

# 2021-2022 大创阶段总结报告

王意晨

2022 年 9 月 5 日

## 1 实验说明

我们实验的总体目标是利用小车验证一系列的编队一致性实验，实验中的小车可以理解为带有 ROS 系统的机器人，小车配备有激光雷达系统、摄像系统。在实验中我们利用路由器将各个小车与自己的电脑连通，也就是说自己的电脑是 master 节点，而各个小车是 branch 节点，各节点之间能够相互 ping 通，在实验中的体现也就是各个小车之间能够相互交换信息。

为了实现编队一致性，我们的首要目标就是要实现跟随，也就是能够跟上前车，当然在后续的实验中我们也会考虑到后车，寻求一种在前后车之间的平衡状态，即尽量处于前后车之间。基于这一目标，我们提出了两种跟随的方式，某种意义上也是获取信息的方式，一个是 WiFi 互联，一个是激光雷达，这两种方式都会在后面做详细的说明。

实验的 GitHub 仓库：[https://github.com/wangyichen191/ROS\\_LAB](https://github.com/wangyichen191/ROS_LAB)

## 2 实验准备

### 2.1 平台搭建

具体的环境配置已经在另一篇文档（《2020-2021 大创阶段报告》）中有具体介绍，这里就不再具体阐述了。因为我们最终的目的是实现多机器人之间的互联，所以我们需要使用 multi\_robot 功能包，另一篇文档（《2020-2021 大创阶段报告》）中也有关于下载 multi\_robot 功能包的介绍。

所谓的实验平台只不过是一些列设备的总称，包括我们的电脑（安装有 ROS 系统的 linux 虚拟机），各个小车以及路由器。我们在电脑端用 ssh 命令登录到小车端，我们可以在小车端输入命令将小车连接到主机端（master 端），当所有小车都连接到 master 端时，小车之间就能互相 ping 通并交换信息。然后我们可以各个小车上编写并运行程序，使得各个小车做相应的运动。

### 2.2 WiFi 互联 + 激光雷达的实现

我们有两种方法合一实现跟随，一种是 WiFi 互联，另一种是激光雷达跟随。在实验中，我们可以同时使用两种方法，比如用激光雷达探测前车的速度和位置，用 WiFi 互联获得后车的速度和位置。增加激光雷达的应用无非就是多订阅了一个话题而已。比如我们执行如下命令：

```
roslaunch multi_robot robot_lidar.launch robot_name:=robot_0
```

就会有如下的一些话题

```
/robot_0/odom  
/robot_0/cmd_vel  
/robot_0/scan  
...
```

## Odometry.msg

```
#消息描述了自由空间中位置和速度的估计值
#文件位置：nav_msgs/Odometry.msg

Header header
string child_frame_id
PoseWithCovariance pose
TwistWithCovariance twist
```

图 1: odometry 消息类型

我们向 `cmd_vel` 话题 push `Twist` 类型的消息以指挥小车的运动。我们订阅 `scan` 话题，拿到小车激光雷达的数据，并进行分析得到前后车的各种信息。具体的各种消息类型可以在《ROS 操作系统入门讲义.pdf》第 3.5 节中查询。

WiFi 互联是从对方小车拿到对方的信息，而不能自己主动探测到，比如说小车 A 想知道小车 B 的速度和位置，那么小车 A 不能自己去探测，而是需要订阅小车 B 的一个话题 `odom`，从这个话题中抽取小车 B 的速度和位置。小车 B 会以一定的频率往这个话题 `odom` 中发布自己的各种信息，消息的类型是 `odometry`（如图1所示），在这个消息类型中有包含速度的消息类型 `twist` 和包含位置的消息类型 `pose`。

与 WiFi 互联不同，激光雷达不能通过 WiFi 来获取前后车的信息，而是对激光雷达返回数据进行分析才能得到前后车的信息。由于获得的激光雷达信息是点云数据，所以还需要一定的算法。这一点我们会在后面阐述。

### 2.2.1 WiFi 互联

WiFi 互联的话我们可以通过订阅话题的方法拿到任意小车的速度位置信息，比如我们想知道 1 号车的速度位置信息就可以使用如下代码：

```
self.selfsub = rospy.Subscriber('/robot_1/odom',Odometry,self.update_robot1,queue_size=1)
#Odometry 是消息类型，update_robot1 是回调函数（每有一个消息被发布在 Odometry 里面，
# 本程序就会调用一次 update_robot1 函数来处理）
#queue_size=1 代表消息缓存队列中只能有一个消息数据
```

在代码的具体实现时，我们的思路就是考虑距离，速度，以及角度差。不管是我最初实现的简单跟随，还是后来根据实验要求设计的利用加速度的方式实现的跟随，都要先获得距离，速度。下面的程序是单个小车跟随的程序，多个小车跟随可以类似地扩展，我们只是拿下面的程序简要说明跟随的思路。

```
#!/usr/bin/env python
# 必须有这一行注释
from ctypes import pointer
import rospy
import math
import numpy
from geometry_msgs.msg import Twist
from nav_msgs.msg import Odometry
from geometry_msgs.msg import Pose
from geometry_msgs.msg import Quaternion
```

```

#global var
start_distance = 0.5 # 小车起始间距
safe_distance = 0.5 # 小车目标间距
frequency = 0.02 #1/50hz 50hz 是 odom 的信息发布速率

# 计算两个小车之间的位置 pos0 和 pos1
def calculate_distance(self):
    return math.sqrt((self.pos1.position.x-self.pos0.position.x)**2 +
        (self.pos1.position.y-self.pos0.position.y)**2)

# 更新小车 0 的位置信息
def update_position_0(self,data):
    global start_distance

    # 以小车 1 的起始位置为原点建立的坐标系，所以需要加上起始偏移
    self.pos0.position.x = start_distance + data.pose.pose.position.x
    self.pos0.position.y = data.pose.pose.position.y

# 更新小车距离向量（由小车 1 指向小车 0 的向量）
def update_distance_vector(self):
    self.distance_vector = [self.pos0.position.x-self.pos1.position.x,
        self.pos0.position.y-self.pos1.position.y]

# 更新小车 1 的车头向量（指示小车 1 车头指向的向量）（这里我们用到了四元数）
def update_robot1_vector(self,data):
    a = 2 * numpy.arccos(data.pose.pose.orientation.w)
    if data.pose.pose.orientation.z >= 0:
        b = 1
    else:
        b = -1
    self.robot1_vector = [numpy.cos(a),b*numpy.sin(a)]

# 根据右手定则，计算出与 vector 垂直的向量，并且该向量在 vector 逆时针左边
def left_vertical(vector):
    z = numpy.array([0,0,1])
    x = numpy.array([vector[0],vector[1],0])
    y = numpy.cross(z,x)
    return [y[0],y[1]]

def calculate_angular_dis(self):

    # 计算小车 1 和 0 的距离向量和小车 1 车头向量之间夹角的 cos
    cos = numpy.dot(self.distance_vector,self.robot1_vector) /
        (numpy.linalg.norm(self.distance_vector) * numpy.linalg.norm(self.robot1_vector))

    # 计算垂直于小车 1 车头向量的向量，并且该向量在车头向量逆时针左边
    vector_vertical = left_vertical(self.robot1_vector)

    # 计算距离向量执行车头向量的左边或右边，如果 symbol 大于 0 指向左边，否则指向右边
    symbol = numpy.dot(vector_vertical,self.distance_vector)

```

```

        if symbol >= 0:
            return numpy.arccos(cos)
        else:
            return numpy.arccos(cos)*(-1)

# 根据当前距离与预期距离之间的差距以及当前速度的大小设定下一时刻速度大小
def check_distance(distance,data):
    global safe_distance
    global frequency
    dis = distance - safe_distance
    if dis > 0:
        return dis * 1.2 + data.twist.twist.linear.x
    else:
        return dis * 0.8 + data.twist.twist.linear.x

class SubThenPub:

    def __init__(self):
        global start_distance
        self.distance_vector = numpy.array([1,0])
        self.robot1_vector = numpy.array([1,0])
        self.pos0 = Pose()
        self.pos1 = Pose()
        self.pos0.position.x = start_distance
        self.pos0.position.y = 0
        self.pos1.position.x = 0
        self.pos1.position.y = 0
        self.__pub_ = rospy.Publisher('/robot_1/cmd_vel',Twist,queue_size=1)
        self.__sub_ = rospy.Subscriber('/robot_0/odom',Odometry,self.callback,queue_size=1)
        self.selfsub = rospy.Subscriber('/robot_1/odom',Odometry,self.update_robot1,queue_size=1)

    def callback(self,data):
        global frequency
        update_position_0(self,data)
        update_distance_vector(self)
        angular_dis = calculate_angular_dis(self)

        # 根据角度差调整角速度，当角度差 >0 往左转弯，当角度差 <0 往右转弯
        angular_speed = angular_dis * 0.8
        distance = calculate_distance(self)
        linear_speed = check_distance(distance,data)
        msg = Twist()
        msg.linear.x = linear_speed
        msg.angular.z = angular_speed

        # 发布 Twist 消息控制运动
        self.__pub_.publish(msg)

        # 在屏幕上打印消息
        rospy.loginfo("angular_dis: %0.2f distance_dis: %0.2f"

```

```

        pub velocity [%0.2f m/s,%0.2f rad/s] leader speed:%0.2f x:%0.2f y:%0.2f",
        angular_dis,distance,
        linear_speed,angular_speed,data.twist.twist.linear.x,
        data.pose.pose.position.x,data.pose.pose.position.y)

    def update_robot1(self,data):
        update_robot1_vector(self,data)
        self.pos1.position.x = data.pose.pose.position.x
        self.pos1.position.y = data.pose.pose.position.y

def main():
    rospy.init_node('position2',anonymous=True)
    SubThenPub()
    rospy.spin()

if __name__ == "__main__":
    main()

```

### 2.2.2 激光雷达

激光雷达的数据是一个列表，列表的每一个元素是一个方向的信息，也是用一个列表表示，存储角度和距离。激光雷达的探测距离最远是 12m，超过 12m 距离将是 INF。下面我们展示一段用雷达探测前车的程序（也可以探测后车）。

我们的思路是先将列表中相邻的元素进行合并（视为一个物体），如果这两个点在空间中的距离很远，那么这两个点不属于同一个物体，如果这两个点小于一定值，把这两个点视为一个物体上的点。就这样先选出一系列物体，然后我们量取小车的宽度，再把与这一宽度相距太大的物体剔除，剩余的就是类似于小车的物体了。在开始阶段，前后车相对于本车的方向和距离大致确定，所以我们将剩余的物体进行排序，与期待的方向和距离差距加权最小的物体就会被视为小车。

在运动过程中也是按照这样的思路探测小车，不同的是，每探测一次，我们就会将前后车相对于本车的方向和距离记录下来，然后在下一次探测的时候选择那些与记录的方向和距离差距较小的物体。

如果想要利用激光雷达，在订阅 scan 话题之后就可以复制粘贴使用其中的函数 ScanCallback（全局变量 cos,has\_ros,car\_width 等也要复制），具体的结果体现在 Call\_Check 函数的返回值 result 中。

```

#!/usr/bin/env python
import rospy
import numpy
import math
import sys
import time
from sensor_msgs.msg import LaserScan
from geometry_msgs.msg import Twist
from nav_msgs.msg import Odometry
from geometry_msgs.msg import Pose
from geometry_msgs.msg import Quaternion
from geometry_msgs.msg import Accel

#global var
start_distance = 0.5
safe_distance = 0.5

```

```

cos = 0
has_ros = 0
car_width = 0.2

def deal_angle(angle):
    if angle < math.pi:
        return angle
    else:
        return angle - 2 * math.pi

def judge_object(distance, LastDistance):
    #judge whether two point belong to the same object
    global cos
    result = []
    dis = math.sqrt(distance**2 + LastDistance**2 - 2*distance*LastDistance*cos)
    if dis > 0.03:
        result = [0,0]
    else:
        result = [dis,1]
    return result

def Call_Check(self, arrays, scan_data):
    #judge the leader's position according to the point cloud
    result = []
    filter_result = []
    for array in arrays:
        if array[3] < car_width + 0.2 :
            #and array[0] < safe_distance + 1 and array[0] > safe_distance - 0.4:
            angle = (array[1] + array[2]) * scan_data.angle_increment / 2
            #follow the car which lie in the left front or right front of the follower
            if True:
                array.append(angle)
                filter_result.append(array)

    for array in filter_result:
        temp_dis1 = array[0] - self.lastdistance1
        temp_ang1 = deal_angle(array[4]) - self.lastangle1
        temp_dis2 = array[0] - self.lastdistance2
        temp_ang2 = array[4] - self.lastangle2

        array.append(numpy.abs(temp_ang1))
        array.append(numpy.abs(temp_dis1))
        array.append(numpy.abs(temp_ang2))
        array.append(numpy.abs(temp_dis2))
        array.append(array[5]*0.5+array[6]*0.5)
        array.append(array[7]*0.5+array[8]*0.5)
    filter_result.sort(key = lambda x:x[9],reverse=False)
    if len(filter_result) == 0:
        result.append([20, 'No Filter Result'])
    else:
        if filter_result[0][5] > 0.2:

```

```

        result.append([20, 'The Angle Out of Prediction', filter_result])
    elif filter_result[0][6] > 0.5:
        result.append([20, 'The Distance Out of Prediction'])
    else:
        self.lastdistance1 = filter_result[0][0]
        self.lastangle1 = deal_angle(filter_result[0][4])
        result.append(filter_result[0])
filter_result.sort(key = lambda x:x[10],reverse=False)
if len(filter_result) == 0:
    #return None
    result.append([20, 'No Filter Result'])
else:
    if filter_result[0][7] > 0.2:
        result.append([20, 'The Angle Out of Prediction', filter_result])
    elif filter_result[0][8] > 0.5:
        result.append([20, 'The Distance Out of Prediction'])
    else:
        self.lastdistance2 = filter_result[0][0]
        self.lastangle2 = filter_result[0][4]
        result.append(filter_result[0])

return result

class laserTracker:
    def __init__(self):
        self.acc = 0
        self.leaderlinear = 0
        self.leaderangle = 0
        self.mylinear = 0
        self.myangle = 0
        self.followerlinear = 0
        self.followerangle = 0
        self.lastangle1 = 0
        self.lastdistance1 = safe_distance
        self.lastangle2 = math.pi
        self.lastdistance2 = safe_distance
        self.delta10 = 0
        self.delta12 = 0

        self.f = open('../output/output_1.txt', 'w')
        self.f.write("test1 robot1\n")
        self.f.write("time(s) position_x(m) position_y(m) linear_speed(m/s)
                      angular_speed(rad/s) acceleration(m/s^2)\n")

        self.scanSubscriber = rospy.Subscriber('/robot_1/scan', LaserScan, self.ScanCallback)
        self.Publisher = rospy.Publisher('/robot_1/cmd_vel', Twist, queue_size=1)
        self.mySubscriber = rospy.Subscriber('/robot_1/odom', Odometry, self.MyCallback)

    def ScanCallback(self, scan_data):
        global has_ros
        dis_seq = []
        LastDistance = float('inf')

```

```

k = -1

if has_ros == 0:
    global cos
    cos = numpy.cos(scan_data.angle_increment)
    has_ros = 1

for i, distance in enumerate(scan_data.ranges):
    if distance != float('inf'):
        #we create the first dis_seq
        if LastDistance == float('inf'):
            temp = [distance,i,i,0]
            dis_seq.append(temp)
            k = k + 1
            LastDistance = distance
        else:
            # 判断两个相邻方向的点是否属于同一个物体
            distance_result = judge_object(distance,LastDistance)
            if distance_result[1] == 1:
                dis_seq[k][2] = i
                dis_seq[k][3] = dis_seq[k][3] + distance_result[0]
                dis_seq[k][0] = (dis_seq[k][0] + distance) / 2
            else:
                temp = [distance,i,i,0]
                dis_seq.append(temp)
                k = k + 1
                LastDistance = distance
result = Call_Check(self,dis_seq,scan_data)

#result[0][0]!=20 说明探测到前车, 且 result[0][0],result[0][1],result[0][2]
# 分别表示与前车的距离, 前车相对本车的始端扫描角度, 前车相对本车的末端扫描角度
#result[1][0]!=20 说明探测到后车, 且 result[1][0],result[1][1],result[1][2]
# 分别表示与后车的距离, 后车相对本车的始端扫描角度, 后车相对本车的末端扫描角度
# 扫描的角度是从正前方开始逆时针扫描的
if(result[0][0]!=20):
    angle = ((result[0][1] + result[0][2]) * scan_data.angle_increment) / 2

    # 在车头左前方的方向接近 0, 在车头右前方的方向接近 2pi, 而我们要把方向转换为越接近
    # 车头前方的方向绝对值越接近 0, 越接近车尾方向的方向的绝对值越接近 pi
    angle_result = deal_angle(angle)
    distance = result[0][0]
    msg = Twist()
    msg.linear.x = (distance - safe_distance) * 1.3
    msg.angular.z = 0.7 * angle_result
    self.Publisher.publish(msg)
    rospy.loginfo("Distance_leader: %0.2f Angle_Leader: %0.2f rad Linear Speed: %0.2f m/s
                  Angle Speed: %0.2f rad/s Accel: %0.2f",
                  result[0][0],angle_result,msg.linear.x,msg.angular.z,self.acc)
else:
    msg = Twist()
    msg.linear.x = self.mylinear

```



```

        msg.angular.z = self.leaderangle
        self.Publisher.publish(msg)
        rospy.loginfo("Linear Speed: %0.2f m/s Angle Speed: %0.2f rad/s",
                        msg.linear.x, msg.angular.z)

def MyCallback(self, data):
    self.mylinear = data.twist.twist.linear.x
    self.myangle = data.twist.twist.angular.z

    ticks = time.time()
    position_x = data.pose.pose.position.x - start_distance
    position_y = data.pose.pose.position.y
    linear_speed = data.twist.twist.linear.x
    angular_speed = data.twist.twist.angular.z
    accel = 0
    self.f.write(str(ticks) + " " + str(position_x) + " " + str(position_y) + " "
                 + str(linear_speed) + " " + \
                 str(angular_speed) + " " + str(accel) + "\n")

def main():
    rospy.init_node('robot_1_lidar', anonymous=True)
    laserTracker()
    rospy.spin()

if __name__ == "__main__":
    main()

```

## 3 实验一

### 3.1 车队描述

已知一列车队，包含 leader 和 follower，如图2所示。

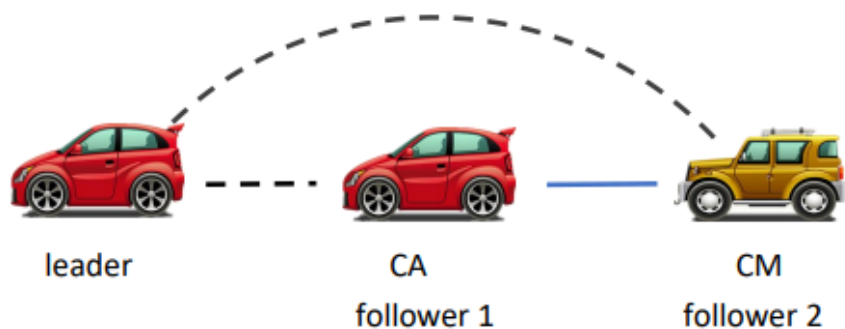


图 2: 实验一车队图

上述三种车辆的具体描述如下：

Leader：以期望速度  $v_0$  行驶，加速度为 0；

Follower 1：是 CA 车辆，能传递准确信息给其他网联车，也可以受到控制；

Follower 2: 是 CM 车辆，能传递准确信息给其他网联车，但是不能准确受控，也就是其决策权在司机；

### 3.2 模型构建

整体来看，任意车辆的加速度满足下式：

$$a_i = \sum_{j=0}^n k_p(p_j - p_i + \Delta_{ij}) + k_v(v_j - v_i)$$

具体来说：

第一辆车（leader）的加速度满足

$$a_0 = 0$$

第二辆车（Follower 1）的加速度满足：

$$a_1 = k_{p1}[(p_0 - p_1 + \Delta_{10}) + (p_2 - p_1 + \Delta_{12})] + k_{v1}[(v_0 - v_1) + (v_2 - v_1)]$$

第三辆车（Follower 2）的加速度满足：

$$a_2 = k_{p2}[(p_0 - p_2 + \Delta_{20}) + (p_1 - p_2 + \Delta_{21})] + k_{v2}[(v_0 - v_2) + (v_1 - v_2)]$$

### 3.3 实验目的

给定初始状态（即每辆车的速度、位置），调节参数（ $k_{p1}$ 、 $k_{p2}$ 、 $k_{v1}$ 、 $k_{v2}$ ）使得车队实现编队一致性（也就是说，车队中所有车辆的速度是相等的，相邻车之间的间距是相等的）

### 3.4 实验要求

1. 确定 leader 的速度（该速度即车队的期望速度），给定 follower 1 和 follower 2 的初始状态；
2. 确定参数  $k_{p1}$ 、 $k_{p2}$ 、 $k_{v1}$ 、 $k_{v2}$ ；
- $k_{p1}$ 、 $k_{v1}$  需要满足下述关系式：

$$w_1^3 k_{v1}^2 + (k_{p1}^2 + 3k_{v1}^4 - 4k_{v1}^2 k_{p1})w_1^2 + (6k_{p1}^2 k_{v1}^2 - 4k_{p1}^3)w_1 + 3k_{p1}^4 > 0$$

其中， $w_1 = \sqrt{\frac{4k_{p1}^3 k_{v1}^2 + k_{p1}^4}{k_{v1}^4}} - \frac{k_{p1}^2}{k_{v1}^2}$   
 $k_{p2}$ 、 $k_{v2}$  需要满足下述关系式：

$$w_2^3 k_{v2}^2 + (k_{p2}^2 + 3k_{v2}^4 - 4k_{v2}^2 k_{p2})w_2^2 + (6k_{p2}^2 k_{v2}^2 - 4k_{p2}^3)w_2 + 3k_{p2}^4 > 0$$

其中， $w_2 = \sqrt{\frac{4k_{p2}^3 k_{v2}^2 + k_{p2}^4}{k_{v2}^4}} - \frac{k_{p2}^2}{k_{v2}^2}$

3. 观察车队中车辆的轨迹，如果没有实现编队一致性，则重复步骤 2，直到实现编队一致性。

### 3.5 实验原理

车队中的 Leader 是按照一定的速度运行的，所以我们只需要给其设定速度就行；第二辆小车则是网联车辆，可以拿到所有车的信息，我们用 WiFi 来实现。

在这个实验中，由于第三辆小车是 CM 车辆，其决策权在司机，所以为了模拟人工驾驶的场景，我们第三辆小车的控制暂时不需要 3.2 节的加速度以及 3.4 节中参数的限制。我们使用如下的模型，如图3所示。每一时刻，follower2 通过雷达能够探测到前方小车与自己的距离 distance 以及角度 a，然后我们使用这两个数据确定自己的线速度 linear 和角速度 angular：

$$linear = (distance - safe\_distance) * arg1$$

$$angular = arg2 * a$$

这样设计的原因就是在现实生活中，我们在驾驶汽车的时候，为了与前车保持一定的距离，当目前距离比安全距离 `safe_distance` 大的时候，自己就要加速逐渐缩短与前车的距离，当目前距离比安全距离小的时候，自己就要减速逐渐加大与前车的距离。而且当自己与前车偏离一条直线的时候，自己就要根据偏转的角度调整自己的转向速度，尽量使得车头朝向前车车尾。参数 `arg1` 和 `arg2` 的值会在实验结果中给出。

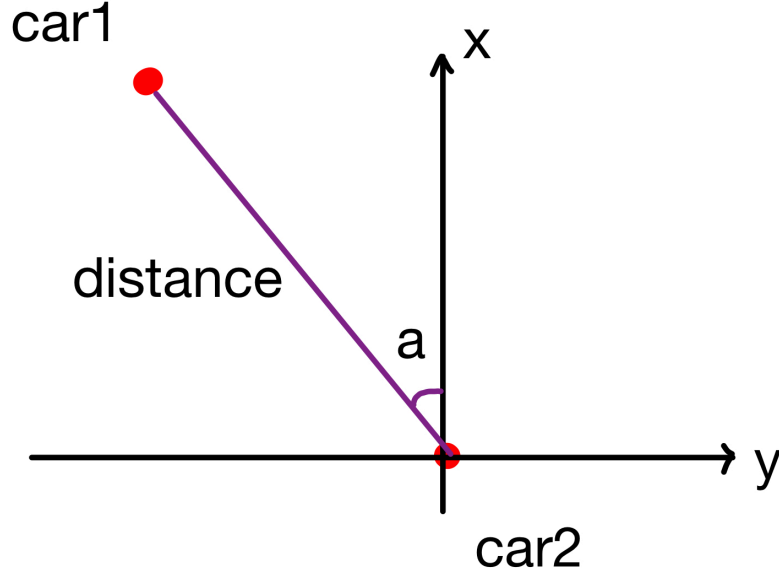


图 3: follower2 的运动物理模型

### 3.6 实验结果

实验过程中车队行驶状态良好，车与车之间的间距相对稳定且与设定的值比较接近。最终我们得到的参数为：

$$k_{p1} = 12, k_{v1} = 6, arg1 = 1.3, arg2 = 0.7$$

图4为我们完整视频的截图。完整的视频（lab1.mp4）存放在压缩包和 GitHub 仓库中。

## 4 实验二

### 4.1 实验目的

验证满足条件的混合车队可以实现编队一致性。

### 4.2 实验原理

#### 4.2.1 混合车队中车辆的分类

混合车队中的车辆按照位置进行分类，包括两类车辆：领导车辆（leader）和跟随车辆（follower），参考车辆的位置和速度一直是理想的位置和速度（即目标位置和目标速度），而跟随车辆的位置和速度起初是随机的。

混合车队中的车辆按照是否受控和网联进行分类，包括三类车辆：网联受控车辆、网联不受控车辆和不网联不受控车辆，网联车辆是指可以收到其他网联车辆的信息（包括速度、位置等信息）并且可以将自己的信息传递给其他车辆的车辆；受控车辆是指收到指令并及进行执行的车辆；由此可以组合出三类车辆。

表 1 将 reference 和 follower 两大类车辆的图示进行了说明，以便于后文的区分。



图 4: 实验一结果展示

#### 4.2.2 车队描述

本实验中的混合车队一共有四辆车，其中第一辆车是 leader，其他三辆车的位置不确定，具体来说可以将该车队分成四种情况讨论。车队的排列和说明如表 2 所示，在车队的排列图中，蓝色线代表雷达跟踪，黑色虚线代表无线传递信息。

#### 4.3 实验内容及步骤

总的来看，本实验按照表 2 的 8 种情况进行实验，分别给定初始状态（即每辆车的速度、位置），调节参数（即 HDV 和 CAV 的速度系数和位置系数）使得车队实现编队一致性（也就是说，车队中所有车辆的速度是相等的，相邻车之间的间距是相等的）。

下面及进行步骤的说明：1、确定 leader 的速度（该速度即车队的期望速度），给定 follower 1、follower 2 和 follower 3 的初始状态（即初始位置、速度以及加速度）；

2、确定参数（即速度系数和位置系数，例：在编号为 1 的车队中需要确定  $k_{p2}$ 、 $k_{v2}$ 、 $k_{p3}$ 、 $k_{v3}$  这四个参数的值），不同编号的车队中的四个参数需要满足的条件见表 2 中的第四列（即参数需满足的条件 1）、第五列（即参数需满足的条件）

3、观察车队中车辆的轨迹，如果没有实现编队一致性，则重复步骤 2，直到实现编队一致性

#### 4.4 实验过程

拿表 2 中的第一种情况为例，leader 是领车，按照一定的速度运行（加速度为 0）；follower1 是不网联不受控小车，我们用激光雷达来实现，但是其他小车不能订阅这个车的话题来获取信息；follower2 是网联受控车辆，该车需要启动激光雷达探测 follower1 的位置和速度，并且可以通过 WiFi 互联获取 leader 和 follower3 的位置

表 1 车辆图示说明

领导车辆 (leader)		
跟 随 车 辆 (follower)	网联受控 (CAV)	
	网联不受控 (HDV)	
	不网联不受控 (HDVW)	

和速度信息；follower3 是网联不受控车辆，可以通过 WiFi 互联获取 leader 和 follower2 的位置和速度信息。在这个实验中，我们初步的思路是让四个小车直线行驶，所以为了防止网联数据误差造成小车之间方向偏移过大，我们对每一个 follower 都启动激光雷达来根据前车的方向调整自己的角速度，尽量使得四个小车在同一条直线上，从而可以很大程度上避免小车与周围障碍物（比如墙壁）发生碰撞。

关于算法的设计我们只需要利用前面的两种方法（WiFi 互联、激光雷达）即可，对于 WiFi 互联，我们需要订阅更多的话题，但注意不能订阅非网联车的话题，对于激光雷达，所有的小车都可以启动激光雷达并套用前面的函数探测前后小车的速度和位置信息。这里需要说明一点的是，激光雷达的数据反馈率和 WiFi 互联的反馈率并不一样，WiFi 互联的反馈率是 50Hz，而激光雷达的反馈率为 5Hz，这一点在我们探测速度以及生成加速度的时候需要用到。

## 4.5 实验结果

由于时间的限制以及网络问题，我们未能按时完成本次实验，但是相关的部分代码已经给出，见压缩包或者 GitHub 仓库。

## 5 实验中遇到的问题及解决办法

### 更换路由器之后的配置问题

据说水星、小米、360 的路由器很让开发人员头大（本人家用的小米路由器就出现连接网线之后还 ping 不同的情况）。

可以观看 (/ROS 机器人 NanoRobot/ROS 机器人视频教程)，其中第一个和第二个视频就是配置 WiFi 的教程。更换路由器之后，一般来说主机端，小车端的 IP 地址都会改变，所以会比较麻烦一点。我们应该先看上述的两个视频更改小车端的配置，使得我们可以通过无线方式连接小车。

能够无线连接小车之后，我们会获得小车的 IP 地址，比如说是 192.168.31.106，那么我们在主机端输入命令：

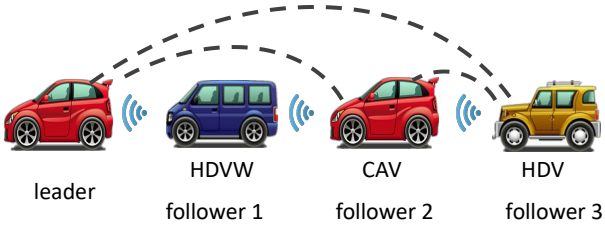
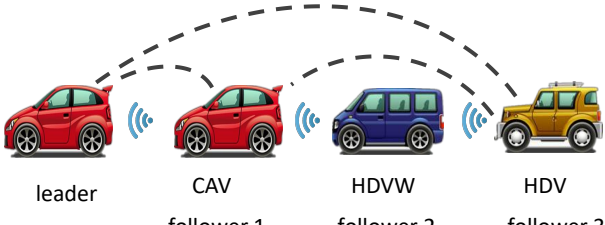
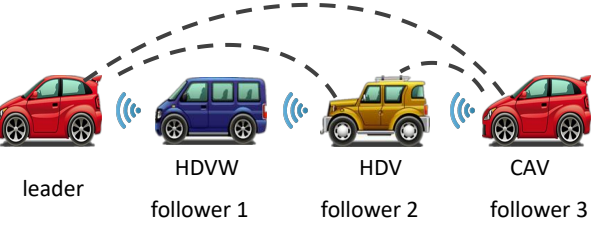
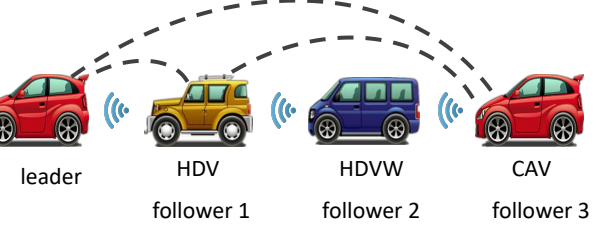
```
ssh nanorobot@192.168.31.106
```

登录小车端，然后参照另一个文档修改小车端的 hosts 和 bashrc 文件。

### 主机 IP 发生变化

主机 IP 地址发生改变之后，我们需要修改所有小车的 hosts 文件，将其中的 master\_ip 更换为新的主机 IP 地址。

表 2 混合车队排列说明

编号	车队排列	数学模型	参数需满足的条件 1	参数需满足的条件 2	参数需满足的条件 3
1	 <p>leader      HDVW      CAV      HDV</p> <p>follower 1      follower 2      follower 3</p>	$\ddot{\tilde{p}}_2 = a_2 = k_{p2}(e_{CAV-P}\tilde{p}_1 - 3\tilde{p}_2 + \tilde{p}_3) + k_{v2}(e_{CAV-V}\dot{\tilde{p}}_1 - 3\dot{\tilde{p}}_2 + \dot{\tilde{p}}_3)$ $\ddot{\tilde{p}}_3 = a_3 = k_{p3}(\tilde{p}_2 - 2\tilde{p}_3) + k_{v3}(\dot{\tilde{p}}_2 - 2\dot{\tilde{p}}_3)$	$w_{11}^3 k_{v2}^2 + (k_{p2}^2 - 6k_{v2}^2 k_{p2})w_{11}^2 - (12k_{p2}^2 k_{v2}^2 + 6k_{p2}^3)w_{11} > 0$ $w_{11} = \sqrt{\frac{6k_{p2}^3 k_{v2}^2 + k_{p2}^4}{k_{v2}^4}} - \frac{k_{p2}^2}{k_{v2}^2}$ $w_{12}^3 k_{v3}^2 + (k_{p3}^2 + 3k_{v3}^4 - 4k_{v3}^2 k_{p3})w_{12}^2 + (6k_{p3}^2 k_{v3}^2 - 4k_{p3}^3)w_{12} + 3k_{p3}^4 > 0$ $w_{12} = \sqrt{\frac{4k_{p3}^3 k_{v3}^2 + k_{p3}^4}{k_{v3}^4}} - \frac{k_{p3}^2}{k_{v3}^2}$	$k_{p2} > k_{p3} > 0$ $k_{v2} > k_{v3} > 0$	
2	 <p>leader      CAV      HDVW      HDV</p> <p>follower 1      follower 2      follower 3</p>	$\ddot{\tilde{p}}_1 = a_1 = k_{p1}(-2\tilde{p}_1 + \tilde{p}_3) + k_{v1}(-2\dot{\tilde{p}}_1 + \dot{\tilde{p}}_3)$ $\ddot{\tilde{p}}_3 = a_3 = k_{p3}(\tilde{p}_1 + e_{HDV-P}\tilde{p}_2 - 3\tilde{p}_3) + k_{v3}(\dot{\tilde{p}}_1 + e_{HDV-V}\dot{\tilde{p}}_2 - 3\dot{\tilde{p}}_3)$	$w_{21}^3 k_{v1}^2 + (k_{p1}^2 - 4k_{v1}^2 k_{p1})w_{21}^2 - 4k_{p1}^3 w_{21} > 0$ $w_{21} = \sqrt{\frac{4k_{p1}^3 k_{v1}^2 + k_{p1}^4}{k_{v1}^4}} - \frac{k_{p1}^2}{k_{v1}^2}$ $w_{22}^3 k_{v3}^2 + (k_{p3}^2 - 4k_{v3}^2 k_{p3})w_{22}^2 - 4k_{p3}^3 w_{22} > 0$ $w_{22} = \sqrt{\frac{4k_{p3}^3 k_{v3}^2 + q^4}{k_{v3}^4}} - \frac{k_{p3}^2}{k_{v3}^2}$	$k_{p1} > k_{p3} > 0$ $k_{v1} > k_{v3} > 0$	$0 < e_{HDV-P} < 1.5$ $0 < e_{HDV-V} < 1.5$ $0 < e_{CAV-P} < 1.5$
3	 <p>leader      HDVW      HDV      CAV</p> <p>follower 1      follower 2      follower 3</p>	$\ddot{\tilde{p}}_2 = a_2 = k_{p2}(e_{HDV-P}\tilde{p}_1 - 3\tilde{p}_2 + \tilde{p}_3) + k_{v1}(e_{HDV-V}\dot{\tilde{p}}_1 - 3\dot{\tilde{p}}_2 + \dot{\tilde{p}}_3)$ $\ddot{\tilde{p}}_3 = a_3 = k_{p3}(\tilde{p}_2 - 2\tilde{p}_3) + k_{v3}(\dot{\tilde{p}}_2 - 2\dot{\tilde{p}}_3)$	$w_{31}^3 k_{v2}^2 + (k_{p2}^2 - 6k_{v2}^2 k_{p2})w_{31}^2 - (12k_{p2}^2 k_{v2}^2 + 6k_{p2}^3)w_{31} > 0$ $w_{31} = \sqrt{\frac{6k_{p2}^3 k_{v2}^2 + k_{p2}^4}{k_{v2}^4}} - \frac{k_{p2}^2}{k_{v2}^2}$ $w_{32}^3 k_{v3}^2 + (k_{p3}^2 + 3k_{v3}^4 - 4k_{v3}^2 k_{p3})w_{32}^2 + (6k_{p3}^2 k_{v3}^2 - 4k_{p3}^3)w_{32} + 3k_{p3}^4 > 0$ $w_{32} = \sqrt{\frac{4k_{p3}^3 k_{v3}^2 + k_{p3}^4}{k_{v3}^4}} - \frac{k_{p3}^2}{k_{v3}^2}$	$k_{p3} > k_{p2} > 0$ $k_{v3} > k_{v2} > 0$	$0 < e_{CAV-V} < 1.5$ $e_{HDV-P} < e_{CAV-P}$ $e_{HDV-V} < e_{CAV-V}$
4	 <p>leader      HDV      HDVW      CAV</p> <p>follower 1      follower 2      follower 3</p>	$\ddot{\tilde{p}}_1 = a_1 = k_{p1}(-2\tilde{p}_1 + \tilde{p}_3) + k_{v1}(-2\dot{\tilde{p}}_1 + \dot{\tilde{p}}_3)$ $\ddot{\tilde{p}}_3 = a_3 = k_{p3}(\tilde{p}_1 + e_{CAV-P}\tilde{p}_2 - 3\tilde{p}_3) + k_{v3}(\dot{\tilde{p}}_1 + e_{CAV-V}\dot{\tilde{p}}_2 - 3\dot{\tilde{p}}_3)$	$w_{41}^3 k_{v1}^2 + (k_{p1}^2 - 4k_{v1}^2 k_{p1})w_{41}^2 - 4k_{p1}^3 w_{41} > 0$ $w_{41} = \sqrt{\frac{4k_{p1}^3 k_{v1}^2 + k_{p1}^4}{k_{v1}^4}} - \frac{k_{p1}^2}{k_{v1}^2}$ $w_{42}^3 k_{v3}^2 + (k_{p3}^2 - 4k_{v3}^2 k_{p3})w_{42}^2 - 4k_{p3}^3 w_{42} > 0$ $w_{42} = \sqrt{\frac{4k_{p3}^3 k_{v3}^2 + q^4}{k_{v3}^4}} - \frac{k_{p3}^2}{k_{v3}^2}$	$k_{p3} > k_{p1} > 0$ $k_{v3} > k_{v1} > 0$	



## 小车 IP 发生改变

如果小车之前的 IP 地址为 192.168.31.106，改变之后的 IP 地址为 192.168.31.199，这时候我们就不能再使用下述命令登录小车端了，而是要用新的 IP 地址。

```
ssh nanorobot@192.168.31.109
```

一般情况下小车的 IP 地址不会改变，但也不排除特殊情况，我们可以登录路由器的管理界面查看都有哪些设备连接到了路由器，并排查小车的 IP 地址有没有改变。

## 小车位置数据在程序开始不为默认位置

在我们执行如下命令后，任意触碰小车导致下车的车轮或者车头方向改变都会使得小车发布在 odom 话题上的数据变化，比如我们运行一个程序使得小车前进 0.5m，在程序执行结束后，odom 话题中小车的位置仍然是程序结束前一刻的内容。我们实验设定的初始状态就是四个小车相隔一定的距离，但是小车程序结束之后的相对位置并不是我们期待的（即使我们将小车放回原位置也不行，因为小车是靠轮子电机的积分算出自己的位置的），这就需要我们关闭下述命令的执行，并重新执行该命令，相当于初始化小车数据。

```
roslaunch multi_robot robot_lidar.launch robot_name:=robot_x
```

## 订阅的话题没有数据（触发不了回调函数）

首先在多车系统的配置下，所有的小车都应该是相互 ping 通的（ping 192.168.31.106 等）。这也是困扰我们很久的问题，我个人感觉还是路由器的的问题，在以往的测试中，如果两个小车 ping 不同，就不能互相订阅对方发布的话题（可以订阅但是没有数据）。所以如果出现异常情况，比如说一个小车不动，就可能是这台小车“失联”了。

## 购买新的小车后需要创建新的工作空间与功能包

下述为创建工作空间与功能包的流程，如果想要了解详细的流程可以在哔哩哔哩上搜索古月居 ROS 观看教程，下面给出网址[https://www.bilibili.com/video/BV1zt411G7Vn?p=9&spm\\_id\\_from=pageDriver](https://www.bilibili.com/video/BV1zt411G7Vn?p=9&spm_id_from=pageDriver)。

```
# 创建工作空间
mkdir -p ~/catkin_ws/src
cd ~/catkin_ws/src
catkin_init_workspace

# 编译工作空间
cd ~/catkin_ws/
catkin_make

# 设置环境变量
source devel/setup.bash

# 检查环境变量
echo $ROS_PACKAGE_PATH

# 创建功能包
cd ~/catkin_ws/src
catkin_create_pkg pkg_name std_msgs rospy roscpp

# 编译功能包
```

```
cd ~/catkin_ws
catkin_make
source ~/catkin_ws/devel/setup.bash
```

## 在小车端显示输入的命令有延迟

我们在实验过程中有时候会出现某个小车端很卡顿的情况，比如在小车端输入一个命令，然后这个命令很久才能在电脑屏幕上显示出来，这时候我们可以尝试在主机端和小车端互 ping，查看是否互通，如果不互通就说明出现了问题，如果能 ping 通可能是网络状况不好，这时候我们可以尝试重启小车。

## 6 调试过程中常用的命令

```
# 登录小车端
ssh nanorobot@IP

# 测试与相应的设备是否互通
ping IP

# 查看话题列表
rostopic list

# 打印话题内容（不止是 odom 话题）
rostopic echo /odom

# 查看话题节点的关系图
rqt_graph

# 运行 position.py 程序，dachuang 是功能包，目前四个小车都有一个 dachuang 功能包
roslaunch dachuang position.py

# 启动小车
roslaunch multi_robot robot_lidar.launch robot_name:=robot_x
```