Homework of Chapter 1

Task 1

kdtree.py， octree.py， 以及 benchmark.py 详见
https://github.com/xuezhen5267/3d_point_cloud_processing

Task 2 KNN & RNN on Kdtree & Octree

验证 kdtree 中 KNN 和 RNN 算法。

1. Kdtree KNN 算法结果与暴力搜索结果相同，算法混却，如下:
   KNN 搜索结果
   34 - 0.43
   35 - 0.58
   43 - 0.60
   4 - 0.64
   61 - 0.64
   19 - 0.64
   42 - 0.66
   21 - 0.70
   暴力索索结果
   [34 35 43 4 61 19 42 21]
   [0.4275116 0.5761755 0.60188375 0.63887703 0.63997203 0.6412338 0.66021993 0.69553652]

验证 Octree 中 KNN 和 RNN 算法。

1. Octree KNN 算法结果与暴力搜索结果相同，算法准确，如下:
   KNN 搜索结果
   50270 - 0.02
   53203 - 0.03
   52701 - 0.05
   11939 - 0.06
   19883 - 0.07
   49828 - 0.07
   1241 - 0.07
   31823 - 0.07

暴力搜索结果

[50270 53203 52701 11939 19883 49828 1241 31823]

[0.02373505 0.0334789 0.04578997 0.06435989 0.06624196 0.06947317 0.07003216 0.07258631]

2. Octree RNN 和 fast RNN 的运行时间结果（db_size = 64000, r = 0.5，RNN循环100次）如下:

Radius search normal:

Search takes 47087.141ms

Radius search fast:

Search takes 19429.049ms

可以看出 fast 算法在运行时间上具有明显的优势。

Task 3 KNN，RNN 以及 Fast RNN 的区别

1. KNN 和 RNN 的区别

在 KNN 中，由于 worstDist() 会缩小，因此在搜索子节点的时候要优先搜索 query 点坐在的子节点。如果 worstDist() 在搜索该字节点时变小，与其他子节点不再相交，则可以跳过与之不再相交的子节点。在 KNN 中，确认 query 所在子节点的编号，以及优先搜索该子节点的代码如下：

```
# 则根据查询点的坐标和节点坐标，计算所在子节点的编号
morton_code = 0
if query[0] > root.center[0]:
    morton_code = morton_code | 1
if query[1] > root.center[1]:
    morton_code = morton_code | 2
if query[2] > root.center[2]:
    morton_code = morton_code | 4
# 优先查找该查询点所在的子节点
if octree_knn_search(root.children[morton_code], db, result_set, query): # 如果最坏距
    return True # return 表示提前终止， true 表示如果子节点可以包围查询点的邻域球，则父节点
```

在 RNN 中，由于 worstDIst() 不会缩小，因此与之相交的节点都无法逃脱被搜索的命运，只需要按照子节点的编号依次搜索即可。

```
for child in root.children:
if child is None: # 如果该子节点是查询点坐在的空间（已经被优先查找过了），或者该子节点为空
    continue # 则跳过该子节点
if not overlaps(query, result_set.worstDist(), child): # 如果查询点的最坏距离与该子节点
    continue # 也跳过该子节点
if octree_radius_search(child, db, result_set, query):
    return True
```

2. RNN 和 Fast RNN 的区别

两者之间的区别仅在于 Fast RNN 函数中会提前运行函数 contains()，该函数的作用是用来判断一个节点（立方体）是否被一个邻域球（以 worstDist()为半径）完全包裹。如果该函数返回值为

True，则代表该节点被邻域球包裹，此时则无需继续搜索该节点的子节点（因为该节点的子节点无需搜索就可以知道一定在邻域球内），提高了运行效率。 contain 函数的定义如下：

```
# contain 的声明，判断当前query是否包含octant
 def contains(query: np.ndarray, radius: float, octant:Octant):
     query_offset = query - octant.center
     query_offset_abs = np.fabs(query_offset)
     query_offset_to_farthest_corner = query_offset_abs + octant.extent
     return np.linalg.norm(query_offset_to_farthest_corner) < radius
```

contain 函数在 fast RNN 中的调用如下

```
  if contains(query, result_set.worstDist(), root):
      # compare the contents of the octant
      leaf_points = db[root.point_indices, :]
      diff = np.linalg.norm(np.expand_dims(query, 0) - leaf_points, axis=1)
      for i in range(diff.shape[0]):
          result_set.add_point(diff[i], root.point_indices[i])
      # don't need to check any child
      return False
```

Task 4 展示 benchmark 的结果

benchmark 的结果如下所示，增加了自适应的 kdtree。

octree ----------

Octree: build 4390.440, knn 0.584, radius 0.515, brute 8.200

kdtree ----------

Kdtree: build 117.630, knn 2.483, radius 0.288, brute 7.755

adaptive kdtree ----------

Adaptive kdtree: build 112.729, knn 1.078, radius 0.287, brute 7.755

基于以上结果可以得到以下结论:

1. 无论是 kdtree 还是 octree，用于 knn 和 rnn 都比暴力搜索要快很多。
2. Octree 的建树时间比 kd-tree的要长，但是 knn 的搜索时间比较短，更适用与一次建树多次查找的任务。
3. Adaptive kdtree 和 kdtree 的建树时间相似，而 adaptive kdtee 的 knn 搜索时间较短，这是因为 adaptive kdtee的树更加平衡，搜索起来也更好使。

Task 5 Adaptive Kdtree

adaptive kdtree 的思路是计算当前节点内所有点在每个维度上的标准差，然后选择标准差较大的维度作为子节点的划分维度。代码如下:

```
# 建立一个新的函数来计算坐标轴，以当前节点内存储的点的信息作为输入
def axis_adaptive(db, point_indices):
    std_deviation = np.std(db[point_indices, :], 0) # 求该节点内的所有点在三个轴上的标准差
    axis = np.argmax(std_deviation) # 选择标准差较大的那个坐标轴维度
    return axis
```

然后替换掉原来递归函数中的用来计算坐标轴的函数即可，如下

```
def kdtree_recursive_build_adaptive(root, db, point_indices, axis, leaf_size):
    ###################### other code
    if len(point_indices) > leaf_size:
        ############################## other code
        root.left = kdtree_recursive_build(root.left,
                                           db,
                                           point_indices_sorted[0: middle_right_idx], #
                                           axis_adaptive(db, point_indices), # 使用自适应
                                           leaf_size)
        root.right = kdtree_recursive_build(root.right,
                                           db,
                                           point_indices_sorted[middle_right_idx:],  # r
                                           axis_adaptive(db, point_indices), # 使用自适应
                                           leaf_size)
    return root
```

算法效果见 Task 4