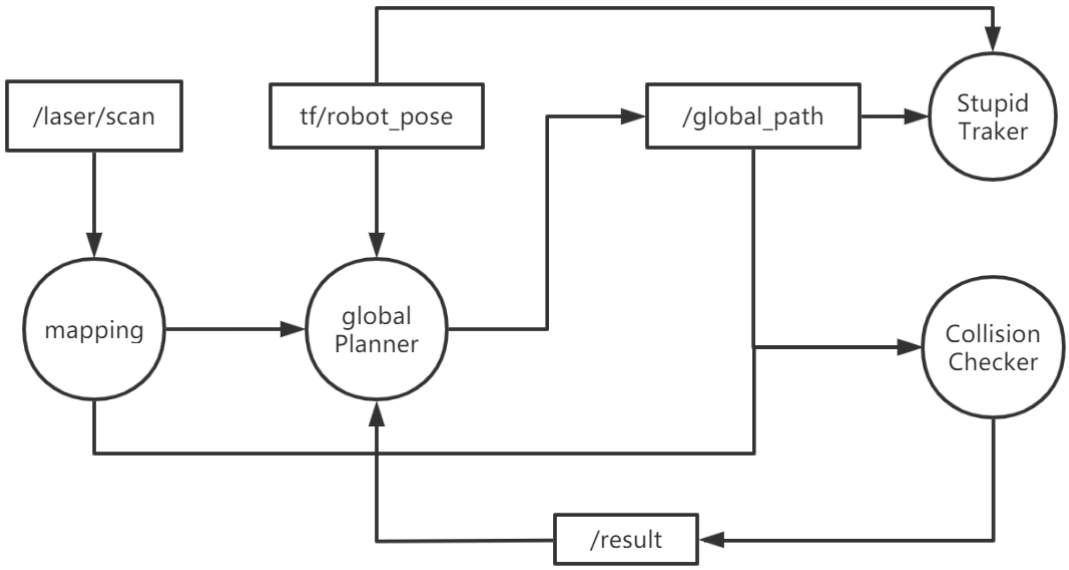
重规划

Michael Gao

Baseline: Collision_Checker + Stupid_Traker



1.global planner 的重规划

重规划执行情况：

- 接收到新的goal时
- Collision Checker节点的客户端请求时

将global planner的重规划函数设置为Service的回调函数，global planner节点接收到Collision Checker节点的客户端请求后，执行回调函数：

```
rospy.Service('/course_agv/global_plan',Plan,self.replan)
```

```
def replan(self,req):
    # 更新当前位姿
    self.updateGlobalPose()
    # A*
    self.plan_rx,self.plan_ry = self.Astar()
    self.publishPath()
    res = True
    return PlanResponse(res)
```

重规划过程：

重规划A*在mapping节点返回的实时栅格占用地图`/grid_map_mine`中进行规划

接收mapping结果，转换成numpy矩阵存于类变量中：

```python
def mapCallback(self,msg):
    if(self.first_map):
        self.map_height = msg.info.height
        self.map_width = msg.info.width
        self.map_reso = msg.info.resolution
        self.first_map = False
    self.map = np.array(msg.data).reshape((-1,msg.info.height)).transpose()
    pass
```

对`self.map`使用A*算法，代码结构如下：

- Maintain a priority queue to store all the nodes to be expanded
- The heuristic function h(n) for all nodes are pre-defined
- The priority queue is initialized with the start state X_s
- Assign $g(X_s)=0$, and g(n)=infinite for all other nodes in the graph
- Loop
    - If the queue is empty, return FALSE; break;
    - Remove the node “n” with the lowest f(n)=g(n)+h(n) from the priority queue
    - Mark node “n” as expanded
    - If the node “n” is the goal state, return TRUE; break;
    - For all unexpanded neighbors “m” of node “n”
        - If g(m) = infinite
            - Push node “m” into the queue
        - If $g(m) > g(n) + C_{nm}$
            - $g(m)= g(n) + C_{nm}$
    - end
- End Loop

改进A*中的路径平滑算法，依次对3个连续坐标点操作，若位于边缘的2个点连线中间点没有与当前时刻map发生碰撞则将中间点赋值为中间点坐标。

2.Collision Checker

与初始版本不同，Collision Checker 接收2个topic做碰撞检测：

- path：`/course_agv/global_path`
- mapping的结果：`/grid_map_mine`

```python
# ros topic
self.path_sub =
rospy.Subscriber('/course_agv/global_path',Path,self.pathCallback)
self.map_sub = rospy.Subscriber('/grid_map_mine', OccupancyGrid,
self.mapCallback)
```

每次接收到path后更新类变量中的path：

```
def pathCallback(self,msg):
    self.lock.acquire()
    self.path = self.pathToNumpy(msg)
    self.lock.release()
```

每次接收到map后检查path是否经过了map中的障碍物，如果发生碰撞就像replan Service发送请求：

```
def mapCallback(self,msg):
    self.map = np.array(msg.data).reshape((-1,msg.info.height)).transpose()
    if self.collision_check():
        self.publish_collision()
        try:
            resp = self.replan_client.call()
            rospy.logwarn("Service call res: %s"%resp)
        except rospy.ServiceException, e:
            rospy.logwarn("Service call failed: %s"%e)
```

碰撞检测主函数：

由于路径点足够稠密，直接用路径点坐标查询栅格判断是否有障碍物

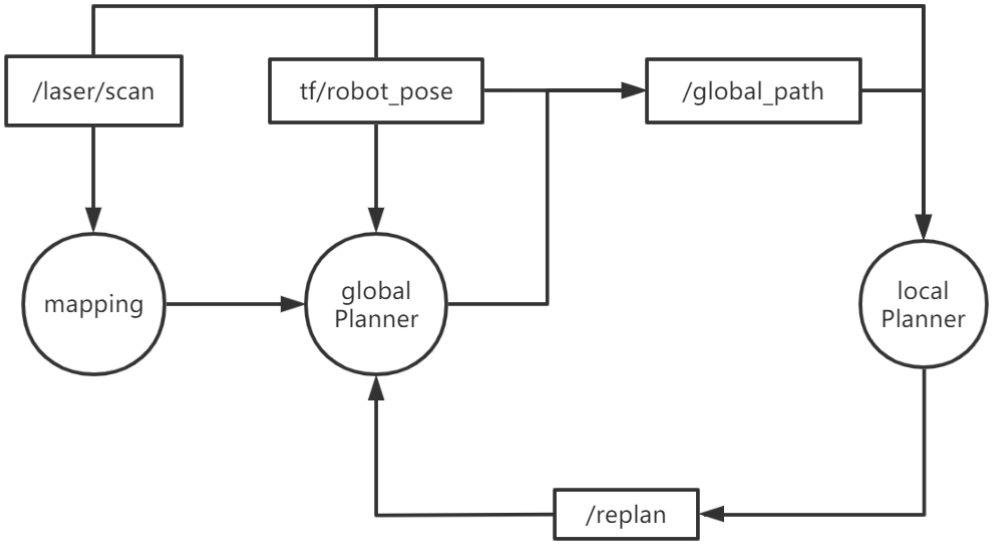
```
def collision_check(self):
    self.lock.acquire()
    res = False
    if self.path.shape[1] == 0:
        self.lock.release()
        return res

    for i in range(self.path.shape[1]):
        xinx = self.global2inx(self.path[0,i])
        yinx = self.global2inx(self.path[1,i])
        for m in range(xinx-2,xinx+3):
            for n in range(yinx-2,yinx+3):
                if m < 0 or n < 0 or m >= self.map_height or n >=
self.map_width:
                    continue
                if self.map[m,n] > 50:
                    res = True
                    break

    self.lock.release()
    return res
```

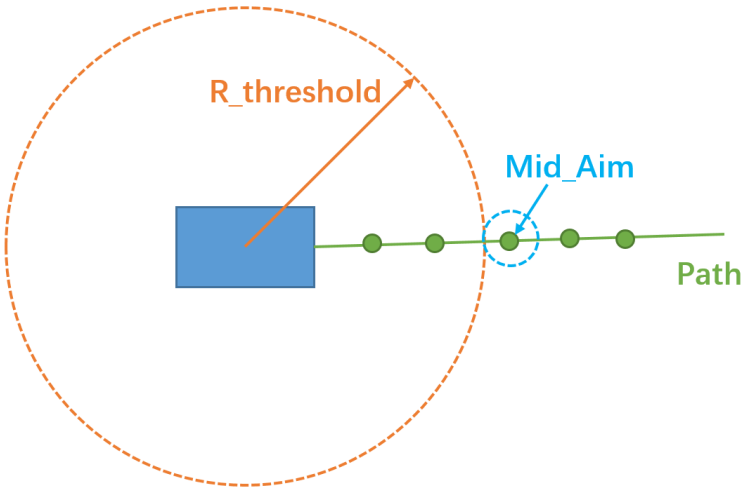
注意此处需要加互斥锁，否则会出现check中途path更新导致访问越界问题。

Task+：DWA + Replan



1.DWA Replan规则

选择当前path上距离小车距离在阈值范围之外的第一个点作为中间目标点：




```
for i in range(self.goal_inx, len(self.path.poses)):
    p = self.path.poses[ind].pose.position
    dis = math.hypot(p.x-self.x,p.y-self.y)
    if dis > self.threshold:
        self.goal_inx = ind
        break
if dis < self.threshold:
    self.goal_inx = len(self.path.poses)-1
goal = self.path.poses[self.goal_inx]
```

记录小车在更新下一个`Mid_Aim`之前DWA被调用的次数，如果调用DWA的次数超出最大阈值而小车仍然未到达下一个目标点，则此时发出replan请求

```
if self.goal_inx - self.goal_index_last > 0:
    self.dwa_cnt = 0
    self.goal_index_last = self.goal_inx

if self.dwa_cnt > self.dwa_maxtime:
    self.dwa_cnt = 0
    try:
        self.replan_client.call()
        break
    except rospy.ServiceException, e:
        break
```

2.DWA速度连续性

为了防止每次replan `vx/vw`重置为0，取消在`initPlanning`函数中的初始化：

```
if self.firstv:
    self.vx = 0.0
    self.vw = 0.0
    self.firstv = False
```

实验结果

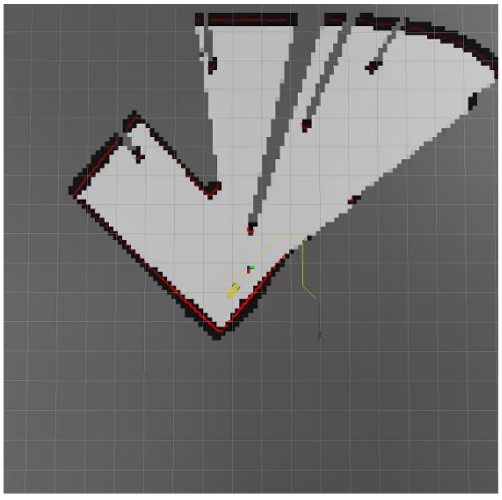
运行方法：

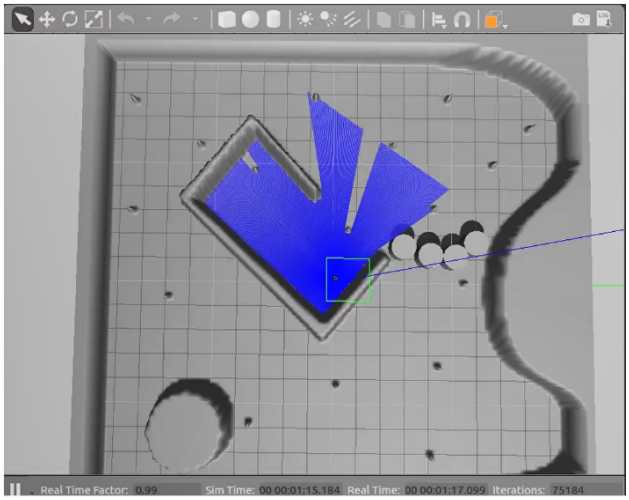
```
roslaunch course_agv_nav replan.launch
roslaunch course_agv_slam_task mapping.launch
```

1.Baseline: Collision_Checker + Stupid_Traker

视频见`/video/baseline.mp4`

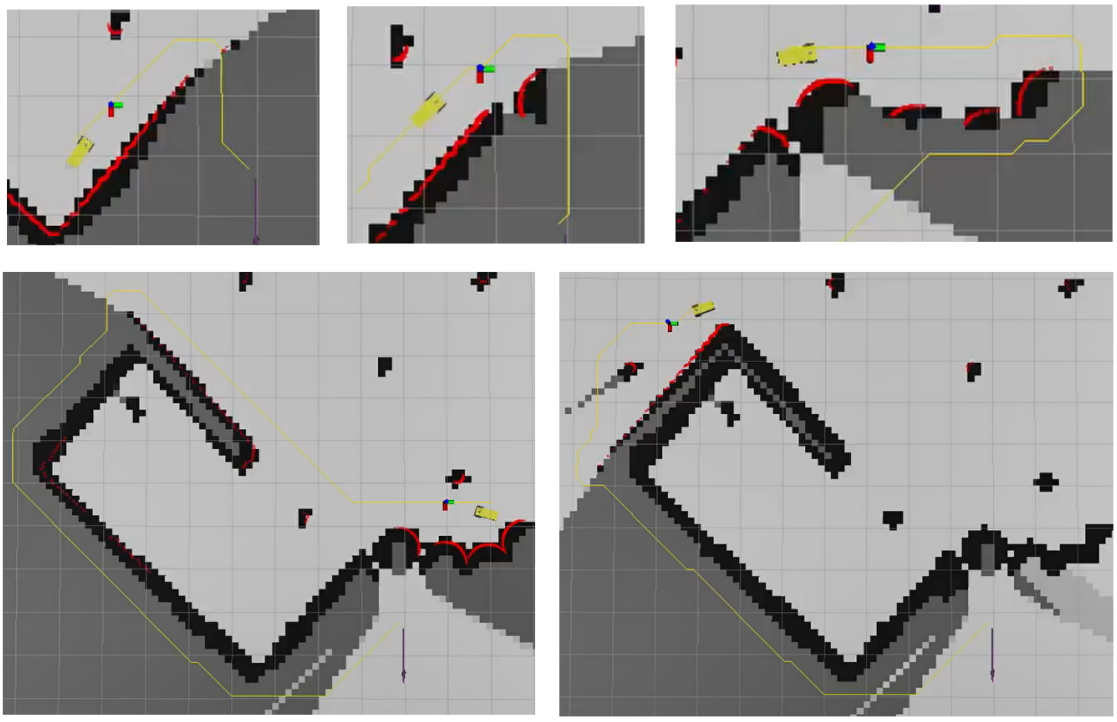
在地图中摆放如下障碍物，设置目标点如下：





一共进行了10次以上replan，同时由于小车的抖动，有时会出现一小格的障碍物噪声，也会造成反复重规划，直接用Collision Checker作为重规划准则会导致大量调用A*，完全忽略了小车的局部避障能力。

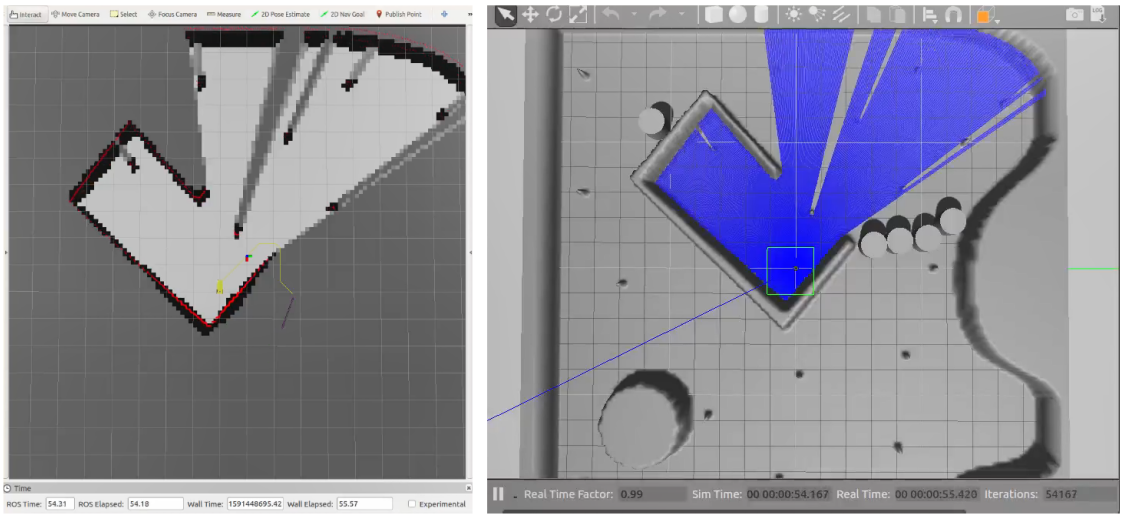
下图展示了其中几次关键replan：



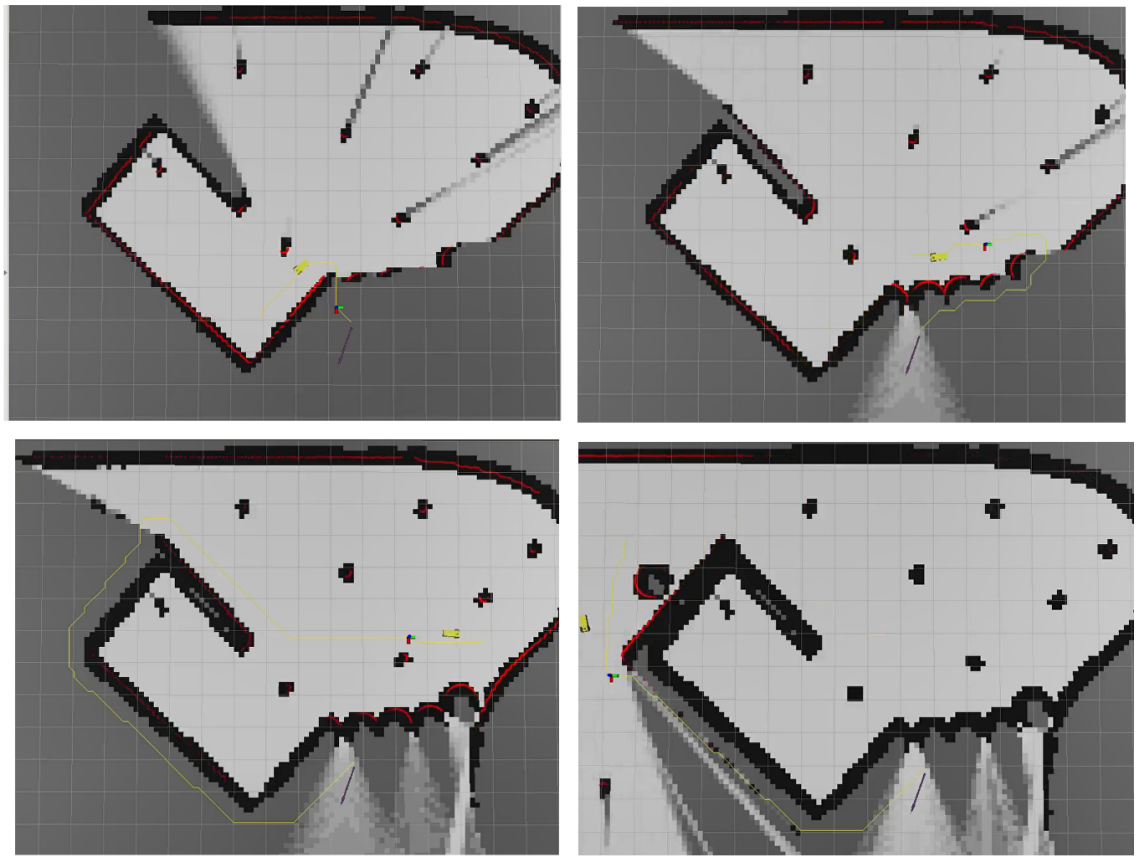
2.Task+：DWA + Replan

视频见`/video/task+.mp4`

在地图中摆放如下障碍物，设置目标点如下：



整体效果较好，能使小车发挥出局部规划能力，共进行了4次路径replan：



与纯DWA的对比：

性能对比：

与纯DWA相比，代码执行区别是Task+加入了Replan，每次Replan Service客户端请求时耗时大概为$0.005s/m$，下图为统计时截图：

```
('dwa_cnt:', 49)
('dwa_cnt:', 50)
('dwa_cnt:', 51)
get request for replan!!!!!!!!
Search arrive goal!
('replan, Time: ', 0.019999999999999574)
exit planning thread!!
running planning thread!!
('dwa_cnt:', 1)
```

障碍物复杂程度对比：

对于一定路径上的单个小半径障碍物DWA局部规划能够起作用，但对于大范围障碍物或根本失去可行路径的情况纯DWA失效。