第 9 章作业讲评

首先确定配准方法，我采用的是 ICP 方法进行配准。由于 ICP 方法不能保证收敛到全局最优解，因此，我们需要提供一个初始解（旋转矩阵 R 和平移向量 T），这个初始解越准越好。

提供初始解的方法很多，在 ROS 的 Navgation Stack 的 amcl 定位包中，初始解是用户在 rviz 交互界面中用鼠标点出来的。在无人驾驶场景下，初始解可以是低精度 GPS 和 IMU 的融合结果。而在本次作业中，我们没有这些获取初始解的渠道，因此，我们需要通过两帧点云的特征点匹配来求一个粗糙的初始解。

首先是特征点的求解。这个在前面的课程中已经讲过了，这里不再赘述。我对比了 PCL 库中的 ISS 和 Sift，最终选择了 Sift 特征点，其表现还是比较稳定的。当然可能是我 ISS 的特征提取方法相关参数没设置好。

```
key_pts_1 = libpcl.keypointSift(np_pts_1, min_contrast=0.09)
key_pts_2 = libpcl.keypointSift(np_pts_2, min_contrast=0.09)
```

其次是特征点的描述子提取。这个在前面的课程中已经讲过了，这里不再赘述。我使用的是 PCL 库中的 SHOT352（和上一节课中其他方法对比下来我觉得表现最好的）。

```
descriptors_1 = libpcl.featureSHOT352(np_pts_1, key_pts_1, feature_radius=5.0)
descriptors_2 = libpcl.featureSHOT352(np_pts_2, key_pts_2, feature_radius=5.0)
```

提取到的描述子可能存在 nan 值，比如周围没有其他点云，就无法建立描述子。这种无效特征点需要删掉。

```
index = np.where(np.isnan(descriptors_1[:, 0]))[0]
descriptors_1 = np.delete(descriptors_1, index, axis=0)
key_pts_1 = np.delete(key_pts_1, index, axis=0)
index = np.where(np.isnan(descriptors_2[:, 0]))[0]
descriptors_2 = np.delete(descriptors_2, index, axis=0)
key_pts_2 = np.delete(key_pts_2, index, axis=0)
```

直觉上，两个特征点能互相匹配上（意味着它们是场景中的同一个点），那他们的特征描述子一定非常相似。特征描述子的相似度计算也很简单。每个特征点的描述子都是一个 352 维的向量，等同于 352 维空间中的一个点。所以直接求两个点的欧式距离，就是两个描述子之间的距离。

得到任意两个描述子之间的距离后，就可以建立两帧点云中特征点的匹配关系。这里采用的是暴力匹配方法。即对于第一帧点云中任意一个特征点 p1，找到第二帧点云中，与其描述子最相似的特征点 p2。若恰好 p1 的描述子也是第一帧的所有特征点中与 p2 描述子最相似的，就算找到了一对匹配点。

```
pairs = []
for i in range(descriptors_1.shape[0]):
  nnid_of_1_in_2 = find_nn_dep(descriptors_1[i], descriptors_2)
  dep_2 = descriptors_2[nnid_of_1_in_2]
  nnid_of_2_in_1 = find_nn_dep(dep_2, descriptors_1)
  if nnid_of_2_in_1 == i:
    pairs.append([i, nnid_of_1_in_2])
```

其中，find_nn_dep(ref_dep, deps)表示从 deps 中找到距离 ref_dep 最近的描述子。

```
def find_nn_dep(ref_dep, deps):
  min_dist = 10e9
  min_id = -1
  for i in range(deps.shape[0]):
    dep = deps[i]
    dist = dist_between_descriptor(ref_dep, dep)
    if dist < min_dist:
      min_dist = dist
      min_id = i
  assert(min_id >= 0 and min_id < deps.shape[0])
  return min_id
```

我们将成功匹配上的特征点对保留下来，其他的特征点全部丢弃。

```
matched_key_pts_1 = key_pts_1[[p[0] for p in pairs]]
matched_key_pts_2 = key_pts_2[[p[1] for p in pairs]]
assert(matched_key_pts_1.shape[0] == matched_key_pts_2.shape[0])
```

确定了所有的匹配结果后，就可以计算初始 R、T。

理想情况下，假设特征点的提取和匹配都是正确的，我们可以找到一组 R 和 T，使得第一帧中（剩下的）的特征点通过对应的旋转和平移后能够和第二帧中（剩下的）的特征点重合。因此，求 R、T 就是求解一个 procrustes 问题。

procrustes 问题求解代码如下：

```
def procrustes(pts_1, pts_2):
  pts_1_nor = normalize_pts(pts_1)
  pts_2_nor = normalize_pts(pts_2)
  u, s, vT = np.linalg.svd(np.matmul(pts_2_nor.T, pts_1_nor), full_matrices=1, compute_uv=1)
  R = np.matmul(u, vT)
  t = np.expand_dims(np.mean(pts_2 - np.matmul(R, pts_1.T).T, axis=0), 0)
  return R, t
```

课堂上的理论证明比较复杂，但代码实现就是把公式用代码的语法抄一遍，所以很简单。

实际操作中，总有些特征点提取错了，或者匹配错了。所以要采用 ransac 的方法，抑制错误匹配带来的负面影响。

Ransac 过程如下：从所有的匹配点对中随机选取 3 对匹配点，求解 procrustes 问题后得到一组 R 和 T，利用该组解对第一帧中的所有特征点进行空间变换;若第一帧中某个特征点在转换后与其在第二帧中的匹配点的空间位置小于阈值(10m，这个拍脑袋定的，可以再调一调)，则记一次投票。得票最多的一组解作为初值。

```
N = 70
outlier_p = 0.1
tau = 10.0
max_inlier_vote = -1
rt_final = []
for step in range(N):
  # selected 3 pairs of points randomly
  selected_pairs = [np.random.randint(0, len(pairs)) for _ in range(3)]
  pts_1 = key_pts_1[[pairs[i][0] for i in selected_pairs]]
  pts_2 = key_pts_2[[pairs[i][1] for i in selected_pairs]]
  # cal r, t
  R, t = procrustes(pts_1, pts_2)
  # vote
  diffs = matched_key_pts_2 - (np.matmul(R, matched_key_pts_1.T).T + t)
  vote = 0
  for i in range(diffs.shape[0]):
    diff = diffs[i]
    if np.linalg.norm(diff) < tau:
      vote += 1
  # record the best vote
  if vote > max_inlier_vote:
    rt_final = [R, t]
    max_inlier_vote = vote
  # stop earlier if reach expectation
  if max_inlier_vote > (1 - outlier_p) * len(pairs):
    # print("stop at[" + str(step) + "].")
    break
# print("max vote: ", max_inlier_vote)
return rt_final[0], rt_final[1]
```

得到初值后，将第一帧点云按照初值进行坐标变换。理想情况下，变换后的第一帧点云和第二帧点云应该已经比较重合了。然后进行 ICP 操作。

ICP 操作其实就是重复上述过程。不同的点在于：

1. ICP 不用求特征点，而是对第一帧中每个点都找到其在第二帧中的匹配点。

2. ICP 不用求描述子，而是直接用两个点的距离建立匹配关系。

详细来说，对于第一帧点云中任意一个点 p1，找到第二帧点云中，与其位置最接近的的特征点 p2，就算找到了一对匹配点。（注意这里不用要求 p1 也是第一帧点云中距离 p2 最近的!!）。

```
# build correspondence
tree = KDTree(np_pts_2, leaf_size=4)
nn_id_array = tree.query(np_pts_1, k=1, return_distance=False)
pairs = []
for i in range(np_pts_1.shape[0]):
  nn_id = nn_id_array[i][0]
  pt_1 = np_pts_1[i]
  pt_2 = np_pts_2[nn_id]
  if np.linalg.norm(pt_1 - pt_2) < 100.0:
    pairs.append([i, nn_id])
assert(len(pairs) > 3)
```

建立好点与点的匹配关系后，求解 procrustes 问题得到一组 R 和 T。将第一帧点云按照 R、T 进行坐标变换。这就完成了一次迭代。理想情况下，每一次迭代，第一帧点云和第二帧点云都会重叠的更好些。

```
# solve r, t
R, t = procrustes(np_pts_1[[p[0] for p in pairs]], np_pts_2[[p[1] for p in pairs]])
```

不断重复上述过程，直到收敛（R 接近单位阵且 T 接近 0 向量）。也可以根据点云的坐标变化程度或者两帧点云的重叠程度来确认是否收敛，如下图。

```
diff = cal_diff(np_pts_1[[p[0] for p in pairs]], np_pts_2[[p[1] for p in pairs]])
if diff < 0.01 or math.fabs(diff - last_diff) < 10e-5:
  break
last_diff = diff
```

其中，cal_diff 的实现如下：

```
def cal_diff(np_pts_1, np_pts_2):
  assert(np_pts_1.shape[0] == np_pts_2.shape[0])
  diffs = np_pts_1 - np_pts_2
  sum_diff = 0.0
  for i in range(diffs.shape[0]):
    sum_diff += np.linalg.norm(diffs[i])
  return sum_diff / np_pts_1.shape[0]
```

需要注意的是，传入该函数的 np_pts_1 和 np_pts_2 要按顺序一一匹配。

在整个过程中，要不断累乘每次算出来的 R，累加每次算出来的 T。比如初始解为 R0，T0；第 n 次迭代算出来的解为 Rn，Tn；3 次迭代后，收敛完成。则最终得到的 Rf 和 Tf 分别为：

Rf = R3 * R2 * R1 * R0，Tf = T0 + T1 + T2 + T3 = T3 + T1 + T2 + T0。

注意：Rf 的计算公式中，Rn 的顺序不能变。对应到代码中，每次算出一个 Rn，更新公式为：

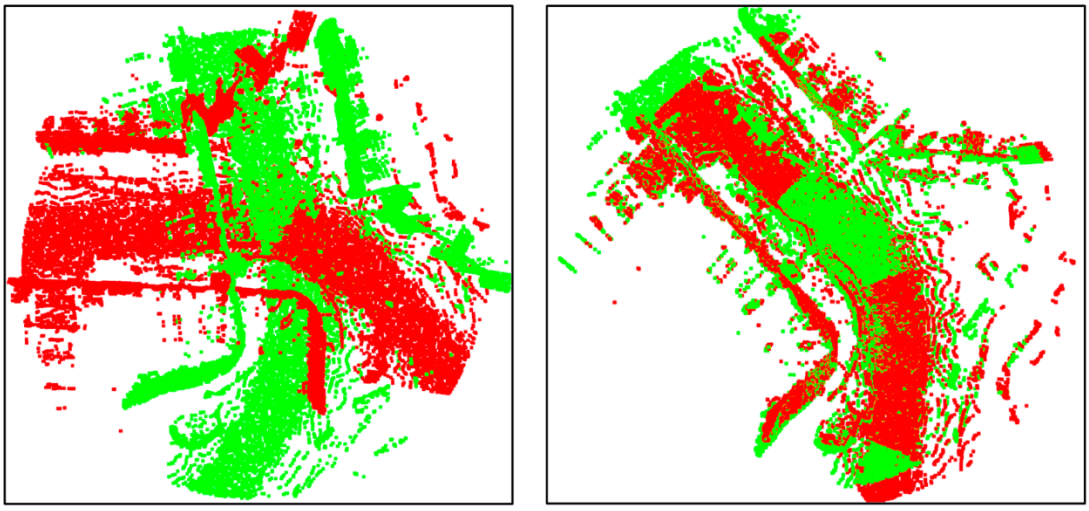
Rf = Rn * Rf

```
# update data
np_pts_1 = np.matmul(R, np_pts_1.T).T + t
R_final = np.matmul(R, R_final)
t_final += t
```

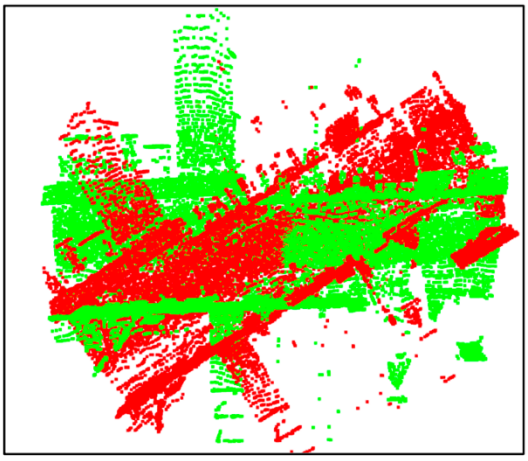
如果等号右边的 Rn 和 RF 写反了，就求错了，变成了 Rf 正确解的逆矩阵。

迭代收敛后，Rf 和 Tf 就是最终的解。

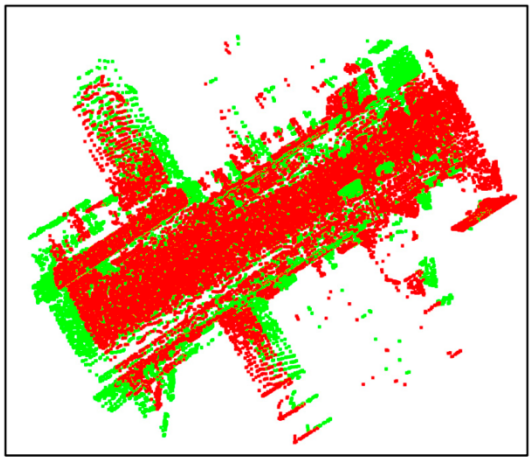
如图是最终效果：



左：初始状态　　右：最终状态



左：初始状态



右：最终状态