

4 实验结果

simple模式用时1min20s，全程无碰；hard模式用时1min57s，仅在障碍物“夹击”的情况下有少数碰撞。

5 分工贡献

张嘉浩：33.33%，RRT*部分；朱姜宇轩：33.33%，反馈控制法部分；蒋颜丞：33.33%，动态窗格法部分。