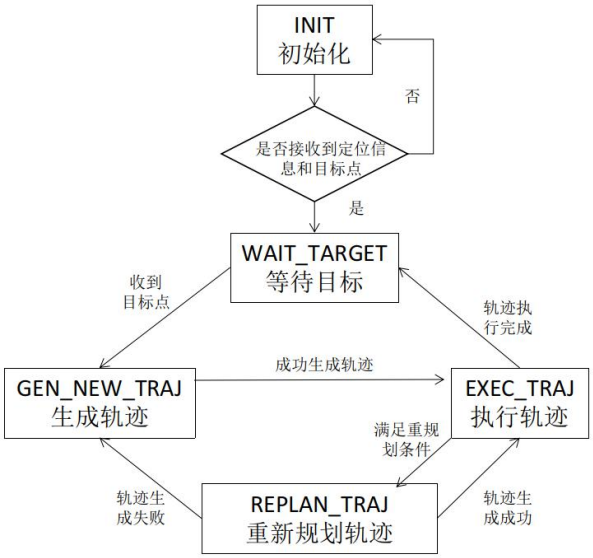
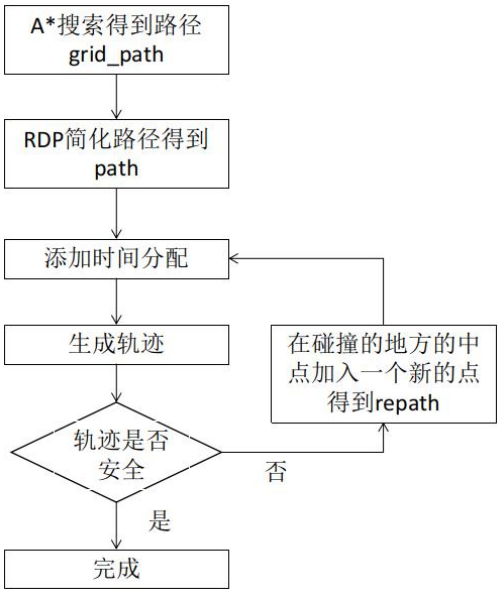
作业要求完成的功能集中在 trajectory_generator_node 中，首先需要厘清程序的运行流程，对于程序的状态转移如下图所示：



作业需要完成的就是 trajGeneration()函数，该函数分为前端的路径搜索加后端的轨迹优化，函数的流程如下所示：



按照作业的提示依次完成各部分代码，其中 path planning 部分采用了 A*、JPS、RRT*三种算法，三种算法相互对比可以发现 A*容易贴近障碍物走，对后面的优化不是很友好，JPS 面对作业这种障碍物不是特别多的情况，表现有时候不是很稳定，OMPL 的 RRT*表现还算可以，前端采用 RRT*比另外两个算法要好一点。

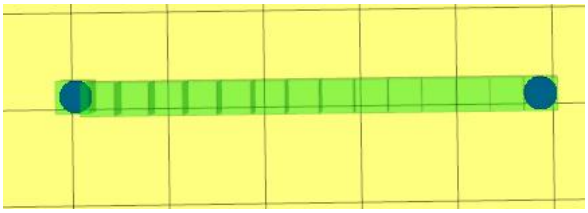
simplify the path 采用 RDP 算法，根据给的伪代码翻译成 C++难度不是很大。

trajectory optimization 部分采用上课的 minimum snap trajectory generation。

safe checking 采用模拟的方法，取一个小的时间间隔，根据多项式参数算出位置，检查各个位置是否有碰撞。

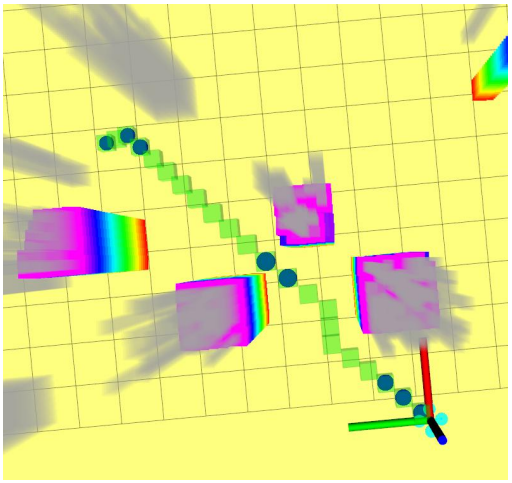
trajectory reoptimization 采用论文中的方法，如果第 i 段轨迹发生了碰撞，那么就在第 i 段 path 上插入一个中点，轨迹会往远离障碍物的地方拉，不断重复上面的过程直到没有碰撞为止。

simplified path 的作用直观的就是提取整条 path 中关键 waypoints，一个最简单的例子如下：



原本 A*的 path 是一条直线，经过 RDP 算法提取关键点过后就成为了两个点，这些关键的 waypoints 就可以送入后端的轨迹生成中。

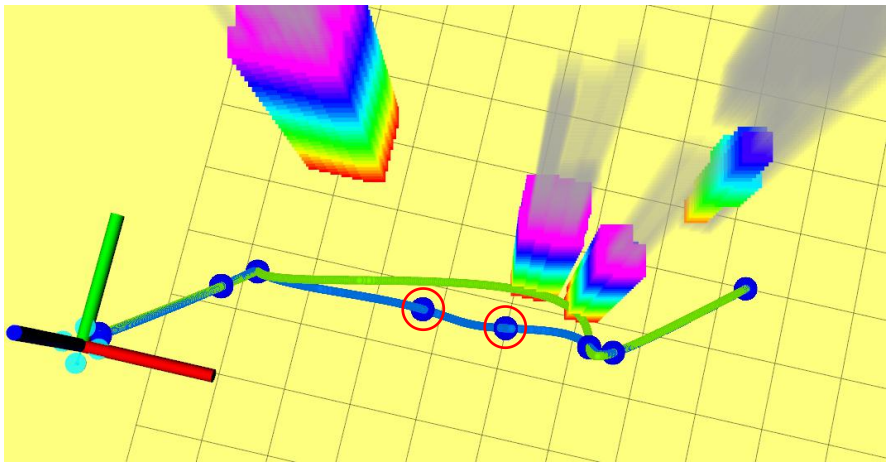
一个 RDP 算法的完整效果如下：



经过 RDP 算法 A*的路径被简化成了几个关键点。

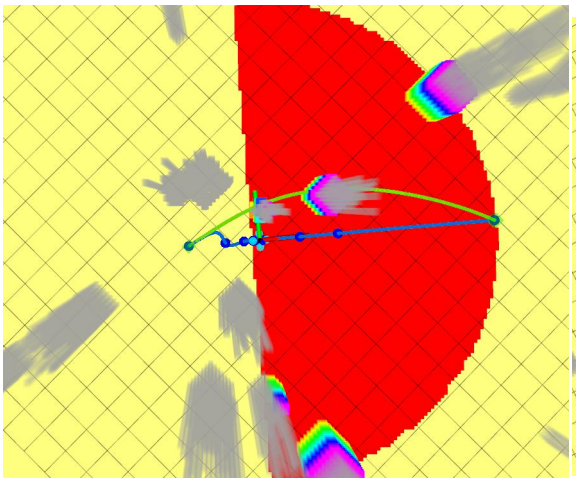
采用 minimum snap trajectory generation 的轨迹生成没有考虑障碍物碰撞的问题，所以生成的轨迹就有可能发生碰撞，这个时候就需要进行重规划生成安全的轨迹，策略采用所给论文中的方法，如果第 i 段发生了碰撞就在第 i 段 path 中插入该段的中点，然后在生成新的轨迹，不断重复这个步骤。因为 minimum snap trajectory generation 要求轨迹必须经过 waypoints，这就相当于把轨迹往往远离外面拉，因为两个 path 之间的直线是没有碰撞的，所以只要重复上述步骤足够多次，一定能够保证轨迹安全。

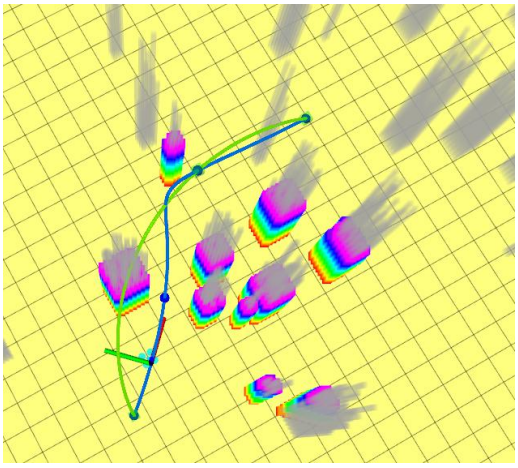
实验结果如下图所示：



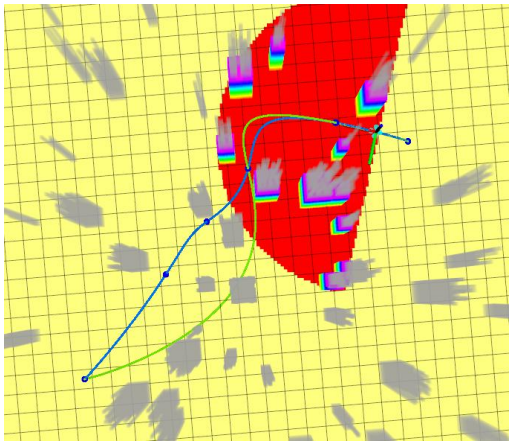
绿色的轨迹是原始的轨迹，这个时候发生了碰撞，这个时候插入了红色圈圈出的两个点过后，新生成的蓝色轨迹被约束远离障碍物了，完成了一次轨迹安全的重规划。

经过多组测试，算法都能够正常运行，实时进行规划，无碰撞的到达目标点。

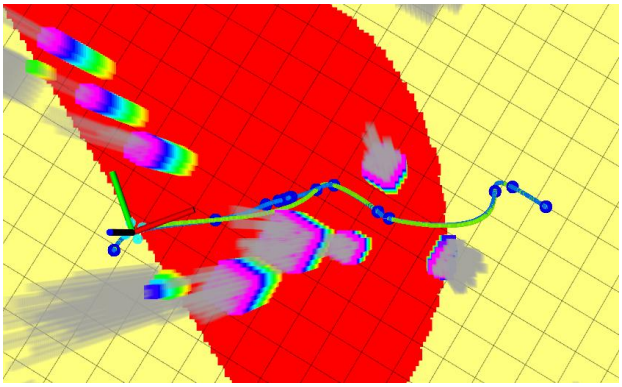




面对长距离的规划任务算法也能够胜任，在实际的测试当中可以发现前端路径规划和后端轨迹优化的关系是非常紧密的，最终的目的是要生成轨迹，前端如果给出的 path 不好，后端就会出现很奇怪的轨迹。所以选择一个好的前端是非常有必要的。



这种方法的弊端面对障碍物比较密集的场景就体现出来了，如果轨迹非常容易发生碰撞，就会插入很多的点，这样不但计算量大而且也飞的比较的慢。



综上，本次作业系统的完成了整个运动规划的功能，从前端的路径规划到后端的轨迹优化，规划出的轨迹也能够正确的控制飞行完成飞行的任务。对其中的算法进行了分析。整个运动规划相对来说还是一个比较大的系统，对于前端改进的方向主要集中在尽量生成一些满足后端需求的 path，减少后端的压力，后端改进就是如何在发生碰撞的时候能够进行高效的轨迹重规划。前端后端是一个整体，相辅相成。