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　5 月 3 日

轮式机器人实验5-8 实验报告

姓名：肖瑞轩　　学号：3180103127

一、实验目的：

1.1实验五：根据虚拟的控制律，下发速度与下速度，从而实现简单的路径跟踪

1.2实验六：在实验4的global_planner的基础上，添加局部规划的动态窗口法dwa，检测是否规划可以成功规划路线，分发速度并避障

1.3实验七：a.在gazebo中载入地图，使用icp匹配的方式获取变换矩阵

b.通过连续的坐标变换获得ICP模型下的小车定位位置

1.4实验八：a.整理icp.py用于本次实验

b.填写ekf.py以及localization.py两个文件

c.使用拓展卡尔曼滤波器对小车的位置进行预测与更新

二、实验原理与内容

2.1 实验五：

路径跟踪的简单实现方法：不停循环，在即将靠近时切换到下一个路径点，运行跟踪控制器，跟踪当前的目标路径点

简单方法存在的问题：路径点选取过远，路径点选取过近，动态状态物，地图上没有的障碍物等

引入虚拟控制律与实际控制律

虚拟控制律为：$\alpha = arctan(-k_1\beta)$,它可以使得小车运动的角度更接近目标方向的角度

实际的控制律为：

$w = -\frac{v}{\rho}[k_2(\alpha - arctan(-k_1\beta)) + (1 + \frac{k_1}{1+(k_1\beta)^2})sin\alpha]$

采用这样的路径跟踪的优点是：简单，计算较快/参数较少，参数对应的效果比较明确

存在问题：在运动过程中考虑单步的速度和加速度的边界，由于机器人的路径隐式，可能出现多步不满足。在切换目标点时，曲率不连续

2.2 实验六：

动态窗口方法 (Dynamic window Approach，DWA) 是一种常用的避障规划方法。

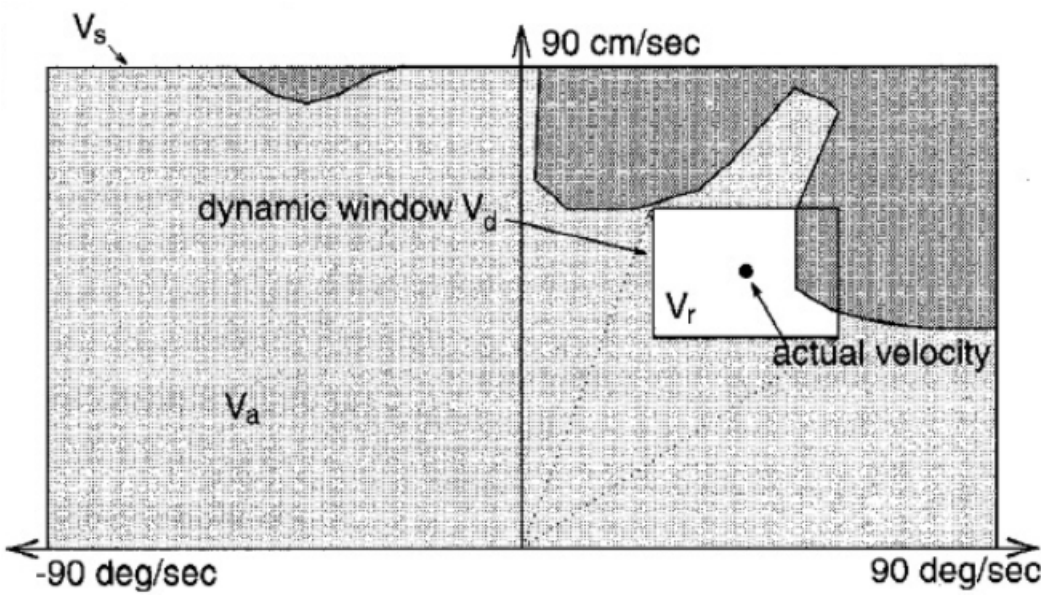
DWA 是一种选择速度的方法。DWA 结合了机器人的动力学特性，通过在速度空间 （v,w）中采样多组速度，并对该速度空间进行缩减，模拟机器人在这些速度下一小段时间间隔内的运动轨迹，机器人的轨迹可以通过分段圆弧（和直线弧）来近似。

当 $w_i = 0$，机器人将沿直线运动。

当 $w_i \neq 0$时，机器人的运动轨迹可被描述为一个圆，满足：

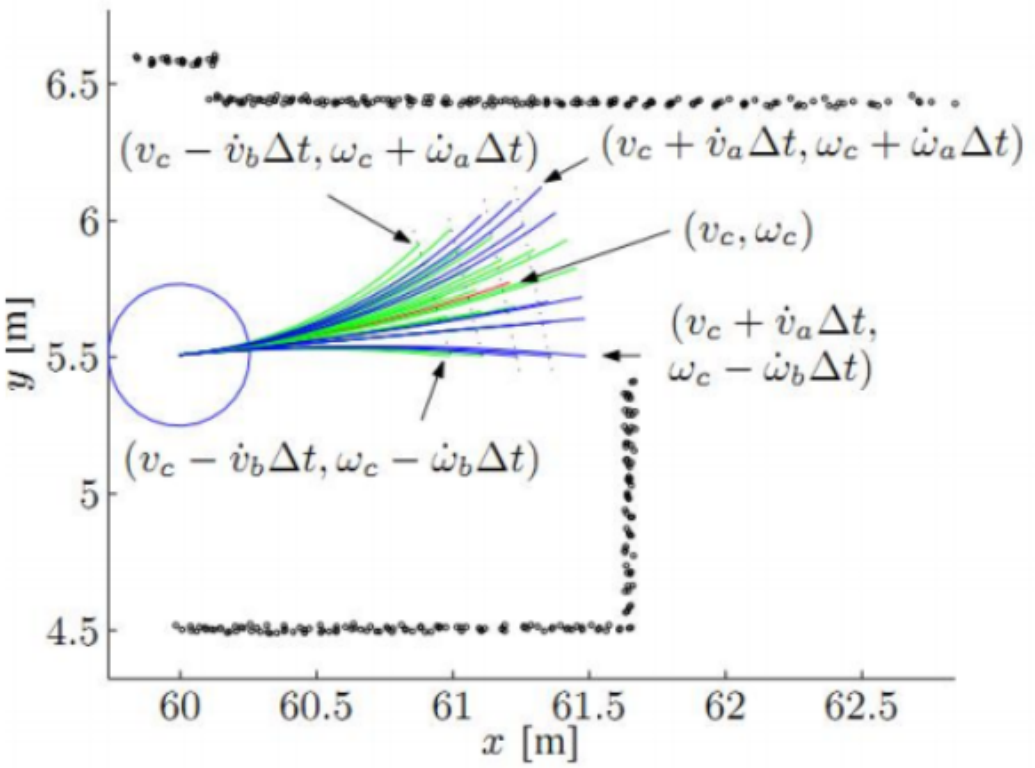
$$M_x^i = -\frac{v_i}{\omega_i} \cdot \sin\theta(t_i)$$
$$M_y^i = \frac{v_i}{\omega_i} \cdot \cos\theta(t_i)$$
$$\left(F_x^i - M_x^i\right)^2 + \left(F_y^i - M_y^i\right)^2 = \left(\frac{v_i}{\omega_i}\right)^2$$

是一个以$M^i(M_x^i, M_y^i)$为圆心的圆，半径为$M_r^i = \frac{v_i}{w_i}$



picture：dwa的速度输入窗口示意

在这之后，我们会根据机器人模型参数中的最高速度，最高角速度，加速度，旋转加速度等参数计算出机器人在一定的模拟时间内速度和角速度可以到达的最大最小范围，将这个窗口范围内的速度（包含角速度）进行采样，输入到三个不同项目的评价函数中，计算出每个采样速度对应的评价值，选出评价最优的一组速度与角速度进行作为当前下发的速度与加速度。



picture：dwa的采样轨迹示意图

其中dwa动态窗口法的伪代码如下：

```
function dwa():
  dw = get_dynamic_window()
  for v in dw:
    _pict_trajectory()
    cost := calc_cost()
    ▷ update best_velocity
  return best_velocity

function calc_cost(goal,obstacles):
    to_goal_cost = dist(me,goal)
    speed_cost = (max_speed - speed_now)
    ob_cost = dist_to_nearest_obstacle()
    return to_goal_cost + speed_cost + ob_cost
```

2.3 实验七：

ICP算法的目的是为了通过把不同坐标系中的点，通过最小化配准误差，变换到一个共同的坐标系中。

ICP算法的基本步骤大致可以分为三步：

a.搜索最近点：取P中一点p，在M中找出距离p最近的m，则（p，m）就构成了一组对应点对集,p与m之间存在着某种旋转和平移关系（R，T）。

b.求解变换关系（R，T）：已经有n对点（p，m），对于n个方程组，那么就一定能运用数学方法求解得出（R，T），但是为了求解更加精确的变换关系，采用了迭代算法。

c.应用变换并重复迭代：如果某次迭代满足要求，则终止迭代，输出最优（R，T），否则继续迭代，但是要注意一点：在每一次迭代开始时都要重新寻找对应点集。也就是说要把结果变换的P_n带入函数E中继续迭代。

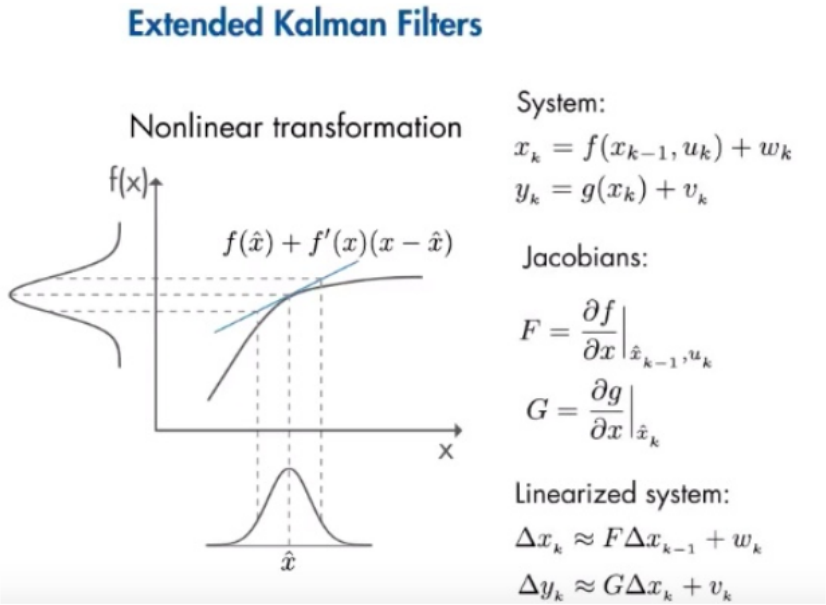
ICP匹配的基本流程可以总结为下图：

picture:ICP基本流程图

2.4 实验八：

扩展卡尔曼滤波（Extended Kalman Filter，EKF）是标准卡尔曼滤波在非线性情形下的一种扩展形式，它是一种高效率的递归滤波器（自回归滤波器）。

EKF的基本思想是利用泰勒级数展开将非线性系统线性化，然后采用卡尔曼滤波框架对信号进行滤波，因此它是一种次优滤波。EKF的核心思想是：据当前的"测量值" 和上一刻的 "预测量" 和 "误差",计算得到当前的最优量. 再预测下一刻的量。



picture:扩展卡尔曼滤波器的数学理解

扩展卡尔曼滤波的基本步骤为：

a.进行初始化：初始化扩展卡尔曼滤波器时需要输入一个初始的状态量x_0，用以表示障碍物最初的位置和速度信息，一般直接使用第一次的测量结果。

b.运用预测模型，得到观测的位置信息

c.运用观测模型，得到观测的位置信息

d.通过预测模型与观测模型进行更新，然后重复上述过程。

扩展卡尔曼滤波的过程预测与更新的数学公式写成矩阵形式如下：

预测过程：

$x_p = Fx_t + Bu_t$

$P_p = J_F P_t J_F^T + Q$

更新过程：

$z_p = Hx_p$

$y = z - z_p$

$S = J_H P_p . J_H^T + R$

$K = P_p . J_H^T S^{-1}$

$x_{t+1} = x_p + Ky$

$x_{t+1} = x_p + Ky \; P_{t+1} = (I - KJ_H)P_p$

三、实验环境

ubuntu1604虚拟机、ros与catkin框架、gazebo与rviz仿真软件 numpy与rospy等第三方库等

四、实验方法、步骤与程序代码

4.1 实验五：

实验五的代码工作量相对而言会更少一些，关键的几个步骤都是关于公式的调用与参数的调节过程，需要注意不同角度的定义域要一致，最好按照atan2()的值域设定为（$-\pi, +\pi$）

```
    def planOnce(self):
        self.lock.acquire()
        self.updateGlobalPose()
        target = self.path.poses[self.goal_index].pose.position
        k1=0.3
        k2=1
        dx = target.x - self.x
        dy = target.y - self.y
        if(self.yaw<0):
            self.yaw=2*3.14159+self.yaw
        distp = math.hypot(dx,dy)
        target_angle = math.atan2(dy, dx)
        if(target_angle<0):
            target_angle=2*3.14159+target_angle
        target_theta=(target_angle-self.yaw)
        if target_theta>3.14159:
            target_theta-=2*3.14159
        garma=(k2*(target_theta-math.atan(k1*target_angle))+(1+k1/(1+
(k1*target_angle)**2))*math.sin(target_theta))
        qulv=-garma/distp
        k3=2
        self.vx = k3*distp
        self.vw = k3*garma
        if self.vw>2:
            self.vw=2
            self.vx=0.5
        if self.vw<-2:
            self.vw=-2
            self.vx=0.5
        if self.vx>1:
            self.vx=1
        self.publishVel()

        self.lock.release()
        pass
```

4.2 实验六：

通过圆弧模型计算在一定的速度与角速度下未来的圆弧轨迹的运动函数motion函数如下：

```
def motion(x, u, dt):
    if u[1]!=0:
        x[0] =x[0]+ (u[0]/u[1])*(math.sin(x[2]+u[1]*dt)-math.sin(x[2]))
        x[1] =x[1]+ (u[0]/u[1])*(math.cos(x[2])-math.cos(x[2]+u[1]*dt))
    else:
        x[0] += u[0] * math.cos(x[2]) * dt
        x[1] += u[0] * math.sin(x[2]) * dt
    x[2] += u[1] * dt
```

```
    if(x[2]>math.pi):
        x[2]=x[2]-2*math.pi
    if(x[2]<-math.pi):
        x[2]=x[2]+2*math.pi
    x[3] = u[0]
    x[4] = u[1]
    return x
```

而计算完整的模拟时间predict_time内的小车经过的点集并储存的calc_posmatrix函数如下

```
def calc_posmatrix(xinit, v, y, config):

    x = np.array(xinit)
    posmatric = np.array(x)
    time = 0
    while time <= config.predict_time:
        x = motion(x, [v, y], config.dt)
        posmatric= np.vstack((posmatric, x))
        time += config.dt
    return traj
```

通过最大（角）速度，最大（角）加速度划分出可行的速度空间的函数calc_dynamic_window函数如下：

```
def calc_dynamic_window(x, config):
    Vs = [config.min_speed, config.max_speed,
          -config.max_yawrate, config.max_yawrate]
    Vd = [x[3] - config.max_accel * config.dt,
          x[3] + config.max_accel * config.dt,
          x[4] - config.max_dyawrate * config.dt,
          x[4] + config.max_dyawrate * config.dt]
    dw = [max(Vs[0], Vd[0]), min(Vs[1], Vd[1]),
          max(Vs[2], Vd[2]), min(Vs[3], Vd[3])]

    return dw
```

对于采样速度的评价与估计汇总函数calc_final_input如下：

```
def calc_final_input(x, u, dw, config, ob):
    xinit = x[:]
    min_cost = 10000.0
    min_u = u
    min_u[0] = 0.0
    mydiction=dict()
    for v in np.arange(dw[0], dw[1], config.v_reso):
        for w in np.arange(dw[2], dw[3], config.yawrate_reso):
            traj = calc_trajectory(xinit, v, w, config)
            mydiction[(v,w)]=traj
            to_goal_cost = calc_to_goal_cost(traj, config) * config.to_goal_cost_gain
            speed_cost = config.speed_cost_gain * \ (config.max_speed - traj[-1, 3])
            ob_cost = calc_obstacle_cost(traj, ob, config) * config.obs_cost_gain
            final_cost = to_goal_cost + speed_cost + ob_cost
            if min_cost >= final_cost:
                min_cost = final_cost
                min_u = [v, w]
    return min_u
```

为了方便结果的可视化分析，我们最好将未来的最优虚度规划出的路径在rviz中显示出来便于进行观察的工作。

4.3 实验七：

实验七的ICP代码部分主要可以分为数据处理、计算部分与接收激光并进行迭代处理的两个部分。

（1）getTransform函数则负责对两组已经一对一匹配好的点云进行SVD分解，找到变换矩阵R与T并返回，在进行SVD分解的时候我们可以直接调用numpy库中的linalg板块的svd函数。代码如下：

```
    def getTransform(self,src,tar):
        T = np.identity(3)
        pm = np.mean(src, axis=1)
        cm = np.mean(tar, axis=1)
        p_shift = src- pm[:, np.newaxis]
        c_shift = tar - cm[:, np.newaxis]
        W = np.dot(c_shift,p_shift.T)
        try:
            u, s, vh = np.linalg.svd(W)
        except Exception as e:
            print(e)
            print(src)
            print(tar)

        R = (np.dot(u,vh)).T
        T = pm - (np.dot(R,cm))
        return R, T
```

（2）对于findnearest函数我们可以有多种方式进行实现：

a.我们可以直接进行for循环的暴力遍历

```
def findNearest_1(self,src,dst):
    for i in src:
        for j in dst:
            find the nearest j to i in dst
            remove matched i,j from origin set to a new set
    remove the matched pairs which distance from each other exceeds threshold
```

这样的循环的优点在于可以方便设立阈值与舍弃掉不需要的点，缺点则是机械循环，在点的数量较多时运算速度会特别慢。

b.查阅numpy的相关资料之后，我们也可以通过对两个矩阵进行repeat与tile两种不同的方式进行复制拓展从而巧妙的实现一种“错位相减”的效果，

```
    def findNearest_2(self,src, dst):
        delta_points = src - dst
        d = np.linalg.norm(delta_points, axis=0)
        deviation = sum(d)
        d = np.linalg.norm(np.repeat(dst, src.shape[1], axis=1)
           - np.tile(src, (1, dst.shape[1])), axis=0)
        orders= np.argmin(d.reshape(dst.shape[1], src.shape[1]), axis=1)
        return orders, deviation
```

c.为了优化运算的速度，我们也可以调用机器学习数据集处理sklearning库中的neighbors模块中的的寻找K近邻回归的kneighbors方法来帮我们进行最近邻的匹配

```
def findNearest_3(self,src, dst):
    assert src.shape == dst.shape
    neigh = NearestNeighbors(n_neighbors=1)
    neigh.fit(dst)
    distances, indices = neigh.kneighbors(src, return_distance=True)
    return distances.ravel(), indices.ravel()
```

我们也可以调用二叉查找的KDtree进一步优化搜索的速度，帮我们进行找到最近邻匹配，

（3）每次接受激光主题laser_callback的主回调函数如下：

```
    def laserCallback(self,msg):
        if self.isFirstScan:
            self.tar_pc = self.laserToNumpy(msg)
            self.isFirstScan = False
            self.laser_count = 0
            return
        self.laser_count += 1
        if self.laser_count <= 5:
            return
        self.laser_count = 0
        time_0 = rospy.Time.now()
        self.src_pc = self.laserToNumpy(msg)
        print('input cnt: ',self.src_pc.shape[1])
        transform_acc = np.identity(3)
        m = self.src_pc.shape[1]
        mysrc= np.copy(self.src_pc)
        mydst= np.copy(self.tar_pc)
        prev_error = 0
        iter_cnt=0
        for _ in range(self.max_iter):
            indexes,error= self.findNearest(mydst, mysrc)
            R,T= self.getTransform(mydst[:, indexes],mysrc)
            mysrc = (np.dot(R,mysrc)) + T[:, np.newaxis]
            self.H = self.update_homogeneous_matrix(self.H, R, T)
            dError = abs(prev_error - error)
            if dError < self.tolerance:
                break
            prev_error = error
            iter_cnt += 1
            pass
        T=self.H
        transform_acc=T
        print("total_iter: ",iter_cnt)
        self.tar_pc = self.laserToNumpy(msg)
        self.publishResult(transform_acc)
        time_1 = rospy.Time.now()
        print("time_cost: ",time_1-time_0)
        self.laser_count =0
```

需要注意的是，由于我们将每次的ICP变换矩阵（3*3）与之前的运动变换矩阵直接乘在了一起，因此实际上我们需要的定位信息中的角度，x，y等已经是矩阵中的值，不用再进行其他转换。

4.4 实验八：

（1）整理好ICP相关的代码，新建ICP类，整理出一个process的接口提供给localization进行调用

（2）新建EKF类,将拓展卡尔曼滤波器预测观测更新的过程放到类中的函数进行处理，将处理的函数接口提供给localization调用

通过v w的预测与观测模型如下：

```
    def odom_model(self,x,u):
        F = np.array([[1, 0, 0],
                      [0, 1, 0],
                      [0, 0, 1]])

        B = np.array([[self.DT * math.cos(x[2, 0]), 0],
                      [self.DT * math.sin(x[2, 0]), 0],
                      [0.0, self.DT])
        x = F.dot(x) + B.dot(u)
        return x


    def observation_model(self,x):
        A = np.array([
            [1, 0, 0],
            [0, 1, 0]
        ])
        r = np.dot(A,x)
        return r
```

对于观测模型与预测模型的雅克比矩阵的计算如下：

```
    def jacob_f(self,x,u):
        yaw = x[2, 0]
        v = u[0, 0]
        jF = np.array([
            [1.0, 0.0, -self.DT * v * math.sin(yaw), self.DT * math.cos(yaw)],
            [0.0, 1.0, self.DT * v * math.cos(yaw), self.DT * math.sin(yaw)],
            [0.0, 0.0, 1.0, 0.0]])
        return jF

    def jacob_h(self):
        jH = np.array([
        [1, 0, 0],
        [0, 1, 0] ])
        return jH
```

提供给localization的estimate的接口如下：

```
    def estimate(self,xEst, PEst, z, u):
        #  predict
        x_p = self.odom_model(xEst, u)
        jF = self.jacob_f(x_p, u)      #jF 3*3
        P_p = jF.dot(PEst).dot(jF.T) + R   #3*3
        #  Update
        jH = self.jacob_h()                 #2*3
        z_p = self.observation_model(x_p)   #z_p 2*1
        y = z.T - z_p                #Y 2*1
```

```
        S = jH.dot(P_p).dot(jH.T) + Q   #S 2*2
        K = P_p.dot(jH.T).dot(np.linalg.inv(S))# K 3*2
        xEst = x_p + K.dot(y)                    #K*y   3*1   X  3*1
        PEst = (np.eye(len(xEst)) - K.dot(jH)).dot(P_p)  #PEst  3*3
        return xEst, PEst
```

（3）对localization中不完整的代码部分进行补充

进行激光模拟的LaserEstimation的函数：对于前一个ekf输出的坐标点进行全局的激光模拟，输出一个类型为Laserscan的仿message值，

代码如下，激光相对于小车的角度为(-π,+π)，要注意激光在- + 交界处的特殊处理

```
    def laserEstimation(self,msg,x):
        data=LaserScan()
        data=msg
        data.ranges=list(msg.ranges)
        data.intensities=list(msg.intensities)
        for i in range(0,len(data.ranges)):
            data.ranges[i]=30
            data.intensities[i]=0.5
        for i in range(0,len(self.obstacle)):
            ox=self.obstacle[i,0]-x[0,0]
            oy=self.obstacle[i,1]-x[1,0]
            alpha=math.atan2(oy,ox)-x[2,0]
            rou=math.hypot(ox,oy)
            delta_alpha=math.atan2(self.obstacle_r/2,rou)
            alpha_min=math.atan2(math.sin(alpha-delta_alpha),math.cos(alpha-
delta_alpha))

 alpha_max=math.atan2(math.sin(alpha+delta_alpha),math.cos(alpha+delta_alpha))
            index_min=int(np.floor(abs((alpha_min+math.pi)/msg.angle_increment)))
            index_max=int(np.floor(abs((alpha_max+math.pi)/msg.angle_increment)))
            if index_min<=index_max:
                for index in range(index_min,index_max+1):
                    if data.ranges[index]>rou:
                        data.ranges[index]=rou

            else:
                for index in range(0,index_max+1):
                    if data.ranges[index]>rou:
                        data.ranges[index]=rou
                for index in range(index_min,len(msg.ranges)):
                    if data.ranges[index]>rou:
                        data.ranges[index]=rou
        self.target_laser=data;
        return data;
```

接受激光信息的lasercallback主调用函数如下：

```
    def laserCallback(self,msg):
        print('------seq:  ',msg.header.seq)
        if self.isFirstScan:
             self.lasttime=msg.header.stamp.secs+msg.header.stamp.nsecs/1000000000
             self.isFirstScan = False
             self.laser_count = 0
             return
        self.laser_count += 1
        if self.laser_count <= 20:
```

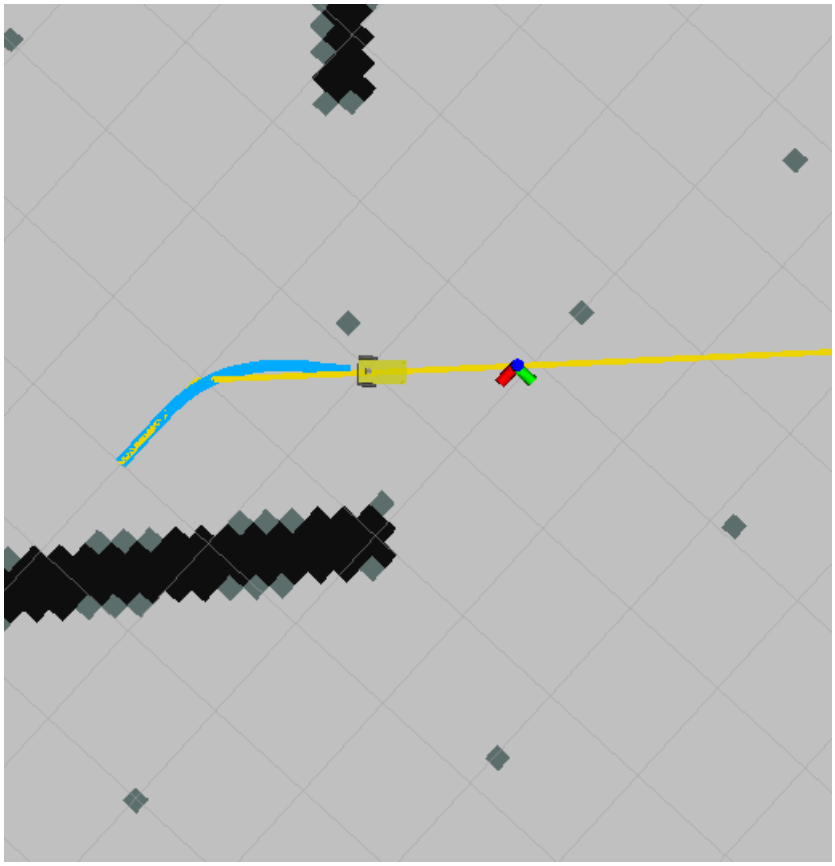
```
            return
        self.laser_count = 0
        tmp1=self.calc_odometry(msg)
        if self.veloacc is None:
            self.veloacc=tmp1
        else:
            self.veloacc=np.dot(self.veloacc,tmp1)
        u=np.zeros((2,1))
        u[0,0]=math.sqrt(veloacc[0,2]**2+veloacc[1,2]**2)/self.ekf.DT
        u[1,0]=math.atan2(veloacc[1,0],veloacc[0,0])/self.ekf.DT
        tmp2=self.calc_map_observation(msg,self.xOdom)
        if self.transform_acc is None:
            self.transform_acc=tmp2
        else:
            self.transform_acc=np.dot(self.transform_acc,tmp2)

        self.xEst[0]=self.transform_acc[0,2]
        self.xEst[1]=self.transform_acc[1,2]
        self.xEst[2]=math.atan2(self.transform_acc[1,0],self.transform_acc[0,0])
        self.xOdom[0]=self.veloacc[0,2]
        self.xOdom[1]=self.veloacc[1,2]
        self.xOdom[2]=math.atan2(self.veloacc[1,0],self.veloacc[0,0])
        z=np.ones((1,2))
        z[0,0]=self.transform_acc[0,2]
        z[0,1]=self.transform_acc[1,2]
        self.xEKF[0,0]=self.xEst[0,0]
        self.xEKF[1,0]=self.xEst[1,0]
        self.xEKF[2,0]=self.xEst[2,0]
        self.xEKF, self.PEst=self.ekf.estimate(self.xEKF, self.PEst, z, u)
        self.publishResult()
        pass
```

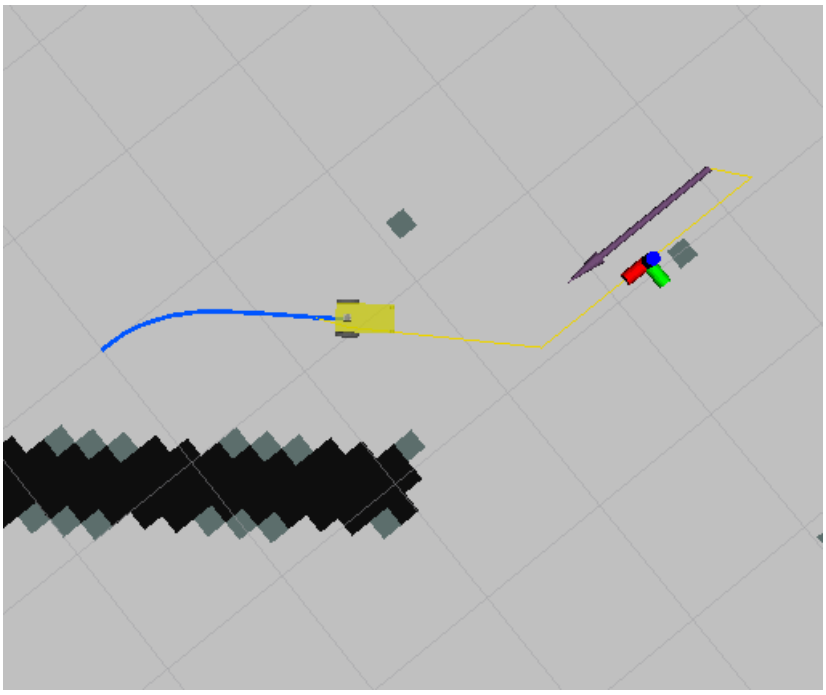
需要特别注意的是对于每一次时序前后的控制以及在进行矩阵赋值与复制时的深浅拷贝的问题。除了时候用Odometry的方式进行展示，我们也可以使用PointCloud的行形式来对两种不同定位方式的结果进行记录散点的展示方式。

五、实验运行结果与分析

5.1 实验五：



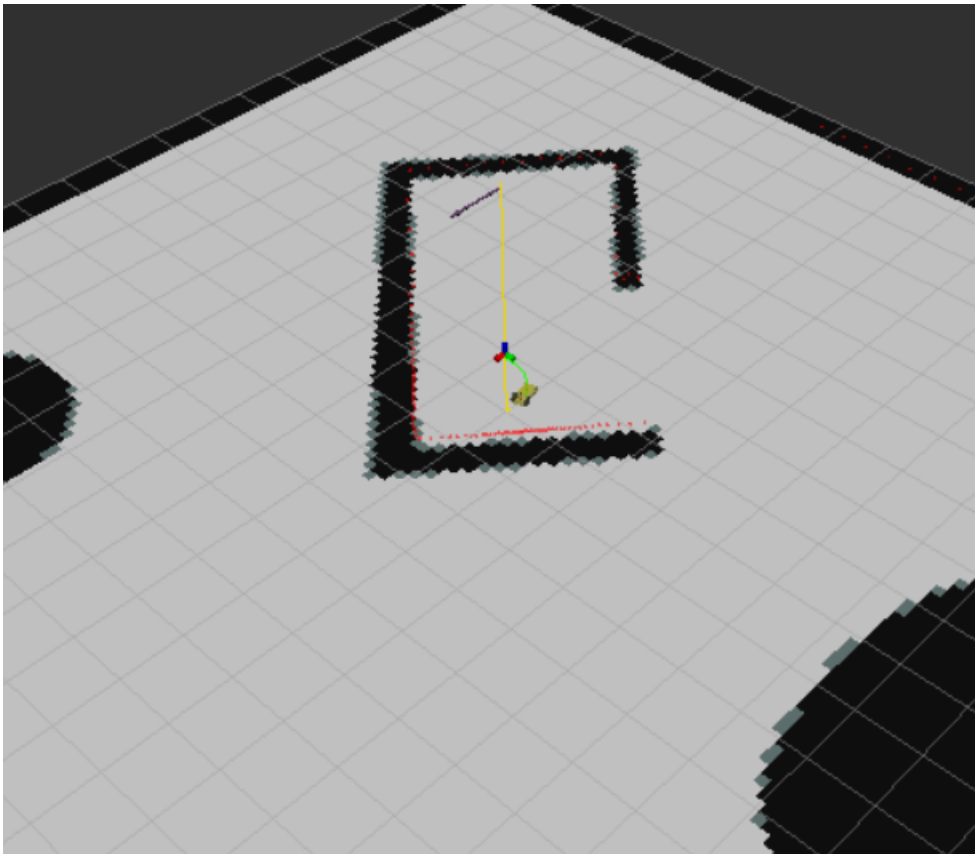
其中蓝色的线段为小车行驶的线段，黄色的线则为全局规划出的路径



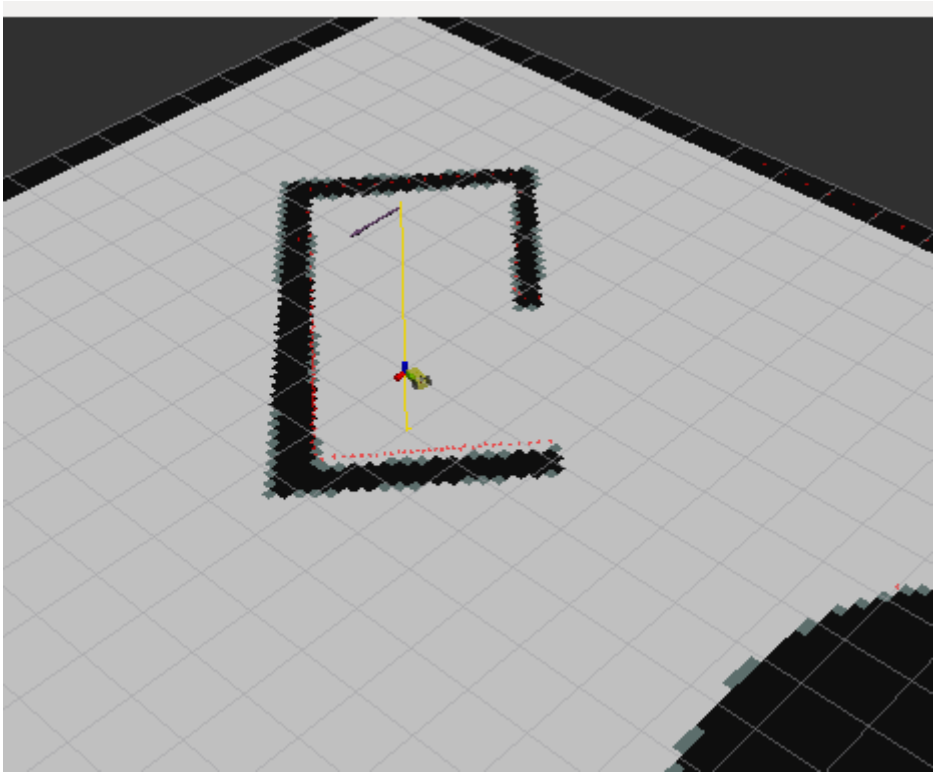
实验五的结果比较简单，小车可以顺利驶向终点，小车行驶的轨迹也较为平滑，而小车的也不会出现需要转弯时先将速度变为0，之后再进行转弯的现象（w4里的stupid_tracking则会出现这样的现象），跟随轨迹的能力强于w4中的stupid_tracking

5.2 实验六:

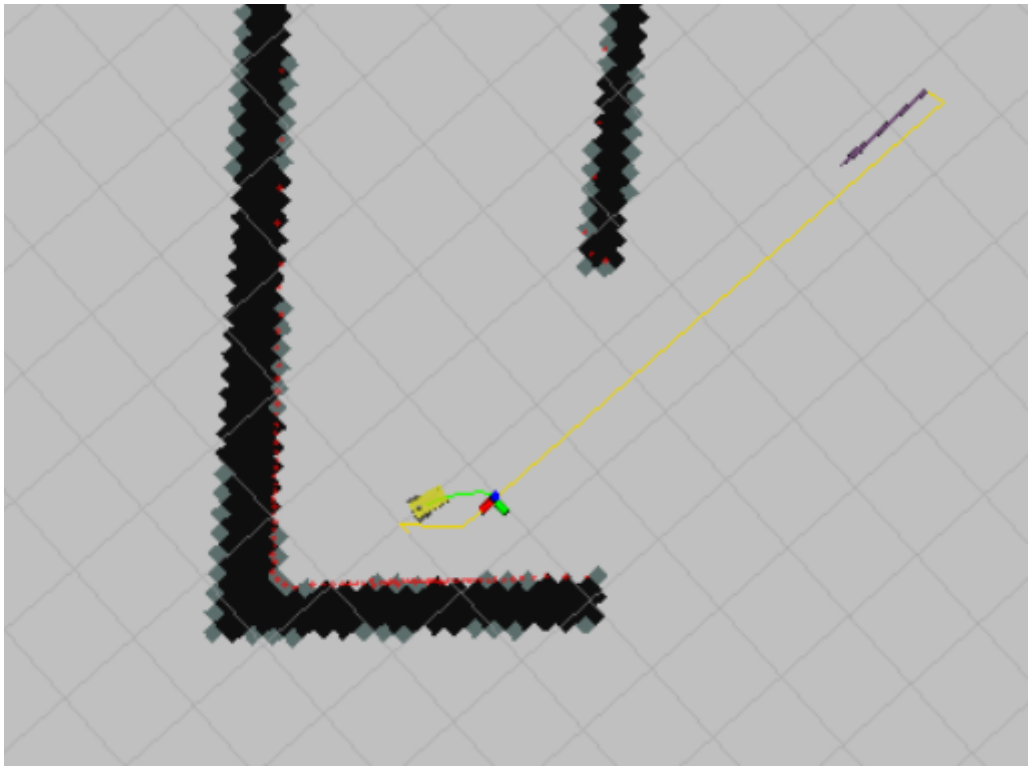
在global planner规划出来的路径上选取间隔格式的点作为每次dwa控制运动跟随的终点，将按照当前选出的最优速度与角速度进行轨迹跟随，其中rviz中绿色的path轨迹是按照当前规划的最优速度与角速度前进时未来的轨迹。



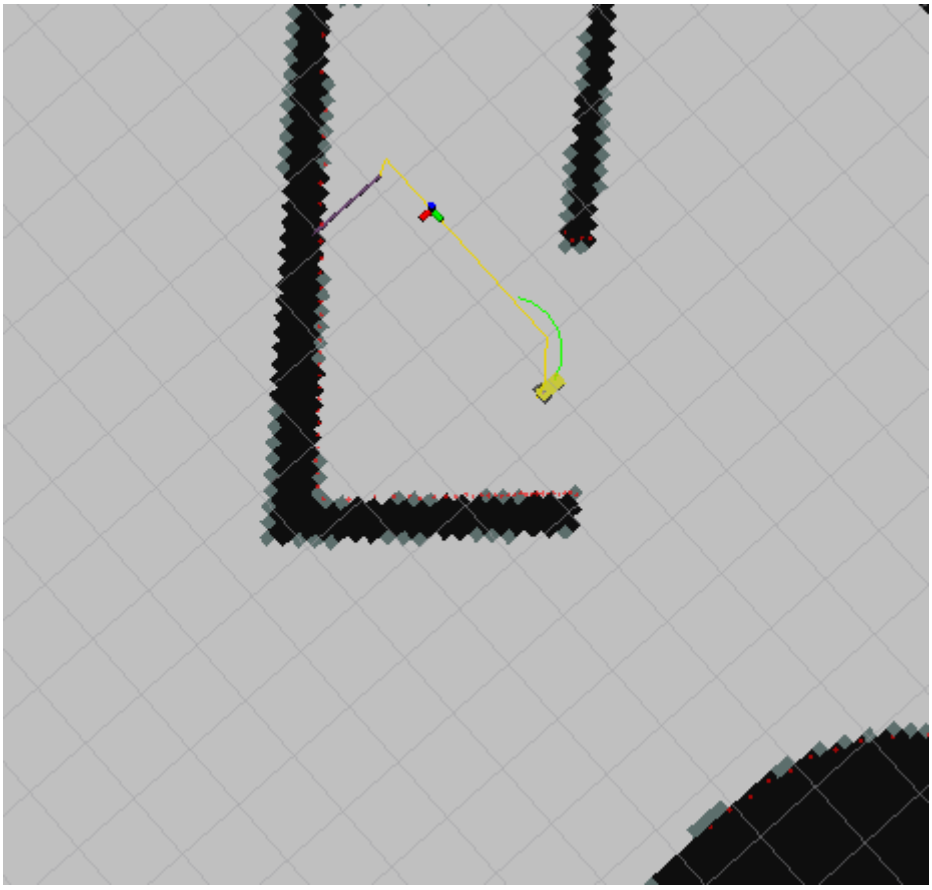
经过一定的运动时间后，可以对规划出的最优路径进行成功的跟随，到达指定的终点



对于**不同的角度、路径方向**、倒向开与正向开都能合理规划与跟随：

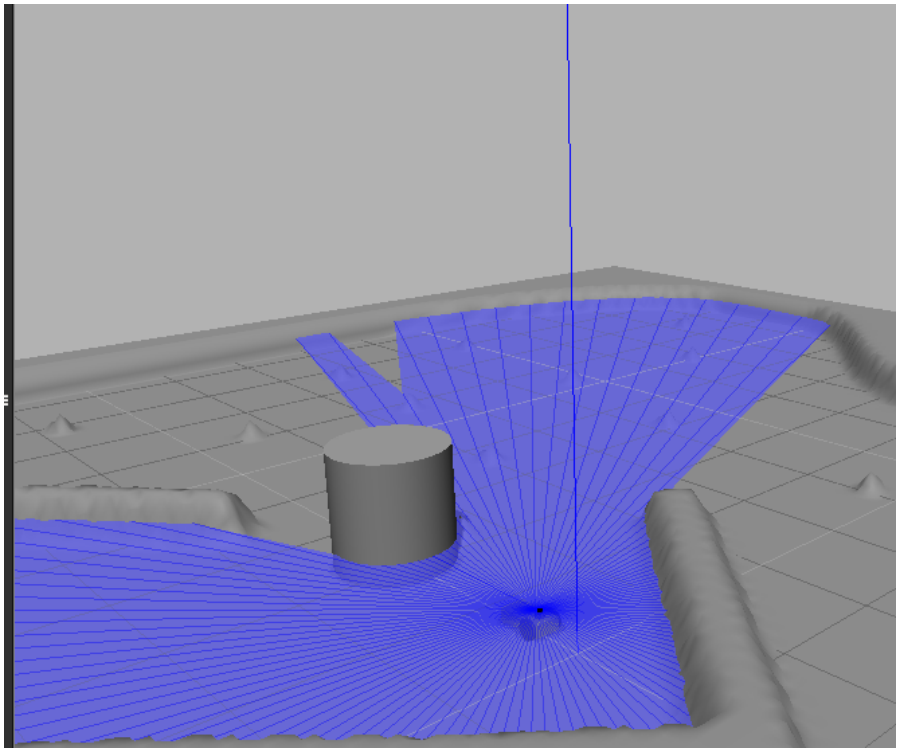


当我们把**目标设置为很远**，超出模拟的最大运动时间时，可以看到，dwa规划可以规划出靠近目标终点的一条光滑路径，达到实验要求

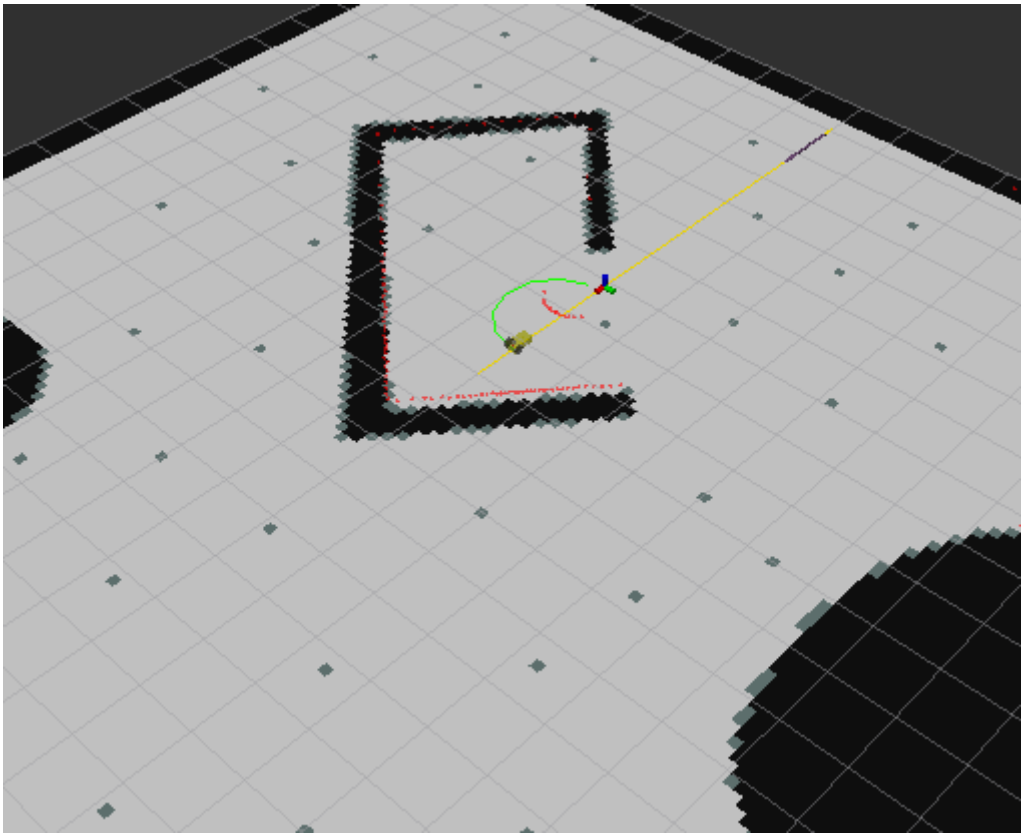


检测添加障碍物之后小车**绕开障碍物**的检测：

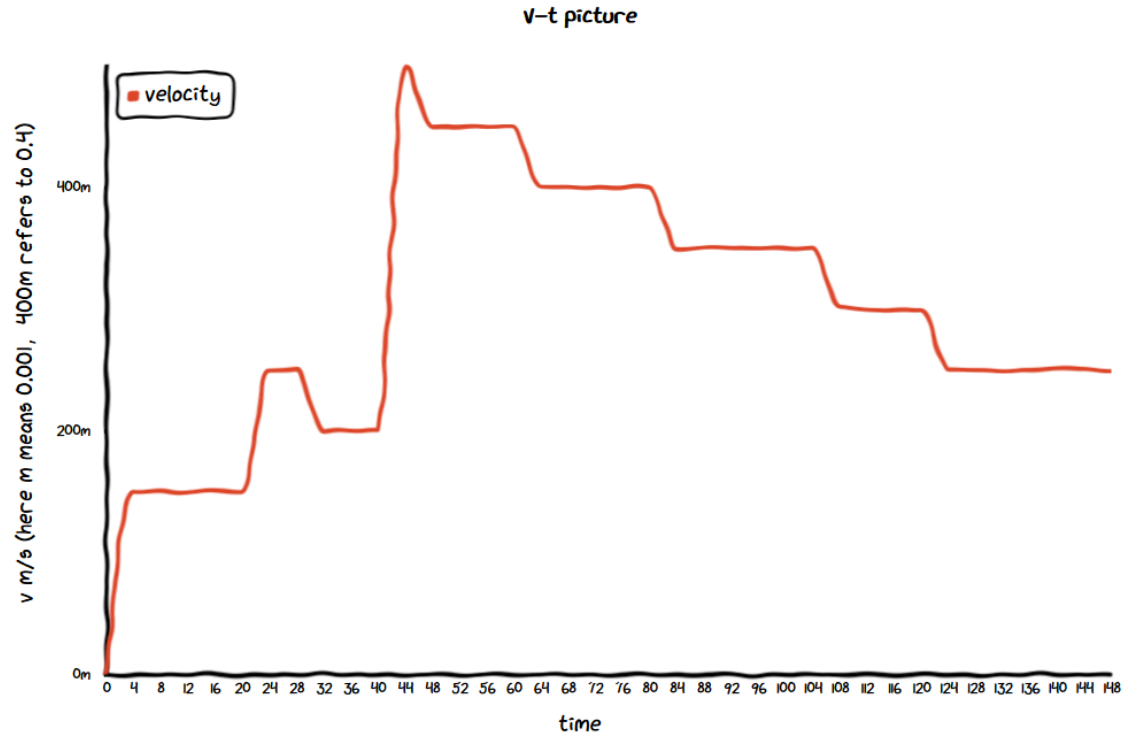
dwa算法相比于全局规划非常重要的一点就是在运动过程中遇到障碍物变化的情况可以“灵活应变”，因此我们需要在实验过程中在gazebo中添加障碍物



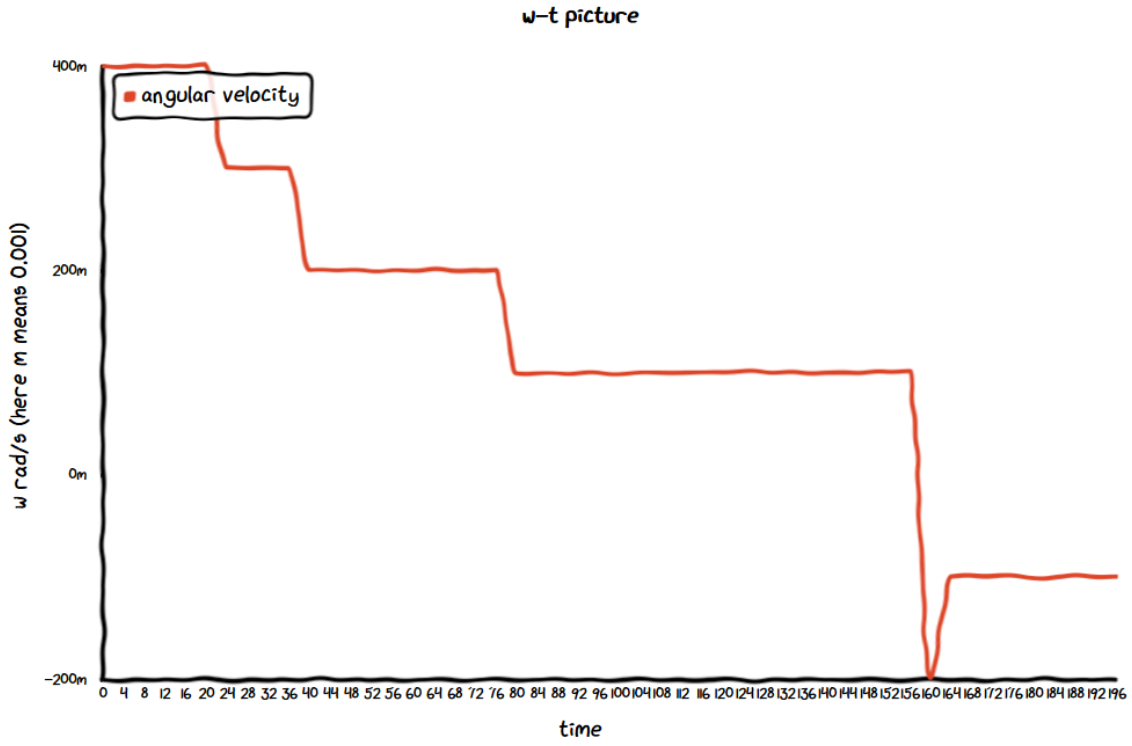
按照原有的规划，走直线是小车的最优路径，添加障碍物之后，小车可以规划出一条绕过障碍物的路线，标明dwa算法拥有良好的绕障碍能力。



对于速度与角速度平滑性的检测：



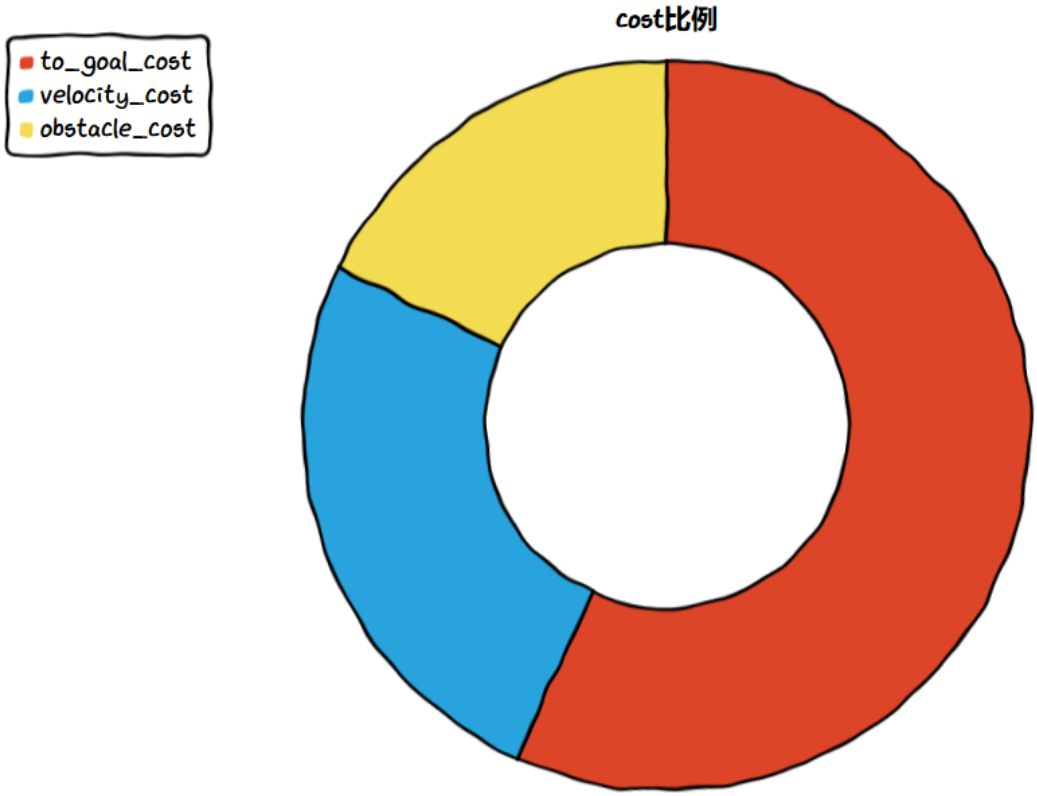
picture：速度v随时间的变化曲线



picture:角速度w随时间的变化曲线

由上图所示，我们可以看到**速度和加速度的变化较为平滑**，由于速度和加速度测试输入的都是离散值，速度的步长为0.05，而角速度的步长为0.1，可以看到在运动的过程中，速度和加速度的变化**大多数都是一个步长的大小**，只有在中间到达某个目标点，下一个目标点相对于之前的运动方向偏移比较多时，才会出现v与w连续性中断的情况（这里v波动了0.3,w波动了0.3),因此我们可以认为我们的dwa算法让v与w相对而言比较平滑，不会出现较大的加速度与角加速度，这也证明了我们在进行输入可行区域的约束时对加速度与角加速度的约束是比较成功的。

接下来我们对之前设定的各种参数带来的三个**cost的大小分配**进行分析，我们对运动过程中三个cost值分别对时间进行求和，算出整个运动过程中的*to_goal_cost*、*velocity_cost*以及*obstacle_cost*的总值，计算其比例分别为 $E_{obstacle\_cost} = 17.88\%$ $E_{togoal\_cost} = 56.77\%$ $E_{velocity\_cost} = 25.44\%$，作图如下，可以知道在这样的参数设置下规划出的路线到目标点的距离影响最大，而*obstacle_cost*看似很小，但由于实际上在小车规划轨迹到障碍物距离小于小车外接圆半径r的时候我们特别地将*obstacle_cost*设为无穷大，并不会影响小车绕过障碍物的性能，同时给to_goal_cost足够大的分配，保证了小车可以在不碰到障碍物的路线中找到最快最近的路。

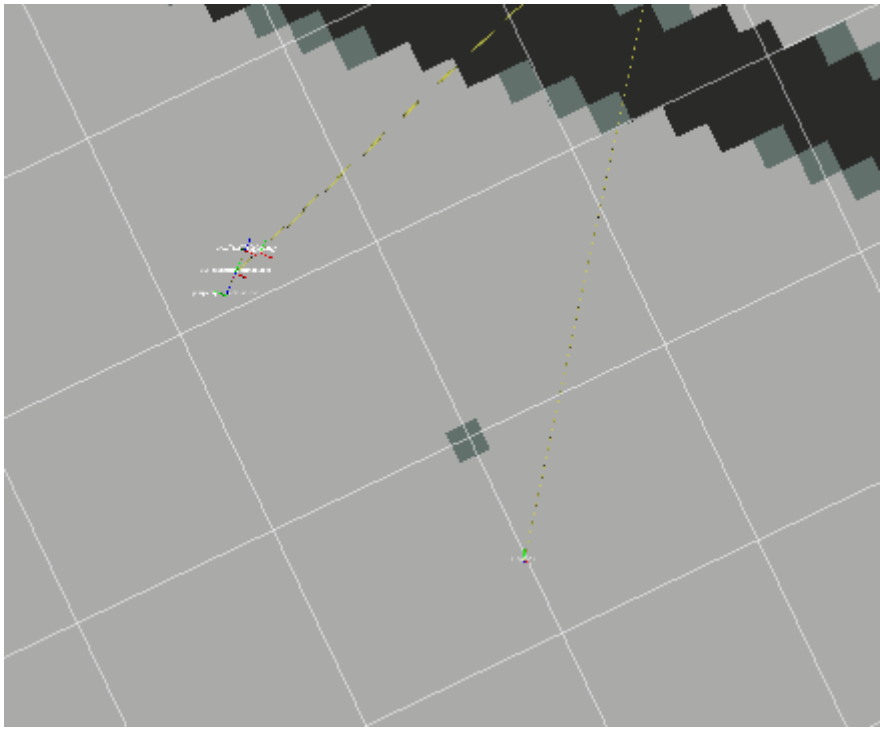


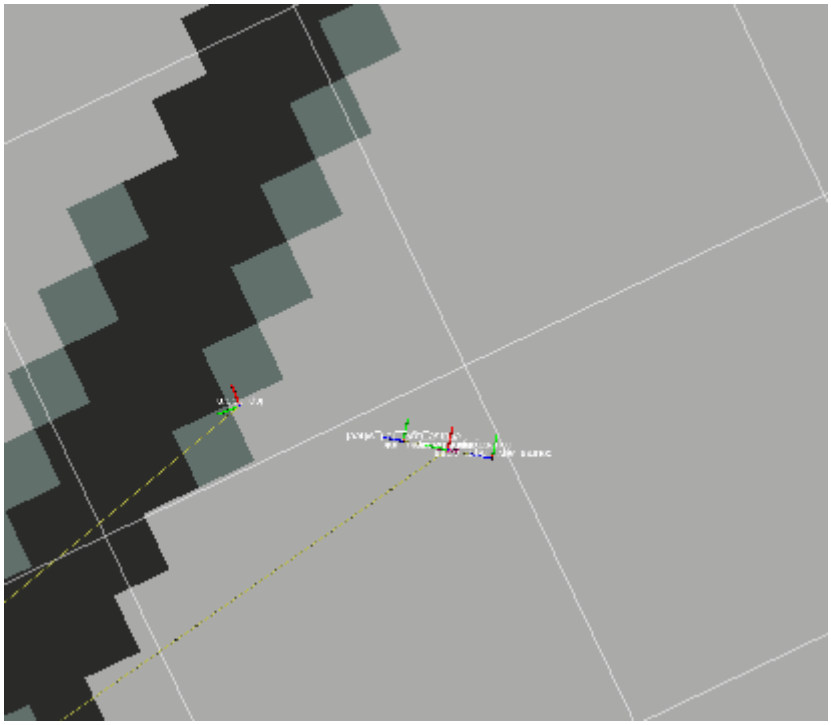
picture:在运动过程中三种cost的求和比例饼状图

5.3 实验七:

由于icp匹配算法在计算时对性能的消耗非常大，因此如果不使用rosbag进行调试可能会出现icp匹配结果相比于小车当前所在的位置会出现较为明显的延后，因此我们需要使用rosbag进行调试的工作。

运动中对于细节放大的展示:

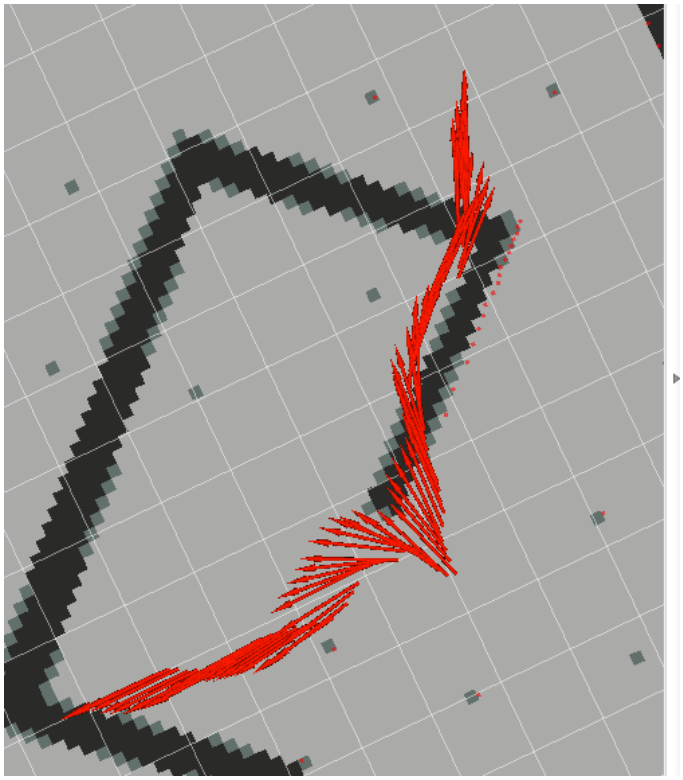




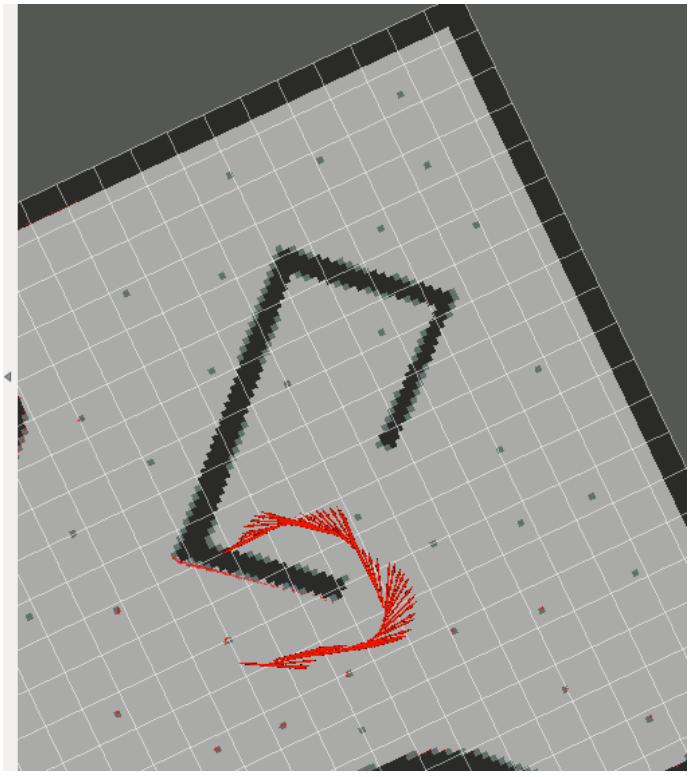
可以看到，在运动过程中，icp相对于真实位置有着明显的侧向偏移，在运动方向上有时候会出现一些滞后的现象。

完整的在过程中的位姿分布如下：

a.使用第一个bag:

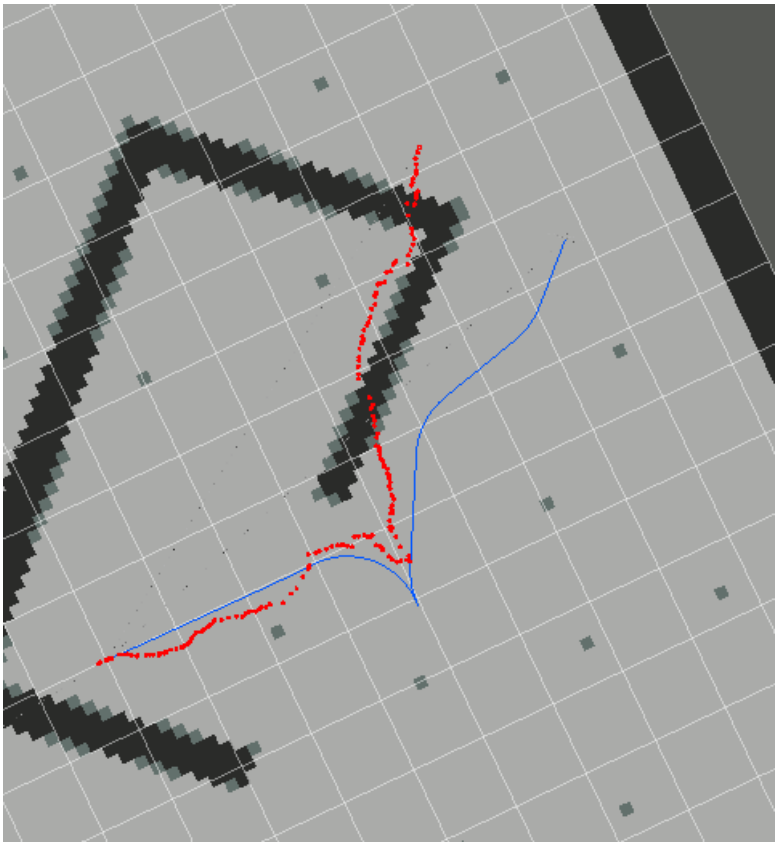


b.使用第二个bag:

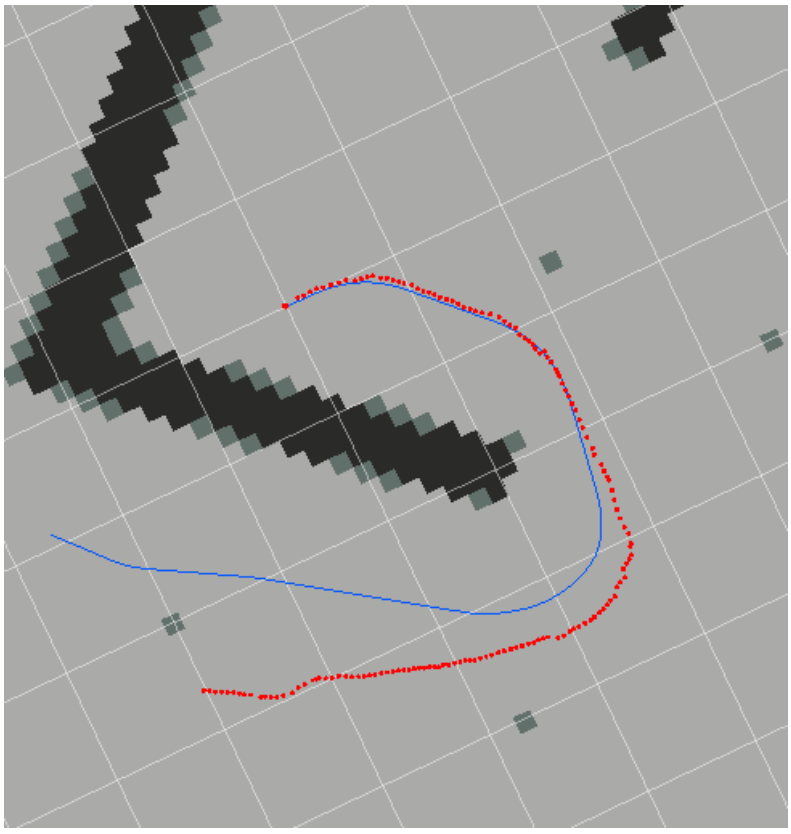


为了更清楚的看到全过程ICP激光里程对真实的路径的跟随情况，我们对小车的真实位置（通过tftransform得到）进行跟踪，使用Path的形式进行输出，表示为图中蓝线，而对ICP的结果进行散点分布跟踪，以PointCloud的形式输出，在图中以红点的方式表示

rosbag1:



rosbag2:

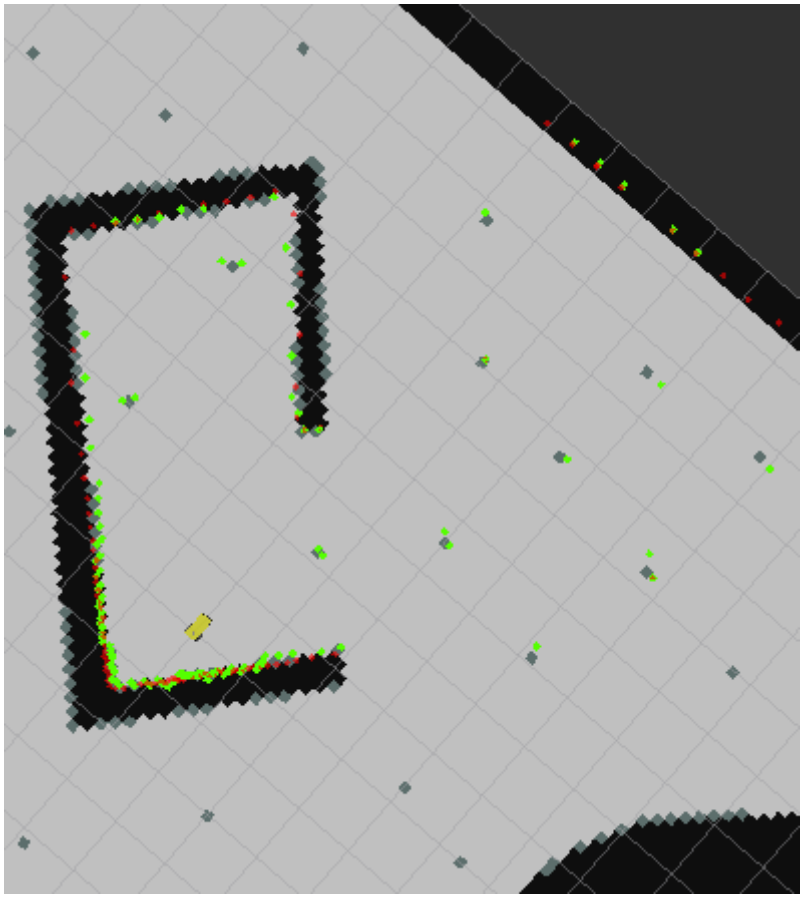


通过全程的轨迹分析，我们可以发现，ICP激光里程对于前期的位置估计较为准确，但在后期累计误差会越来越大，而且在有较大转弯的部分，ICP明显对于转弯的响应会产生较大的误差（表现为轨迹转过的角度没有真实情况多）。

5.4 实验8：

对于实验8，由于ICP运算量进一步加大，为了避免虚拟机性能不足所带来的问题我们同样使用了rosbag对其进行调试。

laser_estimation模拟激光的效果

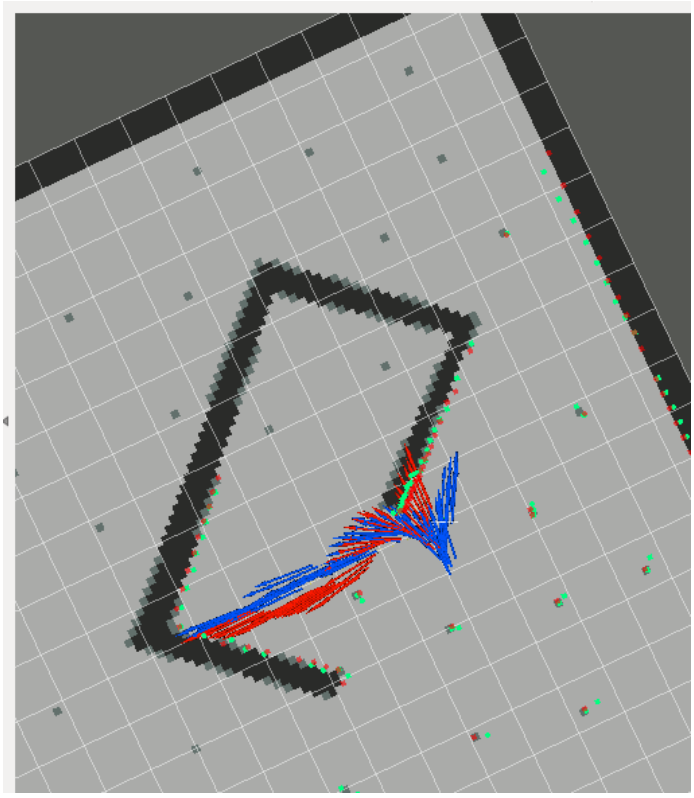


图中绿色的为模拟激光，红色的为真实激光，可以看到激光的模拟较为成功，虽然存在一定的误差，但误差较小，存在误差的原因也可能是因为将地图分解为一个一个像素点，一定程度上破坏了地图的连续性。

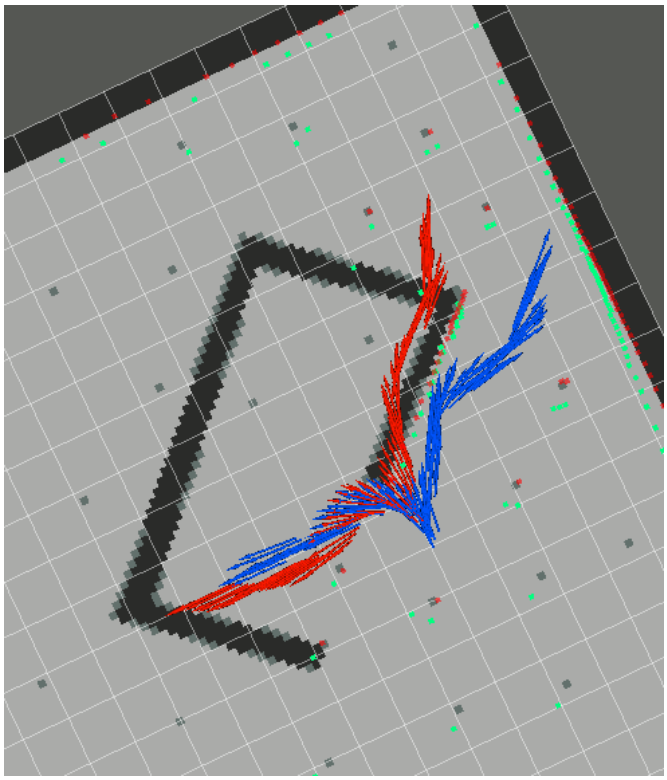
使用第一个rogbag调试的结果：

红色的标记为ICP匹配的结果，蓝色的轨迹标记为ekf输出的结果

a.处在运动中的时候，其中绿色的激光为上一次ekf输出结果的位置的模拟激光，红色的为真实的激光，



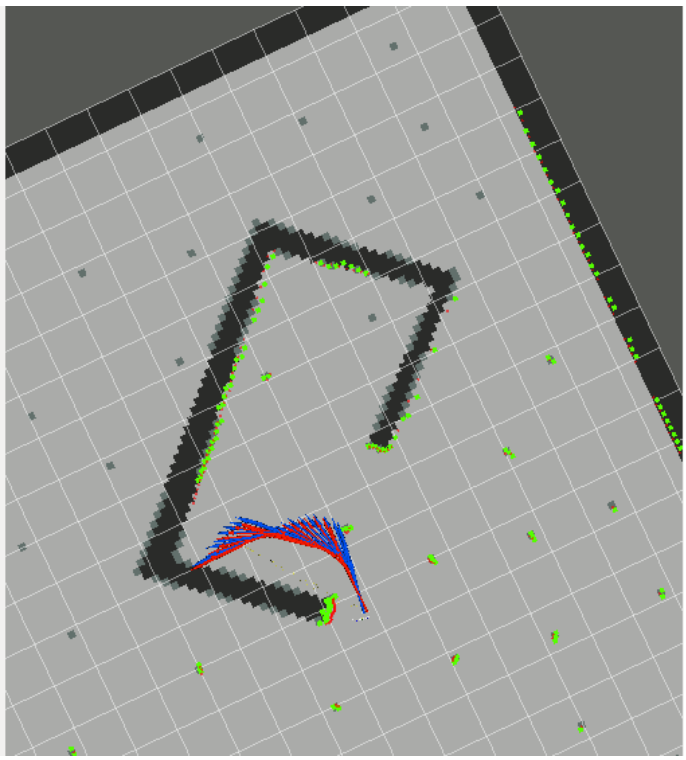
b. rosbag运动结束时：



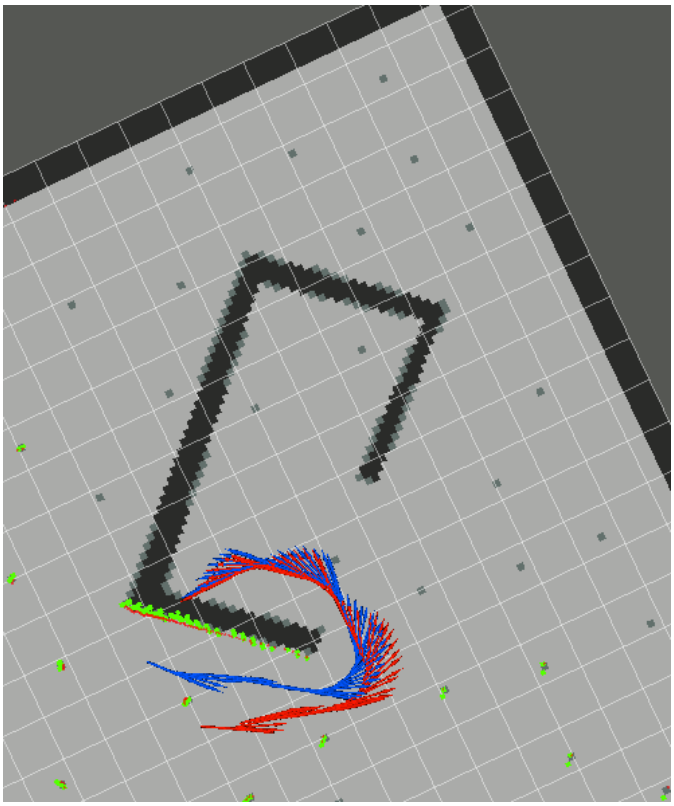
使用第二个rogbag调试的结果：

红色的标记为ICP匹配的结果，蓝色的轨迹标记为ekf输出的结果

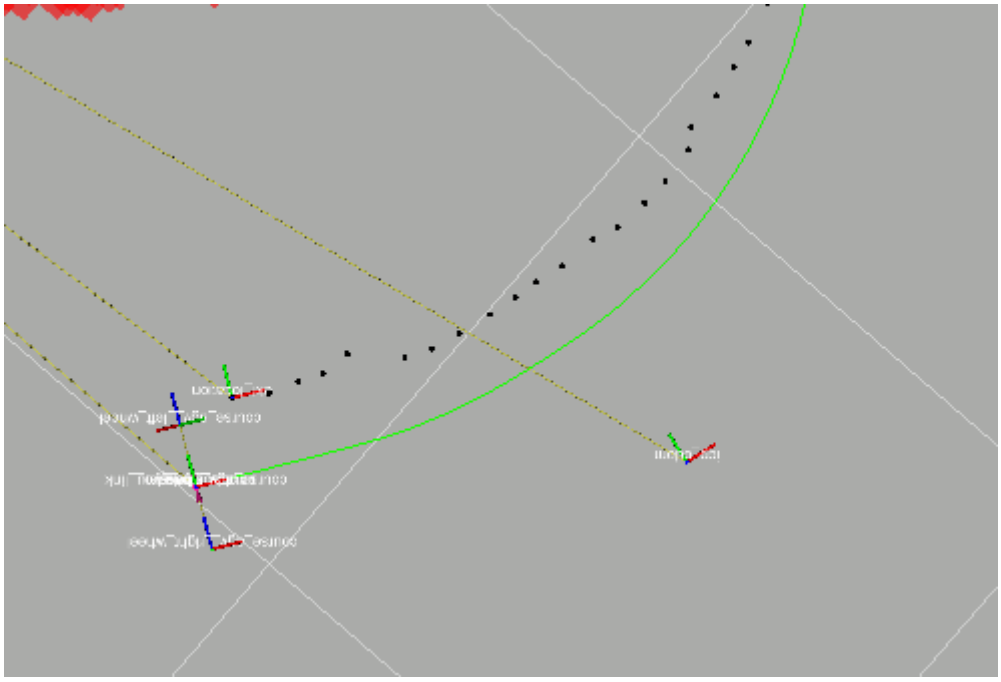
处在运动中，其中绿色的激光为上一次ekf输出结果的位置的模拟激光，红色的为真实的激光，



完成运动之后



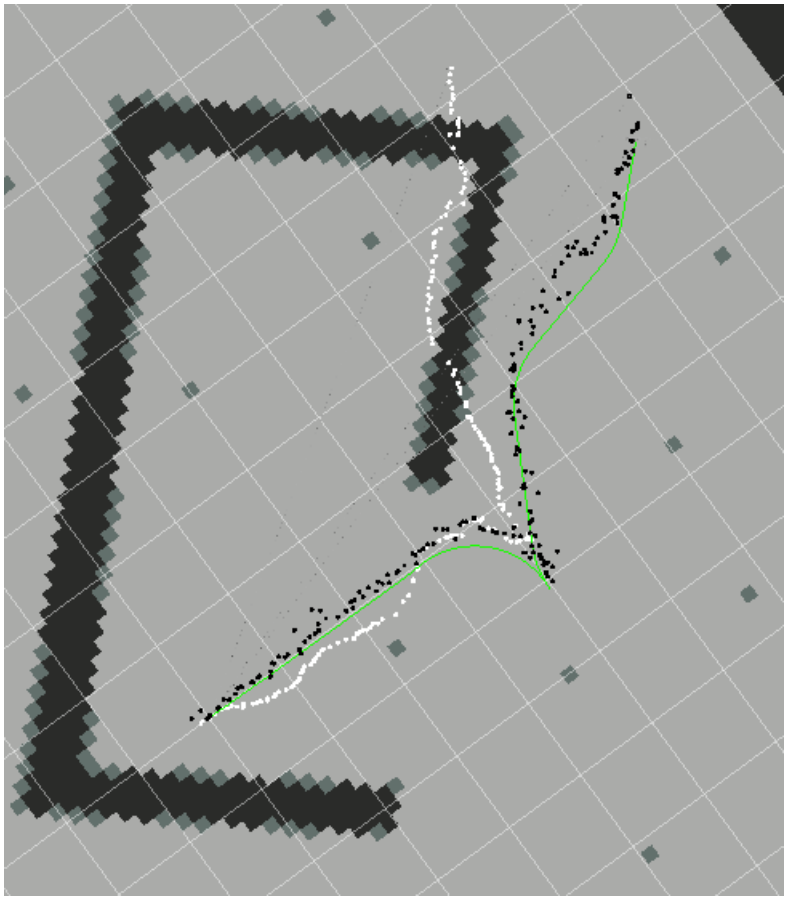
运动中细节大图：



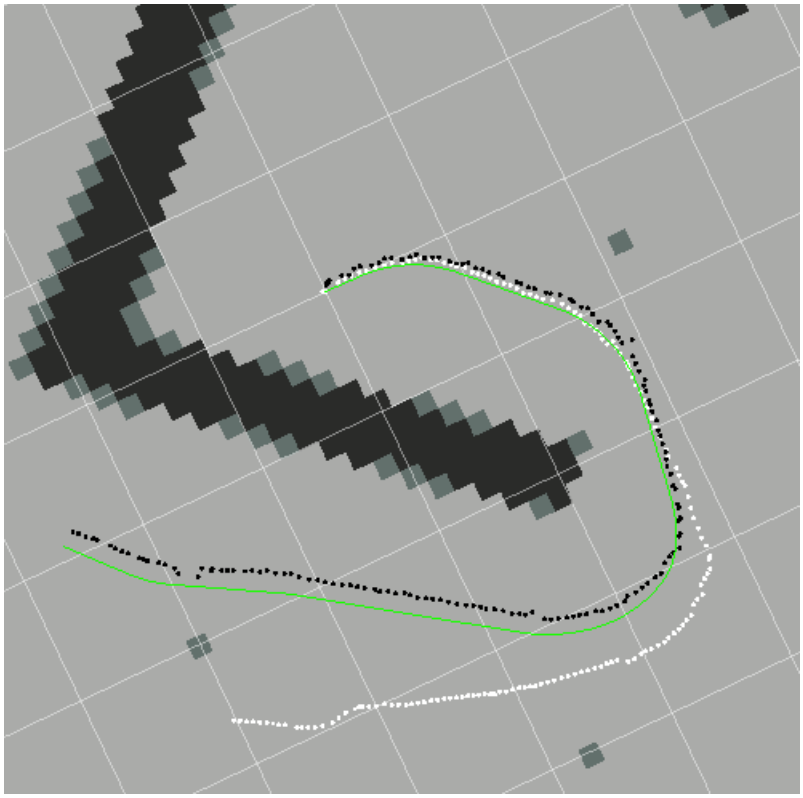
通过对运动中图片的放大（左上为ekf,右下为icp激光里程），我们可以看到ekf定位的结果比较准确，而ICP激光里程不仅有较大的误差，而且在累计跟踪的过程中有较为明显的滞后现象。

对于全程运动过程中的定位跟随，为了更清楚得看到不同方式的定位效果，我们去掉rviz中的激光与小车角度的可视化，将定位的真实路线与icp激光里程以及ekf拟合输出的结果进行路线显示与打点的方式进行展示，其中白色的pointcloud为ICP激光里程的结果，黑色的pointcloud为ekf输出的定位结果，绿色的path则为小车的真实路线

bag1的路线图结果



bag2的路线图结果



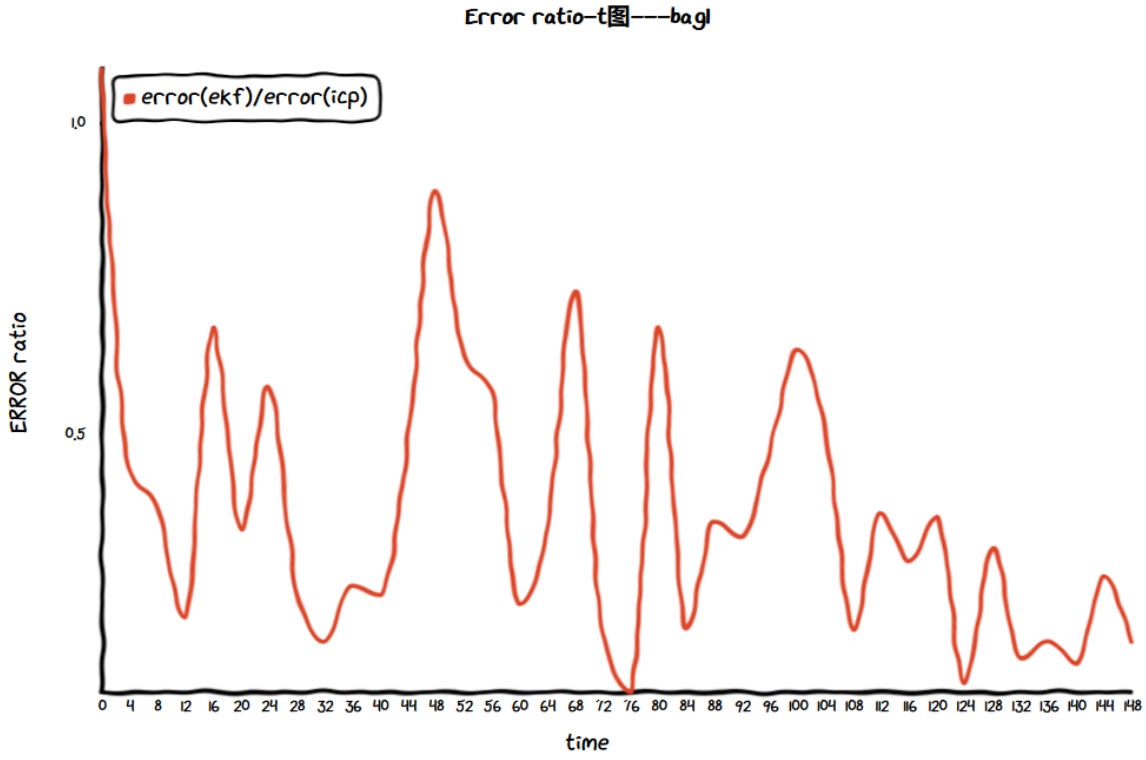
在通过粗略的观察，我们可以明显地看出ICP定位的结果误差明显大于ekf卡尔曼扩展滤波输出的误差，在直线的路段与刚开始的时候，这样的误差并不是特别明显，ICP明显对于转弯的响应会产生较大的误差，且随着路径的增长，到后期ICP的累计误差明显大于ekf滤波的结果

我们对误差的结果进行定量分析，误差即为与真实值的偏移量，我们定义两种定位方式的绝对误差大小为$error = dist(dx, dy) = \sqrt{(x - x_{true})^2 + (y - y_{true})^2)}$

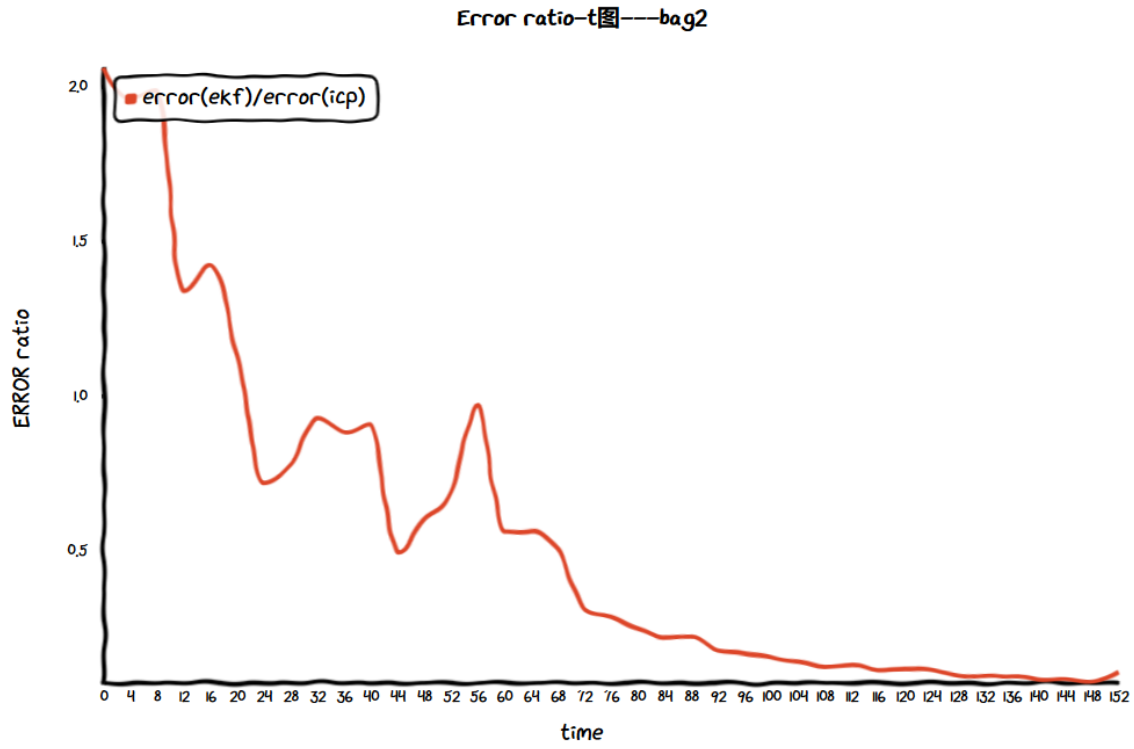
为了对两种定位方式的误差进行比较，我们定义两种定位方式的误差比例$error\ \ ratio = \frac{error_{ekf}}{error_{icp}}$

对全过程的$error\ ratio$进行追踪，并以条形图拟合的形式进行展示

bag1的误差结果:



bag2的误差结果:



ekf与icp激光里程计的定位方式在开始的时候误差差不多，比例都接近于1，甚至ekf在开始的时候误差更大（但开始的时候误差都很小，比值不具有意义），但随着运行时间的增多，与运行里程的增大，efk的误差相对于icp的误差逐渐呈现变小的趋势，到最后的时候error ratio甚至出现了小于0.1的情况。由于是离散取点，可能会出现一些比值的上下波动（特别是bag1）,但是总体的变化趋势符合实验预期。

六、实验总结与心得

6.1 dwa局部规划算法与A*、RRT等全局规划算法的比较

DWA是局部规划器，不过理论上来说完全可以作为独立的路径规划算法，但是局部规划器容易陷入局部最优解出不来，所以一般是需要全局规划器来搭配使用的。因此我们在实验六中采用先全局规划，之后一步步选取全局规划路径上的点进行dwa的控制跟踪。

存在的优点

（1）反应速度较快, 计算较为简单。

（2） 通过速度与角速度可以快速得出下一时刻规划轨迹的最优解，方便进行观察与参数调整

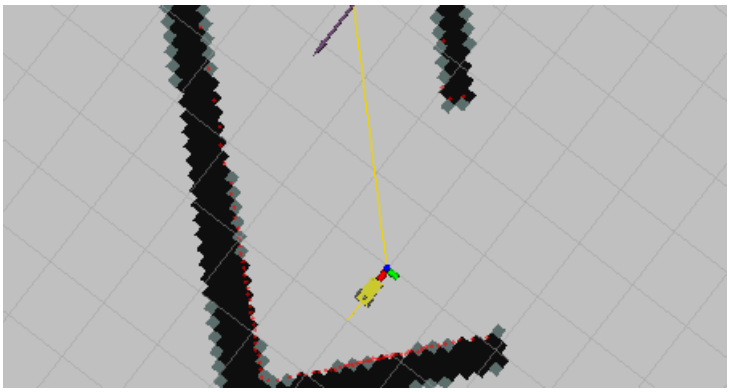
存在的不足

（1）对于局部规划中，较高的灵活性会极大的降低行驶的平稳性。

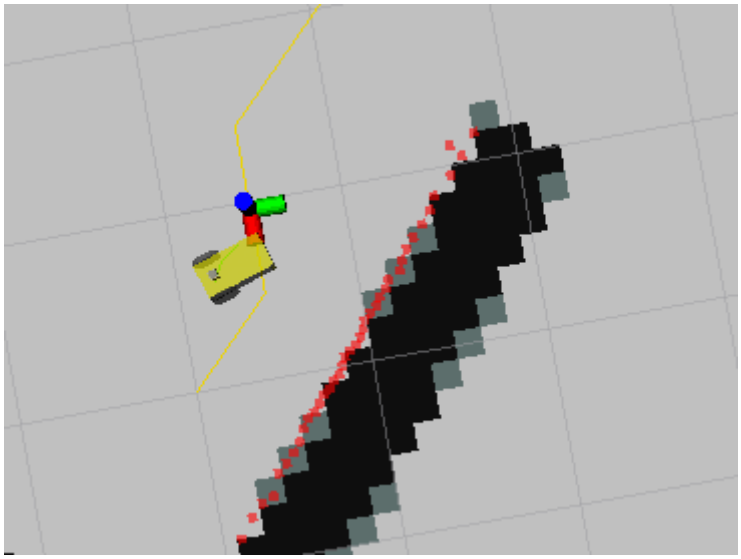
（2）可能需要对卡在局部最优值的情况做特殊的讨论与处理。

6.2 dwa中对下发参数的跟随调试

dwa能起到很好的效果的一个重要前提是publish的参数v与w，能够通过左右轮子的command转动角速度的运动参数可以对规划出的路线进行很好的跟随，要是不能做到很好的跟随，对于v、w与leftwheel/rightwheel command的关系是实验2、3的内容，我们可以通过轮子半径与轮距等参数对运动量下发过程中的参数进行理论上的调整，但实际上我们也需要进行实践的验证，比较方便的方式就是在rviz中显示出完整的规划路线，观察是否能够进行较好的跟随，是否能够完成规划的转弯角度等



picture :较为精确的跟随



picture:存在误差的跟随：小车转过的角度比预期更大:

只有在dwa下发一定的速度之后，运动学分解等实现的效果能够成功跟随dwa规划出的路线，才能为后续dwa的精确的pathtracking打下基础。

6.3 dwa动态窗口法的参数分配与避免局部最优

（1）对于不同cost的计算的分配参数的调整：

对于不同cost的分配，由于找到更快到目标的路总是最重要的，因此togoal_cost应该是占比最多的，而由于obstacle_cost在小车离障碍物距离小于某个阈值时为正无穷，我们加入的比例并不需要太大，当分配给obstacle_cost的参数太大之后，可能会出现小车会更趋向与往离障碍物更远的方向运动而不是朝着目标点运动。

我们也可以考虑加入归一化的处理方法

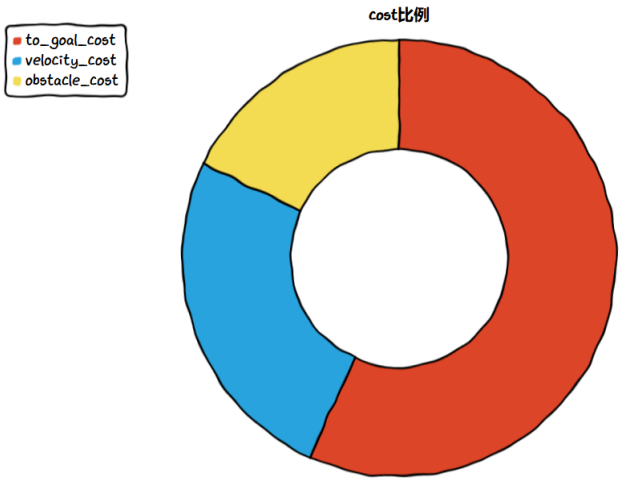
$$normal\ togoal\_cost(i) = \frac{togoal\_cost(i)}{\sum_{i=1}^{n} togoal\_cost(i)}$$

$$normal\ obstacle\_cost(i) = \frac{obstacle\_cost(i)}{\sum_{i=1}^{n} obstacle\_cost(i)}$$

$$normal\ velocity\_cost(i) = \frac{velocity\_cost(i)}{\sum_{i=1}^{n} velocity\_cost(i)}$$

这样做可以让评价标准更加连续，而且不会让某一项在评价函数中占过大的比例

在我们实验六中的比例结果如下图所示：



（2）如何解决可能出现的局部最优值：

dwa是一个localplanner，在极少数情况下，我们可能会观察到机器人进入局部最优值附近的循环中。 如果小车进入了局部最优值且没有合适的路径可以进行规划，则就是这种情况。 当我们发现小车在原理障碍物的地方原地打转时，进行人为的干预即可。

6.4 对ICP激光定位变现出的不足可能的改进方法

由实验7结果中的实验结果可知，单纯地使用ICP激光定位存在许多问题，结果也有一定的误差，我们对改进方法进行专门的讨论。

单纯使用icp激光定位的不足：存在误差，对转弯处反应不足，运动时间变长后累计误差变大，要求数据点云里的每一点在模型点云上都要有对应点，需要遍历点云的每一点，配准速度慢，并且易陷于局部最优解。

可能的改进方法：

（1）对于误差的减小：

a.可以引入基于速度与加速度的里程计，通过速度里程计与激光定位的相互结合与校准来减少误差。

b.可以像实验八中一样，引入绝对观测与卡尔曼滤波器，不只是使用激光里程，使用观测模型与预测模型融合的方式进行预测与更新。

c.对因为小车抖动，发生倾斜带来的激光误差：产生这样的误差的本质原因还是在运动约束与路径跟随中的加速度与速度约束还不够小，从而使激光测试障碍物产生误差，从而影响ICP结果

（2）对于运算速度的优化：

经过相关资料的阅读，对于ICP算法的优化，一种可行的路径是：利用二次曲面逼近方法求每一个点与其邻域的最小二乘拟合平面的法向量（称为这个点的方向矢量）以及曲率，之后再根据曲率确定特征的点集，根据方向矢量进行一一对应关系的查询，这么做虽然数学形式复杂，但却免去的N^2的复杂的遍历与点与点之间两两距离的计算，把目标函数从点云到点云的计算，转变到点云到面的计算，减少了ICP的迭代计算次数，提高ICP算法的速度。

6.5 关于ekf扩展卡尔曼滤波器的扩展学习与讨论

由实验八与加入landmark之后的ekf可知，这样的定位方法相比于单纯的ICP定位已经比较精确，但仍然存在雅克比矩阵（一阶）及海塞矩阵（二阶）计算困难等问题，因此我们仍然需要对ekf相关的一些知识与方法进行简答的学习与讨论

（1）关于卡尔曼滤波与粒子滤波：ekf是通过线性化处理来实现非线性滤波估计，而粒子滤波是利用样本来进行预测，逼近下一次状态，ekf计算比pf更快，但它的性能随着非线性强度变大而明显下降。ekf用高斯分布来预测系统状态。如果系统状态分布不是高斯分布，误差会变大。pf则使用随机样本集，可以用在任何复杂环境下，但代价却是远远更大的计算量。所以，应该根据实际的需要来选择合适的滤波器。

（2）对非线性分布可以考虑使用无损卡尔曼滤波：无损卡尔曼滤波又称无迹卡尔曼滤波（Unscented Kalman Filter），UKF的基本思想是卡尔曼滤波与无损变换，它在卡尔曼滤波的基础上添加了构造Sigma点集，做非线性变换，计算变换后的均值与方法的这样一个UT变换的过程，这样的算法能有效地克服EKF估计精度低、稳定性差的问题，因为不用忽略高阶项，所以对于非线性分布统计量的计算精度高。

实验心得与体会：

实验5-8相对于实验1-4难度上有一个比较大的提升，特别是实验7后期与实验8的全过程，也花费了非常多的精力，特别是一遍遍看着几百行的代码，梳理完整的运行流程却暂时不知道是哪个地方出问题的时候是非常痛苦的，但在发现问题与做出成果之后却是异常快乐与豁然开朗的。

实验5-8除了难度上的提升还有一个很重要的点就是更体现了环环相扣的思想，因为运动学分解的参数调参不够就可能对dwa产生影响，因为dwa对于速度的约束不够就可能使得ICP激光出现波动与漂移的现象，因此最后在之前的基础调试地比较好的基础上再进行后续的工作。

实验5-8除了过程的工作之外，在完成实验要求之后，关于实验结果的分析与比较也是非常重要的一部分，自己在对于后期的数据处理与展示，误差的分析等上面也花费了不少的时间，但看着自己九千多字的报告的完成也是一件非常有意义的事情，希望自己可以在之后的实验中可以吸取之前的一些经验与总结，做出更好的实验成果！