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　7　月　3　日

目录

实验报告4

3180103127　　肖瑞轩

一、实验目的：

完成EKF_slam的定位系统并在ekf_slam定位的基础上进行地图构建的mapping任务。

二、实验原理与内容

2.1 ICP匹配与变换算法

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。

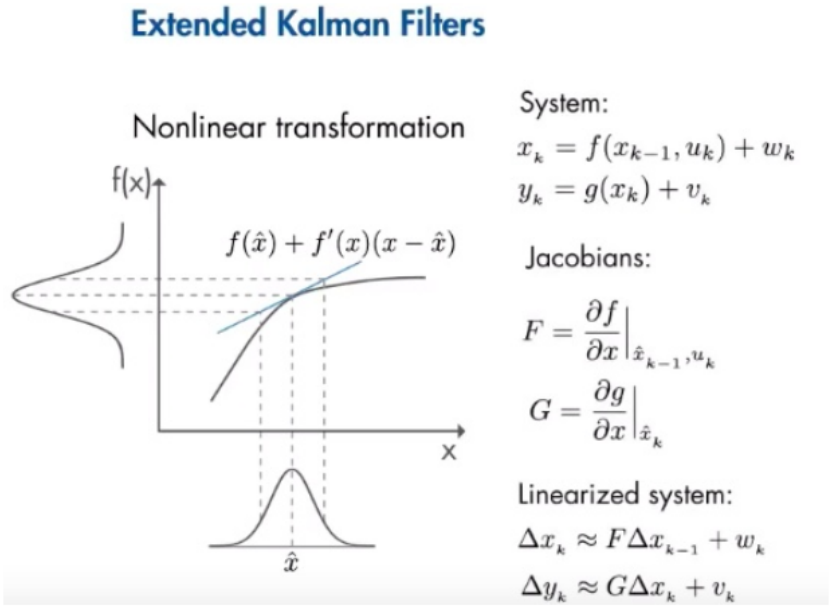
ICP匹配的基本流程可以总结为下图：

picture:ICP基本流程图

2.2 EKF拓展卡尔曼滤波

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。



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数学的矩阵形式如下：

预测过程：

首先进行坐标的预测

$$x_{_p} = Fx_t + Bu_t \tag{1}$$

然后需要求解预测模型的方差：

$$P_{_p} = J_F P_t J_F^T + Q \tag{2}$$

更新过程：

首先我们需要建立观测模型：

$$z_{_p} = Hx_{_p} \tag{3}$$

$$y = z - z_{_p} \tag{4}$$

然后需要对几个系数进行计算

$$S = J_H P_{_p} . J_H^T + R \tag{5}$$

$$K = P_p . J_H^T S^{-1} \tag{6}$$

最后对坐标与方差进行更新

$$x_{t+1} = x_{_p} + Ky \tag{7}$$

$$P_{t+1} = (I - KJ_H)P_{_p} \tag{8}$$

2.3 Landmark的提取

在本次实验中，我们输入到EKF中的ICP匹配变成了地标的提取，我们定义特征是地图上容易被识别的一系列二维点的集合

$$L = \{l_i\} \tag{9}$$

我们在进行推导之前作出假设：假设圆柱直径先验已知，假设激光的间隔至少在一定范围内小于圆柱直径

在激光数据上检测间断区域，将激光分成若干簇，该步骤称为聚类，其中将相邻两帧划到不同的簇的条件为

$$ranges_i - ranges_{i-1} > threshold \tag{10}$$

其中$threshold$为我们设立的临界阈值。

而对于本次仿真实验中运用到的小圆柱，每一簇符合特征筛选的标准为：

计算每一簇的坐标质心为x_{ci}，则满足特征检测的簇的条件为：

$$|X_{ci} - X_0| < 2 \times radius_{max} \tag{11}$$

其中$radius_{max}$为我们设定的最大半径的阈值，之后我们对找到的每一个特征计算其x、y坐标并赋予单独的id(1-N)即可。

2.4 EKF_slam定位算法

EKF_slam的算法主要分为如下的步骤：

1. State prediction
2. Measurement prediction
3. Measurement
4. Data association
5. Update

EKF_slam算法的伪代码如下所示：

$$\begin{aligned}
&\text{Algorithm EKF SLAM}(\mu_{t-1}, \Sigma_{t-1}, u_t, z_t, N_{t-1}):\\
&N_t = N_{t-1}\\
&F_x = \begin{pmatrix} 1 & 0 & 0 & 0\cdots 0 \\ 0 & 1 & 0 & 0\cdots 0 \\ 0 & 0 & 1 & 0\cdots 0 \end{pmatrix}\\
&\bar{\mu}_t = \mu_{t-1} + F_x^T \begin{pmatrix} -\frac{v_t}{\omega_t}\sin\mu_{t-1,\theta} + \frac{v_t}{\omega_t}\sin(\mu_{t-1,\theta} + \omega_t\Delta t) \\ \frac{v_t}{\omega_t}\cos\mu_{t-1,\theta} - \frac{v_t}{\omega_t}\cos(\mu_{t-1,\theta} + \omega_t\Delta t) \\ \omega_t\Delta t \end{pmatrix}\\
&G_t = I + F_x^T \begin{pmatrix} 0 & 0 & \frac{v_t}{w_t}\cos\mu_{t-1,0} - \frac{v_t}{\omega_t}\cos(\mu_{t-1,\theta} + \omega_t\Delta t) \\ 0 & 0 & \frac{v_t}{\omega_t}\sin\mu_{t-1,\theta} - \frac{v_t}{\omega_t}\sin(\mu_{t-1,\theta} + \omega_t\Delta t) \\ 0 & 0 & \end{pmatrix}
\end{aligned} \tag{12}$$

EKF_slam的两步过程：

预测过程prediction

EKF_SLAM_Prediction $(\mu_{t-1}, \Sigma_{t-1}, u_t, z_t, c_t, R_t)$

2 : $F_x = \begin{pmatrix} 1 & 0 & 0 & 0\cdots 0 \\ 0 & 1 & 0 & 0\cdots 0 \\ 0 & 0 & 1 & 0\cdots 0 \end{pmatrix}$

3 : $\bar{\mu}_t = \mu_{t-1} + F_x^T \begin{pmatrix} -\frac{v_t}{\omega_t}\sin\mu_{t-1,\theta} + \frac{v_t}{\omega_t}\sin(\mu_{t-1,\theta} + \omega_t\Delta t) \\ \frac{v_t}{\omega_t}\cos\mu_{t-1,\theta} - \frac{v_t}{\omega_t}\cos(\mu_{t-1,\theta} + \omega_t\Delta t) \\ \omega_t\Delta t \end{pmatrix}$

4 : $G_t = I + F_x^T \begin{pmatrix} 0 & 0 & -\frac{v_t}{w_t}\cos\mu_{t-1,\theta} + \frac{v_t}{\omega_t}\cos(\mu_{t-1,\theta} + \omega_t\Delta t) \\ 0 & 0 & -\frac{v_t}{\omega_t}\sin\mu_{t-1,\theta} + \frac{v_t}{\omega_t}\sin(\mu_{t-1,\theta} + \omega_t\Delta t) \\ 0 & 0 & 0 \end{pmatrix} F_x$

5 : $\bar{\Sigma}_t = G_t\Sigma_{t-1}G_t^T + \underbrace{F_x^T R_t^x F_x}_{R_t}$

矫正过程correction

EKF_SLAM_Correction

6 : $Q_t = \begin{pmatrix} \sigma_r^2 & 0 \\ 0 & \sigma_\phi^2 \end{pmatrix}$

7 : for all observed features $z_t^i = \left(r_t^i, \phi_t^i\right)^T do$

8 : $j = c_t^i$

9:

if landmark j never seen before

10 : $\begin{pmatrix} \bar{\mu}_{j,x} \\ \bar{\mu}_{j,y} \end{pmatrix} = \begin{pmatrix} \bar{\mu}_{t,x} \\ \bar{\mu}_{t,y} \end{pmatrix} + \begin{pmatrix} r_t^i\cos\left(\phi_t^i + \bar{\mu}_{t,\theta}\right) \\ r_t^i\sin\left(\phi_t^i + \bar{\mu}_{t,\theta}\right) \end{pmatrix}$

11: endif

12 : $\delta = \begin{pmatrix} \delta_x \\ \delta_y \end{pmatrix} = \begin{pmatrix} \bar{\mu}_{j,x} - \bar{\mu}_{t,x} \\ \bar{\mu}_{j,y} - \bar{\mu}_{t,y} \end{pmatrix}$

13 : $q = \delta^T \delta$

14 : $\hat{z}_t^i = \begin{pmatrix} \sqrt{q} \\ \operatorname{atan2}(\delta_y, \delta_x) - \bar{\mu}_{t,\theta} \end{pmatrix}$

15:

$$F_{x,j} = \begin{pmatrix} 1 & 0 & 0 & \underbrace{0 \cdots 0}_{2j-2} & 0 & 0 & \underbrace{0 \cdots 0}_{2N-2j} \\ 0 & 1 & 0 & 0 \cdots 0 & 0 & 0 & 0 \cdots 0 \\ 0 & 0 & 1 & 0 \cdots 0 & 0 & 0 & 0 \cdots 0 \\ 0 & 0 & 0 & 0 \cdots 0 & 1 & 0 & 0 \cdots 0 \\ 0 & 0 & 0 & \underbrace{0 \cdots 0}_{2j-2} & 0 & 1 & \underbrace{0 \cdots 0}_{2N-2j} \end{pmatrix}$$

16: $H_t^i = \frac{1}{q} \begin{pmatrix} -\sqrt{q}\delta_x & -\sqrt{q}\delta_y & 0 & +\sqrt{q}\delta_x & \sqrt{q}\delta_y \\ \delta_y & -\delta_x & -q & -\delta_y & +\delta_x \end{pmatrix} F_{x,j}$

17 : $K_t^i = \bar{\Sigma}_t H_t^{iT} \left(H_t^i \bar{\Sigma}_t H_t^{iT} + Q_t \right)^{-1}$

18 : $\bar{\mu}_t = \bar{\mu}_t + K_t^i \left(z_t^i - \hat{z}_t^i \right)$

19 : $\bar{\Sigma}_t = \left(I - K_t^i H_t^i \right) \bar{\Sigma}_t$

20: end for

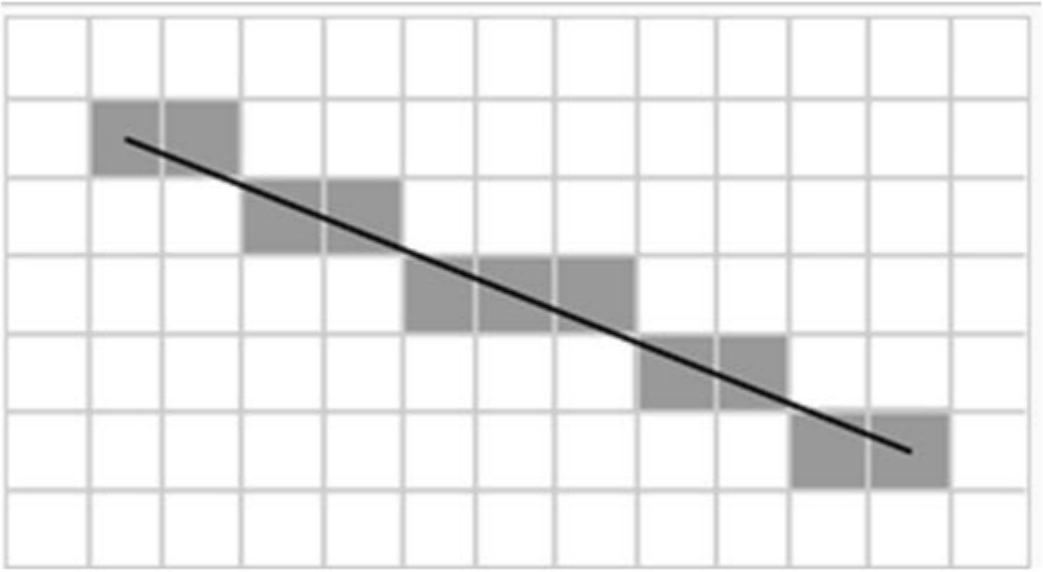
21 : $\mu_t = \bar{\mu}_t$

22 : $\Sigma_t = \bar{\Sigma}_t$

23: return μ_t, Σ_t

2.5 mapping地图构建

Bresenham的直线算法是一种算法，它确定应该选择的n维光栅的点，以便形成两点之间的直线的近似近似。 它通常用于在位图图像中（例如在计算机屏幕上）绘制线基元，因为它仅使用整数加法，减法和位移，所有这些都是标准计算机体系结构中非常便宜的操作。 它是一种增量错误算法。 它是计算机图形学领域最早开发的算法之一。 原始算法的扩展可用于绘制圆圈。



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 Landmark的提取部分

我们需要对激光laserscan的结果进行分簇，然后依次检测每一簇内是否满足要求（），将满足要求的视为提取出的特征，加入Landmarkset中

```
    def process(self,msg,trust = False):
        labels = []
        landmarks_=[]
        if(trust):
            laser_extr=msg
            landmarks_ = self.extractLandMark(laser_extr,labels,trust)
        return landmarks_
```

其中extractLandMark接口如下：

```
    def extractLandMark(self,msg,labels,trust):
        landMarks=LandMarkSet()
        total_num=len(msg.ranges)
        tmpranges=None
        groups=[]
        for i in range(0,total_num):
            tmpangle=msg.angle_min +i*msg.angle_increment

            if(tmpranges is None):
                tmpranges=np.zeros((1,2))
                tmpranges[0,0]=msg.ranges[i]*np.cos(tmpangle)
                tmpranges[0,1]=msg.ranges[i]*np.sin(tmpangle)

            elif(i==total_num-1):
                if(abs(msg.ranges[i]-msg.ranges[i-1])>self.range_threshold):
                    groups.append(tmpranges)
                    tmpranges=np.zeros((1,2))
                    tmpranges[0,0]=msg.ranges[i]*np.cos(tmpangle)
                    tmpranges[0,1]=msg.ranges[i]*np.sin(tmpangle)
                    if(abs(msg.ranges[i]-msg.ranges[0])<=self.range_threshold):
                        groups[0]=np.vstack((groups[0],tmpranges))
                    else:
                        groups.append(tmpranges)
                else:
                    tmpranges=np.vstack((tmpranges,
[msg.ranges[i]*np.cos(tmpangle),msg.ranges[i]*np.sin(tmpangle)]))
                    if(abs(msg.ranges[i]-msg.ranges[0])<=self.range_threshold):
                        groups[0]=np.vstack((groups[0],tmpranges))
                    else:
                        groups.append(tmpranges)

            else:
                if (i>0 and abs(msg.ranges[i]-msg.ranges[i-
1])>self.range_threshold) :
                    groups.append(tmpranges)
                    tmpranges=np.zeros((1,2))
                    tmpranges[0,0]=msg.ranges[i]*np.cos(tmpangle)
                    tmpranges[0,1]=msg.ranges[i]*np.sin(tmpangle)

                else:
```

```
                    tmpranges=np.vstack((tmpranges,
[msg.ranges[i]*np.cos(tmpangle),msg.ranges[i]*np.sin(tmpangle)]))


        idnum=0
        for i in range(0,len(groups)):
            dx=abs(groups[i][0][0]-groups[i][-1][0])
            dy=abs(groups[i][0][1]-groups[i][-1][1])
            if(math.hypot(dx,dy)<2*self.radius_max_th):
                landMarks.position_x.append((groups[i][0][0]+groups[i][-1][0])/2)
                landMarks.position_y.append((groups[i][0][1]+groups[i][-1][1])/2)
                landMarks.id.append(idnum)
                idnum=idnum+1

        #print(landMarks.position_x)
        #print(landMarks.position_y)

        return landMarks
```

4.2 ICP点云匹配与变换部分

（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，代码如下：

```
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()
```

4.3 EKF_slam接口部分：

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_lm的estimate的接口如下：

```
    def estimate(self,xEst,PEst,z,u):
        G,Fx=self.jacob_motion(xEst,u)
        covariance=np.dot(G.T,np.dot(PEst,G))+np.dot(Fx.T,np.dot(self.Cx,Fx))

        # Predict
        xPredict=self.odom_model(xEst,u)
        zEst=self.observation_model(xPredict)
        #self.publishLandMark(zEst,color="r",namespace="estimate",frame="ekf_icp")
        neighbour=self.icp.findNearest(z,zEst)
        zPredict=zEst[:,neighbour.tar_indices]
        zPrime=z[:,neighbour.src_indices]
        length=len(neighbour.src_indices)
        variance=self.alpha/(length+self.alpha)
        if length<self.min_match:
            print("Matching points are too little to execute update.")
            #  no update under this condition.
            return xPredict, covariance


 self.publishLandMark(zPredict,color="g",namespace="paired",frame="ekf_icp")
        m=self.jacob_h(self.tar_pc,neighbour,xPredict)
        # z from(2*n)array to (2n*1) array
        zPredict=np.vstack(np.hsplit(zPredict,np.size(zPredict,1)))
        zPrime  =np.vstack(np.hsplit(zPrime,np.size(zPrime,1)))
        #print("delta z: \n{}\n\n".format(zPrime-zPredict))
        # K factor

 K=np.dot(np.dot(covariance,m.T),np.linalg.inv(np.dot(m,np.dot(covariance,m.T))+np
.diag([variance]*2*length)))
        # update xEst and PEst
        xEst=xPredict+np.dot(K,zPrime-zPredict)
        deltax=np.zeros((3,1))
        PEst=covariance-np.dot(K,np.dot(m,covariance))

        return xEst, PEst
```

对检测到新的landmark需要计算全局坐标并加入已有的landmark集合中

```
            if (len(self.lm_set.position_y)==0):
                tmp_lm=self.u2T(self.xEst[0:3]).dot(self.lm_src_pc)
                for i in range(tmp_lm.shape[1]):
                        self.lm_set.position_x.append(tmp_lm[0][i])
                        self.lm_set.position_y.append(tmp_lm[1][i])
            else:
```

```
                tmp_lm=self.u2T(self.xEst[0:3]).dot(self.lm_src_pc)
                for i in range(tmp_lm.shape[1]):
                    min_dist=10
                    for j in range(len(self.lm_set.position_y)):
                        tmp_dist=math.hypot(tmp_lm[0][i]-
self.lm_set.position_x[j],tmp_lm[1][i]-self.lm_set.position_y[j])
                        if tmp_dist<min_dist:
                            min_dist=tmp_dist
                    if(min_dist>1.5):
                        self.lm_set.position_x.append(tmp_lm[0][i])
                        self.lm_set.position_y.append(tmp_lm[1][i])
```

mapping地图构建中的bresenham算法

```
class bresenham:
    # parameters are simple. there are only 'start' point and 'end' point.
    def __init__(self, start, end):
        self.start = list(start)
        self.end = list(end)
        self.path = []
        self.toggle=0
        # if start point and end point are same, then we don't need to calculate
line.
        if start[0]-end[0]+start[1]-end[1]==0:
            return None
        # we should consider which variable is more steep between x and y.
        # return true if x is more steep than y.
        self.steep = abs(self.end[1]-self.start[1]) > abs(self.end[0]-
self.start[0])
        if self.steep:
            self.start = self.swap(self.start[0],self.start[1])
            self.end = self.swap(self.end[0],self.end[1])
        if self.start[0] > self.end[0]:
            self.toggle=1
            _x0 = int(self.start[0])
            _x1 = int(self.end[0])
            self.start[0] = _x1
            self.end[0] = _x0
            _y0 = int(self.start[1])
            _y1 = int(self.end[1])
            self.start[1] = _y1
            self.end[1] = _y0
        dx = self.end[0] - self.start[0]
        dy = abs(self.end[1] - self.start[1])
        error = 0
        derr = dy/float(dx)
        ystep = 0
        y = self.start[1]
        if self.start[1] < self.end[1]: ystep = 1
        else: ystep = -1
        for x in range(self.start[0],self.end[0]+1):
            if self.steep:
                self.path.append((y,x))
            else:
                self.path.append((x,y))
            error += derr
            if error >= 0.5:
                y += ystep
                error -= 1.0
        if self.toggle==1:
```

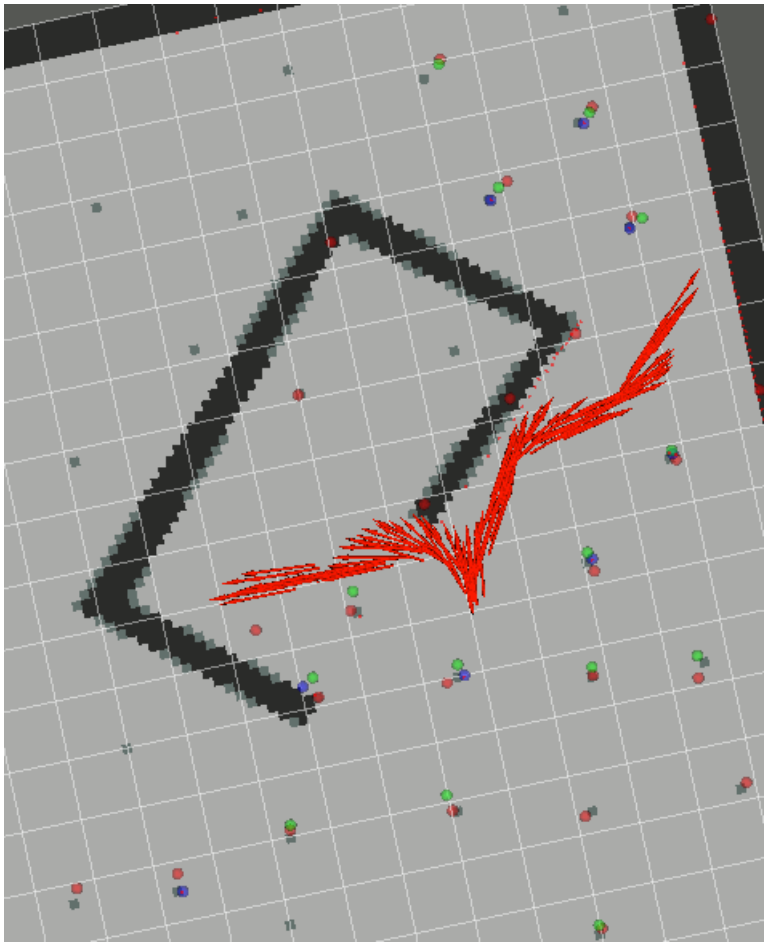
```
            self.path.reverse()
    def swap(self,n1,n2):
        return [n2,n1]
```

五、实验运行结果与分析

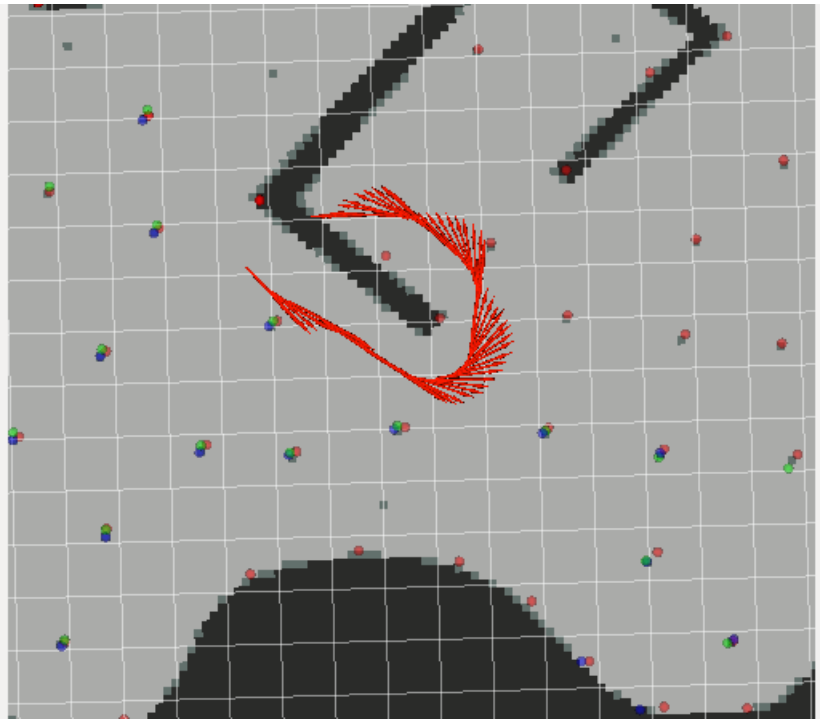
5.1 EKF_slam定位结果与分析：

5.1.1 对于特征点的提取与保存：

rosbag1：



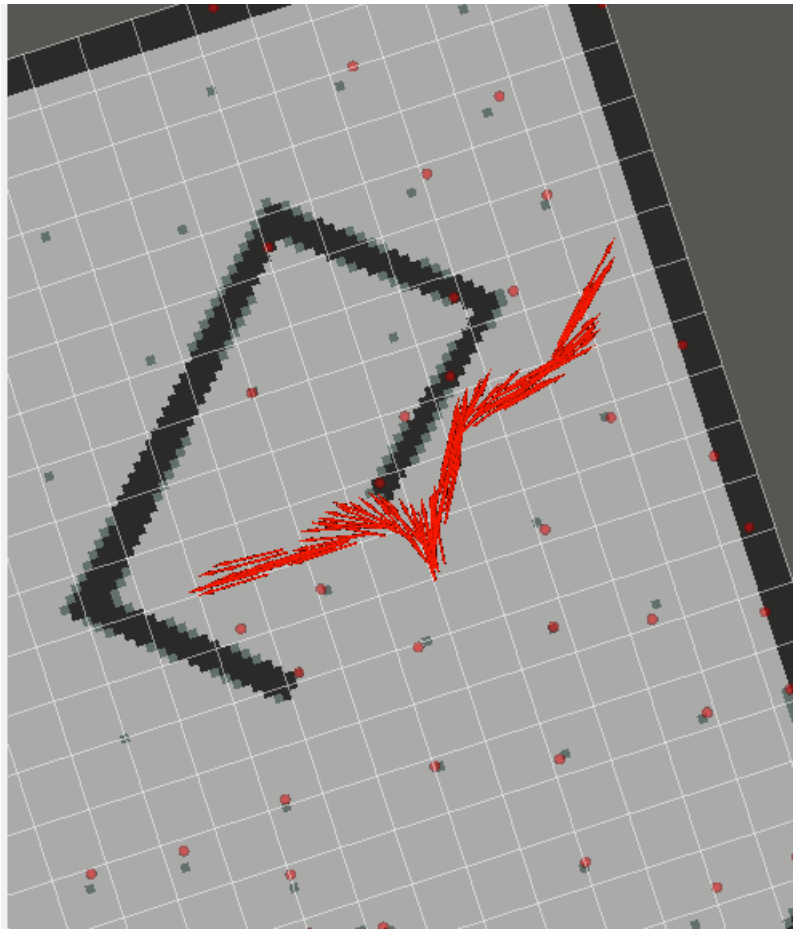
rosbag2



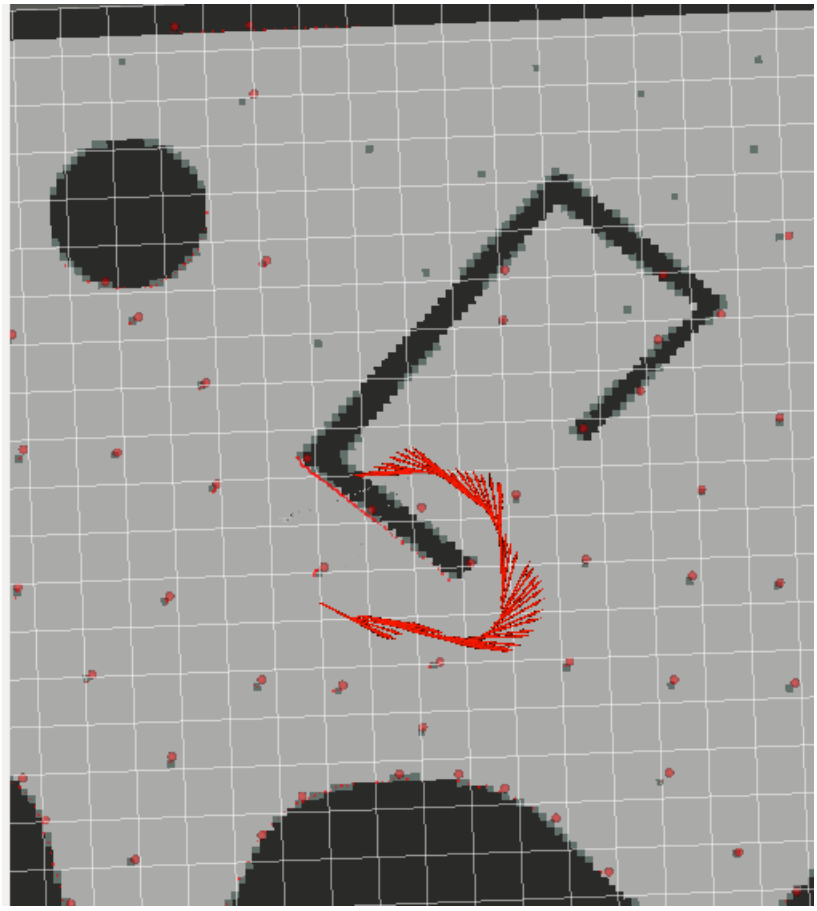
如图所示，蓝色和绿色的landmark代表着该帧下探测到的landmark与在已有的landmark集合中与之最近邻匹配的landmark,而红色的landmark代表着在运动过程中被保存下来的landmark的全集。

为了便于观察我们可以只保留红色的过程中的存储的特征点

rosbag1:



rosbag2:



可以看到，经过特征点的`addlandmark()`的检测新特征之后，小车运动范围内的绝大多数特征都被检测到并被放入landmark集合之中，集合中landmark的坐标位置在全局观测看来较为精确，虽然仍然有不少墙上的点被辨识为landmark，但经过后续的最近邻匹配的筛选后并不会对定位结果产生较大影响。

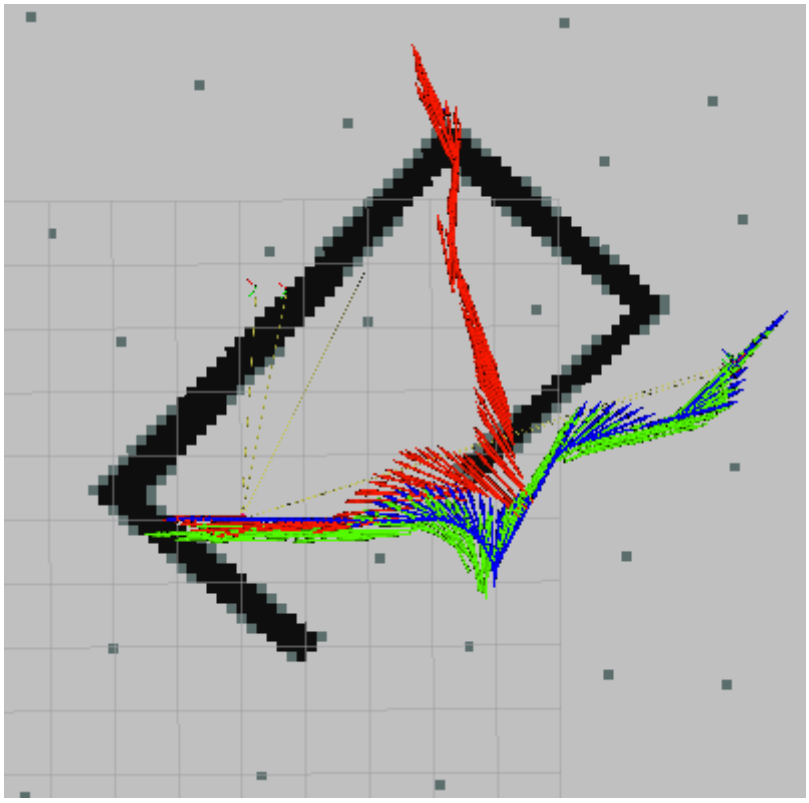
5.1.2 定位的性能分析

5.1.2.1 定位误差

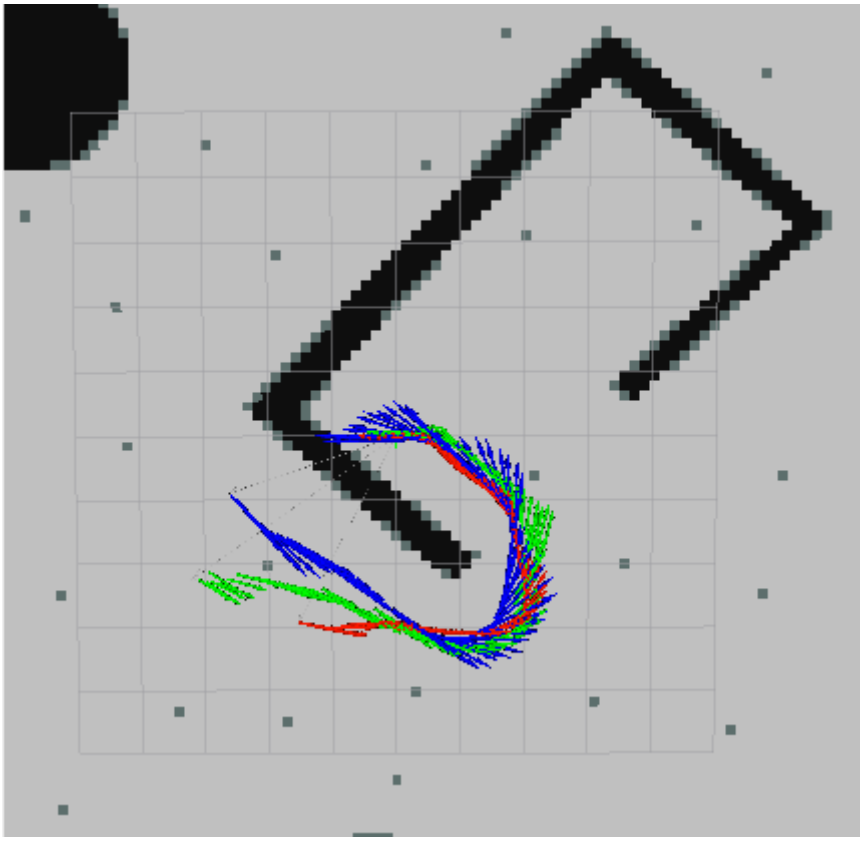
我们首先进行定位误差的定量观察：

我们对之前的rosbag1与rosbag2进行分别的定位观察，如下面的两图所示，其中红色的为纯ICP定位的结果，蓝色的为普通EKF定位的结果，绿色的为EKF_slam的结果。

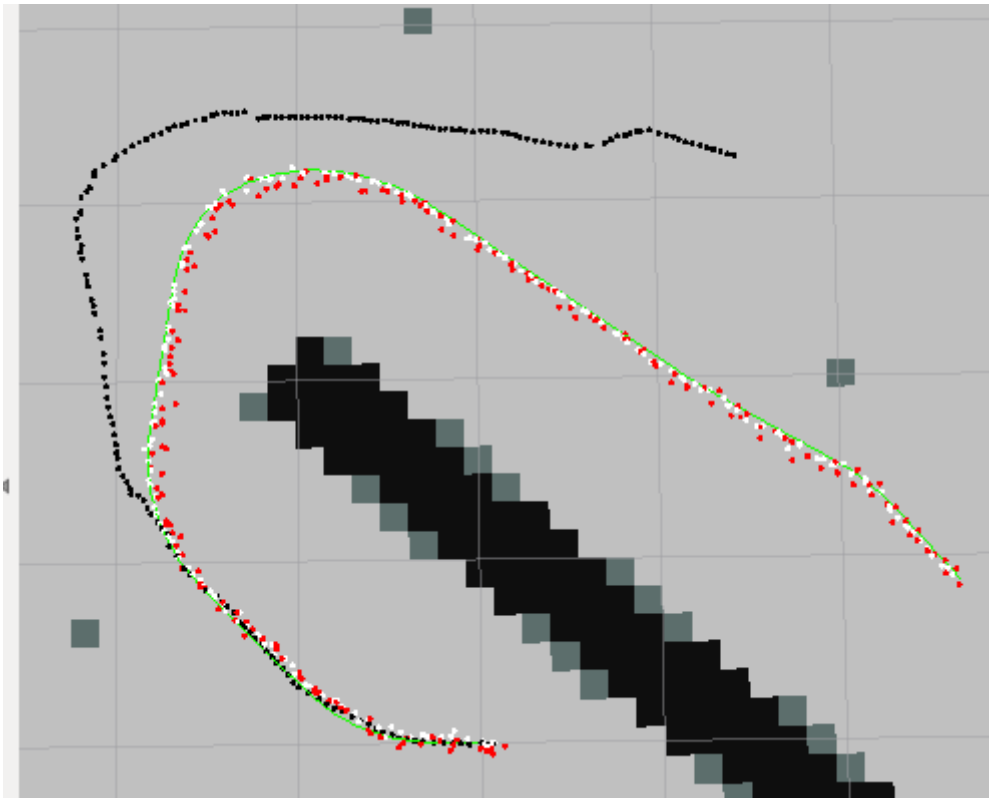
rosbag1:



rosbag2:



为了更仔细的进行观察，我们使用point cloud的点云形式进行观察，对rosbag1路线中的定位进行真实路径与点云的可视化显示，结果如下图所示：其中黑色的散点为纯ICP定位的结果，红色的点云为普通EKF的结果，白色的点云为EKF_slam的结果，而绿色的path为小车真实走过的路径。



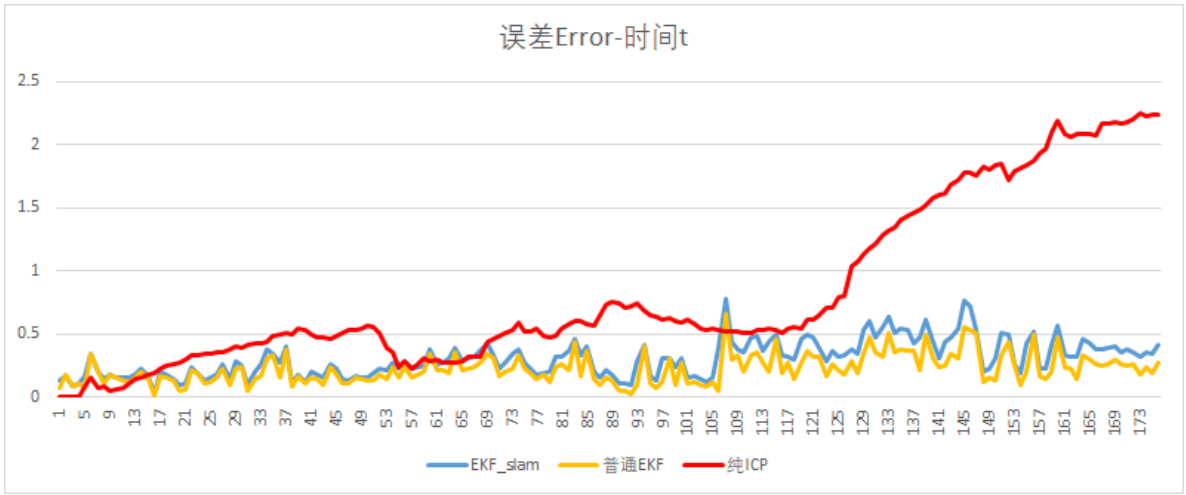
通过定性的观察我们可以发现，纯ICP的误差最大,普通ekf和ekf_slam的定位误差都比较小，但是ekf_slam的定位时左右的波动现象更加明显。

接下来我们对误差进行定量分析:

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，我们定义每一种定位方式的绝对误差大小为

$$error = \mathrm{dist}(dx, dy) = \sqrt{(x - x_{\mathrm{true}})^2 + (y - y_{\mathrm{true}})^2} \tag{13}$$

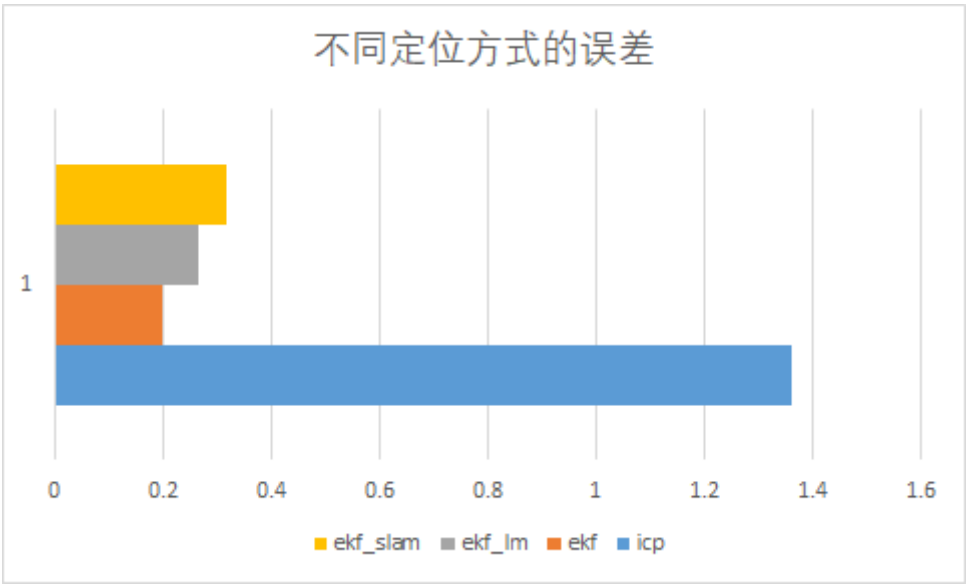
我们统计闭环的一整圈路径中不同定位方式绝对误差随着帧数变换的数据(每8帧取一次)，绘制误差随着小车运行时间的变化图像如下，其中红色线为纯ICP定位的误差，黄色线为普通EKF定位的误差，蓝色线为EKF_slam的定位误差.。

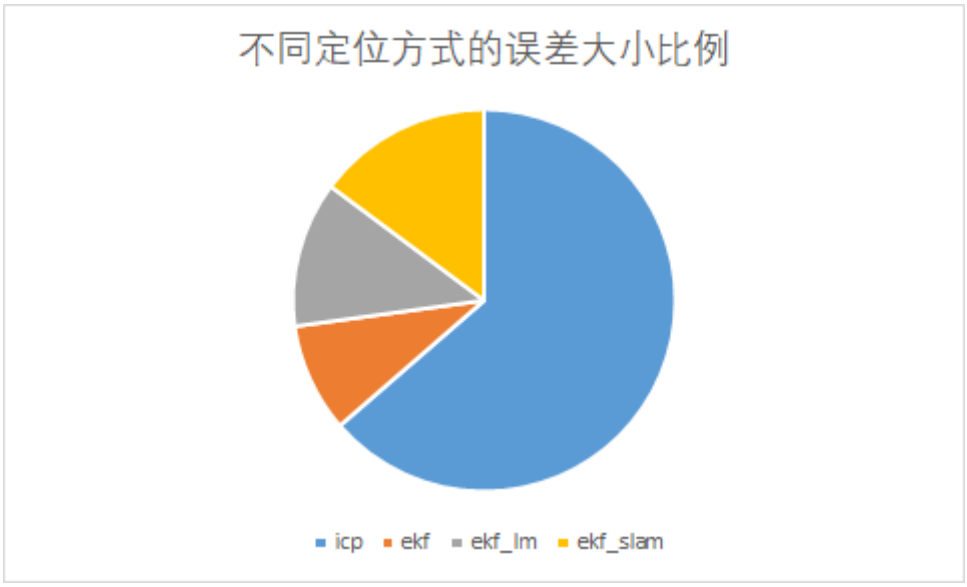


为了更加直观地展示三种定位方式的误差的比例大小，我们使用更加直观的误差比例展示图进行展示。我们对过程中的每个时间的误差进行不同定位方法的分别对每一帧求平均，误差的平均值如下表所示（纯icp由于定位在一圈闭环后累计的误差已经非常非常大导致误差远大于其他的定位方式）：

定位方法	icp	普通的ekf	ekf_lm ekf_slam
平均误差	1.3632	0.1975	0.2642 0.3177

使用饼状图与柱状图进行展示





从误差的结果中我们可以看出，在闭环一圈之后，纯ICP的定位误差很大，而在其他三种定位方式中，定位最精确的为普通的ekf，而ekf_lm的误差会略大于普通的ekf定位（差别不大），而ekf_slam的定位误差又会略大于ekf_lm，但这三种不同的EKF的定位误差大小相差不大，平均误差均远小于纯ICP的定位误差。

5.1.2.2 定位消耗的时间

进行不同定位时单次消耗的时间分析与比较

我们使用python中的时间模块,对不同定位方式(这里给出了四种不同的选择)进行单次定位的计时，由于bag1与bag2的激光密度也不同，对两个不是闭环的bag进行分别计时，四种定位方式分别为纯icp，普通ekf、ekf_lm与ekf_slam，实验数据如下：

对rosbag1:

使用全局激光的ICP激光里程计(W7 纯ICP)，单次定位的平均耗时为：6.55ms

使用全局的地图激光进行EKF定位（W8 ），单次定位的平均耗时为：17.10ms

当观测模型与预测模型都使用特征提取时（W10 EKF_lm），单次定位的平均耗时为:6.29ms

当使用不依赖map_server的EKF_slam时，单次定位的平均耗时为:4.93ms

对rosbag2:

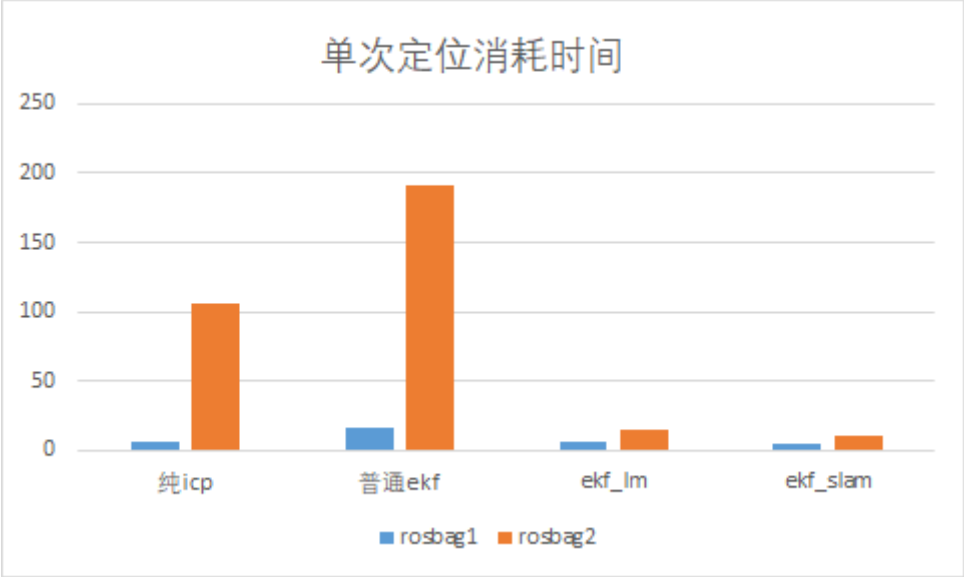
使用全局激光的ICP激光里程计(W7 纯ICP)，单次定位的平均耗时为：106.71ms

使用全局的地图激光进行EKF定位（W8 ），单次定位的平均耗时为：191.23ms

当观测模型与预测模型都使用特征提取时（W10 EKF_lm），单次定位的平均耗时为:15.21ms

当使用不依赖map_server的EKF_slam时，单次定位的平均耗时为：10.67ms

将两个bag的不同定位方式消耗的时间使用柱状图进行可视化，结果如下：



picture:单次定位的消耗时间

由上述结果我们可以知道，由于rosbag2相比rosbag1的激光更密，激光组数更多，因此全地图激光点云匹配的组数也更多(rosbag1最多100组而rosbag2最多达到了300组) 更加密集的激光导致在视野中的提取出来的特征也更多，对激光信息的处理与匹配、ekf的输入输出等很多步骤的时间复杂度都是与激光点数的平方成正比（即$O(N^2)$），少部分的时间复杂度达到了更高的数量级，因此rosbag1与rosbag2之间的定位耗时差别非常大。

接下来我们尝试对时间复杂度进行理论的分析，设进行icp算法时最大的允许的迭代次数为M，而每次进入ICP变换算法的点数最大为N，除去ICP迭代过程中的二次方级别的时间复杂度，其他的地方的时间复杂度均为常数、线性或对数时间复杂度，因此我们可以认为进行单次ICP变换算法时的时间复杂度为

$$T(N) = O(MN^2) \tag{14}$$

而ekf的过程大致可以看做两个ICP（其中一个可以简化为最近邻匹配）与其余的线性时间复杂度的计算过程，因此这样的分析(消耗时间的复杂度约是匹配的点数的平方)与实验数据对照可以发现，这样的变化趋势基本与实验数据一致，使用特征提取代替每一束激光的结果可以很大幅度减少运行的时间，从而减少性能消耗。

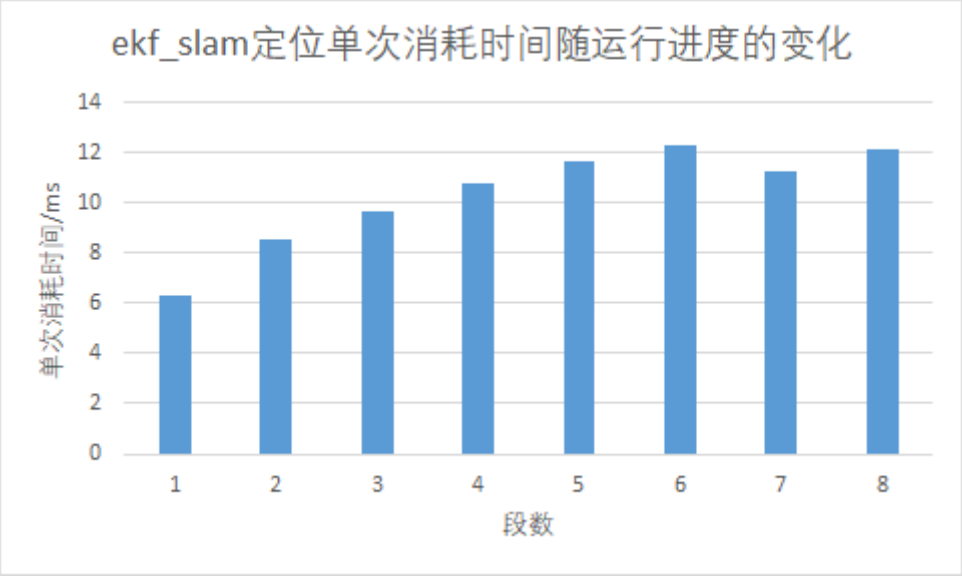
同样都是使用特征提取而不是全局的激光，ekf_slam在ekf_lm的基础上进一步缩短了运行的时间。主要的原因是ekf_lm在进行观测模型校正时会使用初始地图上的所有提取到的所有landmark进行最近邻匹配，而ekf_slam只需要对已经探测到并保存的landmark进行最近邻匹配，后者的规模明显小于前者，因此ekf_slam消耗的时间会比ekf_lm更短。

EKF_slam计算消耗时间随时间的变化：

我们将整个闭环的过程分为较为平均的8段来进行处理（分为8段而不是离散的帧可以避免单帧中因为特征的最近邻匹配产生的随机偏差，分为多段更具有统计意义），每段中我们计算该段的平均每帧EKF_slam的定位耗时，结果如下表所示：

段数	1	2	3	4	5	6	7	8
耗时/(ms)	6.31	8.57	9.69	10.83	11.67	12.32	11.25	12.19

可视化图表的结果如下：



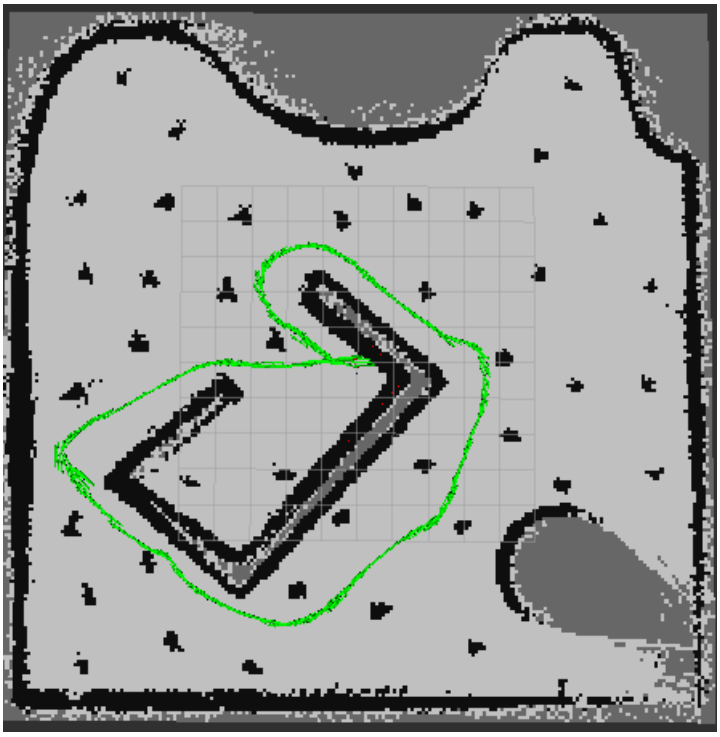
由此结果可知，EKF_slam单次消耗的时间随着小车的运行是一个先逐渐变大，之后变得稳定，在某个值附近上下震荡的结果。根据分析，最开始的时候耗时较少主要是因为刚开始的时候检查到的landmark的集合内landmark的数量比较少，因此进行最近邻匹配的时候遍历的landmark次数较少，消耗时间少，另一个原因是刚开始的时候可能存在连续两帧的landmark匹配上的个数小于阈值导致该帧被跳过，也会拉低平均值。而在后期消耗时间趋于稳定的原因是地图上的大多数特征都已经被检测到并加入landmark集合中，每次加入的新landmark很少，进行遍历时的landmark个数基本为一个固定值，因此单次消耗时间会逐步趋于稳定。

5.2 mapping地图构建的结果

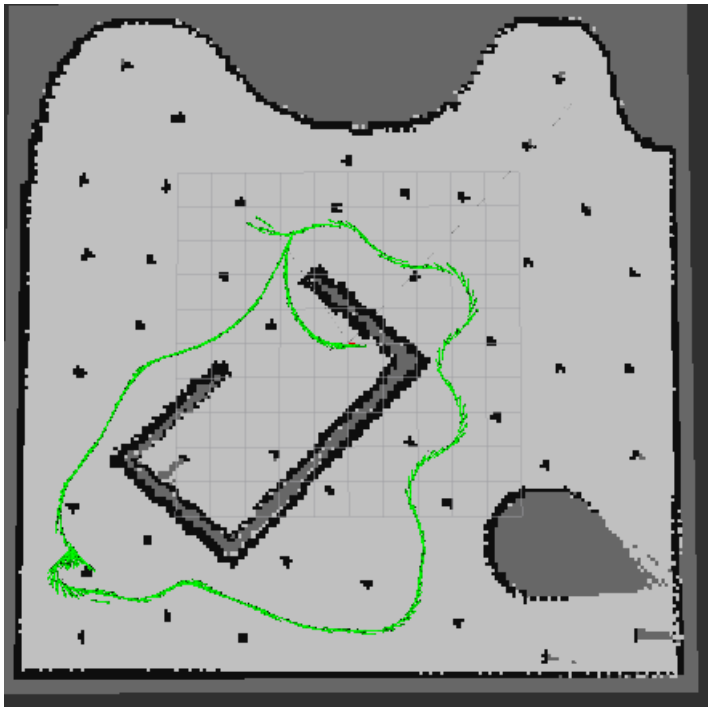
5.2.1 离线地图的构建：

使用助教提供的bag得到的离线地图如下：

其中绿色的轨迹为小车ekf_slam定位的轨迹



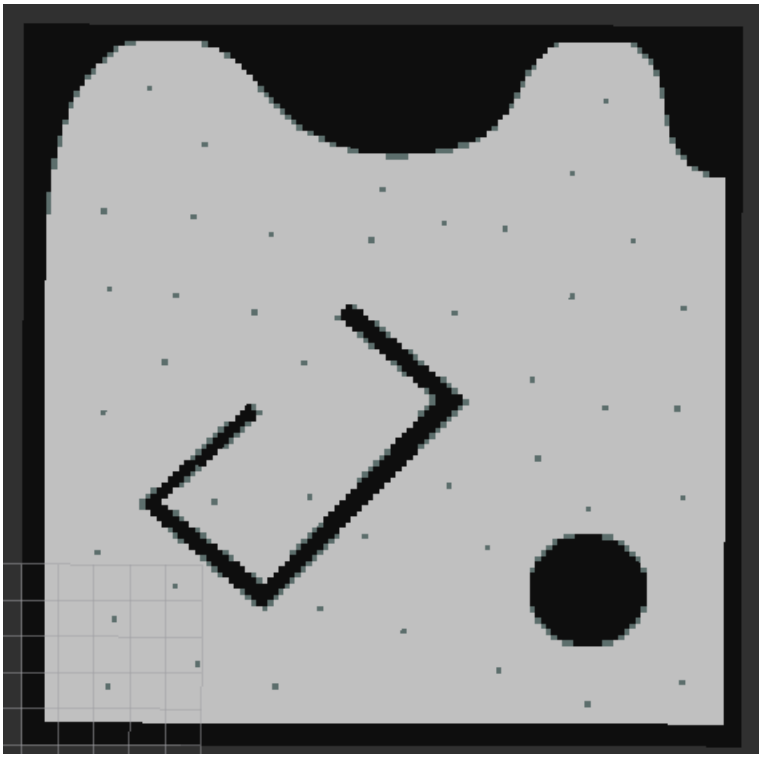
另外一个闭环的bag得到的离线地图：



（注：在实验讨论与心得部分有对离线地图效果的进一步优化）

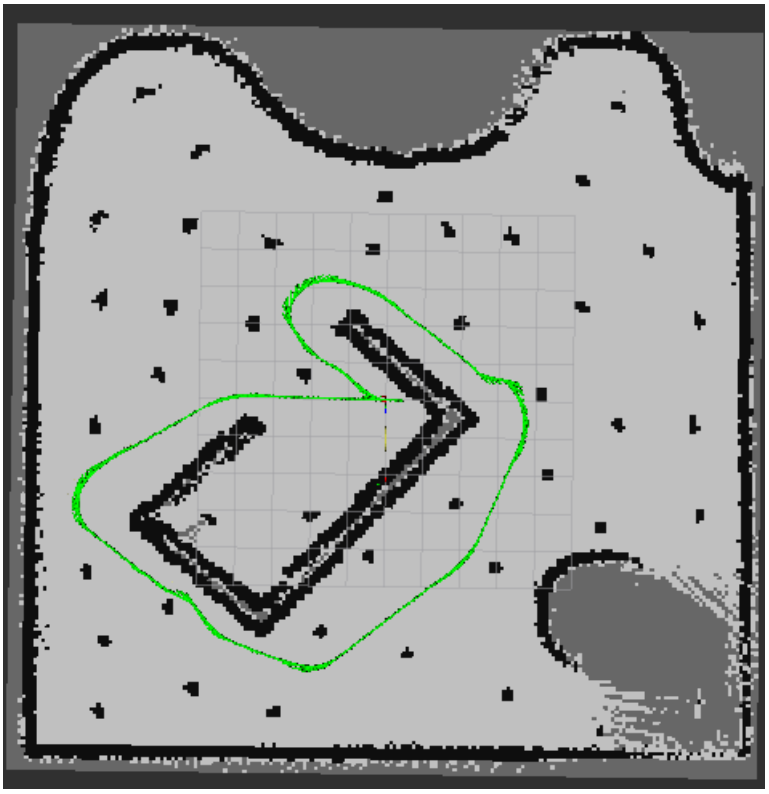
ekf_slam构造的离线地图与真值地图的比较：

真值的map如下所示：

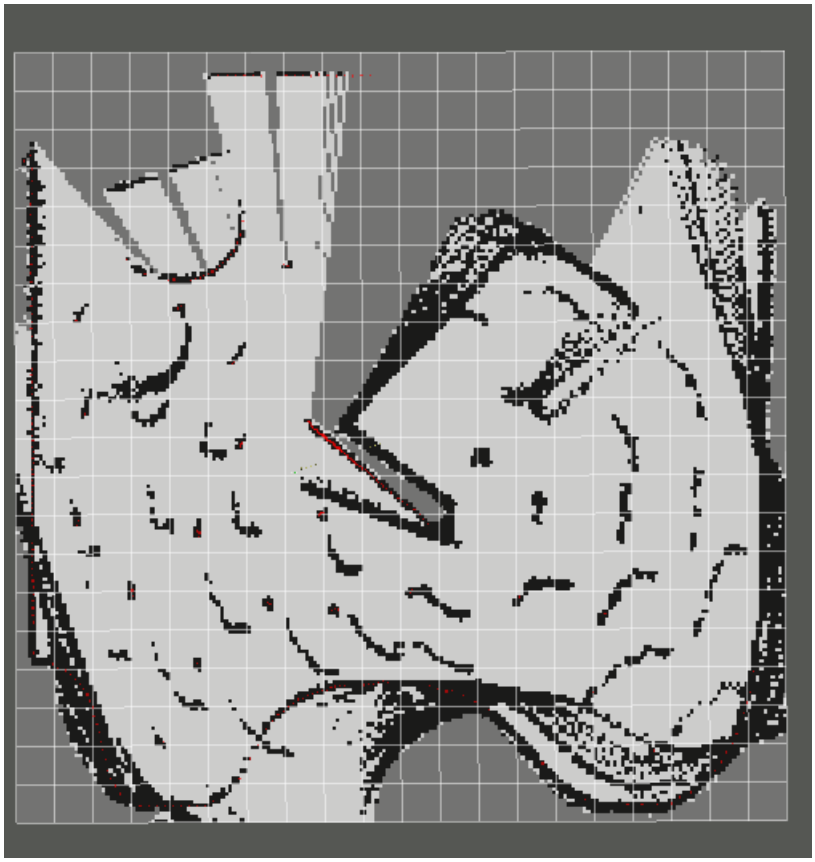


ekf_slam构造的离线地图与普通ekf以及纯ICP得到的离线地图的比较：

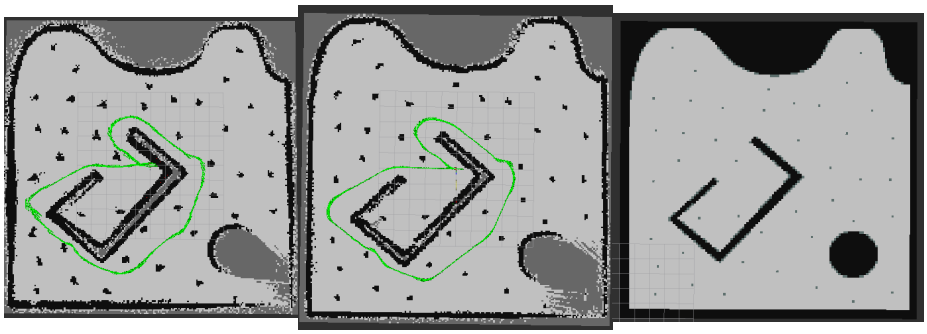
对于第一个bag，使用普通的EKF定位方式来获得mapping的结果时，构成的离线地图结果如下：



而当我们使用纯ICP的定位结果进行mapping地图构建时，则因为定位误差实在累计越来越大，在构建中途的结果内地图内的障碍物就会糊成一团，因此没有进行完整的闭环流程，中间的结果如下图所示：



通过对EKF_slam与普通EKF以及真值地图得到的结果进行比较（纯ICP得到的结果太差，不在这里进行讨论），如下图所示（在上方也有更大的相同的图片展示）：



我们可以发现，相比于普通EKF的结果，ekf_slam运行出来的效果略差于普通ekf构建的结果，在mapping使用的参数相同的情况下：

（1）ekf_slam得到的离线地图中landmark成的团的现象更大（从真值地图上的一个点膨胀为了离线地图上的一小团），说明ekf_slam对于地图上landmark的构建精度略差，对于地图边界障碍物与墙的构建上也略微更粗糙一些

（2）在定位的轨迹方面，ekf_slam明显在轨迹中的左右波动比普通ekf更大（Odometry形成的轨迹更粗），定位略微的左右波动也是造成（1）中现象的一个很重要的因素。

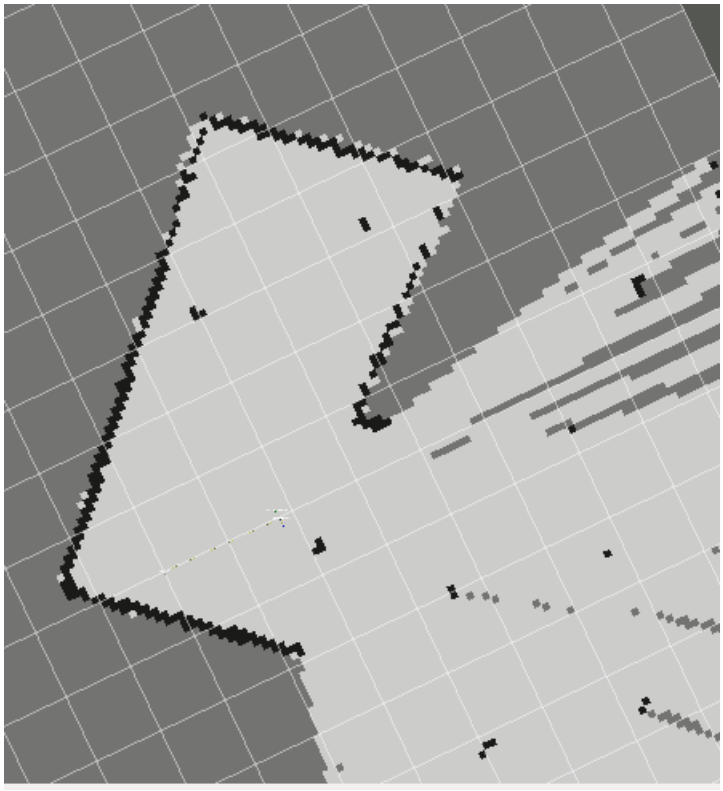
（3）与真实的地图相比，无论是ekf还是ekf_slam都使得图中的障碍物边界更加粗糙，对于右下角的部分区域，由于并没有到达附近的区域，这一片仍然是处于灰色的未知状态。

5.2.2 在线地图的构建：

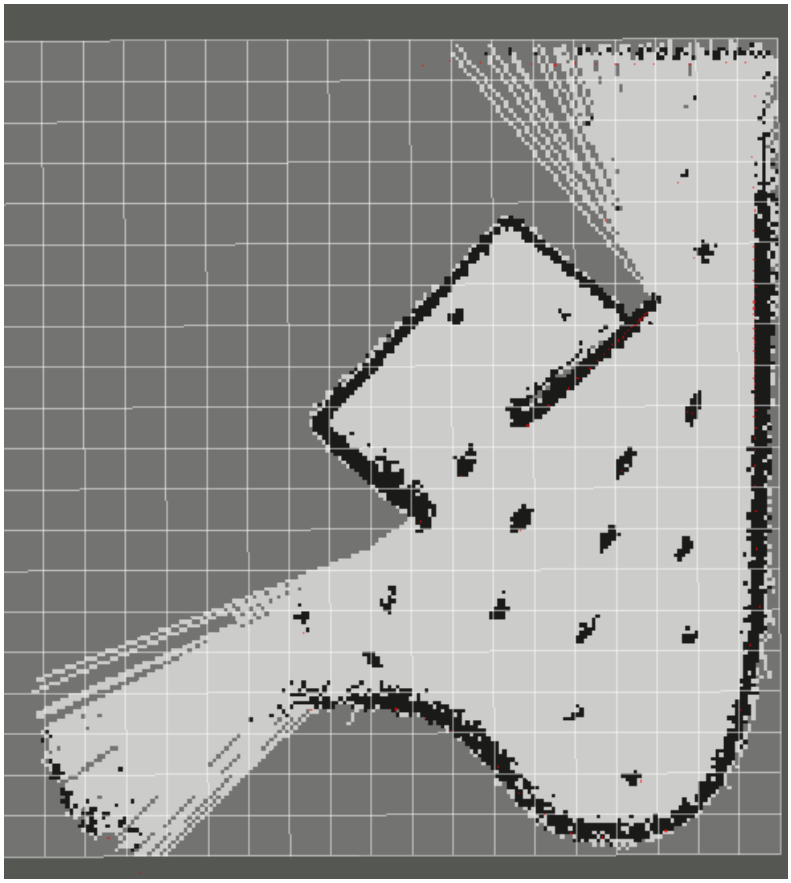
我们选用之前的非闭环的rosbag1和rosbag2来进行在线地图的检测，为了方便地图的观察，我们隐去之前已经展示过的EKF_slam的定位结果。

rosbag1：

运动初期在线地图的细节

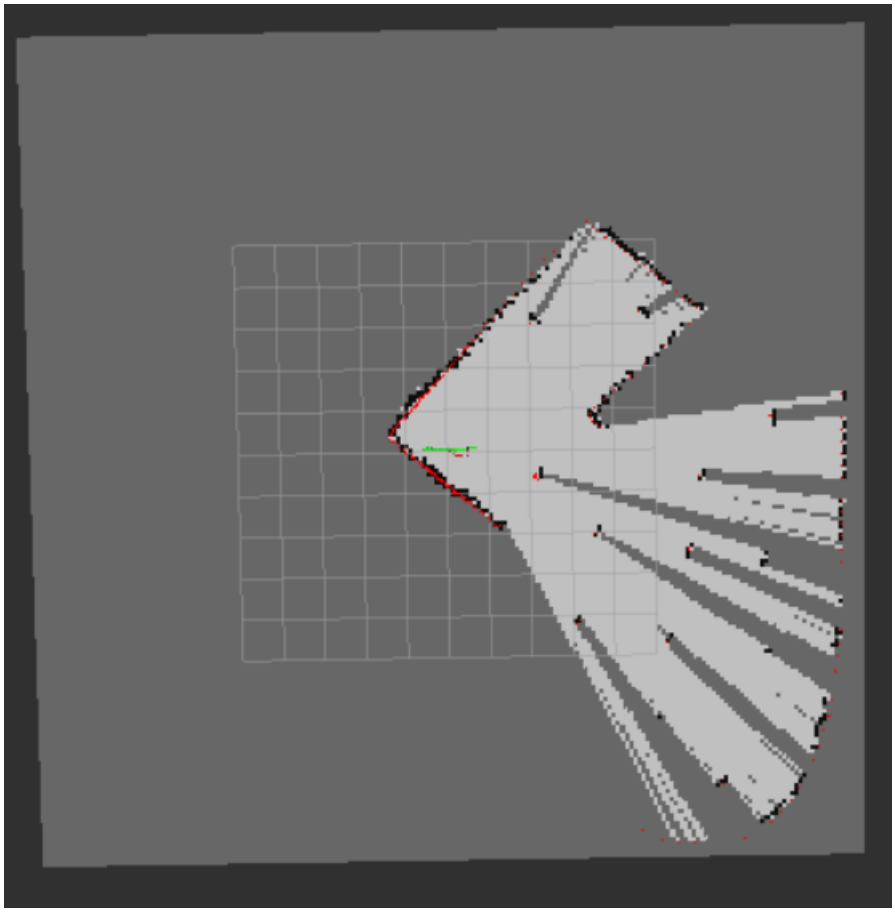


最终结果如图所示（最终结果其实就是离线地图），由于激光束较少，精度比不上rosbag2。

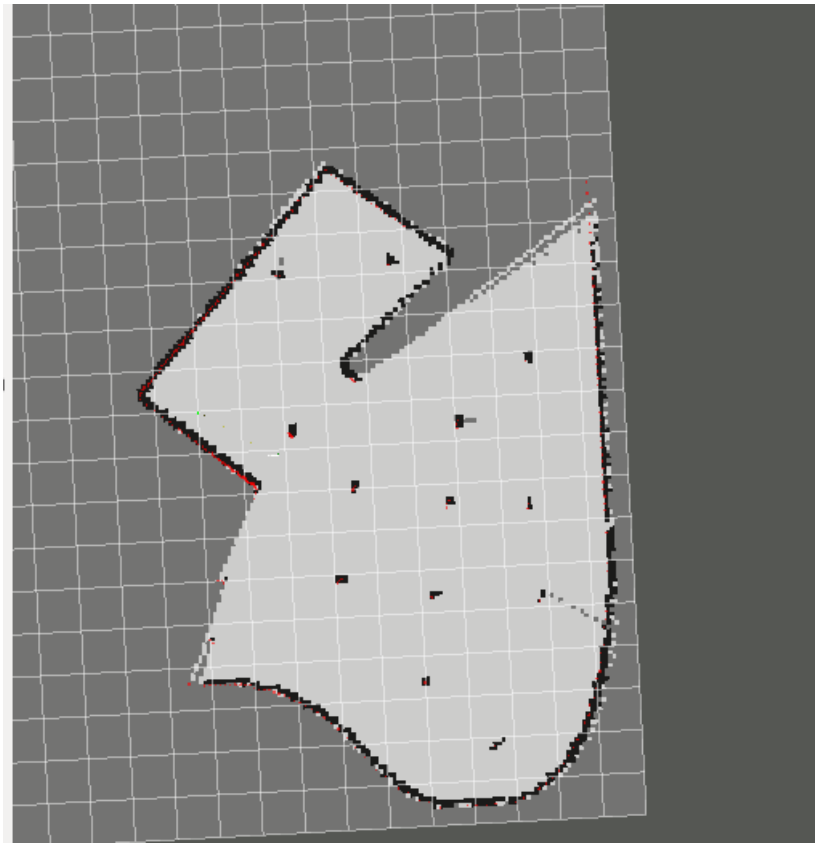


rosbag2:

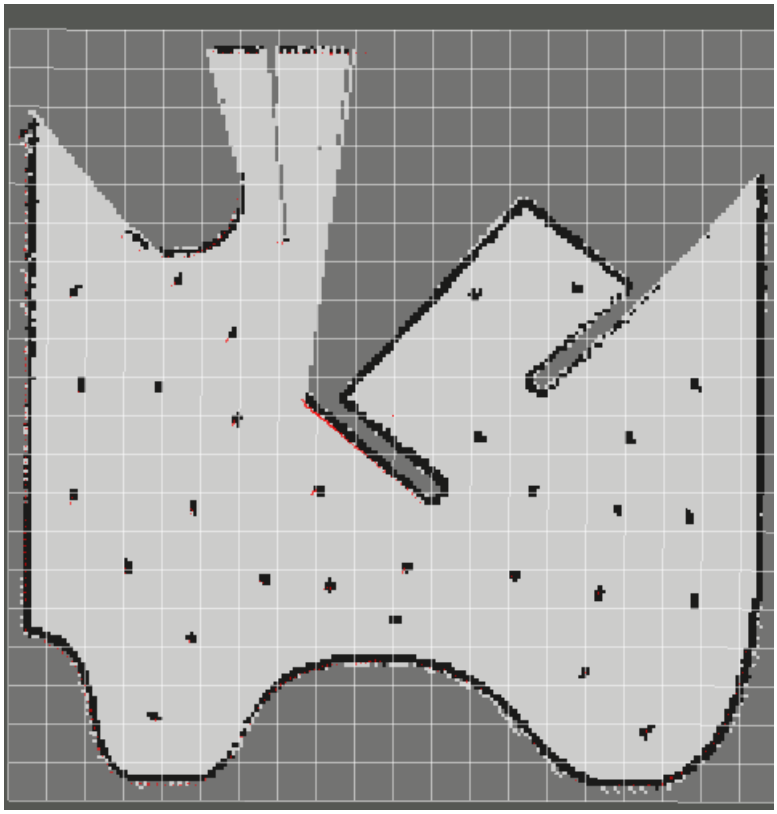
初始时在线地图的细节：



在小车运动过程中，地图被逐渐完善

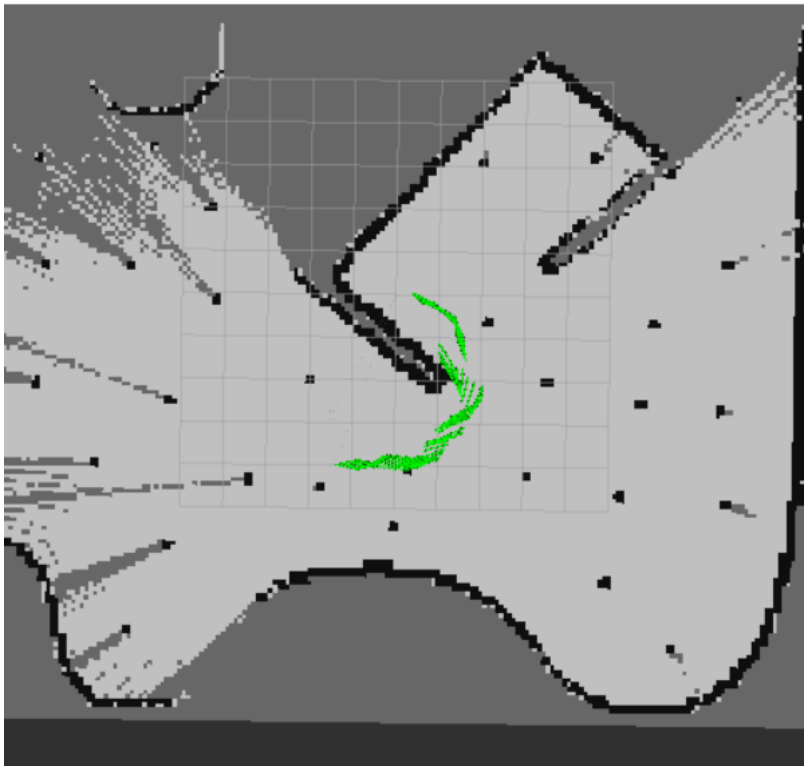


最终结果如图所示：

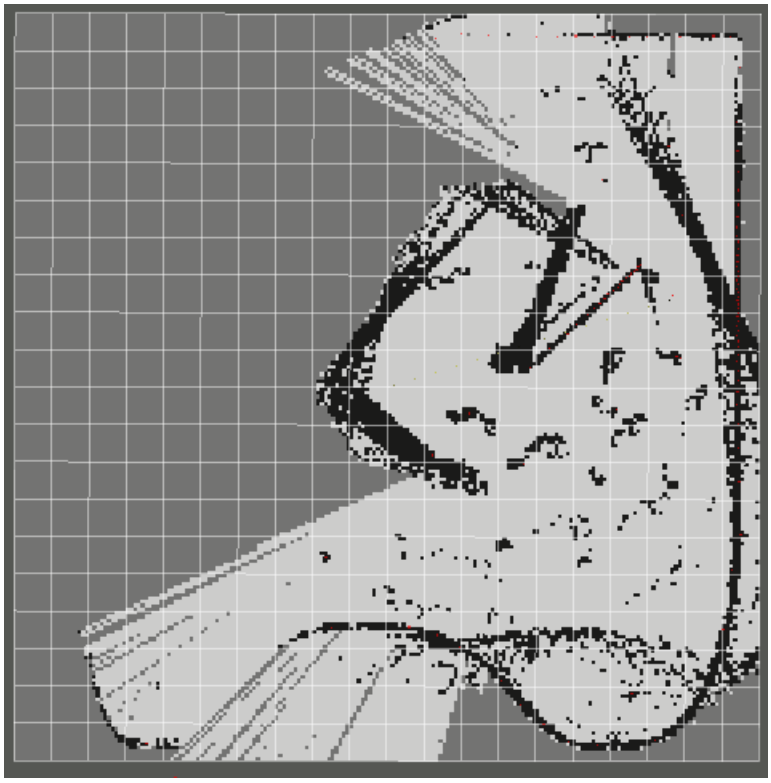


ekf_slam的在线地图与ekf、icp的对比：

使用普通的EKF定位结果来进行mapping构造得到的在线地图结果。



使用纯ICP激光里程的定位结果来进行mapping的结果（即使是在线地图仍然效果非常不好）：



在线地图与离线地图的对比：

由于离线地图的闭环特殊性，如果ekf_slam定位误差较大的话，可能会导致定位出来的路径并不闭环，在mapping过程的后半段可能出现偏移较大的情况（在首尾闭合部分的障碍物检测可能因为绕一圈回来的位置误差较大而造成两次在同一点的定位不一样导致前后同一障碍物mapping出来的位置差别很大）。而在线地图如果不需要闭环的话偏移会比较小（其实应该说是闭环比非闭环更能检测ekf_slam的性能）。

Mapping部分的分析与总结：

对于不同的定位方式有着不一样的**精度**，而地图的效果主要是由定位的精度决定的，因此对于精度方面，$ekf > ekf\_lm > ekf\_slam >> icp$(前三种的定位方式的精度差异是较小的差异)。而对于ekf_slam的**复杂度**，在已知定位和激光message的情况下消耗的复杂度是一样的，因此不同方式的复杂度的区别主要也体现在定位的复杂度上不同。而对于定位的复杂度$O(M \times N^2)$，则是ekf_slam消耗最少，ekf_slam第二少，普通ekf的消耗会远大于前两者的消耗。而对于闭环地图的**收敛性**，主要取决于闭环地图的效果以及该定位方式是否容易出现发散的现象，而这又会归结到定位的精度以及定位的稳定性（误差是否容易累计等）。综上所述，当我们拥有初始的地图数据时，我们可以使用普通ekf获得较高的精度，也可以牺牲部分精度使用ekf_lm获得更快的运行速度。而倘若我们没有初始的地图数据（符合真实世界的情况），ekf_slam会是一个兼顾精度与速度，方便构建地图的定位方法。

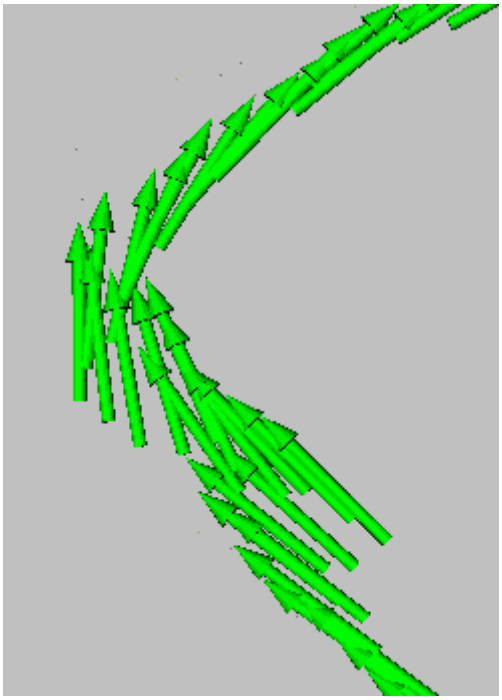
六、实验思考与心得

6.1 EKF_slam长时间运行后可能出现的发散问题与改进方案?

（1）运行过程中有的bag中的激光laserscan数据可能会出现inf与nan的数据，

```
range_max: 20.0
ranges: [8.768767356872559, 9.097431182861328, 9.586639404296875, 10.026311874389648, inf, inf,
3.501473903656006, 3.0698633193969727, 2.7965900897979736, 2.556821346282959, 2.2626545429229736,
2.084261655807495, 1.9138869047164917, 1.8205376863479614, 1.6865249872207642, 1.6390482187271118,
1.4826864004135132, 1.4552686214447021, 1.362685203552246, 1.314389944076538, 1.2500640153884888,
1.2185882329940796, 1.1805055141448975, 1.1630821228027344, 1.103317141532898, 1.0920360088348389,
```

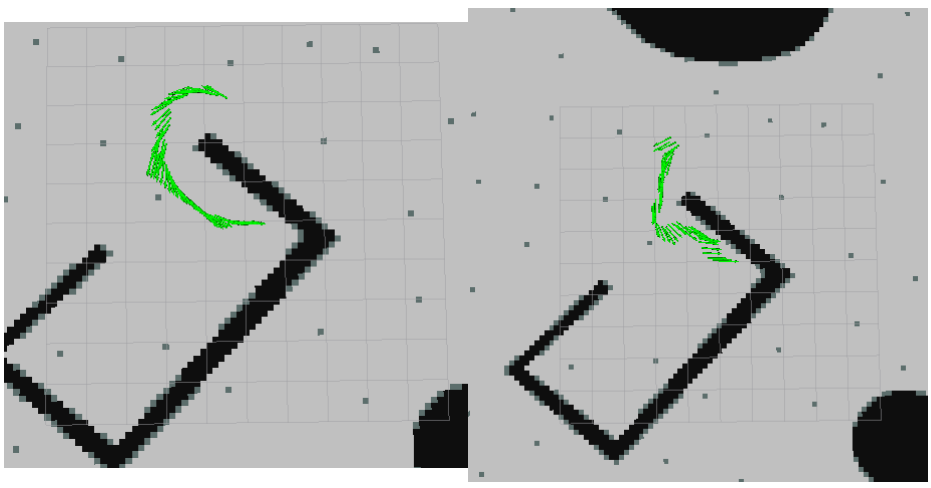
对于如果中间某一帧出现inf和nan的情况，我们可以采用进行特判，跳过这一帧激光的解决方案。但在有的bag中出现了连续多帧存在inf与nan的情况，如果直接跳过这么多帧的话会导致下一次有效的定位位置不但离上一次有效的定位位置距离远，误差也会较大。因此我的处理方式是将inf和nan都设置为LASER_MAXRANGE的一个常数（这里为20），这样做虽然仍然会带来误差和不稳定的现象，但相较于直接跳过多帧的处理方式的误差更小，但仍然会出现不稳定的现象（如下图所示），从而可能导致发散的现象。



（2）EKF_slam对于运行速度比较快的小车，由于两帧之间位置的差比较大，进行最近邻匹配时产生的误差也相对更大，而同时EKF_slam对于比较急的转弯的响应会出现延迟以及转弯角度难以达到真实角度的现象，因此ekf_slam在遇到车速快（相邻两帧距离大）以及转弯比较多的运动中都会导致小车的定位误差变大，而在长时间的运行中，大量的转弯以及高速行驶可能会导致EKF_slam的逐渐发散。

6.2 使用EKF_slam所构建的离线地图来进行EKF定位：

我们将EKF_slam最后构建的离线地图的数据当做下一次的EKF普通定位的地图输入（laser_scan仍然是真实的laser_scan，改变的分别是普通ekf中的laser_estimation以及ekf_lm初始时对地图上所有特征的提取）。



其中左边为真值地图的ekf，右边为离线地图的ekf，可以看出来，使用ekf_slam构建离线地图再直接用在estimation中进行普通EKF定位时，误差较大而定位十分不稳定（也可能是因为laserestimation使用了离线地图数据而真实的激光仍是真值地图中的激光导致不匹配）

而在这样的情况下，我发现EKF_lm进行定位的精度甚至会略微好于普通的EKF（只是从开始到变得发散的时间会长一些些，但仍然随着运行时间的变大会发散），我猜测是因为EKF_lm在进行特征提取的时候虽然原有的单点的特征在图上模糊成了一个小团，但我们在进行特征提取的时候取的是多束激光位置的中心值，因此实际上提取landmark的误差要比在这样的离线地图上进行laser_estimation模拟激光的误差会小一些，导致ekf_lm比普通ekf发散更慢一些。

6.3 在ekf_slam中遇到的困难与解决

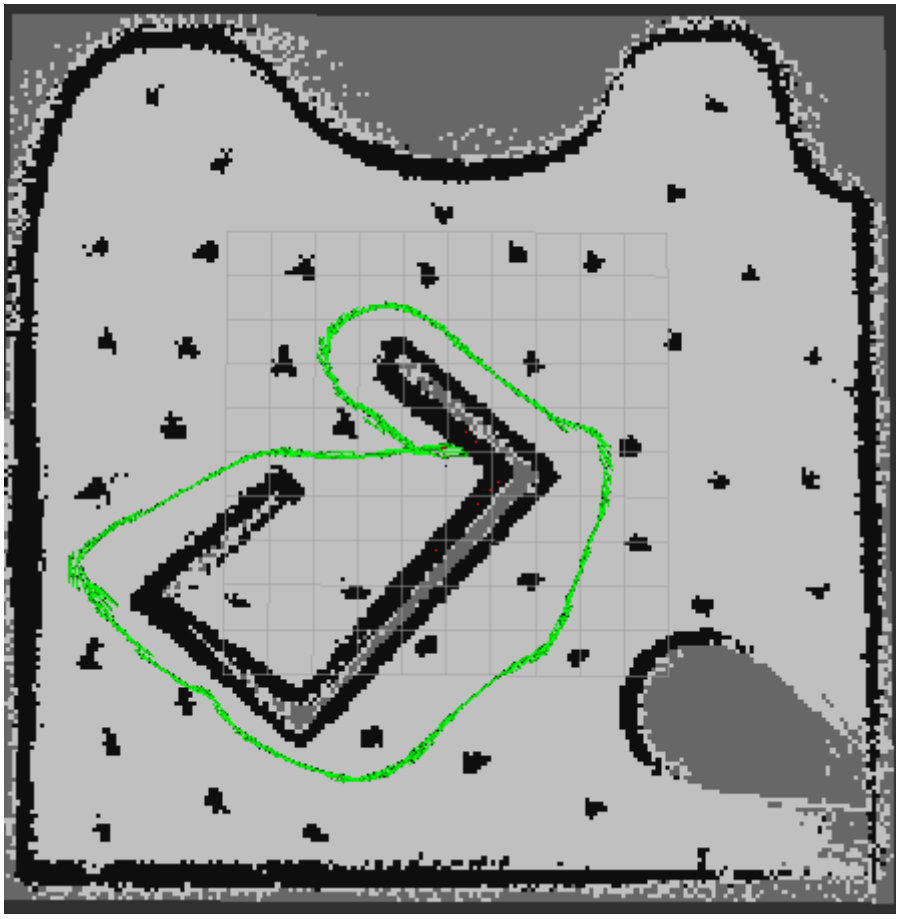
（1）对于不同的bag，ekf_slam中同一种参数设定方式可能对构造的地图的效果产生非常大的影响，对不同的bag都需要重新进行参数的改进与调整以达到较好的效果。

（2）mapping时候出现地图倾斜的现象，最开始我也并不知道这样的现象该如何解决，但我后来发现这样的情况在对激光中的inf以及nan的数据进行处理后这样的现象出现了好转，另一个处理是对间隔帧数的选取，不同的间隔帧数（最开始为间隔5帧处理一次）也会稍微改变偏转角度（暂时还没有发现原因），应该调节到倾斜角度较小为止。

（3）定位的漂移：在定位过程中可能还是会遇到定位在某处出现漂移的问题，对于每一帧定位我们都可以在最后与上一帧进行对比，如果位移超过我们设置的阈值我们则认为定位出现了漂移，应该取消这一帧，进行下一帧的定位处理。

6.4 mapping时对于参数选取的优化

我们最初进行mapping的时候得到的结果如下，可以看到在地图的边缘一圈的墙外也出现了部分的白色栅格（空闲的栅格）



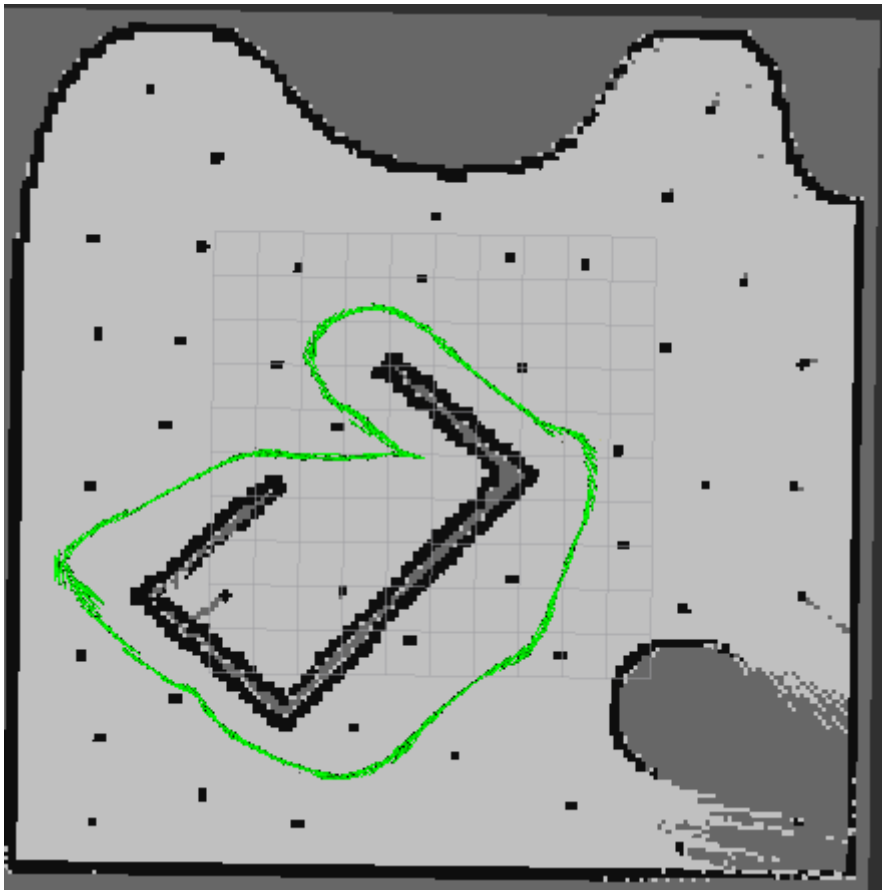
针对这样的问题，我们做出以下的参数的修改：

mapping时候做bresenham计算的时候起点和终点的坐标应该加上round而不是直接向下取int，代码如下修改

```
                px_o=int(round(10*(ox[i]+10)))
                py_o=int(round(10*(oy[i]+10)))
                px_c=int(round(10*(center_x+10)))
                py_c=int(round(10*(center_y+10)))
```

同时，我们在进行栅格空闲的与否的判定中将要求加强，需要多次检测为空闲后（每一次检测到便进行数值的累加）才会将这一块认定为栅格状态（在这里经过实验调参认定为20次）。同时我们对于判定为障碍物占用状态的条件也更加严格，来减少landmark在离线地图上的聚团现象。造成这样的现象的另外一个比较重要的原因就是在我的笔记本电脑中有时会出现激光穿墙的现象，在我的其他设备中则没有出现这样的现象。

参数调整之后得到的结果如下所示：



虽然边界上仍然会出现白点和模糊的状况，但相比之前的结果以及得到了较大的改善。

实验心得：

这次实验是整个轮式机器人课程中的最后一次实验，也是最难的一次实验。在实验的过程中也是遇到了非常多的问题与困难，而因为期末考试的父子安排和其他课程非常多的大作业，导致最后一次实验中间跨度的时间非常大，但好在老师延长了期限让我可以有更多的时间来完成这次实验。这次实验的任务与后期的结果讨论相对而言比较多，报告中需要思考的内容也非常多，报告的文字也是不出意料的突破了万字orz。有的最开始未能理解的部分好在查阅了相关资料之后获得了理解，这次实验中特别是在mapping部分想要达到比较好的效果真的得花非常多的时间与精力对参数进行调整。虽然因为期末的时间比较紧（但也投入了非常非常多的时间与精力），导致或许报告中会有一些细节存在瑕疵，希望老师可以谅解。反思自己四次实验做下来，都是重复着每次实验过程中“大呼好难”到自己慢慢摸索debug尽

量做出较好的效果，这样的过程虽然有一些痛苦，但只要我们有耐心去钻研，对我们的编程能力与自我动手解决问题的能力的提升是非常有帮助的。在课程的最后也要再次对课程中解决了我很多疑惑的老师和助教表示感谢！