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 30 日

目录

实验报告3

3180103127　　肖瑞轩

一、实验目的：

在W8的扩展卡尔曼滤波定位的基础上，完成地图中圆柱的特征提取，使用带有特征提取的拓展卡尔曼滤波器对小车的位置进行预测与更新，再对不同的定位方式的性能进行分析。

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。

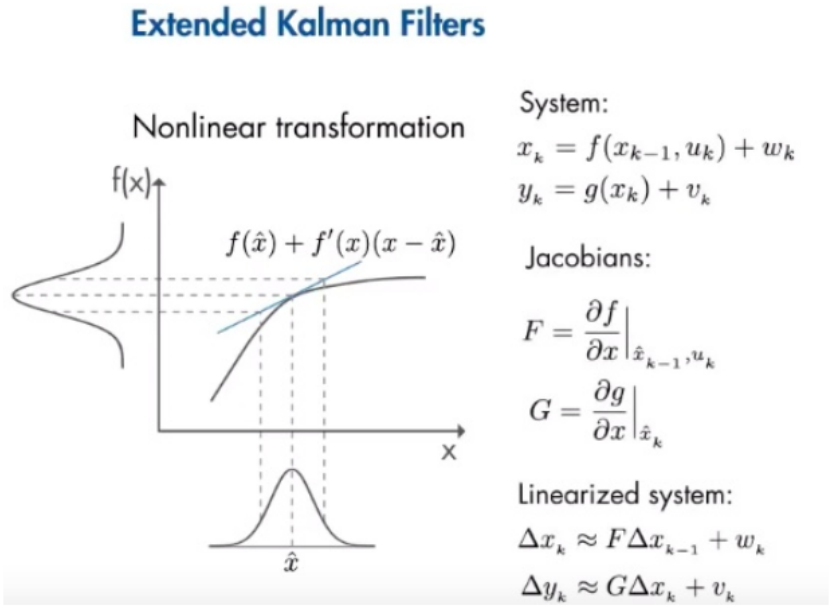
ICP匹配的基本流程可以总结为下图：

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。

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
```

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(- ,+)，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(alphadelta_
        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;
```

对提取出来的两组地标需要在rviz中显示出来，分别以绿色与蓝色的形式显示，代码如下：

```
    def publishLandMark(self,msg,msg2):
        # msg = LandMarkSet()
        if len(msg.id) <= 0:
            return

        landMark_array_msg = MarkerArray()
        for i in range(len(msg.id)):
            marker = Marker()
            marker.header.frame_id = "course_agv__hokuyo__link"
            marker.header.stamp = rospy.Time(0)
            marker.ns = "lm"
            marker.id = i
            marker.type = Marker.SPHERE
            marker.action = Marker.ADD
            marker.pose.position.x = msg.position_x[i]
            marker.pose.position.y = msg.position_y[i]
            marker.pose.position.z = 0 # 2D
            marker.pose.orientation.x = 0.0
            marker.pose.orientation.y = 0.0
            marker.pose.orientation.z = 0.0
            marker.pose.orientation.w = 1.0
            marker.scale.x = 0.2
            marker.scale.y = 0.2
            marker.scale.z = 0.2
            marker.color.a = 0.5 # Don't forget to set the alpha
            marker.color.r = 0.0
            marker.color.g = 0.0
            marker.color.b = 1.0
            landMark_array_msg.markers.append(marker)
        for i in range(len(msg2.id)):
            marker = Marker()
            marker.header.frame_id = "course_agv__hokuyo__link"
            marker.header.stamp = rospy.Time(0)
            marker.ns = "lm"
            marker.id = i+len(msg.id)
            marker.type = Marker.SPHERE
            marker.action = Marker.ADD
            marker.pose.position.x = msg2.position_x[i]
            marker.pose.position.y = msg2.position_y[i]
            marker.pose.position.z = 0 # 2D
            marker.pose.orientation.x = 0.0
            marker.pose.orientation.y = 0.0
            marker.pose.orientation.z = 0.0
            marker.pose.orientation.w = 1.0
            marker.scale.x = 0.2
            marker.scale.y = 0.2
```

```
            marker.scale.z = 0.2
            marker.color.a = 0.5 # Don't forget to set the alpha
            marker.color.r = 0.0
            marker.color.g = 1.0
            marker.color.b = 0.0
            landMark_array_msg.markers.append(marker)
        self.landMark_pub.publish(landMark_array_msg)
```

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 拓展卡尔曼滤波EKF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：

```python
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：

```python
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
```

其中在进行icp输入前，我们需要对不同点数的组进行额外的处理与消除，代码如下：

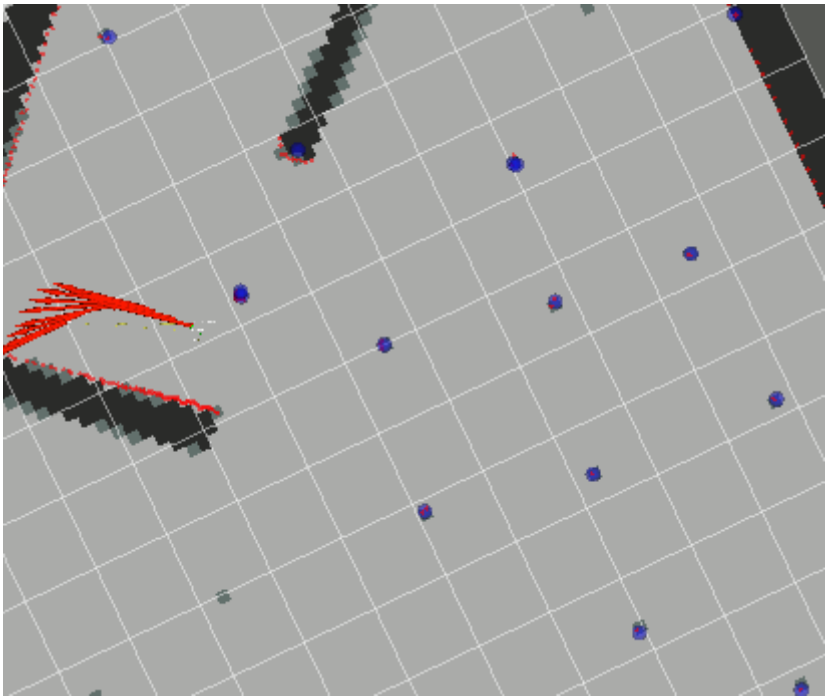
```python
        mylmtar,mylmsrc= self.findNearest(mylmtar,mylmsrc)
        #print("delta")
        lmnorm=np.linalg.norm(mylmtar-mylmsrc,ord=2,axis=0,keepdims=True)
        #print(lmnorm)
        totalrow=lmnorm.shape[1]
        i=0

        while True:
            if lmnorm[0,i]> 0.5:
                mylmtar = np.delete(mylmtar, i, axis=1)
                mylmsrc = np.delete(mylmsrc, i, axis=1)
                lmnorm = np.delete(lmnorm, i, axis=1)
                i=i-1
            i=i+1
            if(i>=mylmsrc.shape[1]):
                break
```

五、实验运行结果与分析

5.1 激光特征检测与提取

我们要提取的激光特征主要有两组，一组是每一次laserscan的消息中提取出来的特征，另一组是上一帧输出定位坐标的模拟激光的特征提取，为了展示激光特征的效果，我们只对每一次真实的laserscan的消息提取出的特征进行展示。



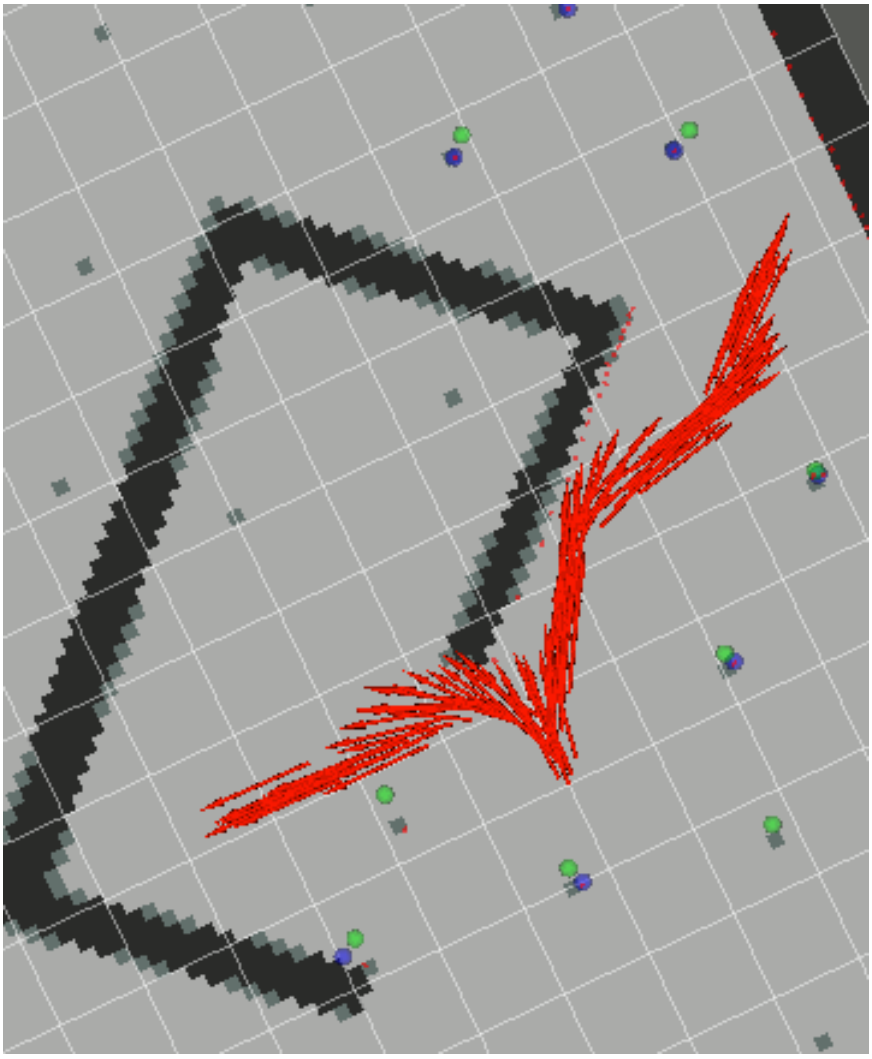
如图所示，可以看到，在激光的视野内，找到的特征点数较为完整，几乎打得到的所有特征都有所显示，而位置也较为精确。

关于提取特征的精确度优化，会在实验心得讨论部分提到。

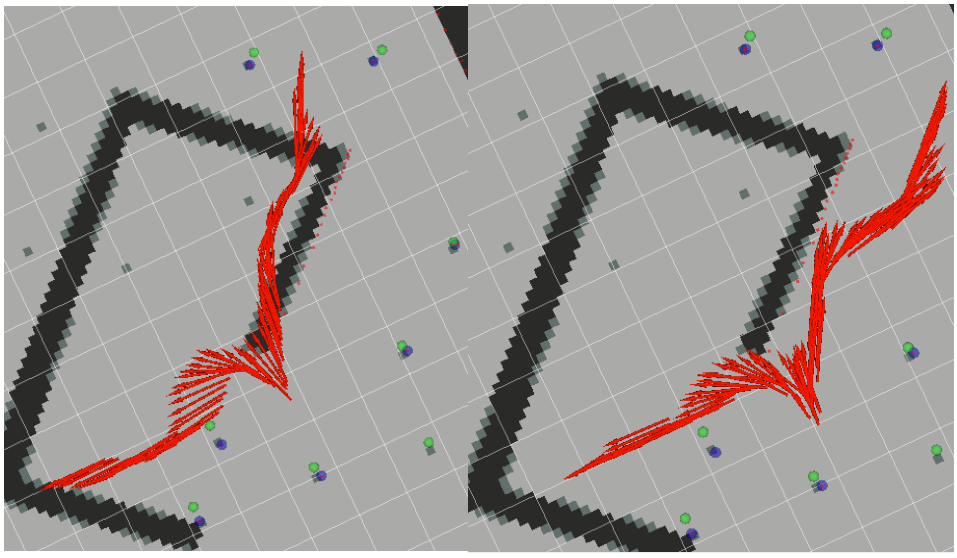
5.2 基于特征观测的卡尔曼滤波的定位效果

rosbag1的结果：

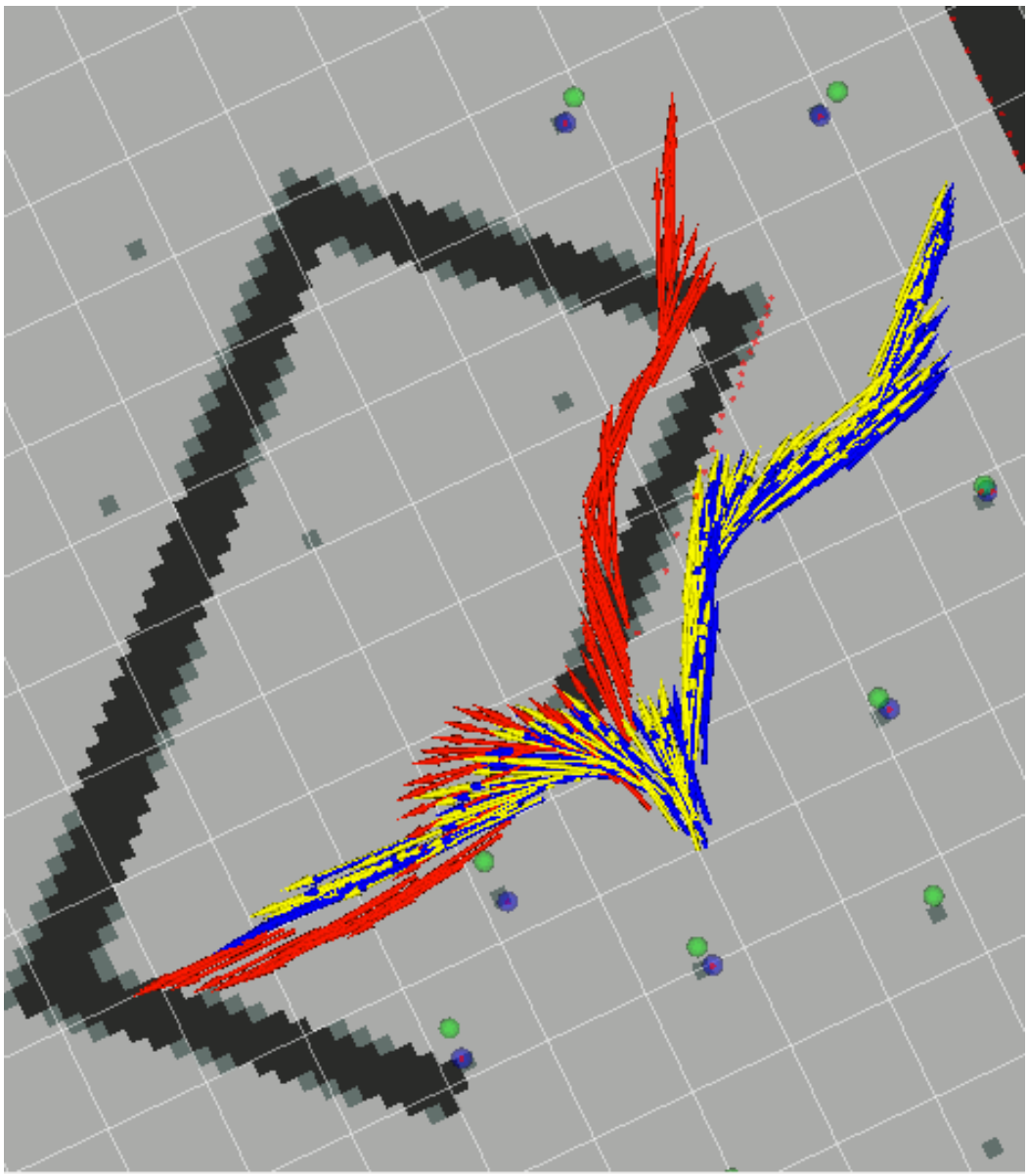
我们首先单独展示基于特征观测的ekf定位(w10结果)：



作为对比，我们将单独的icp激光里程(W7)与基于全局地图的ekf定位(W8)也一同展示如下

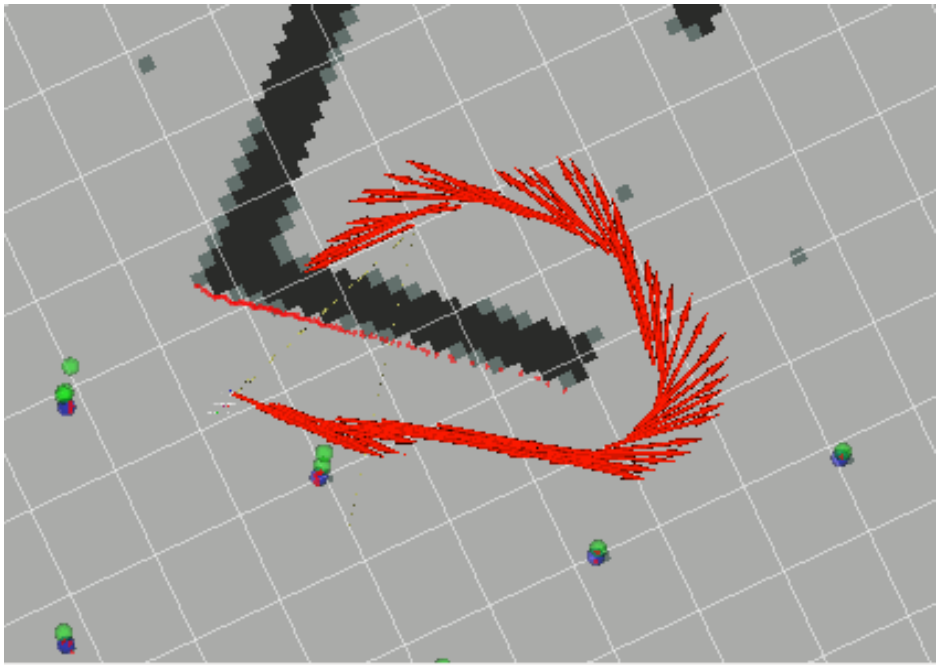


最后将三者合在同一张图中进行展示如下，红色的为单独的ICP，蓝色的为全局激光的EKF而黄色的为特征提取之后的EKF，结果如下：

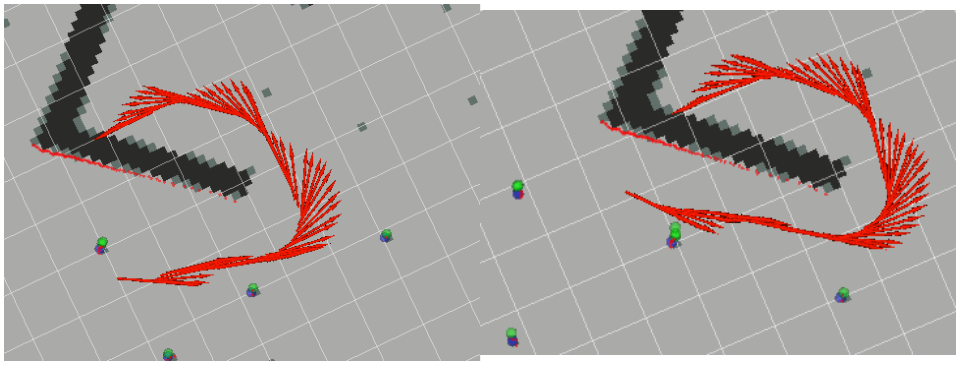


rosbag2的结果

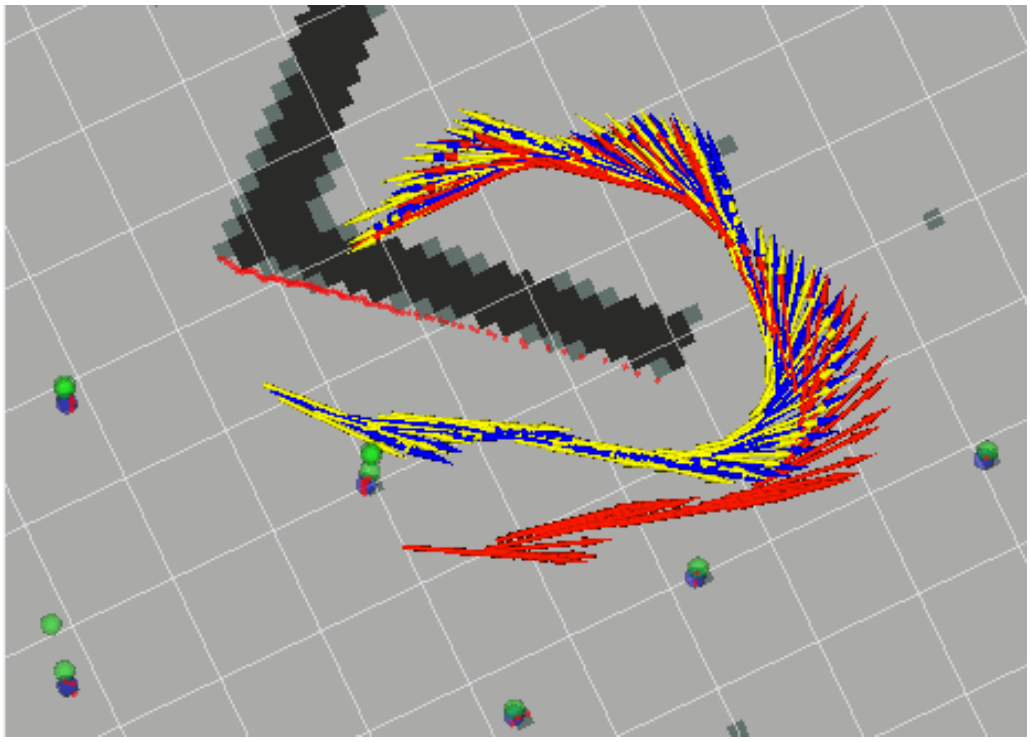
我们首先单独展示基于特征观测的ekf定位(w10结果)：



作为对比，我们将单独的icp激光里程(W7)与基于全局地图的ekf定位(W8)也一同展示如下。其中，左边为单独的icp激光里程，右边为基于全局地图的ekf定位



最后将三者合在同一张图中进行展示如下，红色的为单独的ICP，蓝色的为全局激光的EKF而黄色的为特征提取之后的EKF，结果如下：



picture:运动中某处的细节大图

结合rosbag1与rosbag2的结果，通过对运动中三种定位方式的对比，我们可以初步地看到单独的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，而采用全局激光的和采用特征提取的EKF定位的误差都相对更小，且两者的结果比较接近，使用特征提取的定位结果相对采用全局激光的ekf定位波动稍微会更大一点。但相对而言特征提取与全局激光的ekf定位误差相对于ICP激光里程定位的误差都比较小。

5.3 不同定位方式的性能分析

接下来我们对不同定位方式的性能进行分析，主要从误差大小、运算规模与运行时间等角度入手进行分析。

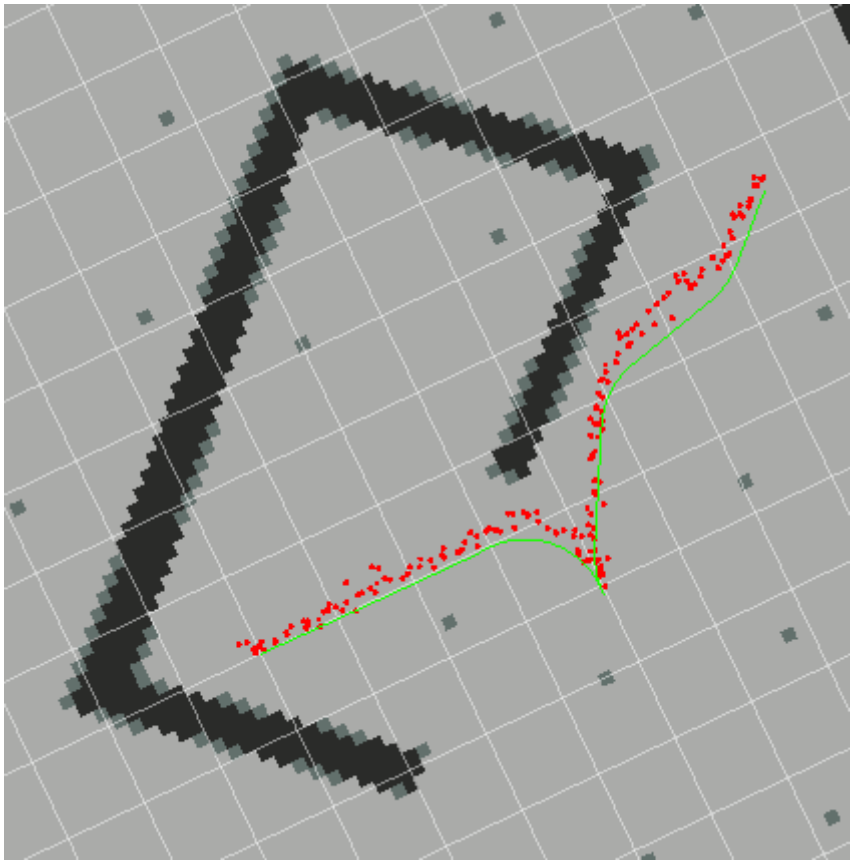
5.3.1定位的误差分析

rosbag1:

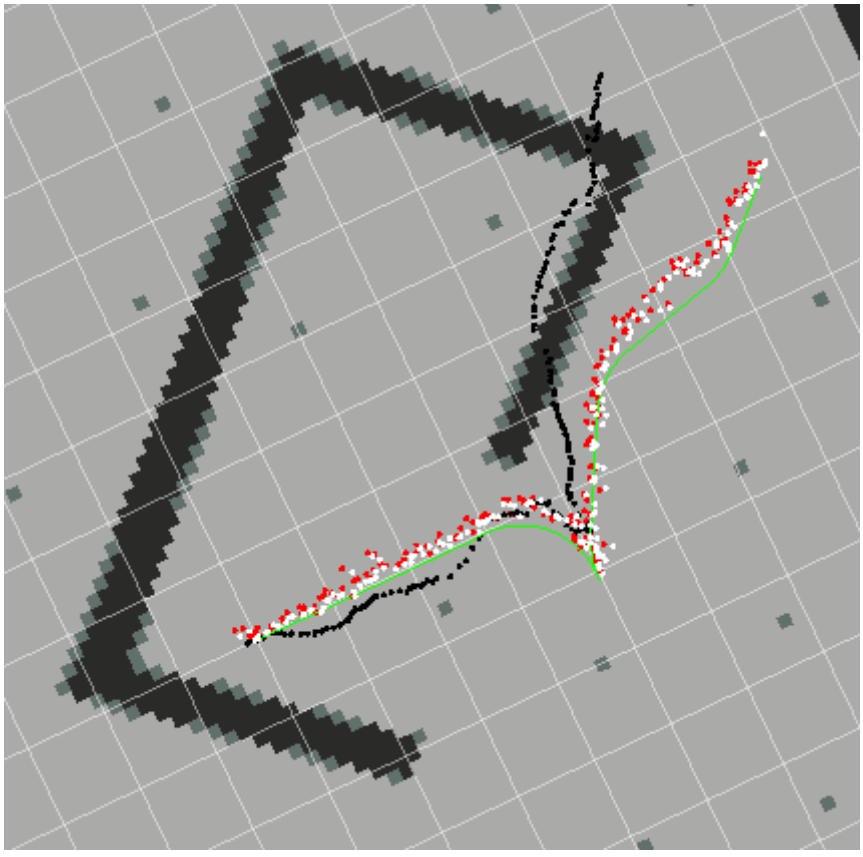
我们先对rosbag1的结果进行定性观察与定量分析

定性观察:

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，表示为图中绿色的线，而对特征提取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。



接下来我们将单纯ICP激光里程、使用全局激光的ekf和特征提取的ekf三种定位方式的散点定位效果进行合并的展示，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激光里程以及全局ekf、提取特征的ekf输出的结果进行路线显示与打点的方式进行展示，其中黑色的pointcloud为单纯ICP激光里程的结果，白色的pointcloud为全局激光的ekf输出的定位结果，红色的的pointcloud为特征提取的ekf输出的定位结果，绿色的path则为小车的真实路线，如下图所示。

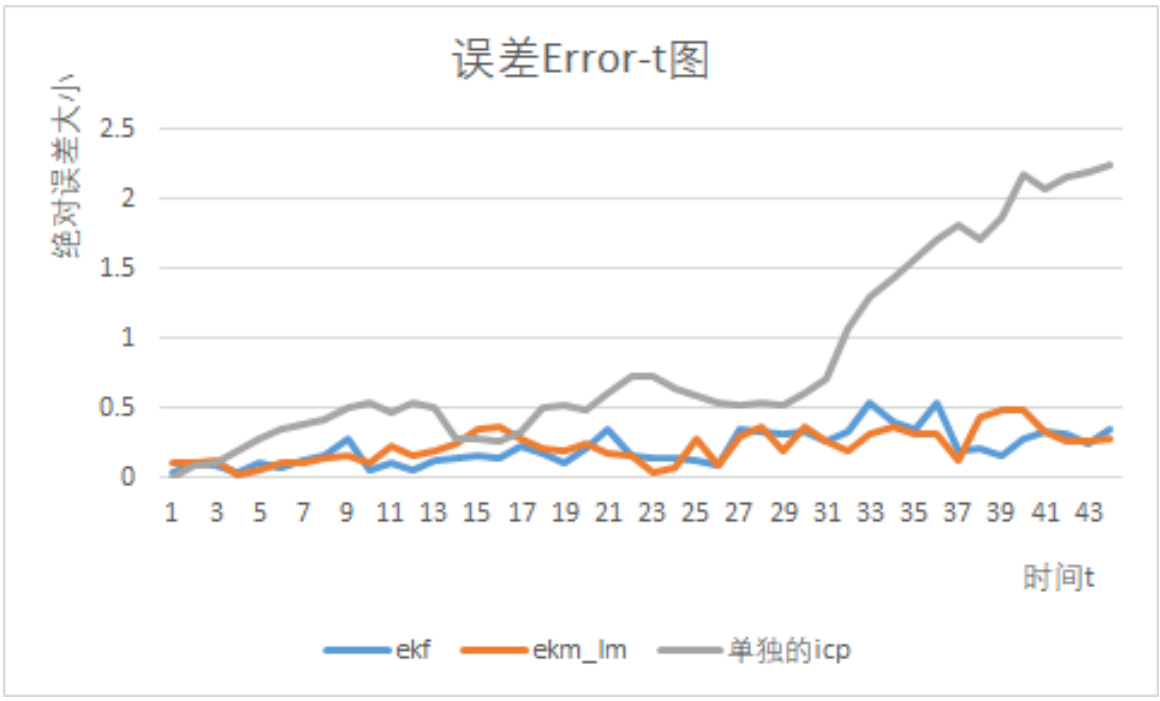


接下来我们对误差进行定量分析：

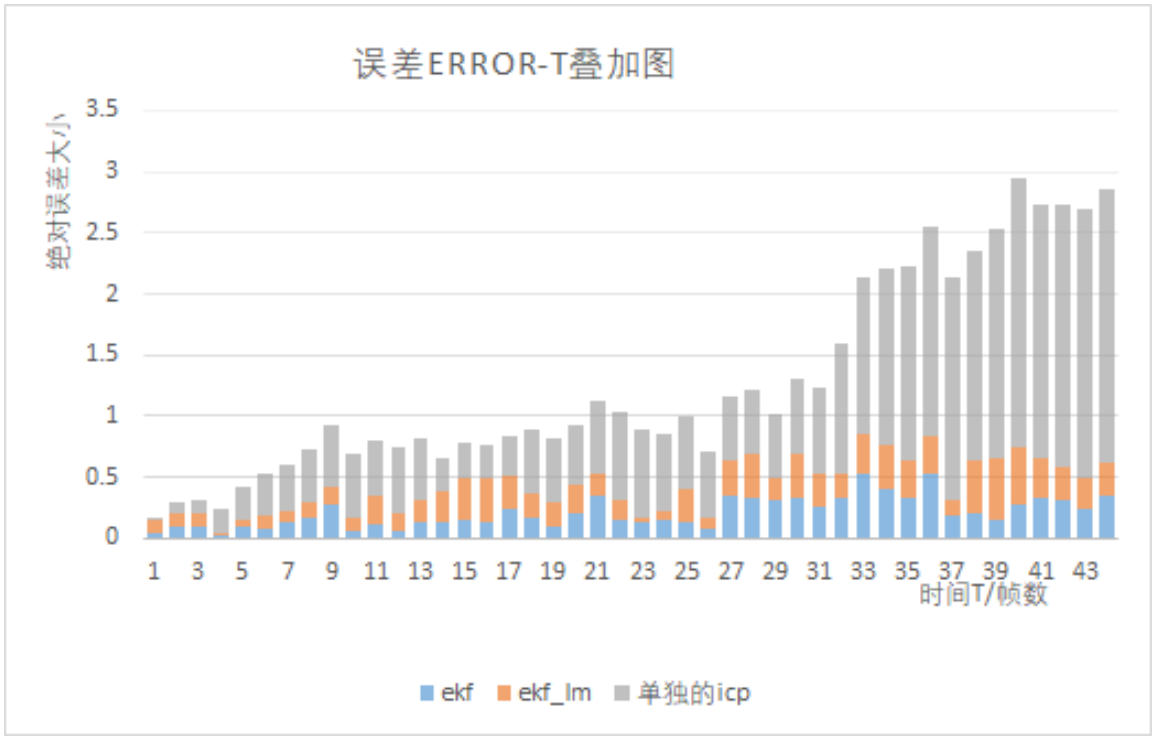
我们对误差的结果进行定量分析，误差即为与真实值的偏移量，我们定义每一种定位方式的绝对误差大小为

$$error = \mathrm{dist}(dx, dy) = \sqrt{(x - x_{\mathrm{true}})^2 + (y - y_{\mathrm{true}})^2}) \tag{12}$$

我们统计不同定位方式绝对误差随着帧数变换的数据(每4帧取一次)，绘制图像如下：



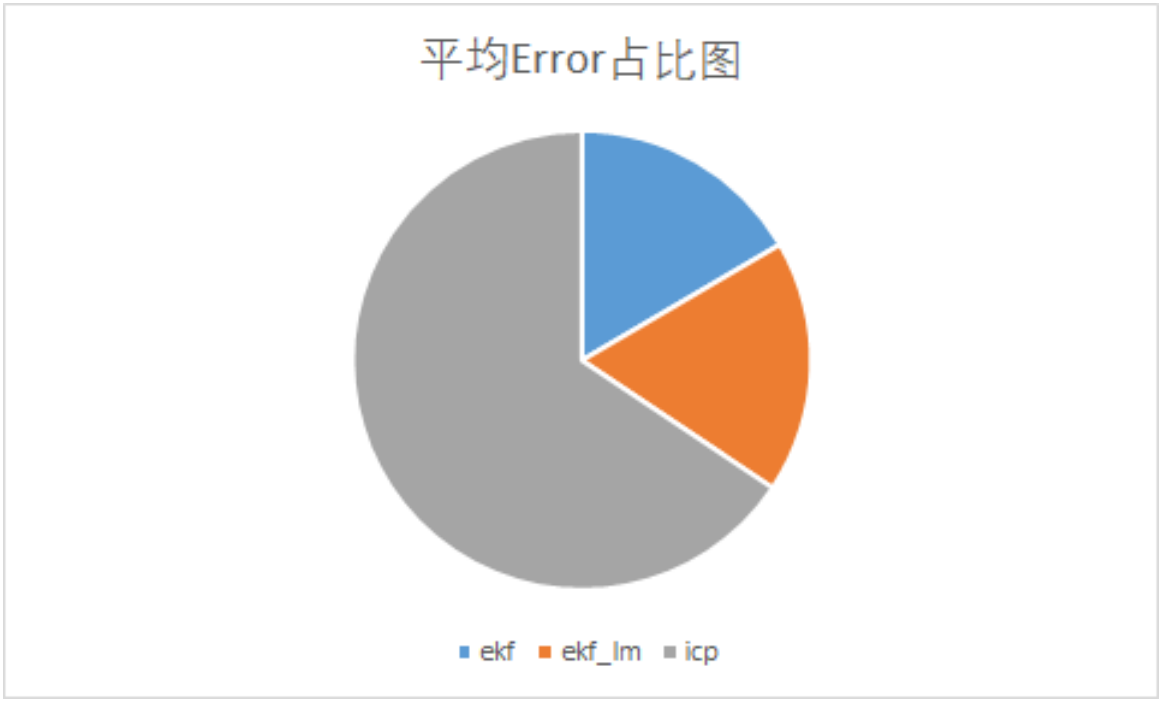
为了更加直观地展示三种定位方式的误差的比例大小，我们使用更加直观的误差比例展示图进行展示。



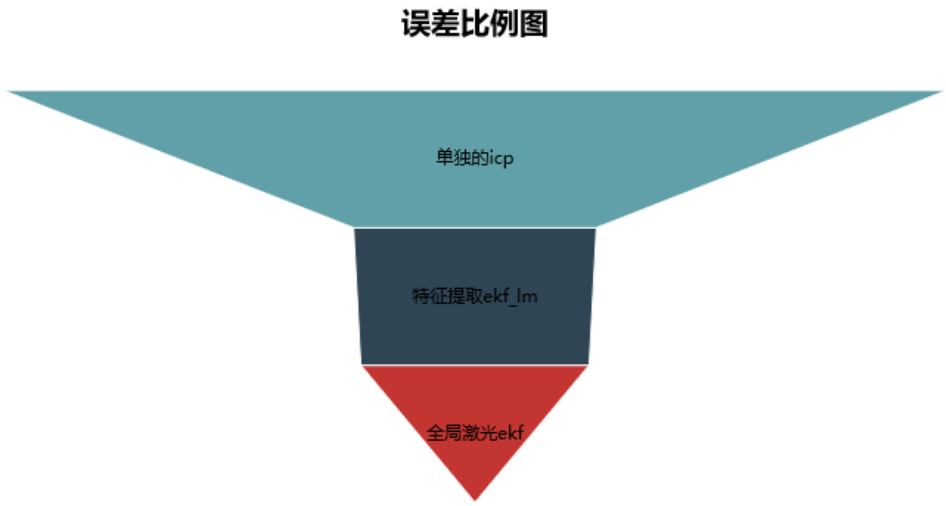
我们对过程中的每个时间的误差进行不同定位方法的分别求平均，误差的平均值如下表所示：

定位方法	ekf	ekf_lm	icp
平均误差	0.2102	0.2435	0.8683

使用饼状图与漏斗图进行展示



漏斗图如下：

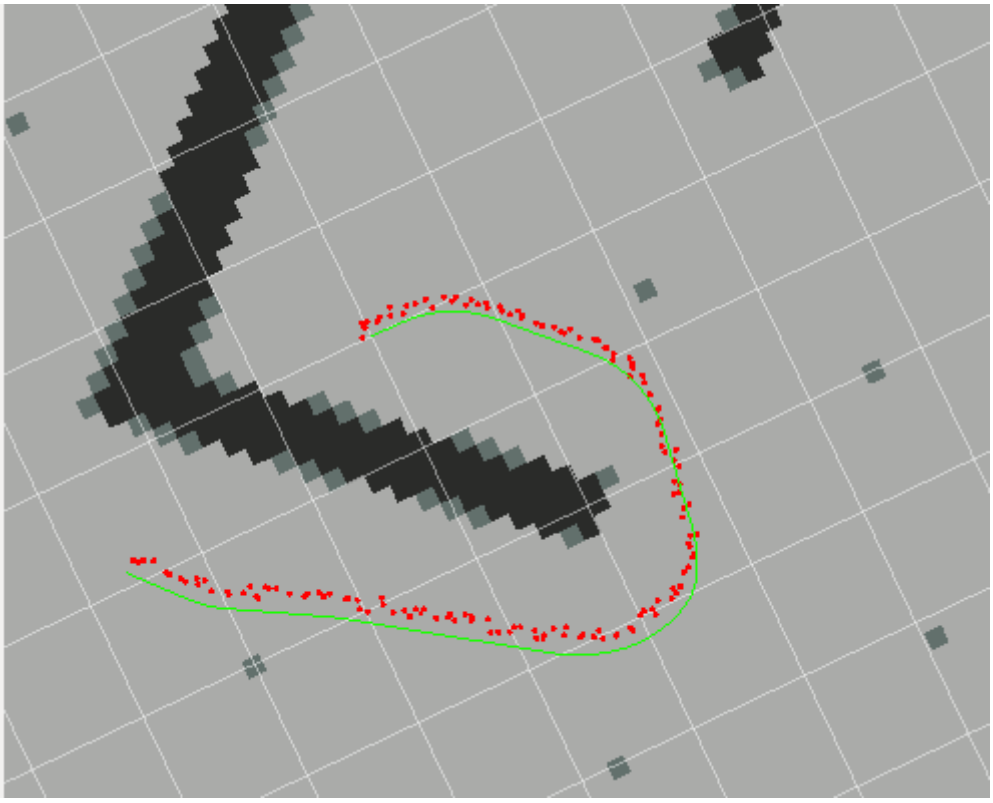


rosbag2的定位误差:

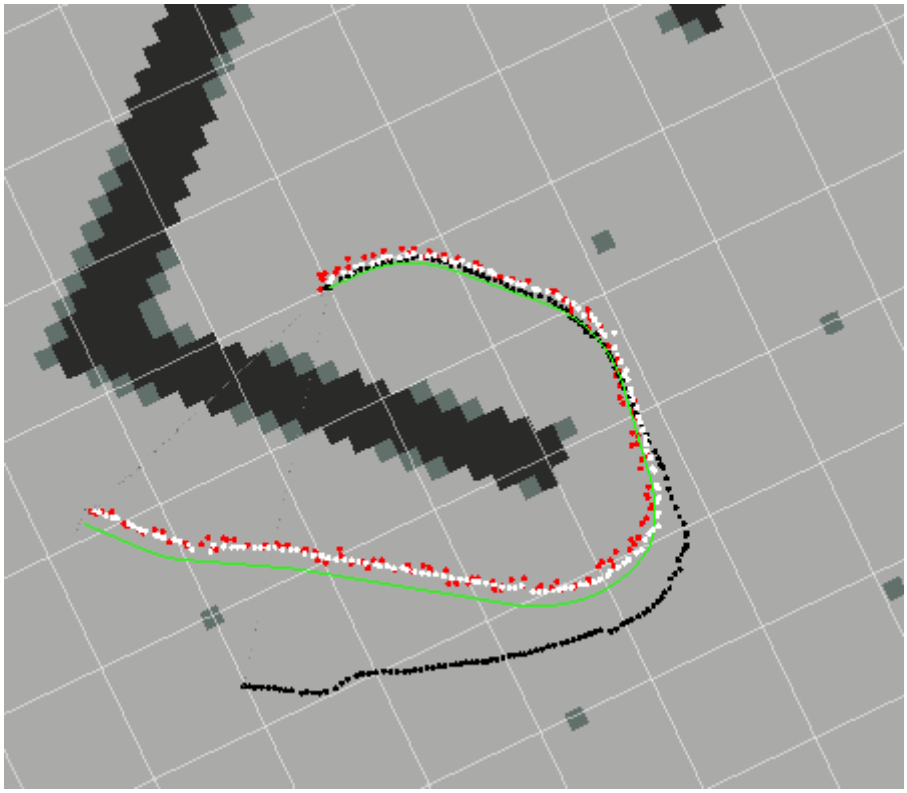
接下来我们对rosbag2的结果进行定性观察与定量分析

定性观察

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，表示为图中绿色的线，而对特征提取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。



接下来我们将单纯ICP激光里程、使用全局激光的ekf和特征提取的ekf三种定位方式的散点定位效果进行合并的展示，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激光里程以及全局ekf、提取特征的ekf输出的结果进行路线显示与打点的方式进行展示，其中黑色的pointcloud为单纯ICP激光里程的结果，白色的pointcloud为全局激光的ekf输出的定位结果，红色的的pointcloud为特征提取的ekf输出的定位结果，绿色的path则为小车的真实路线，如下图所示。

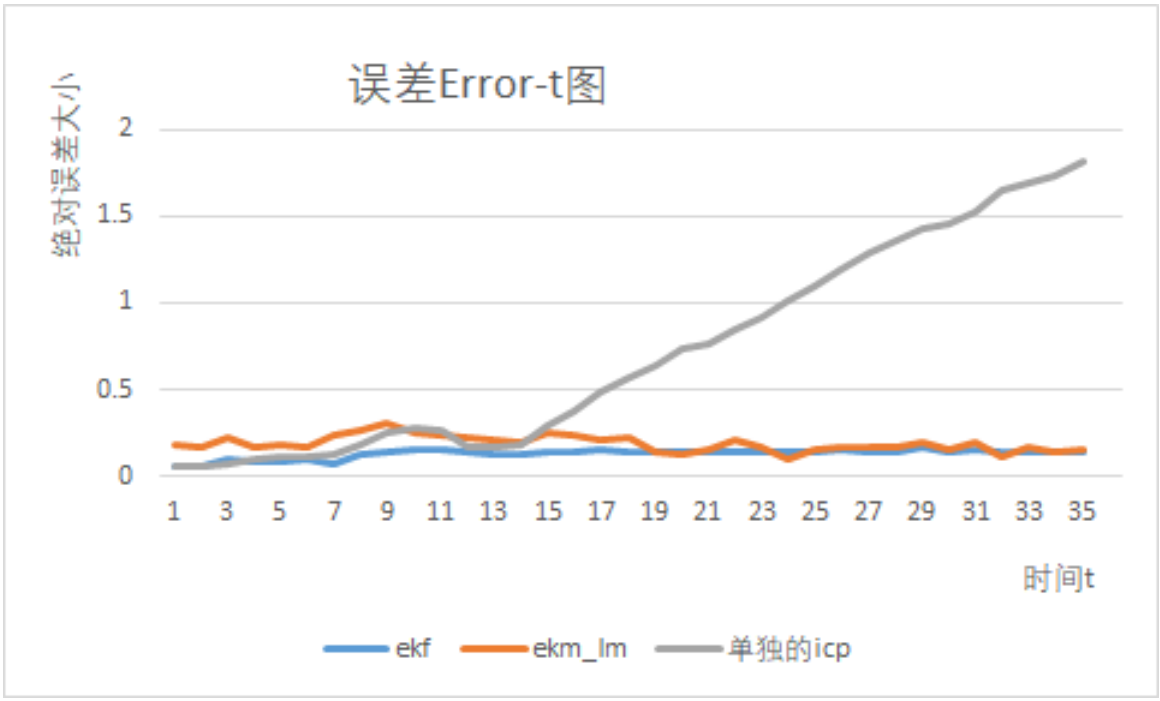


接下来对误差进行定量分析：

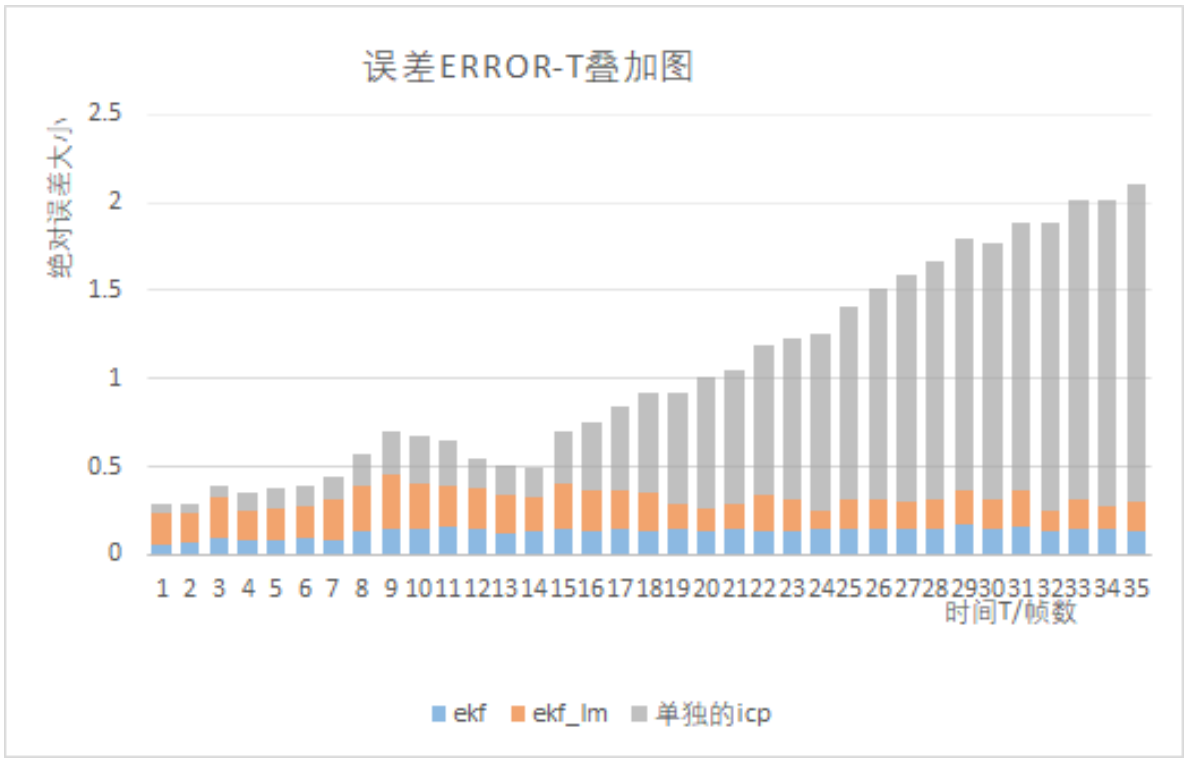
我们对误差的结果进行定量分析，误差即为与真实值的偏移量，我们定义每一种定位方式的绝对误差大小为

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

我们统计不同定位方式绝对误差随着帧数变换的数据(每4帧取一次)，绘制图像如下：



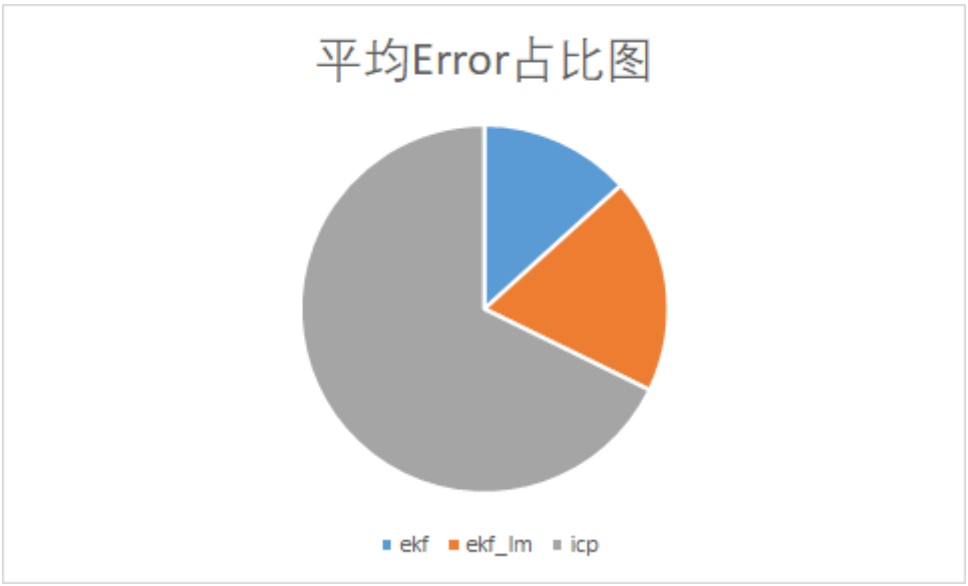
为了更加直观地展示三种定位方式的误差的比例大小，我们使用更加直观的误差比例展示图进行展示。



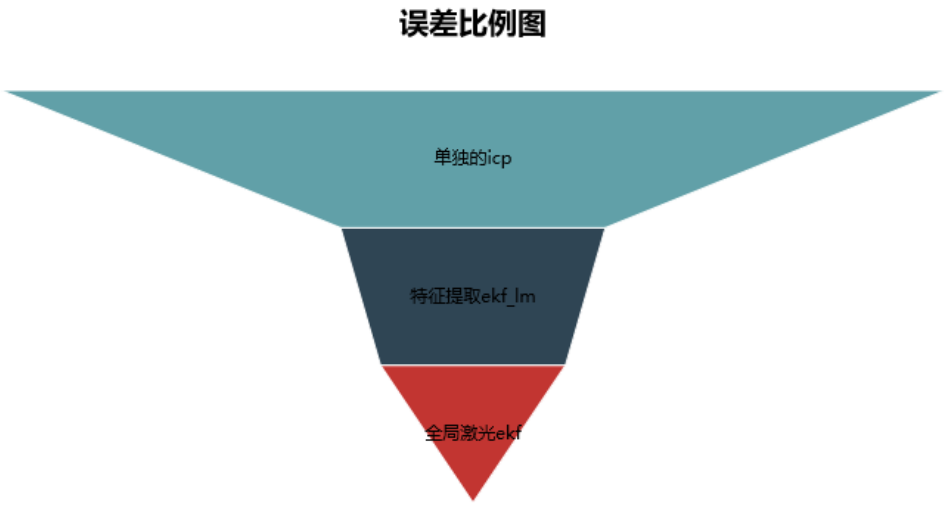
我们对过程中的每个时间的误差进行不同定位方法的分别求平均，误差的平均值如下表所示：

定位方法	ekf	ekf_lm	icp
平均误差	0.130	0.186	0.663

使用柱状图与漏斗图进行展示



漏斗图如下：



定位的误差总结：通过rosbag1与rosbag2的误差的定量观察与定性分析，我们可以看出来，采用全局激光的和采用特征提取的EKF定位的误差都相对更小，且两者的结果比较接近，在两个rosbag中特征提取的误差相比于全局激光的误差会相对更大一点点，主要体现为使用特征提取的定位结果相对采用全局激光的ekf定位波动稍微会更大一点。而采用单纯ICP的激光里程计的误差相比于这两者来说要大很多。

5.3.2定位消耗的性能分析

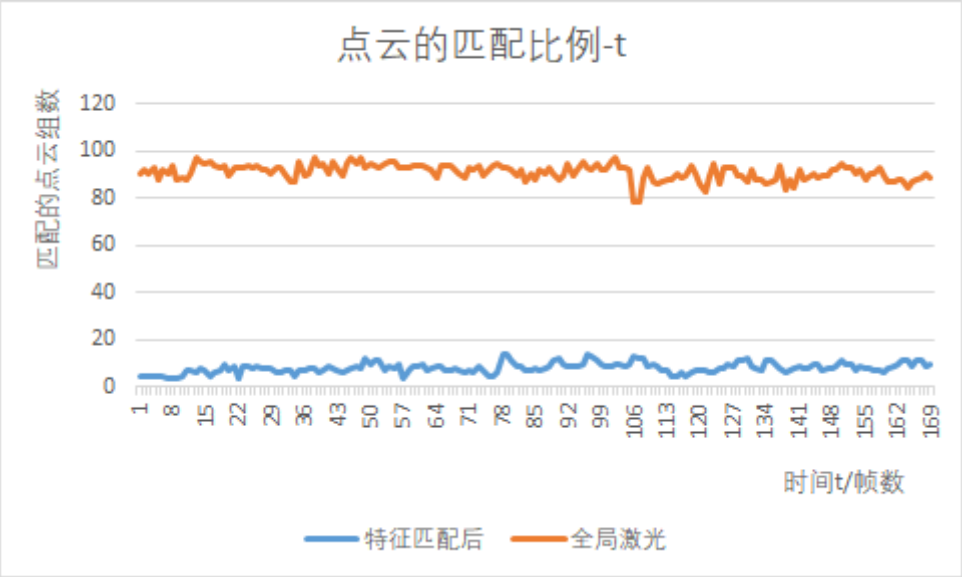
接下来我们对不同定位方式的性能消耗进行测试。

进行ICP匹配时输入的点数规模测试

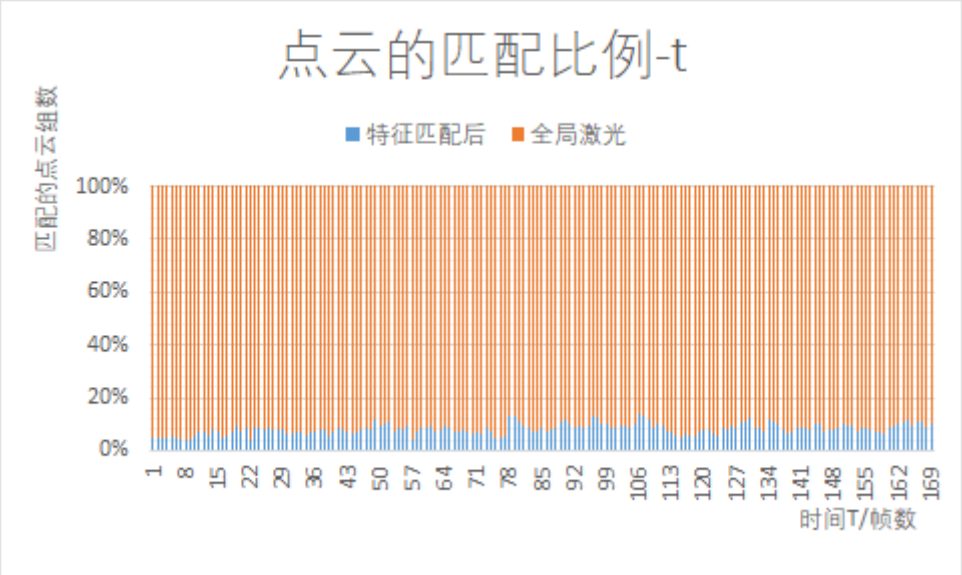
进行运算时，影响时间的一个重要因素就是输入icp匹配的点云组数，而点云组数又直接和laserscan的总激光条数相关(bag1为100条,bag2为300条)，但后续进行icp运算与ekf迭代更新的主要是经过最近邻筛选后输入到icp变换求解中的组数（会略小于总的激光条数），而进行特征提取的时候提取的特征点数也直接与总的激光条数相关，因此先对输入ICP变换的组数规模进行测量是很有必要的。

对rosbag1:

我们分别对全局激光与特征匹配使用的ICP符合要求的匹配点数(输入icp匹配的点云个数)进行测量，注意到全局激光的输入由于有的点两帧之间匹配结果距离过大而被丢弃，这个点数并不是初始的激光线数（300与100），我们对得到的结果进行折线图的绘制。



采用更加直观的百分比柱状图对比例进行展示:

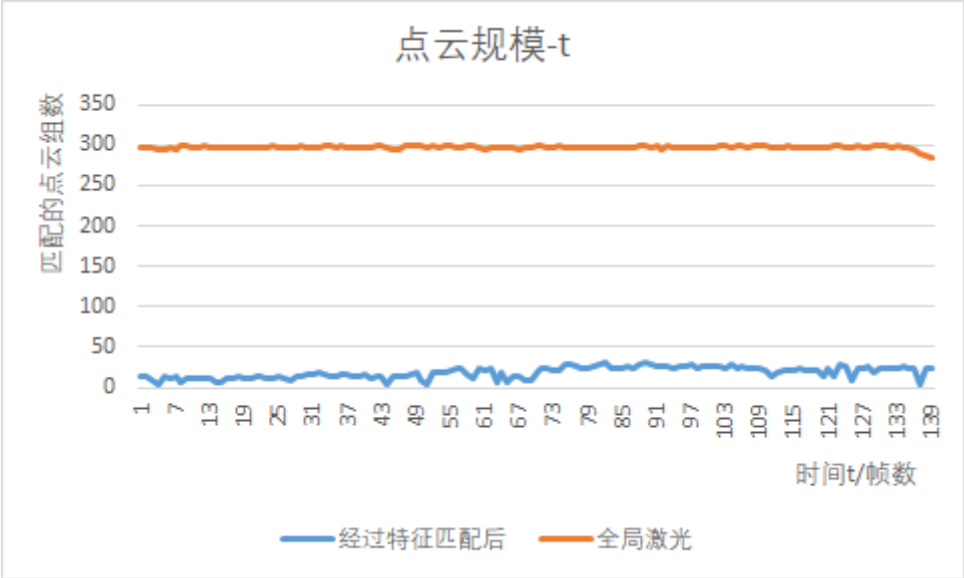


对全过程的每一帧的icp输入组数进行取平均，大小如下表所示

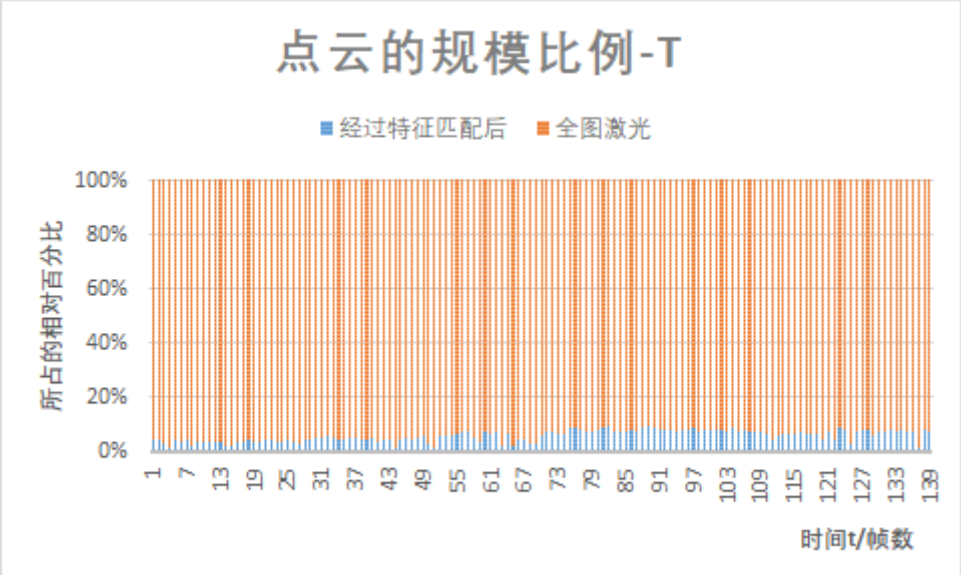
激光的处理方式	使用全局激光	使用特征匹配
平均组数	91.31	8.20

对rosbag2:

我们分别对全局激光与特征匹配使用的ICP符合要求的匹配点数(输入icp匹配的点云个数)进行测量，注意到全局激光的输入由于有的点两帧之间匹配结果距离过大而被丢弃，这个点数并不是初始的激光线数（300与100），我们对得到的结果进行折线图的绘制。



采用更加直观的百分比柱状图对比例进行展示：



对全过程的每一帧的icp输入组数进行取平均，大小如下表所示

激光的处理方式	使用全局激光	使用特征匹配
平均组数	297.54	18.76

进行EKF定位时消耗的时间分析

我们使用python中的时间模块,对不同定位方式(这里给出了五种不同的选择)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bag进行分别计时，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：6.30ms

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：15.10ms

使用特征提取的ICP激光里程计(纯ICP+landmark)，单次定位的平均耗时为：3.97ms

当观测模型使用特征提取时(EKF+观测模型用特征提取)，单次定位的平均耗时为：10.51ms

当观测模型与预测模型都使用特征提取时（EKF+两个模型用特征提取），单次定位的平均耗时为:6.73ms

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：100.39ms

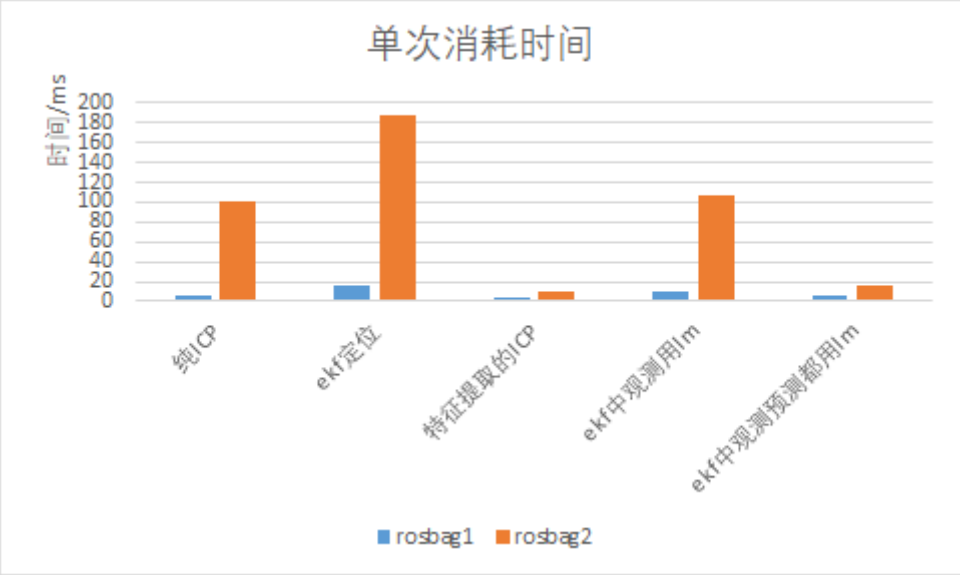
使用全局的地图激光进行EKF定位（W8 ），单次定位的平均耗时为：187.59ms

使用特征提取的ICP激光里程计(纯ICP+landmark)，单次定位的平均耗时为：8.97ms

当观测模型使用特征提取时(EKF+观测模型用特征提取)，单次定位的平均耗时为：107.42ms

当观测模型与预测模型都使用特征提取时（EKF+两个模型用特征提取），单次定位的平均耗时为:16.02ms

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(rosbag2为18.76而bag1为8.20)，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，对于每一个rosbag内部，使用特征提取时，可以大大减少进行icp匹配时的组数，对bag1，有无特征提取时的组数为8.20与91.31，对bag2，有无提取时的组数为18.76与297.54，而这样的规模的变化也直接导致了单次定位的消耗时间差距很大。

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，使用特征提取的EKF定位可以很大幅度减少运行的时间，从而减少性能消耗。

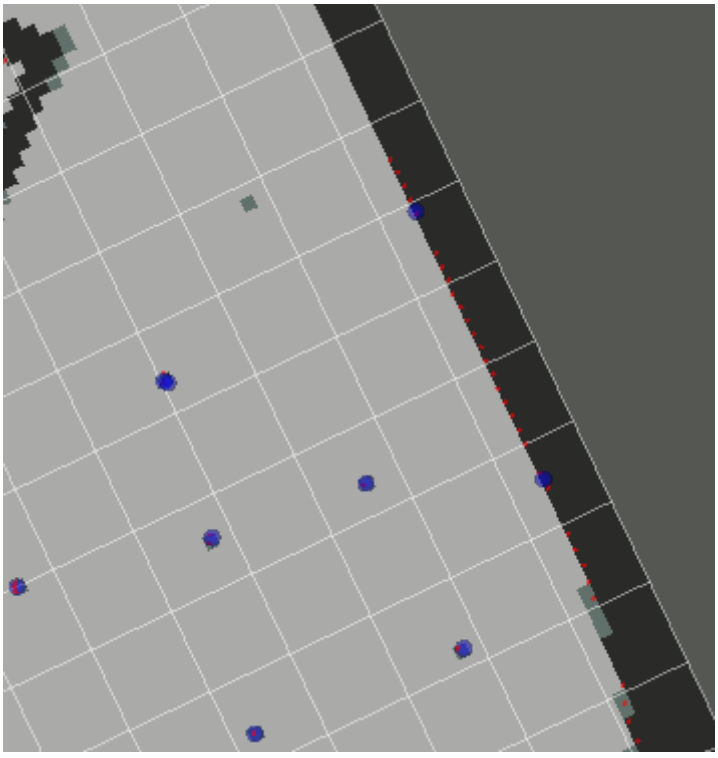
5.3.3不同定位方式的性能分析总结

通过对三种定位方式的性能分析（ICP激光里程，全局激光的EKF定位，特征提取的EKF定位），我们可以发现从误差的角度，单纯的ICP误差很大，而全局激光的EKF定位和特征提取的EKF定位误差相对而言都很小，特征提取的定位相对会大一点点。而从消耗时间的角度来说，使用特征提取的EKF定位消耗的时间远小于另外两者。综上所述，当我们需要兼顾运行时间和定位精度时，我们应当使用本次实验中的基于特征观测的EKF定位，而如果我们追求更好的精度（虽然精度的提升只有一点点）就使用全局激光的EKF定位，而如果没有地图信息的话，现在我们就只能使用ICP激光里程计（当然我们在后面的实验中可以进行地图与特征构建）的方式来进行定位。

六、实验思考与心得

6.1关于对landmark提取时精准度的提高

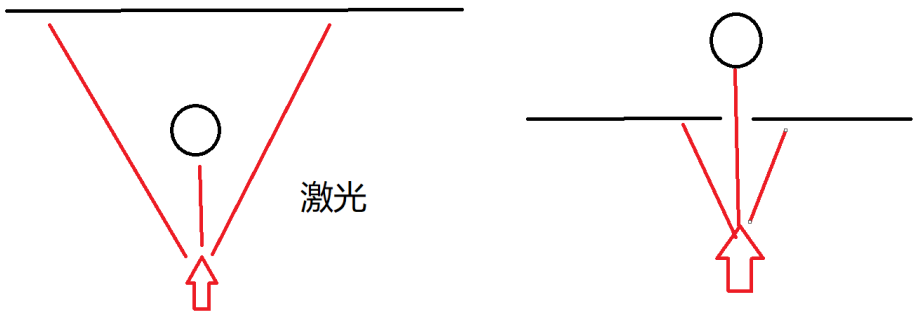
在最开始进行landmark的特征提取时，我发现有的不是landmark的部分也会被检测为landmark，有的墙面处也会出现被检测为landmark的情况，如下图所示：



因此我们需要进行优化：

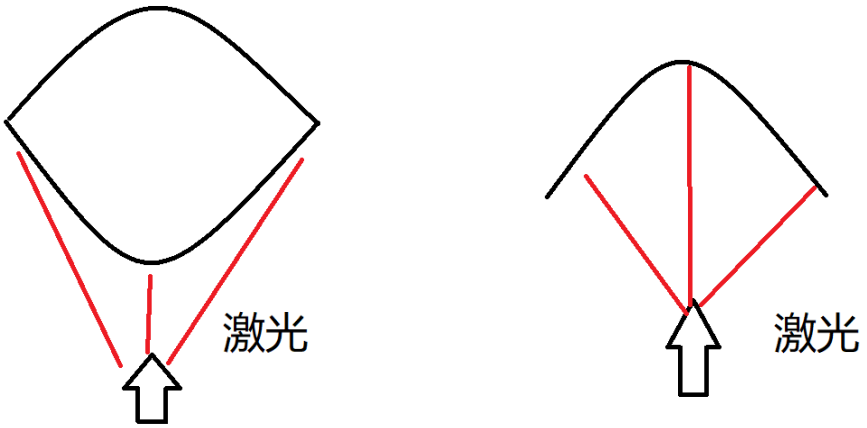
(a)如果该簇只有一个点的情况：该簇内只有一个点的时候，我们需要对该簇与所有簇的ranges进行对比，避免出现两边的簇更近，而单个的簇更长的情况出现。

如下图所示,左边是合理的而右边的情况在我们的地图中几乎没有出现，因此我们可以把右边这样的情况找到的点进行舍弃。



(b)该簇内有多个点的情况：由于我们检测的是圆柱体，一个重要的原则就是同一簇内多个激光的趋势大约是中间更小两边更大的，不可能在内部(非两边端点)出现一个极大值

如下图所示，在同一簇内，左边的激光扫描中间ranges小两边ranges更大，是合理的，而右边的激光扫描中间ranges大两边ranges更小，是不合理的，不应该检测为特征点。



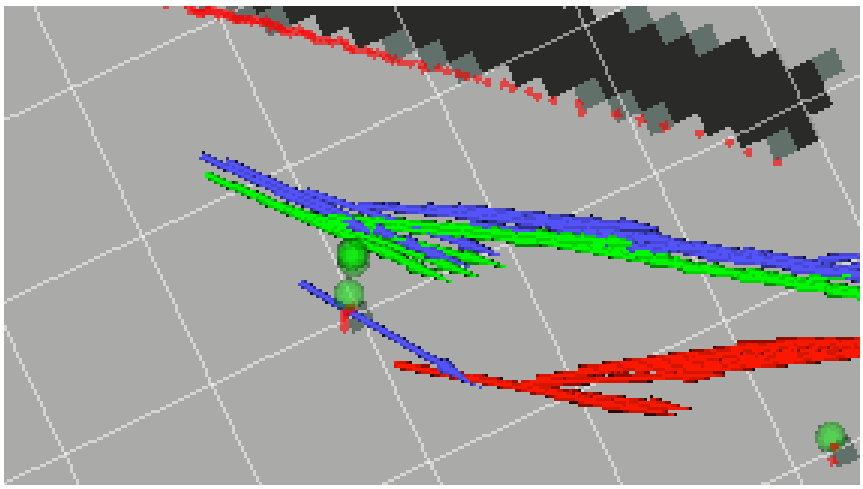
(c)其他细节的处理：即首尾部分的处理，我们需要特别注意到激光的$-\pi$与$+\pi$处的激光实际是连接的，我们需要考虑最后的一簇激光是否需要与最初的一簇激光进行合并，避免出现首尾同一处特征点被分开标记两次

6.2关于尝试减小特征提取的EKF定位误差的心得与讨论

(a)我们应该对特征提取后匹配的点进行检验，与之前的EKF全局激光比较类似，特征提取的比较特殊的一个地方在于可能出现两组点云数量不一致、可能前后两组有前面的特征不再观测到、加入的之前没有观测到的特征等现象(实际这个现象在后面的实验中会有所用途)。因此我们需要对前后两帧特征最近邻匹配的结果进行检测与约束，避免最近邻的两个点不是同一个特征的情况，约束条件如下所示：

$$X_{tar} - X_{src}[in\ matching\ order] < Distance_{maxmoving} \tag{15}$$

(b)我们应该控制特征提取的点数不能过少，在刚开始的实验中，由于在有的帧的地方提取出的特征点过少，会在该处出现匹配后的变换不稳定的现象，导致定位到达该处时出现误差一下子变大的情况。我们需要对每一次进行ICP匹配的点数最小值进行设定，我在实验中设定的阈值为4，如果点数过少，则跳过这一帧，与下一帧匹配。如果不进行约束则可能出现在该点之后出现如下图所示的定位漂移的现象。

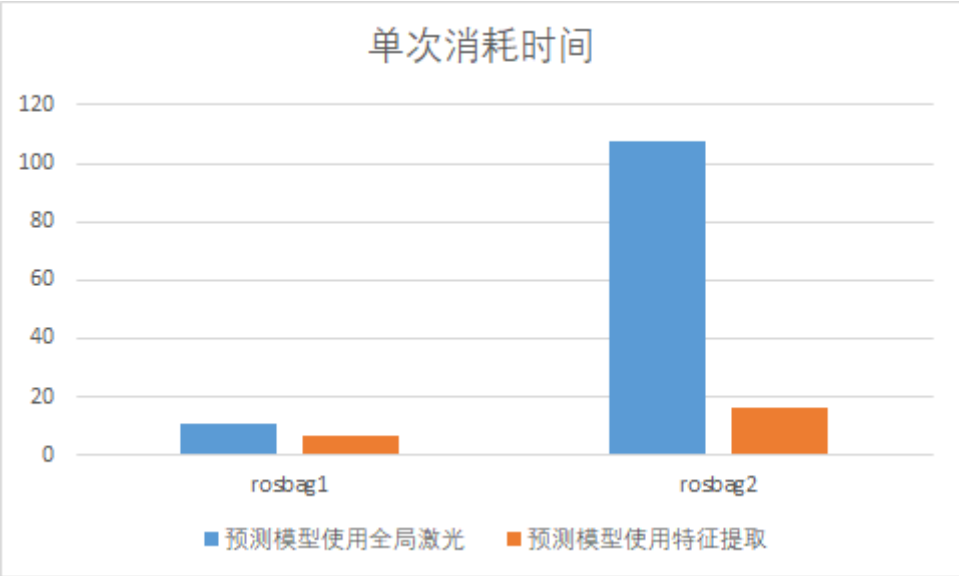


picture:有一帧模拟激光landmark数量过少后出现的漂移

6.3关于进一步缩短消耗时间的尝试

（a）我们在之前的实验中基本采用的是将EKF中的观测模型使用了特征提取的方式，而为了进一步缩短运行时间，我们可以将ekf中的预测模型使用的激光里程计中每相邻两帧激光也从全图激光转化为特征的提取。之前我们的定位等于是一个特征提取的ICP(观测模型)+一个全局激光的ICP(计算预测模型的v)+EKF更新步骤，而其中消耗时间最长的或许就是全局ICP求运动速度的那一部分，现在我们将这一部分也变成相邻两帧的激光的ICP匹配，可以在现有的基础上大幅度降低运行速度。

特别是对rosbag2，我们可以看到，当观测模型使用特征提取时(EKF+观测模型用特征提取)，单次定位的平均耗时为：107.42ms，而当观测模型与预测模型都使用特征提取时（EKF+两个模型用特征提取），单次定位的平均耗时为:16.02ms，对于rosbag1，对应的两个数据分别是10.51ms与6.73ms，如下图所示。



因此我们可以看出，当我们对于误差的要求没有那么严格的时候，我们可以采用EKF中两个模型都使用特征提取，这样可以进一步大幅度减少运算时间，节约运算性能，对于匹配的点数很多的时候效果尤为明显。

值得一提的是，这样做法的误差相对而言并没有太大的变化，得到的定位误差与之前的误差大小差不多，因此这样的提高运行速度的方法是可行的。

（b）我们也可以考虑通过缩短ICP匹配算法本身的$O(M \times N^2)$的时间复杂度来进一步缩短消耗时间，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，使得时间复杂度$O(M \times N^2)$得到降低，提高ICP算法的速度。

6.4 对LaserEstimation的进一步优化

经过调试我发现，有一部分误差出现的原因是`laserEstimation`函数写得不够好，因此在确定某个点的虚拟激光的时候虚拟激光本身就存在一定的误差，从而给后续带来连锁的误差。

因此我们需要结合地图的像素个数(分辨率)对`laserEstimation`函数进行重写与更正。

原有的`laserEstimation`函数如下所示

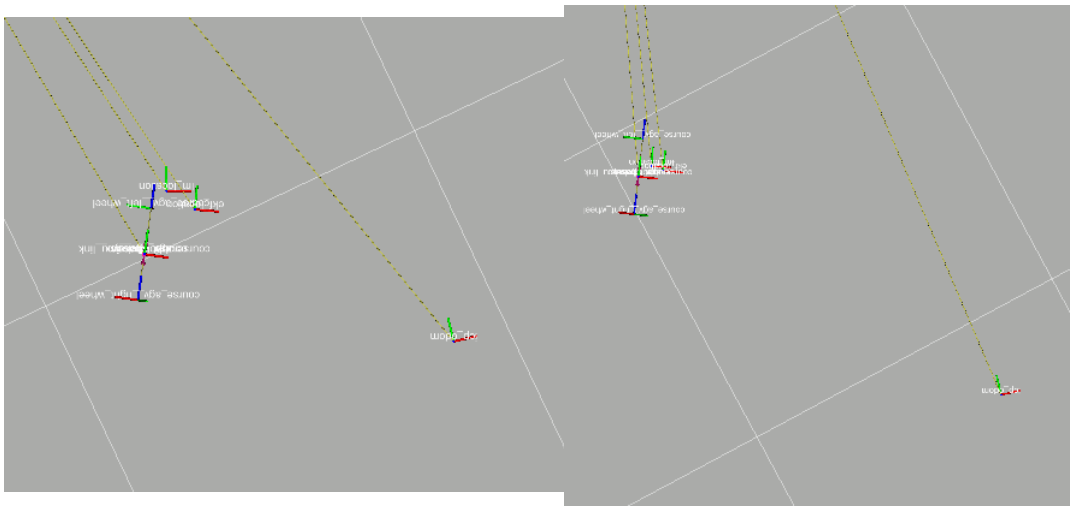
```
def laserEstimation(self,msg,x):
    data=LaserScan()
    data=msg
    data.ranges=list(msg.ranges)
    data.intensities=list(msg.intensities)
    for i in range(0,len(data.ranges)):
        data.ranges[i]=30
        data.intensities[i]=0.5
    for i in range(0,len(self.obstacle)):
        ox=self.obstacle[i,0]-x[0,0]
        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(alphadelta_
        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;
```

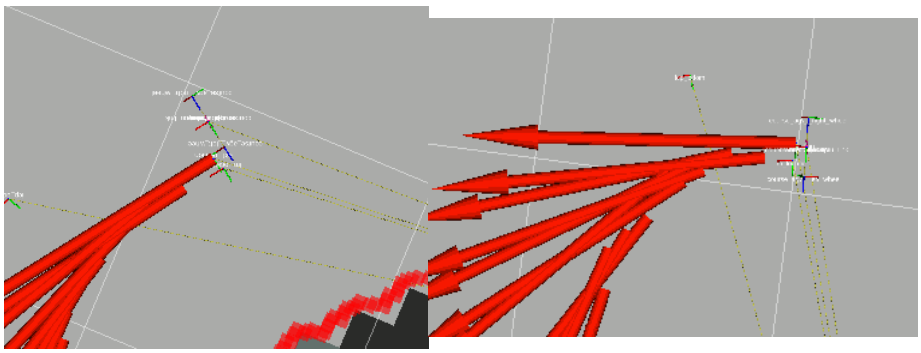
可以看到，原有的地图估计函数采用了无脑的遍历，基本没有考虑到地图真实精度对我们的激光估计的影响，而我们实际把地图的分辨率(resolution)利用起来是很有必要的，我们可以把地图的分辨率作为地图进行遍历时候的单步步长，经过误差修正后的`laserEstimation`函数如下所示

```
    def laserEstimation(self,msg,x):
        data=LaserScan()
        data=msg
        data.ranges=list(msg.ranges)
        data.intensities=list(msg.intensities)
        resolution=self.map.info.resolution
        for i in range(0,len(data.ranges)):
            data.ranges[i]=30
            data.intensities[i]=0.5
            myangle=i*msg.myangle_increment+msg.myangle_min+x[2,0]
            x_num=resolution*math.cos(myangle)
            y_num=resolution*math.sin(myangle)
            for j in range(int(math.floor(data.range_max/resolution))):
                index_x=int(math.floor((j*x_num+x[0,0]-
self.map.info.origin.position.x)/resolution))
                index_y=int(math.floor((j*y_num+x[1,0]-
self.map.info.origin.position.y)/resolution))
                map_data=self.map.data[index_y*self.map.info.width+index_x]
                if(map_data>20 or map_data<-0.5):
                    data.ranges[i]=j*resolution
                    break
        return data;
```

可以看到，我们遍历x_num与y_num时使用的单步步长是地图的分辨率resolution，这样做可以最大利用图片的精度。达到更好的效果，前后的定位效果的对比图如下：左边为修改前，右边为修改之后



在rviz中以箭头的形式显示出来，左边为修改前，右边为修改之后：



可以看到，对比修改前后，对LaserEstimation如果考虑地图分辨率的因素之后可以提高定位的精度，原来的ekf定位相对于精确的位置在左右仍有一定偏差，修改之后定位几乎在非常靠近两轮中心的位置，进一步减少了误差。

心得与体会

第三次实验与报告其实工程量并不算多，但是除了基本的工作之外，在完成实验要求之后，关于实验结果的分析与比较消耗了很多时间与精力，完成最基本的定位并不复杂，但毕竟麻烦的是在后面进行优化与误差的减小，这次实验自己在这方面感觉想了挺多可行的优化方法然后进行一一的尝试，算是一件非常有意义的事情，除此之外，后期的数据处理与展示，误差的分析等方面同样是这次实验的一个超重点，同样这次的报告也突破了万字，，自己也能感受到自己对于编程与调试过程的逐渐熟悉，希望在第四次的实验中可以做得更好！