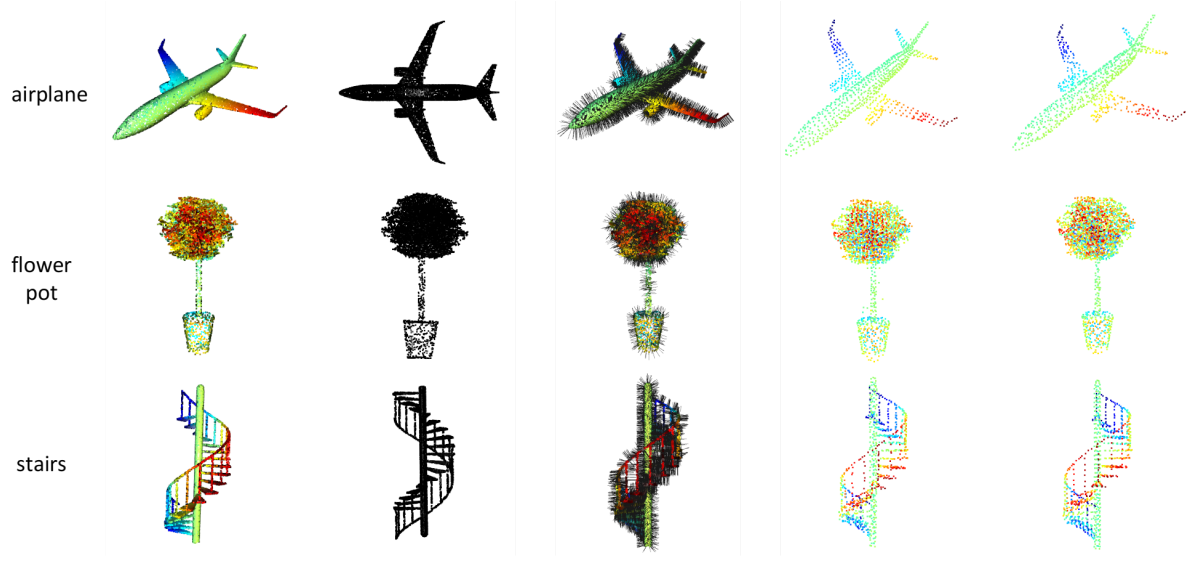
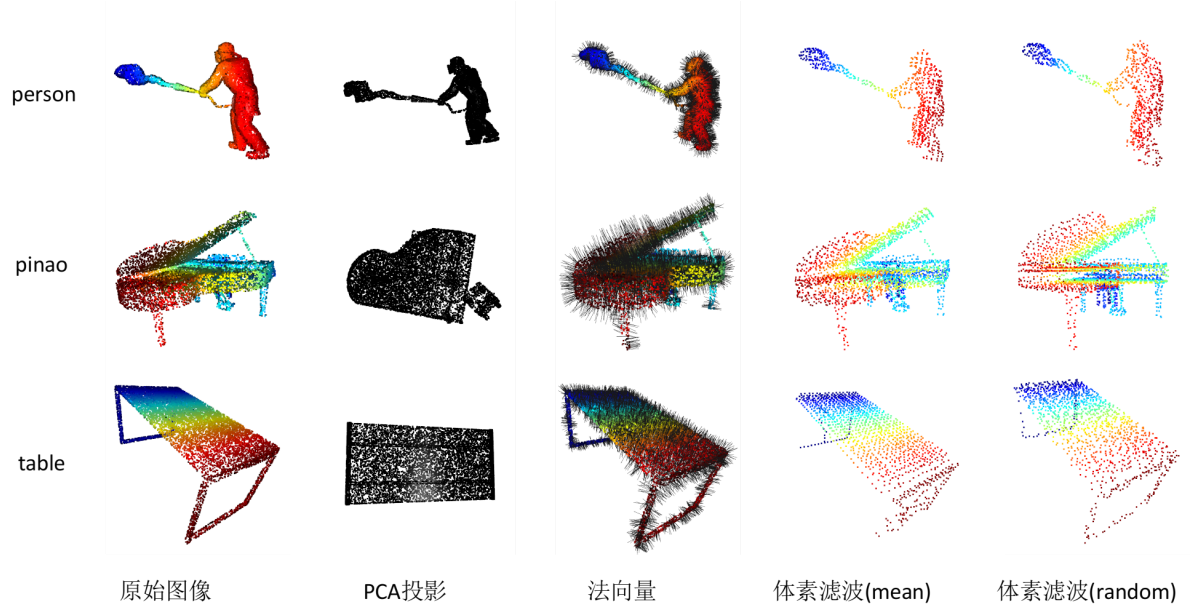
三维点云处理 - 作业 - 第一章

0. 结果展示

在40个类别中挑选了六个比较有代表性的进行展示, 全部40个种类的随机个体展示在附件`results`文件夹中. 从左到右依次是: 原始点云 -- PCA投影 -- 法向量可视化 -- 体素滤波(mean方法) -- 体素滤波(random方法).





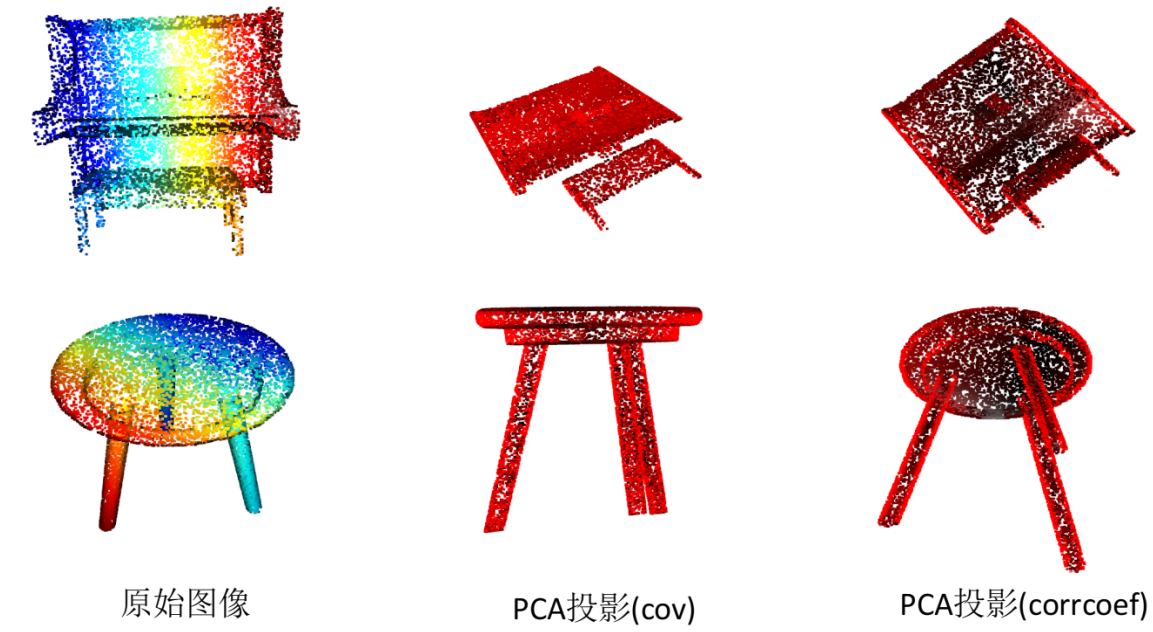
1. PCA

算法核心代码:

```
def PCA(data, correlation=False, sort=True):
    mean = np.mean(np.asarray(data), axis=0)
    # 数据中心化
    X_mean = data - mean
    # 计算协方差矩阵
    if correlation:
        H = np.corrcoef(X_mean.T)
```

```
    else:
        H = np.cov(X_mean.T)
    # SVD分解
    eigenvalues, eigenvectors = np.linalg.eig(H)

    if sort:
        sort = eigenvalues.argsort()[::-1]
        eigenvalues = eigenvalues[sort]
        eigenvectors = eigenvectors[:, sort]
    return eigenvalues, eigenvectors
```

numpy 中计算协方差矩阵可以调用 `corrcoef` 和 `cov` 两个函数. `cov` 即使标准的协方差计算, 而 `corrcoef` 计算的是相关系数, 也可以看作是协方差, 但是剔除了变量量纲的影响. 相关系数R与协方差矩阵C的关系如下: $R_{ij} = \frac{C_{ij}}{\sqrt{C_{ii} * C_{jj}}}$. 因此计算出来的特征值和特征向量也不一样, PCA投影也不相同, 如下图所示:



可以看出这两种方法计算的结果大致一样, 但是又有所不同, 这可能是因为相关系数计算经过了方差的标准化,是一种特殊的协方差.

> **注意:** `cov` 与 `corrcoef` 函数的输入X的每一行代表一个变量, 每一列代表所有这些变量的单一观察. 所以应该传入`3xN`的矩阵. 如果没有做转置, 最终计算的结果将是`NxN`的, 不仅结果不正确, 而且会非常慢.

2. 法向量计算

算法核心代码:

```
def normal_estimation(pcd_tree, points, neighbors_num):
    normals = []
    for p in points:
        # 搜索领域
        [k, idx, _] = pcd_tree.search_knn_vector_3d(p, neighbors_num)
        neighbors = points[idx]
        # PCA得到特征向量
        _, eigVector = PCA(neighbors, sort=True)
        # 最小的特征值对应的特征向量就是法向量方向
        normal = eigVector[:, -1]
        normals.append(normal)
    return normals
```

法向量计算时邻域的选择需要随着物体的特性调整, `search_knn_vector_3d` 选取的是距离目标点距离最近的k个点组成邻居, 对于形状平整, 边缘较少的区域, k值可以设置大一些, 有利于排除噪声的干扰;对于形状复杂, 表面变化频繁的区域, 可以设置小一点的k值, 以得到更加精确的法向量.

3. 体素滤波

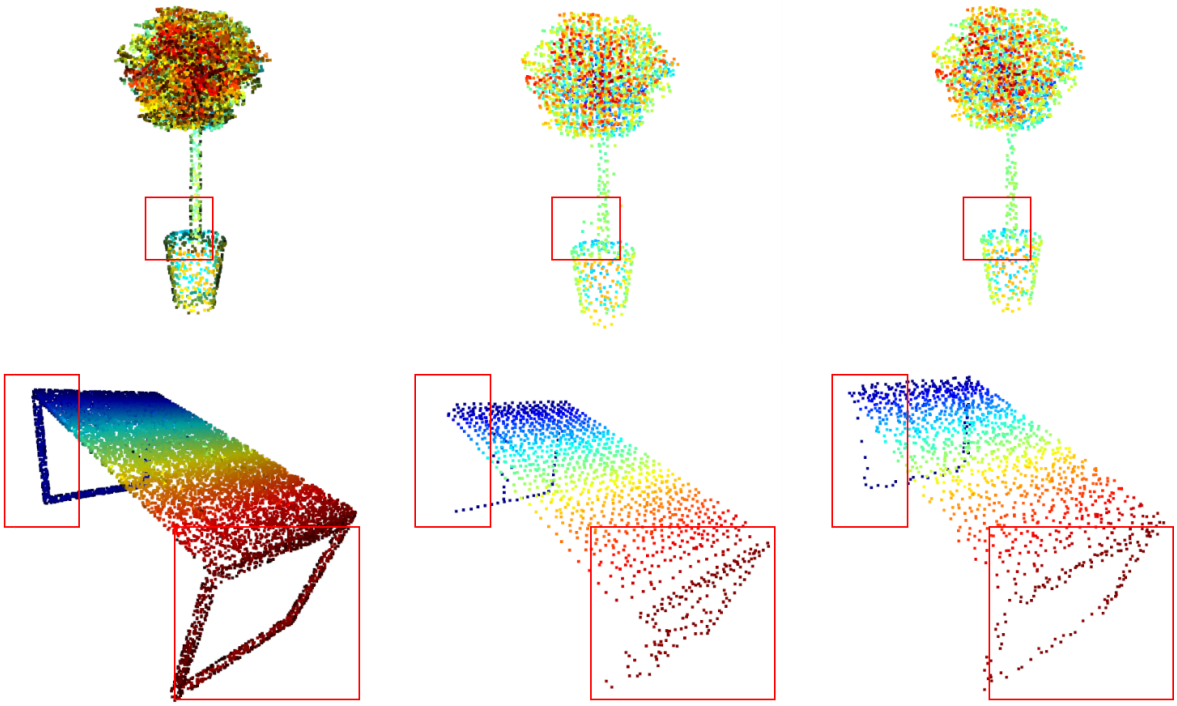
算法核心代码:

```
def voxel_filter(point_cloud, leaf_size, method="mean"):
    filtered_points = []
    points = np.asarray(point_cloud)
    # 计算xyz的范围
    x_min = min(points[:, 0])
    x_max = max(points[:, 0])
    y_min = min(points[:, 1])
    y_max = max(points[:, 1])
    z_min = min(points[:, 2])
    z_max = max(points[:, 2])
    # 确定xyz轴的维度
    D_x = (x_max - x_min) // leaf_size
    D_y = (y_max - y_min) // leaf_size
    D_z = (z_max - z_min) // leaf_size

    # 计算每个点的索引
    ids = []
    for p in points:
        h_x = (p[0] - x_min) // leaf_size
        h_y = (p[1] - y_min) // leaf_size
        h_z = (p[2] - z_min) // leaf_size
        h = h_x + h_y * D_x + h_z * D_x * D_y
        ids.append(h)
    # 对索引进行排序，并记录排序后的索引变化
    sorted_ids = np.sort(ids)
    sort_ids_ind = np.argsort(ids)
    # 遍历排序后的点，进行滤波
    local_points = []               # 存放索引相同的点，用于下采样
    previous_id = sorted_ids[0]     # 存放上一个索引的值
    for i in range(len(sorted_ids)):
        # 如果当前索引等于上一个索引值，则将其加入local_points中
        if sorted_ids[i] == previous_id:
            local_points.append(points[sort_ids_ind[i]])
        # 当前索引值是第一次出现的
```

```
        else:
            # 计算上一个采样点，将其加入最终的输出数组
            if method == "mean":
                new_point = np.mean(local_points, axis=0)
                filtered_points.append(new_point)
            elif method == "random":
                np.random.shuffle(local_points)
                filtered_points.append(local_points[0])
            # 更新 previous_id，清空 local_points 并将当前点加入
            previous_id = sorted_ids[i]
            local_points = [points[sort_ids_ind[i]]]

    filtered_points = np.array(filtered_points, dtype=np.float64)
    return filtered_points
```

实现的体素滤波有两种下采样方法, 一种是将落在同一voxel内的点做平均生成新的点输出, 另一种是在voxel内的点中随机选择一个点输出. 由于计算平均点输出的方法创造了新的点, 因此对数据的修改比较大, 效果没有随机取点好.

下图展示了这两种方法的结果对比, 可以发现mean方法在规则连续的区域效果比较好, 但是在角点, 细长以及边框区域效果很差.



原始图像　　体素滤波(mean)　　体素滤波(random)