

# 浙江大学

## 本科实验报告

课程名称：轮式机器人技术与强化实践

姓 名：肖瑞轩

学 院：计算机科学与技术学院

专 业：混合班

学 号：3180103127

指导教师：王越、黄哲远

2020 年 4 月 3 日

# 轮式机器人实验1-4 实验报告

姓名：肖瑞轩

学号：3180103127

## 一、实验目的：

**1.1实验一：** 安装ros环境,阅读tutorials并熟悉ROS Filesystem、ROS Package、ROS Nodes & Topic、ROS Services and Parameters、rqt\_console and roslaunch等基本操作

**1.2实验二：** 使用urdf在gazebo中构建双轮差动机器人，使其正确显示并通过ros node可以控制轮子速度

**1.3实验三：** a.在gazebo中载入地图，在urdf中添加sensor，可以读取laser以及imu数据

b.添加tf，使rviz可以显示完整的sensor、map以及机器人数据

c.rosbag的使用

**1.4实验四：** a.通过call map\_server的service来获取全局地图信息

b.通过tf获取当前坐标

c.通过订阅消息获取rviz上设置的goal

d.路径规划并发布path

## 二、实验内容与原理：

### 2.1实验一：

(1) ros系统文件管理结构：

```
└─ 'ROOT_DIR' or '~'
   └─ catkin_ws          # catkin workspace
      │   └─ build        #
      │   └─ devel       #
      │       └─ setup.bash #
      └─ src             # src -> ../catkin_ws_backup/xxx_ws_src
└─ catkin_ws_backup    # store packages | multi-workspace backup
```

(2) 创建package并运行脚本的基本步骤

```
catkin build
source devel/setup.bash
roscore
# open another console
roslaunch <package_name> <script_name.py>
```

## 2.2实验二：

- (1) 在rviz上检查你的机器人模型，运行 `roslaunch course_agv_description course_agv_rviz.launch` 命令。
- (2) 在gazebo中运行小车，添加controller,在终端中运行 `roslaunch course_agv_gazebo course_agv_world.launch` 命令查看小车在gazebo中的效果
- (3) 编写 `keyboard_command.py` 键盘控制脚本与 `kinematics.py` 运动学分解脚本控制小车的运动速度与方向。

$$\begin{aligned}\dot{\mathbf{X}}_I &= R(\theta)^{-1} \dot{\mathbf{X}}_R \\ &= R(\theta)^{-1} \begin{bmatrix} \frac{r\dot{\phi}_1}{2} + \frac{r\dot{\phi}_2}{2} \\ 0 \\ \frac{r\dot{\phi}_1}{2l} + \frac{-r\dot{\phi}_2}{2l} \end{bmatrix}\end{aligned}$$

图：地面系与机器人系的运动学变换

其中 `keyboard_command.py` 负责接收键盘的指令并把速度publish到 `/course_agv/velocity` 的topic, 而 `kinematics.py` 负责订阅该主题并通过运动学分解并将左右轮的joint的转速publish到 `/course_agv/leftwheel(rightwheel)/command` 的topic去

## 2.3实验三：

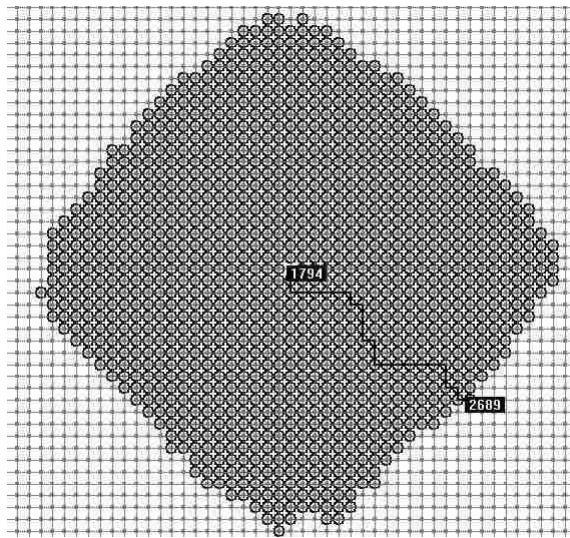
- (1) 添加地图和传感器
  - 在urdf里添加相关link以及joint
  - 在.gazebo文件中添加对应插件运行 `roslaunch course_agv_gazebo course_agv_world.launch` 指令在gazebo终端查看效果
- (2) 在rviz中添加需要显示的信息(laser,imu,map)并编写 `robot_tf.py` 将 `map/world_base->robot_base` 的tf变换广播出去。
- (3) 在rviz中查看广播是否成功与效果。

## 2.4实验四：

### (1) Dijkstra算法

Dijkstra算法用来寻找图形中节点之间的最短路径。在Dijkstra算法中，需要计算每一个节点距离起点的总移动代价。同时，还需要一个优先队列结构。对于所有待遍历的节点，放入优先队列中会按照代价进行排序。

在算法运行的过程中，每次都从优先队列中选出代价最小的作为下一个遍历的节点。直到到达终点为止。



## (2) A\*算法:

A\*算法通过下面这个函数来计算每个节点的优先级。

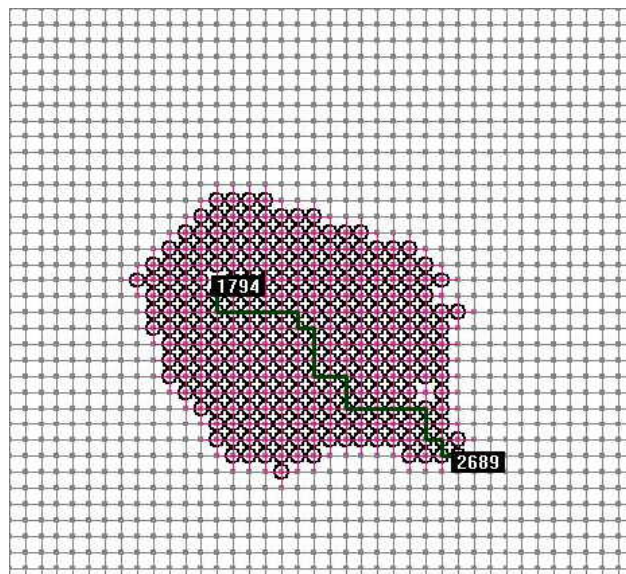
$$f(n) = g(n) + h(n)$$

其中:

- $f(n)$ 是节点 $n$ 的综合优先级。当我们选择下一个要遍历的节点时，我们总会选取综合优先级最高（值最小）的节点。
- $g(n)$ 是节点 $n$ 距离起点的代价。
- $h(n)$ 是节点 $n$ 距离终点的预计代价，这也就是A\*算法的启发函数。关于启发函数我们在下面详细讲解。

A\*算法在运算过程中，每次从优先队列中选取 $f(n)$ 值最小（优先级最高）的节点作为下一个待遍历的节点。

另外，A\*算法使用两个集合来表示待遍历的节点，与已经遍历过的节点，这通常称之为 `open_set` 和 `close_set`。



其中A\*算法的伪代码如下:

```

For each node n in the graph
n.f=Infinity, n.g=Infinity
2.Create an empty list.
3.Strat.g=0,start.f=H(start) add start to list.
4.While list not empty
(1)Let current=node in the list with the smallest f value,remove current from list
(2)If(current==goal node) report success
(3)For each node, n that is adjacent to current
If(n.g>(current.g+cost of edge from n to current))
n.g=current.g+cost of edge from n to current
n.f=n.g+H(n)
n.parent=current
add n to list if isn't there already

```

### 三、实验环境

ubuntu1604虚拟机、ros与catkin框架、gazebo与rviz仿真软件等

### 四、实验方法、步骤与程序代码

#### 4.1实验一：

(1)阅读ros tutorials相关内容

(2)编写hello.py,代码如下：

```

#!/usr/bin/env python
import rospy
print("Hello ROS")
print(rospy.get_param('test_param', "default_value"))

```

(3) 通过 `chmod 777 hello.py` 修改可执行的权限，在通过 `catkin build`、`source devel/setup.bash`、`roscore` 之后新开终端，运行 `roslaunch <package_name> hello.py` 即可

#### 4.2实验二：

(1)阅读ros tutorials中关于publisher与subscriber的相关内容

(2)编写keyboard\_command板块：

getleys函数：

```

def getKey():
    tty.setraw(sys.stdin.fileno())
    select.select([sys.stdin], [], [], 0)
    key = sys.stdin.read(1)
    termios.tcsetattr(sys.stdin, termios.TCSADRAIN, settings)
    return key

```

脚本主体的publisher部分:

```
pub1 = rospy.Publisher('/course_agv/velocity', Twist, queue_size = 1)
rospy.init_node('keyboard_command')
speed = rospy.get_param("~speed", 0.5)
turn = rospy.get_param("~turn", 1.0)
x = 0
y = 0
th = 0
status = 0

try:
    print(vels(speed, turn))
    while(1):
        key = getKey()
        if key in moveBindings.keys():
            x = moveBindings[key][0]
            th = moveBindings[key][1]
        else:
            x = 0
            y = 0
            th = 0
            if (key == '\x03'):
                break
        twist = Twist()
        twist.linear.x = x*speed; twist.linear.y = 0; twist.linear.z = 0;
        twist.angular.x = 0; twist.angular.y = 0; twist.angular.z = th*turn
        pub1.publish(twist)

except Exception as e:
    print(e)
```

(3)编写kinematics运动学分解部分:

```
def callback(twist):
    pub1 =
    rospy.Publisher('/course_agv/right_wheel_velocity_controller/command', Float64,
    queue_size = 1)
    pub2 = rospy.Publisher('/course_agv/left_wheel_velocity_controller/command',
    Float64, queue_size = 1)
    cmd1=Float64()
    cmd2=Float64()
    global data1
    data1 +=(3*(twist.linear.x+twist.angular.z*0.5))
    global data2
    data2 +=(3*(twist.linear.x-twist.angular.z*0.5))
    cmd1=data1
    cmd2=data2
    pub1.publish(cmd1)
    pub2.publish(cmd2)
def main():
    rospy.init_node('kinematics', anonymous=True)
    rospy.Subscriber("/course_agv/velocity", Twist, callback)
    rospy.spin()
```

(4)将上述的脚本添加进launch文件中，运行指令 `roslaunch course_agv_gazebo course_agv_world.launch`，在gazebo中查看小车控制效果。

### 4.3实验三

(1) 通过 `rostopic list`与 `rostopic echo /gazebo/link_states` 查看已有的主题与 `linkstates` 主题此时发送的内容。

```
link_name: "robot_base"
pose:
  position:
    x: 0.0871191867573
    y: -0.111874572568
    z: 0.010829240932
  orientation:
    x: 0.00259996055142
    y: -0.000962195627122
    z: -0.134019751781
    w: 0.990974782988
twist:
  linear:
    x: -0.211269133886
    y: -0.0114803923382
    z: 0.00294595184363
  angular:
    x: -0.0830640043808
    y: -0.00958107405998
    z: -0.568238027364
reference_frame: ''
success: True
status_message: "GetLinkState: got state"
course-ubuntu@ubuntu-VirtualBox:~$
```

(2) 编写 `robot_tf.py` 将 `map/world_base->robot_base` 的tf变换广播出去。

```
def callback(data):
    br = tf.TransformBroadcaster()
    #rospy.loginfo(data.name)
    msg=data.pose[data.name.index('course_agv::robot_base')]
    br.sendTransform((msg.position.x, msg.position.y, 0),
                    (msg.orientation.x,msg.orientation.y,msg.orientation.z,msg.orientation.w),
                    rospy.Time.now(),
                    "robot_base",
                    "map"
                    )
if __name__ == '__main__':
    print("helloworld")
    rospy.init_node('robot_tf')
    rospy.Subscriber("/gazebo/link_states",LinkStates,callback)
    rospy.spin()
```

(3) 运行 `roslaunch rqt_tree rqt_tree` 查看tf树变换效果

### 4.4实验四

#### 1.Dijkstra算法与障碍物膨胀

(1)通过调用rosservice与订阅rostopic获得地图、当前位置与选取的目标位置。

```
def mygetgoal():
```

```

rospy.Subscriber("/course_agv/goal", PoseStamped, callbackgoal)

def callbackpos(data):
    global curposx
    curposx=data.transforms[0].transform.translation.x
    global curposy
    curposy=data.transforms[0].transform.translation.y
    #rospy.loginfo("posx="+str(curposx)+"posy="+str(curposy) )

def mygetposition():
    rospy.Subscriber("/tf",TFMessage,callbackpos)

def mygetmap():
    rospy.wait_for_service('/static_map')
    try:
        getmap = rospy.ServiceProxy('/static_map', GetMap)
        mymap = getmap().map.data
        return mymap
    except rospy.ServiceException, e:
        print "Service call failed: %s"%e

```

(2)因为机器人存在一定的半径范围，所以需要对障碍物进行边缘膨胀，在实验四中我采用了牺牲小车的可移动范围来进行边缘膨胀，因为小车的长度与宽度分别0.4与0.2，因此小车的外接圆半径约为0.23,注意到原本的地图的像素尺寸为129\* 129，长宽在地图上都为20，因此我们把129\*129的地图进行进一步的压缩，**压缩为43 \*43**的地图，新地图中的每一格包含了原地图的九个小格，而只有**当这九个小格都没有包含障碍物的时候**，我们才认为这一个新的格是没有障碍物、可以到达的（其实是到达这个大格子的中心）由计算可得 $20/43=0.466$ ,而 $0.466/2=0.233$ ，即该格中心的到大格子边缘的**最短距离为0.233**>**小车的外接圆半径0.23**，因此我们可以认为在这样的条件下路径规划可以使得小车不会碰上障碍物。这样不仅对地图进行了障碍物边缘的膨胀工作，同时还可以将路径搜索的时间复杂度从129 \*129降到了43 \*43。具体对地图操作的代码如下所示。

```

global mycost
mycost = np.mat(np.zeros((43, 43),dtype=np.int))
global dist
dist = np.mat(np.zeros((1849,1849),dtype=np.int))
dist=dist+ float("Inf")
for i in range(0, 43):
    for j in range(0,43):
        if(mycmap[mypictrans2(3*i,3*j)]<10 and
mycmap[mypictrans2(3*i,3*j+1)]<10 and mycmap[mypictrans2(3*i,3*j+2)]<10 and
mycmap[mypictrans2(3*i+1,3*j)]<10 and mycmap[mypictrans2(3*i+1,3*j+1)]<10 and
mycmap[mypictrans2(3*i+1,3*j+2)]<10 and mycmap[mypictrans2(3*i+2,3*j)]<10 and
mycmap[mypictrans2(3*i+2,3*j+1)]<10 and mycmap[mypictrans2(3*i+2,3*j+2)]<10):
            mycost[i,j] = 100
        else:
            mycost[i,j]=0
for i in range(0, 43):
    for j in range(0, 43):
        dist[mypictrans(i, j),mypictrans(i, j)] = 0
        if (43 > i> 0 and 42 > j >=0 and mycost[i, j] > 10 and mycost[i - 1,
j + 1] > 10 ):
            dist[mypictrans(i, j),mypictrans( i - 1,j + 1)] =
dist[mypictrans(i - 1,j + 1),mypictrans(i, j)] = 14

```



```

        if (43 > i > -1 and 42 > j > -1 and mycost[i, j]>10 and mycost[i, j
+ 1]>10 ):
            dist[mypictrans(i, j),mypictrans(i, j + 1)] = dist[mypictrans(i,
j + 1),mypictrans(i, j)] = 10

        if (42 > i > -1 and 42 > j > -1 and mycost[i, j]>10 and mycost[i +
1, j + 1]>10 ):
            dist[mypictrans(i, j),mypictrans(i + 1,j + 1)] =
dist[mypictrans(i + 1,j + 1),mypictrans(i, j)] = 14
            if (42 > i > -1 and 43 > j > -1 and mycost[i, j]>10 and mycost[i +
1, j]>10):
                dist[mypictrans(i, j),mypictrans(i + 1, j)] = dist[mypictrans(i
+ 1, j),mypictrans(i, j)] = 10

```

(3)编写Dijkstra算法用来寻找图形中节点之间的最短路径，函数主体如下：

```

def shortPath_Dijkstra(graph, v0, v1):
    n = len(graph)
    global final
    global P
    global D
    global k
    k=0
    final, P, D = [0] * n, [0] * n, [0] * n
    for i in range(n):
        D[i] = graph[v0,i]
    D[v0] = 0
    final[v0] = 1
    for v in range(1, n):
        min = float("Inf")
        for w in range(0, n):
            if (final[w]==0 and D[w] < 100000 and D[w]<min and
mycost[w//43,w%43]>10) :
                k = w
                min = D[w]

        final[k] = 1
        for w in range(0, n):
            if (100>graph[k,w]>5 and final[w]==0 and min + graph[k,w] < D[w]):
                D[w] = min + graph[k,w]
                P[w] = k
        #print(v)
    return D[v1]

```

(4) 对得到的路径进行publish,发送到 /course\_agv/global\_path 主题

```

path = Path()
path.header.seq = 0
path.header.stamp = rospy.Time(0)
path.header.frame_id = 'map'
pathdata=mypictrans(int(21+goalx*2.1),int(21+goaly*2.1))
while(P[pathdata]!=mypictrans(int(21+curposx*2.1),int(21+curposy*2.1)) and
P[pathdata]!=0):
    pose = PoseStamped()
    pose.header.seq = i

```

```

pose.header.stamp = rospy.Time(0)
pose.header.frame_id = 'map'
pose.pose.position.y = (pathdata//43-21)/2.1+0.2
pose.pose.position.x = (pathdata%43-21)/2.1+0.2
pathdata=P[pathdata];
pose.pose.position.z = 0.01
pose.pose.orientation.x = 0
pose.pose.orientation.y = 0
pose.pose.orientation.z = 0
pose.pose.orientation.w = 1
path.poses.append(pose)
i=i+1
path.poses.reverse()
path.poses=path.poses[1:]
rospy.loginfo(path)
path_pub.publish(path)

```

## 2.使用A\*算法替代Dijkstra算法

其中，障碍物膨胀、对于rosservice的调用、rostopic的订阅与发送均与Dijkstra算法中一致，变化的是A\*算法替代了原算法。

定义AStar类，其中的基本数据类型、距离与判断函数如下：

```

class Astar:
    def __init__(self):
        self.open_set = []
        self.close_set = []

    def G_N_dist(self, p):
        x_dis = abs(p.x-int(21+curposx*2.1+0.5))
        y_dis = abs(p.y-int(21+curposy*2.1+0.5))
        # Distance to start point
        return x_dis + y_dis + (np.sqrt(2) - 2) * min(x_dis, y_dis)

    def F_N_dist(self, p):
        x_dis = abs(int(21+goalx*2.1+0.5) - p.x)
        y_dis = abs(int(21+goaly*2.1+0.5) - p.y)
        # Distance to end point
        return x_dis + y_dis + (np.sqrt(2) - 2) * min(x_dis, y_dis)

    def TotalCost(self, p):
        return self.G_N_dist(p) + self.F_N_dist(p)

    def JudgeObstacle(self, x, y):
        if x < 0 or y < 0:
            return False
        if x >= 43 or y >= 43:
            return False
        if mycost[x,y]<10:
            return False
        return True

    def WhetherinList(self, p, point_list):
        for point in point_list:
            if point.x == p.x and point.y == p.y:
                return True

```

```

        return False

    def IsInOpenList(self, p):
        return self.whetherinList(p, self.open_set)

    def IsInCloseList(self, p):
        return self.whetherinList(p, self.close_set)

    def IsStartPoint(self, p):
        return p.x == int(21+curposx*2.1+0.5) and p.y ==int(21+curposy*2.1+0.5)

    def IsEndPoint(self, p):
        return p.x == int(21+goalx*2.1+0.5) and p.y == int(21+goaly*2.1+0.5)

```

递归处理邻接点、中途剪枝并回溯跳出、建立path的主体运算函数部分代码如下

```

def Astar_algo(self):
    start_time = time.time()
    start_point = Point(int(21+curposx*2.1+0.5), int(21+curposy*2.1+0.5))
    start_point.cost = 0
    self.open_set.append(start_point)
    while True:
        index = self.SelectPointInOpenList()
        if index < 0:
            print('No path found!')
            return
        p = self.open_set[index]
        if self.IsEndPoint(p):
            return self.select_path(p,start_time)
        del self.open_set[index]
        self.close_set.append(p)
        # Process all neighbors
        x = p.x
        y = p.y
        self.NeighborPoint(x-1, y+1, p)
        self.NeighborPoint(x-1, y, p)
        self.NeighborPoint(x-1, y-1, p)
        self.NeighborPoint(x, y-1, p)
        self.NeighborPoint(x+1, y-1, p)
        self.NeighborPoint(x+1, y, p)
        self.NeighborPoint(x+1, y+1, p)
        self.NeighborPoint(x, y+1, p)

    def NeighborPoint(self, x, y, parent):
        if not self.JudgeObstacle(x, y):
            return # Do nothing for invalid point
        p = Point(x, y)
        if self.IsInCloseList(p):
            return # Do nothing for visited point
        #print('Process Point [' , p.x, ', ', p.y, ']', ', ', cost: ', p.cost)
        if not self.IsInOpenList(p):
            p.parent = parent
            p.cost = self.TotalCost(p)
            self.open_set.append(p)

        if self.IsInOpenList(p):

```

```

        if parent.cost+mylen(p.x,p.y,parent.x,parent.y)<p.cost:
            p.parent=parent
            p.cost=parent.cost+mylen(p.x,p.y,parent.x,parent.y)

def SelectPointInOpenList(self):
    index = 0
    selected_index = -1
    min_cost = sys.maxsize
    for p in self.open_set:
        cost = self.TotalCost(p)
        if cost < min_cost:
            min_cost = cost
            selected_index = index
    index += 1
    return selected_index

def select_path(self, p, start_time):
    global mypath
    mypath = []
    while True:
        mypath.insert(0, p) # Insert first
        if self.IsStartPoint(p):
            break
        else:
            p = p.parent
    end_time = time.time()
    print('==== Algorithm finish in', int(end_time-start_time), ' seconds')

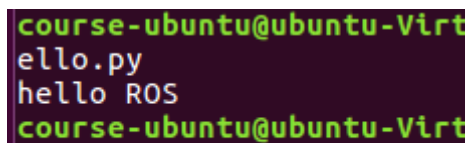
```

3.运行 `roslaunch course_agv_nav nav.launch` 查看运行效果

## 五、实验运行结果与分析

### 5.1实验一

如图所示，通过正确的实验步骤，最后运行hello.py脚本可以在终端中输出hello ROS。

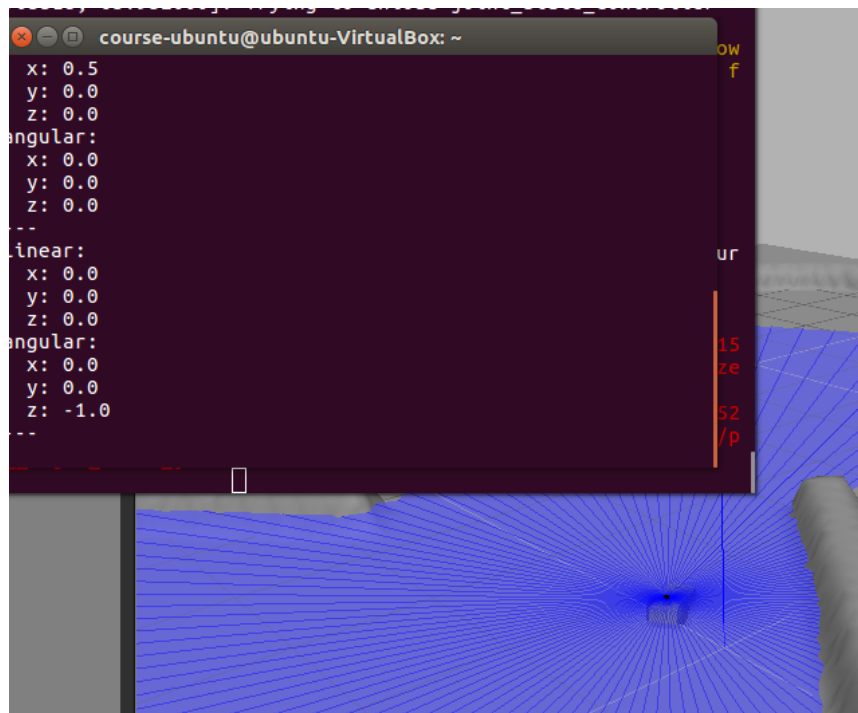


```

course-ubuntu@ubuntu-Virt
ello.py
hello ROS
course-ubuntu@ubuntu-Virt

```

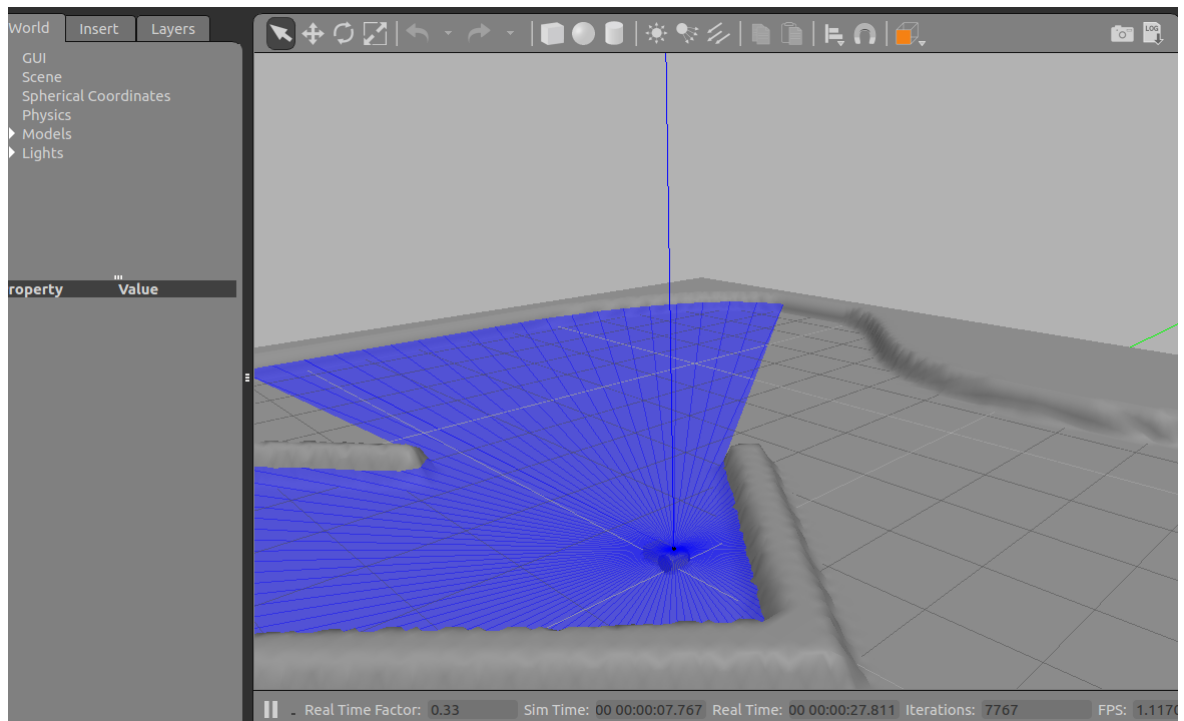
### 5.2实验二



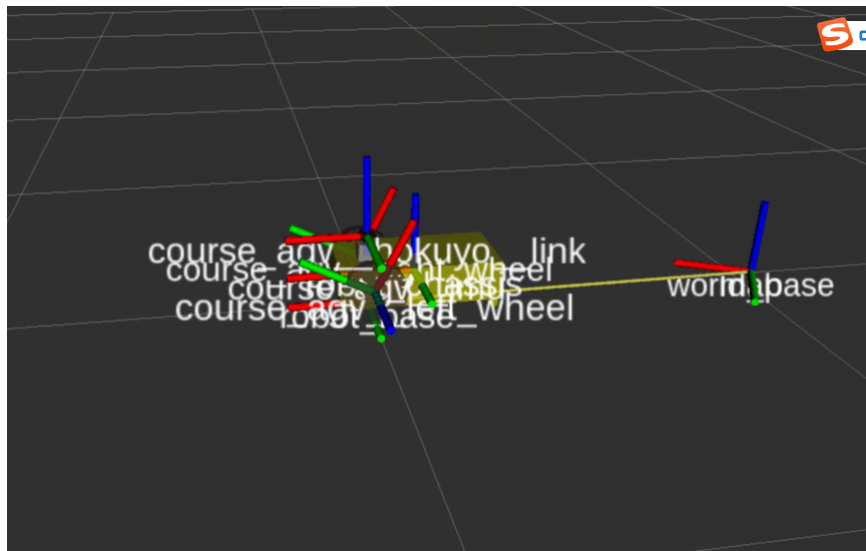
如图所示，通过使用键盘的WASD可以控制小车的速度与方向，符合实验预期与要求（这时候的截图course\_agv\_gazebo里面已经有了laser\_scan与IMU）。

### 5.3实验三

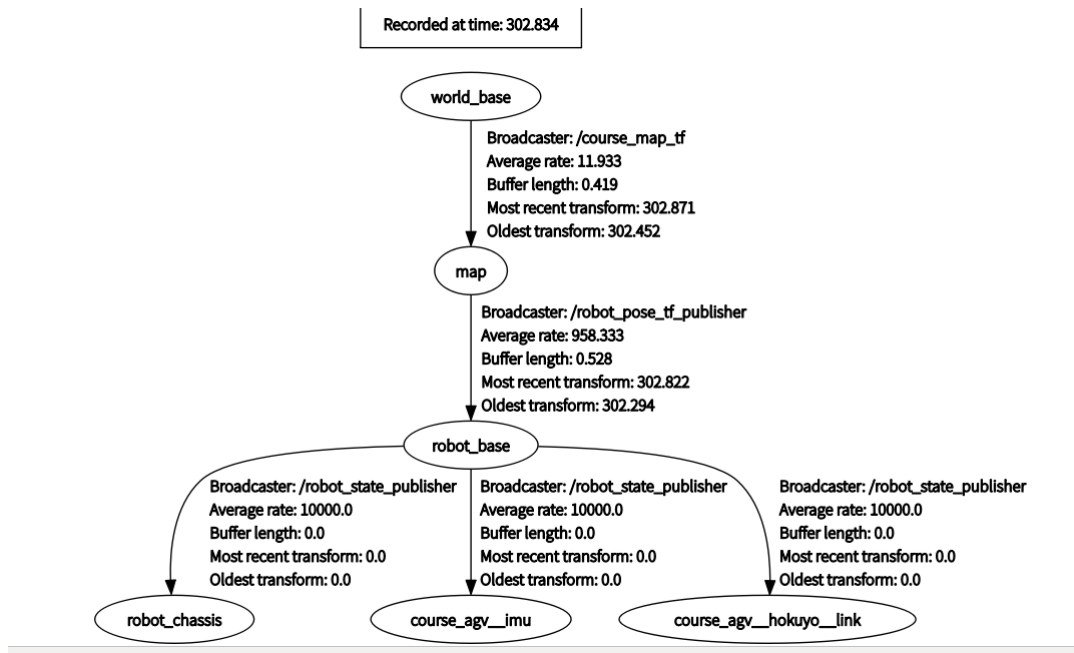
在urdf已经添加好link和joint的情况下，运行命令 `roslaunch course_agv_gazebo course_agv_world.launch` 可以看到在gazebo窗口中，laserscan激光正常显示，



rviz中可以看到不同的joint的坐标系。



在完成robot\_tf.py脚本的编写并进行roslaunch之后, 新建终端输入 `roslaunch rqt_tf_tree` `rqt_tf_tree` 可以在gui终端观察到此时的不同坐标系构成的tf\_tree,可以看到, 通过 `/robot_pose_tf_publisher` 的这个broadcaster,坐标变换从 `map` 坐标系被广播到了 `robot_base` 坐标系



而在rviz界面中我们同样可以看到, 在tf下方的不同frame全部显示transform成功。

Status: Ok	
✓ course_ag...	Transform OK
✓ course_ag...	Transform OK
✓ robot_base	Transform OK
✓ robot_cha...	Transform OK
✓ course_ag...	Transform OK
✓ course_ag...	Transform OK
✓ map	Transform OK
✓ world_base	Transform OK

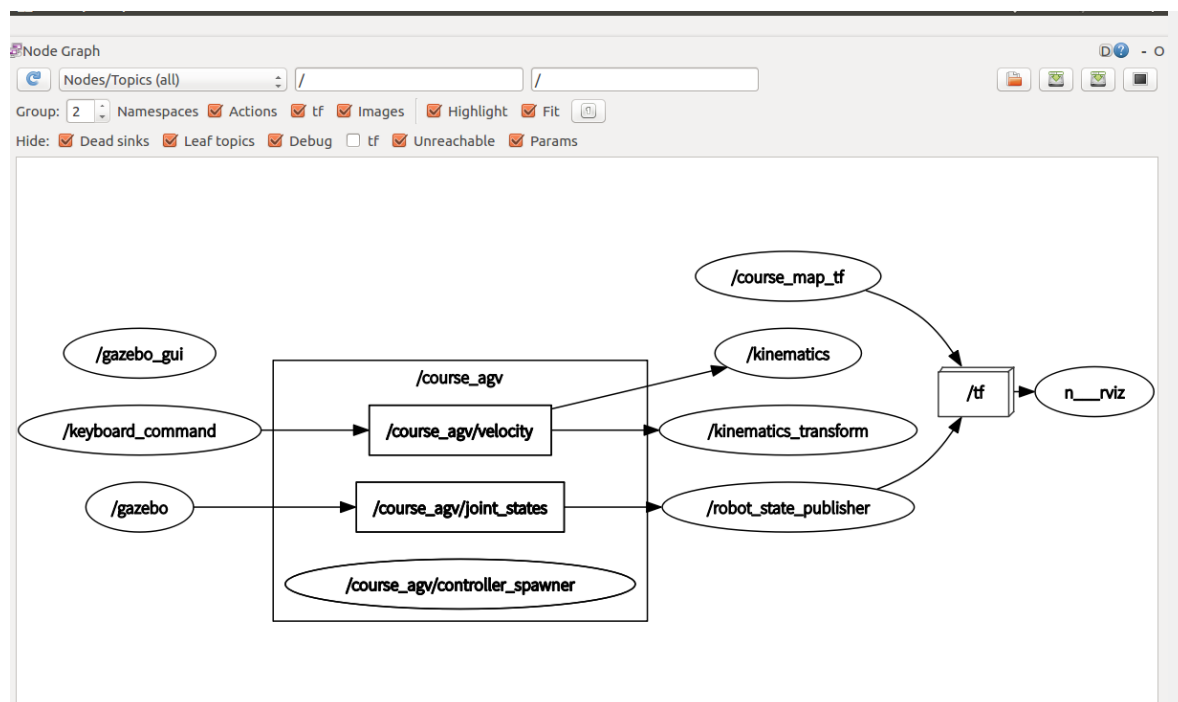
运行rostopic echo /tf可以看到的确有map到robot\_base的坐标转换被广播。

```

stamp:
  secs: 36
  nsecs: 128000000
  frame_id: "map"
  child_frame_id: "robot_base"
transform:
  translation:
    x: 0.000347754614216
    y: 4.09876651128e-05
    z: 0.0
  rotation:
    x: -5.06455379624e-06
    y: 5.77458365827e-07
    z: 0.00137504490319
    w: 0.999999054612
---
transforms:
-
  header:
    seq: 0
    stamp:
      secs: 36
      nsecs: 129000000
      frame_id: "map"
      child_frame_id: "robot_base"
    transform:
      translation:
        x: 0.000347769183082
        y: 4.09777277455e-05
        z: 0.0
      rotation:
        x: -5.49977425562e-06
        y: -2.1683184044e-08
        z: 0.00137510068322
        w: 0.999999054533
  ---

```

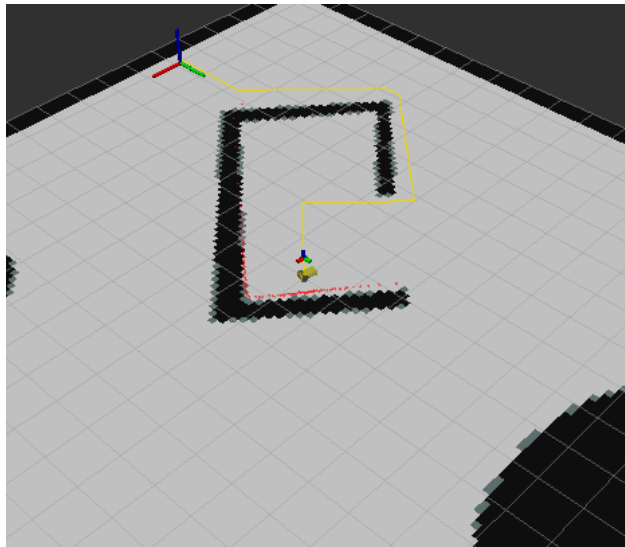
整个实验的rqt\_graph的roscnode关联图如下:



## 5.4实验四

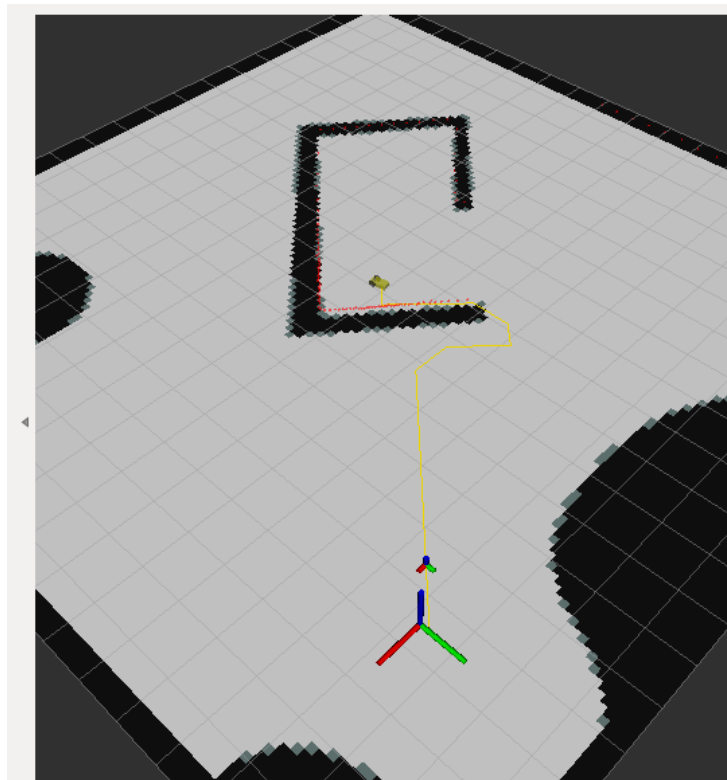
### 1.Dijkstra算法的实验结果

绕障碍物的结果:



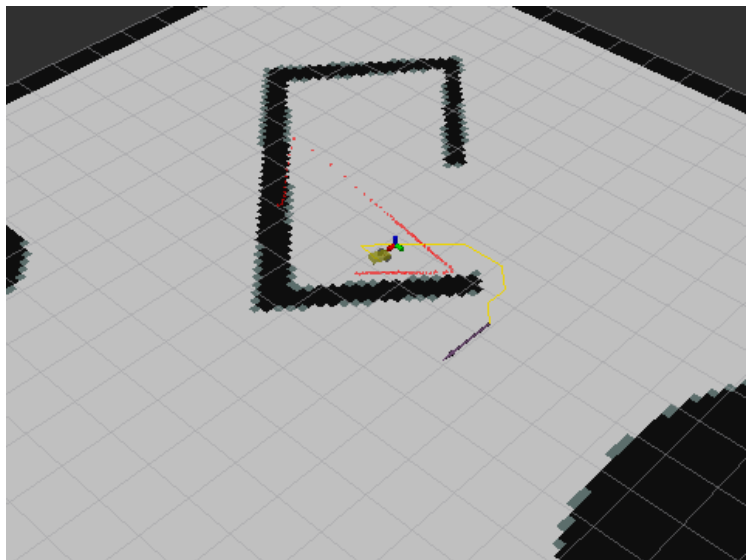
因为计算偏慢，Dijkstra算法对于二次选点与响应的能力很差，基本不具有实用性

**障碍物边缘膨胀的结果：**对于路径规划，若不加边缘膨胀的处理，规划的路径很可能会与障碍物有明显的摩擦，而加上路径规划之后，则可以看到路径与障碍物之间留出了大约一个小车车身的距离。



图：进行边缘膨胀处理之前





图：进行边缘膨胀处理之后

当我们选择`rospy.loginfo(path)`的时候，可以通过终端看到实际的路径中的每一个点的坐标信息，

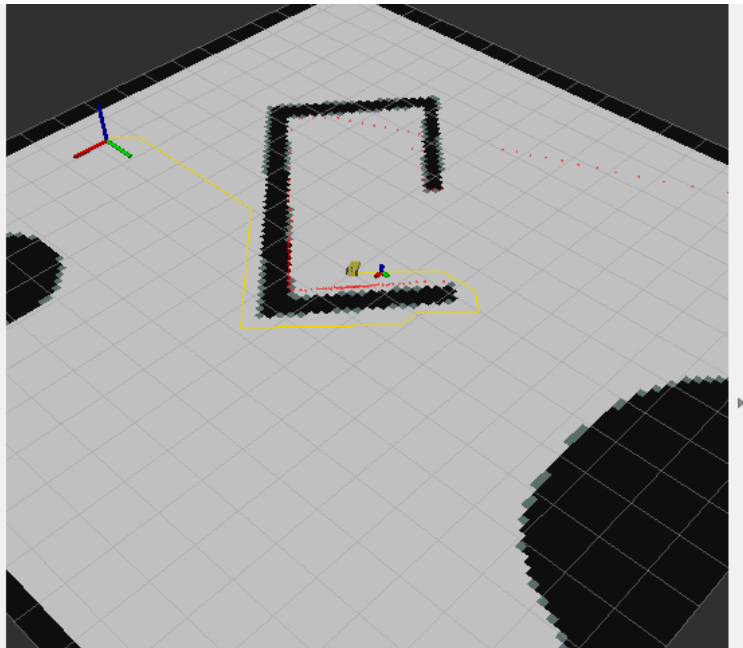
```
orientation:
  x: 0.0
  y: 0.0
  z: 0.0
  w: 1.0
header:
  seq: 9
  stamp:
    secs: 0
    nsecs: 0
  frame_id: "map"
pose:
  position:
    x: -2.85714285714
    y: -4.28571428571
    z: 0.01
  orientation:
    x: 0.0
    y: 0.0
    z: 0.0
    w: 1.0
header:
  seq: 10
  stamp:
    secs: 0
    nsecs: 0
  frame_id: "map"
pose:
  position:
    x: -3.33333333333
    y: -4.7619047619
    z: 0.01
  orientation:
    x: 0.0
    y: 0.0
    z: 0.0
    w: 1.0
header:
  seq: 12
```

**Dijkstra算法的运行时间：**Dijkstra算法相对于A\*算法真的很慢！因为不会提前跳出，所以计算时间和目标点的选取没有太大的关联，平均计算时间通过计时大约在5-9s的范围之内。

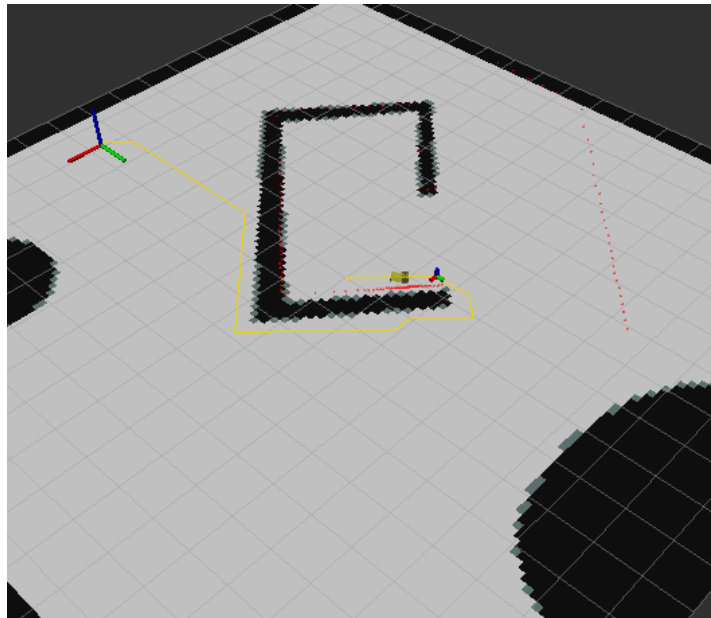
```
[FO] [1585850357.755174, 5.508000]: goalx=3.38497829437goaly=4.66103649139
===== A
lgorithm finish in 7 s
[INFO] [1585850357.795348, 5.519000]: the distance is ('ne
```

## 2.A\*算法的实验结果

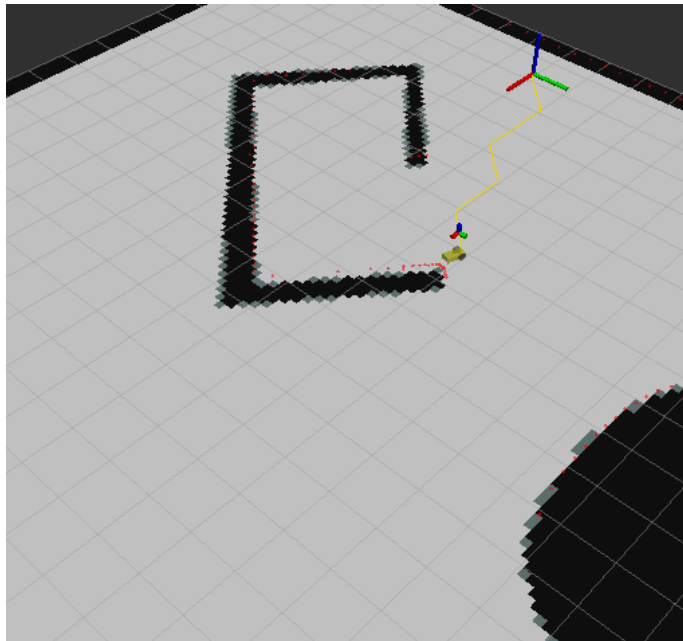
**基本结果：**A\*算法同样可以实现路径规划的过程。如下图所示，路径规划具有比较好的绕障碍物的能力。



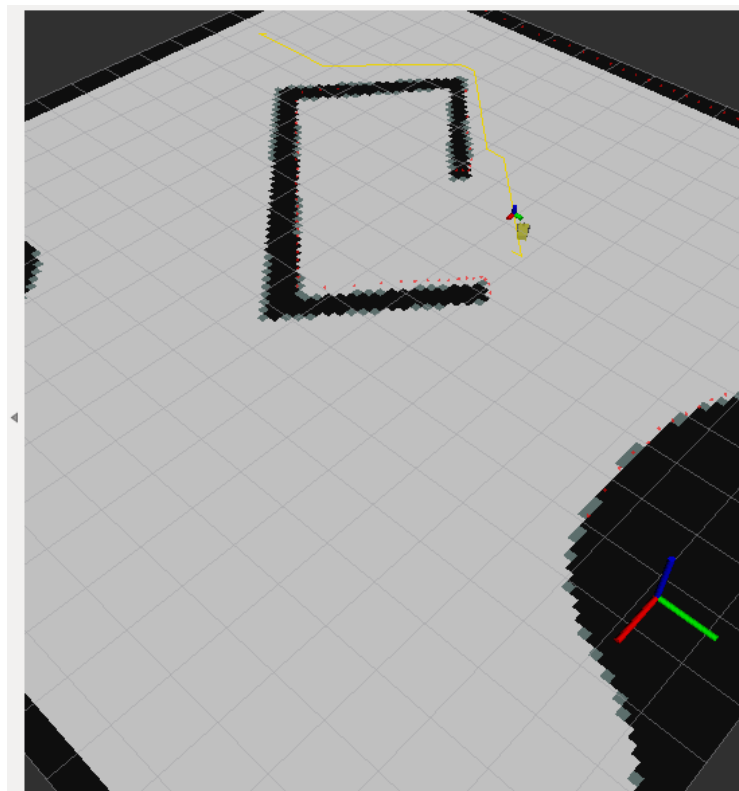
**Tracking的结果：**在路径规划实现后，小车可以在 mytracking 节点的下发之后，可以实现**路径的跟随**，如下图中所示



**A\*算法的二次规划响应与错误选点的提示：**因为较快的计算速度，路径规划对于小车行驶途中的二次响应速度也比较快，在上图的路径过程中进行二次选目标点，可以快速完成规划



对于选中**无效的障碍物**作为终点，则不会有成功的路径规划显示，ros终端也会提示No path found!



No path Found!

**A\*算法的运行时间：**对于A\*算法，计算速度要快很多，通过在程序中设计计时器可以看到算法完成的时间，对于较为简单、直接的路径(路途近，不会绕过障碍物)，运行的时间约在10-100ms之内，而对于较为复杂的绕过障碍物的路径，路径规划的时间也可以控制在300-400ms之内，基本可以满足较为迅速的规划响应。

```
475goalx=-4.52698326111 [INFO] [1585763126.867667, 8.374000]: goalx=-3.25557899
('==== Algorithm finish in', 12, ' ms')

10089417goalx=-7.775806427 [INFO] [1585763293.157014, 47.837000]: goalx=-5.732
('==== Algorithm finish in', 223, ' ms')
[INFO] [1585763293.382409, 47.887000]: the distance is
```

图：简单与复杂路径的时间统计

### 不同的距离启发式函数 (Heuristic Function) 对于结果的影响:

对于A\*算法，有着不同的启发式函数：

a.使用曼哈顿距离:

```
def heuristic(a, b):  
    # 网格上的曼哈顿距离  
    # Manhattan distance on a square grid  
    return (abs(a.x - b.x) + abs(a.y - b.y))*D
```

D是指两个相邻节点之间的移动代价，通常是一个固定的常数。

在使用曼哈顿距离之后，默认为小车只能在地图上进行上下左右方向的移动，但不能进行斜上斜下方向的移动，递归处理邻接点也只能处理邻接的上下左右的节点。

### b.使用对角距离

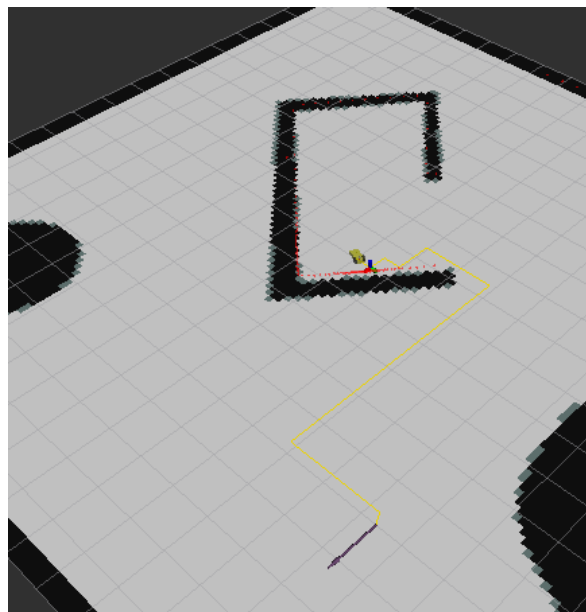
```
function heuristic(node) =
    dx = abs(node.x - goal.x)
    dy = abs(node.y - goal.y)
    return D * (dx + dy) + (D2 - 2 * D) * min(dx, dy)
```

对角距离则在曼哈顿距离的基础上增添了倾斜方向的移动选择

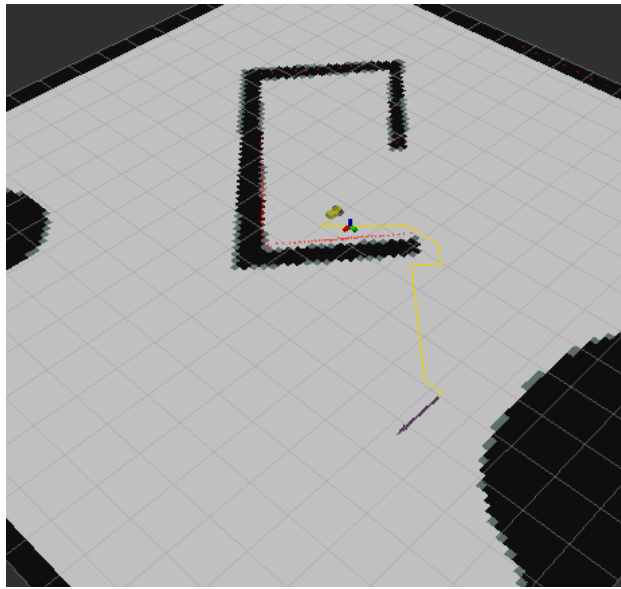
c.使用欧几里得距离:

```
function heuristic(node) =  
    dx = abs(node.x - goal.x)  
    dy = abs(node.y - goal.y)  
    return D * sqrt(dx * dx + dy * dy)
```

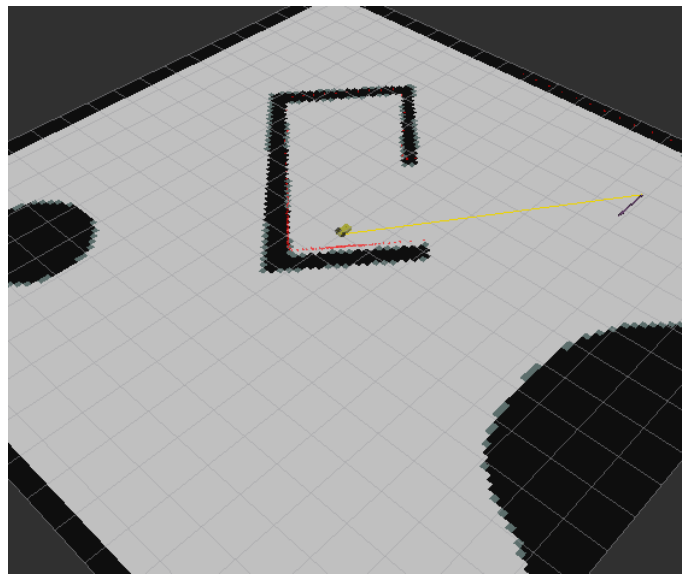
欧几里得距离使用的是直线距离。



图：启发函数使用曼哈顿距离



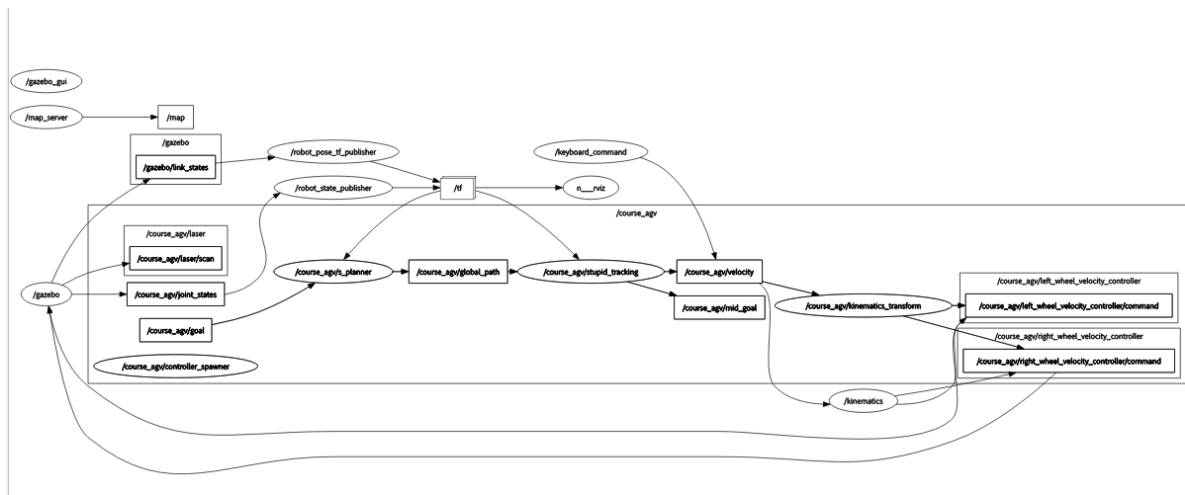
图：启发函数使用对角距离



图：启发函数使用欧几里得距离

如上图所示，使用曼哈顿距离时路径只能沿着x轴与y轴方向进行运动，而对角距离可以使得沿斜上下方向移动，欧几里得距离则在一定的范围内可以沿着任意网格的方向前进，没有障碍物的时候会沿着直线前进

从**运行时间**的角度运行实验分析，对角距离使用时会略微慢于曼哈顿距离的使用，而欧几里得距离的使用因为增加了自由度，运行时间会明显慢于之前的两种启发式函数。



图：程序总体的节点信息rqt\_graph

## 六、实验总结与心得

### 1、关于Dijkstra算法与A\*算法的对比与总结。

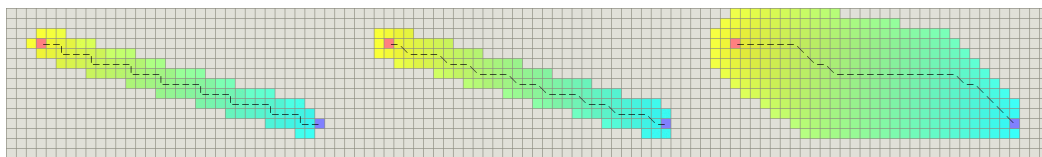
Dijkstra算法的实质是广度优先搜索，是一种发散式的搜索，时间复杂度比较高。而A\*算法不但记录当前点到起点点的代价，还估计当前点到目标点的期望代价，是一种启发式算法，可以认为是一种深度优先的算法。

A\*算法最关键的过程，就在每次选择下一个当前搜索点时，是从所有已探知的但未搜索过点中，选取路径总值最小的结点进行展开。当A\*算法的 $g(n)=0$ 时，该算法类似于Dijkstra算法，当 $h(n)=0$ 时，该算法类似于BFS。

A\*算法将简单的启发式距离预测引入到原有的Dijkstra算法，大幅降低待处理节点的数量，也大幅度加快了运行的速度（对本实验A\*算法运行时间在0.5s内而Dijkstra算法需要5-9s的时间）

### 2、对于A\*中不同启发式距离函数的选取。

使用曼哈顿距离时可以沿上下左右运动，而使用对角距离的时候还可以沿着斜上下的方向移动，使用欧几里得距离时可以在一定范围内对所有可能的点进行递归（即任意方向）。曼哈顿距离对于我们使用的地图来说大大限制了小车的行动自由，而欧几里得距离虽然理论上可以获得更符合真实的最短路径，但对角距离的情况下的路径的长短情况也并不会太差，且欧几里得启发函数因为递归搜索的范围广，运行的时间会显著地长于前两种启发函数。总结起来就是**只能上下移动**选取曼哈顿距离、要求**距离最优最短**选取欧几里得距离，在一般的情况下选取对角距离的启发式函数可以更好的**兼顾路径长度与运算时间**。



图：曼哈顿、对角、欧几里得距离复杂度的示意图

### 3、怎么解决实验二中小车会车头离地，车轮离地的现象

在最开始进行实验二的键盘控制、运动学分解的过程中，由于对于实际控制运动细节的忽视，导致小车有时候会出现车的一头离地或车轮离地的现象，经过分析，这可能是小车瞬时加速度过快的结果。我们可以通过：（1）减少每一次按下键盘后速度的增加量，每一次按下W、S后，速度的增加量直接关系到加速度大小，减少这个量可以有效控制加速度。

（2）设定速度与角速度的上限，小车的速度和角速度都不能过大，应当对速度与角速度设定阈值。

（3）合理设定  $v_x$  与  $v_w$  之间的比例， $v_x = (\text{left\_v} + \text{right\_v}) / 2$ ,  $v_w = (\text{left\_v} - \text{right\_v}) / (\text{双轮间距})$ ，小车双轮之间的距离为  $\text{width} + \text{wheel\_thickness} = 0.23$ ，应当设置按下WS与AD之间对于 $v_x$ 、 $v_w$ 影响的数值分配。

#### 4、对于障碍物膨胀的总结以及可能的优化

对于障碍物膨胀是一个在动态与静态、运动区域与风险概率的权衡，静态的障碍物膨胀本来是与小车当前的朝向相关，但我们为了将其完全的静态化，由于小车规则的形状，我们可以使用小车的外接圆半径作为进行障碍物膨胀的参数，为小车运动留出足够的距离（我使用的是将地图进行压缩，将多个小格转化为一个大格）

**可能的优化：**可以先对静态地图进行一个障碍物的静态膨胀，即对于原有的障碍物将其周围一定范围内的点也定为障碍物，然后再进行地图的压缩工作，比如把129 \* 129的地图压缩为43 \* 43,每一个大格中的每一个小格都不是障碍物才为可行点。在动态方面可以计算小车当前与障碍物的距离，当距离过小的时候对沿该方向的速度进行减小的自修正干预。

#### 5、对于stupid\_tracking的质疑以及不能按照路径tracking的解决措施（按照老师要求对助教代码的质疑）。

stupid\_tracking的缺点：在实验四的stupidtracking中，小车的转弯角度较大时，将转弯与前进分离开，只有当转弯基本完成时再继续前进，这会导致小车的**速度不连续，运动不流畅**等问题，而小车前进的速度为一个恒定的值，而实际上的速度可以设计为一个随着距离目标点的**距离大小而线性变化**的一个函数，同时也与**曲率**相关，在曲率小的平直路加快速度，曲率大的拐弯处减慢速度。同时对于w过大的限制，只做出了 `self.vw > 0.5` 的限制，但是没有考虑到w为负值方向的限制。

不能按照路径tracking的解决措施：可以在小车运动的过程中进行偏移的检测，在每一小段的起点与终点中，可以近似认为小车基本沿一条直线运动，可以在运动过程中检测小车离该直线的距离，当距离大于某个阈值的时候进行偏离纠正操作或进行重新规划。

#### 6、对于更加复杂的障碍物（比如障碍物动起来了）

当障碍物情况更加复杂甚至产生移动之后，实际的规划与tracking可能就会需要很多的改进了，如果障碍物移动到了现有的路径之上，如果还是使用原有的全局规划的路径，好像唯一的方法是是小车停下来，等待障碍物离开并保持安全距离后载通过。

但等待的方法在实际中的用途并不大，我们可以在全局规划的框架之上加上一个局部的规划，对于探测到的障碍物以及可能的移动对小车前面的每一块区域进行危险系数的评定，对于小车可能的不同的  $v_x$ 、 $v_y$ 、 $v_{\theta}$  进行一定空间内的采样与仿真、评分，选出评分最高的路径，并每运行一段时间进行一次这样的局部窗口规划。

**实验心得与体会：**

轮式机器人的课程对我来说是非常有趣而又伴随着一些艰难的，虽然我并不是控制专业的同学，之前对于机器人方面几乎是零基础的，但对于机器人的好奇心和每一次完成实验看着rviz与gazebo窗口中的机器人运动却是很有成就感的。但学习的过程中也遇到了非常多的困难，比如自己之前并没有系统地学习过python语言的语法，很多时候自己写python代码在有的语法部分会有一些吃力。每一次实验也需要去阅读非常多的英文资料，去查看每一个message类型怎么用，每个service该怎么调用等等，这需要很多的耐心。在实验过程中对于bug的发现与调试也是一个艰难的过程，因为终端下暂时还不会像在ide里面那样方便的观察数据变化等等，所以调试过程也会存在困难。希望我可以保持这样的热情，享受觉得自己不可能完成到最后完成的过程！