Assignment2 Report

王聪

目录

1 绘制泰森多边形

泰森多边形的绘制其实主要以下分成两个步骤：

1. Delaunay 三角形网的构建。
2. 三角形网外接圆圆心心连线。

所以为了为了成功构建出给定离散点的泰森多边形，我们需要先构建出对应离散点的 Delaunay 三角形网。然后按照上面步骤所说，将三角形网外接圆圆心连线，最终绘制泰森多边形。

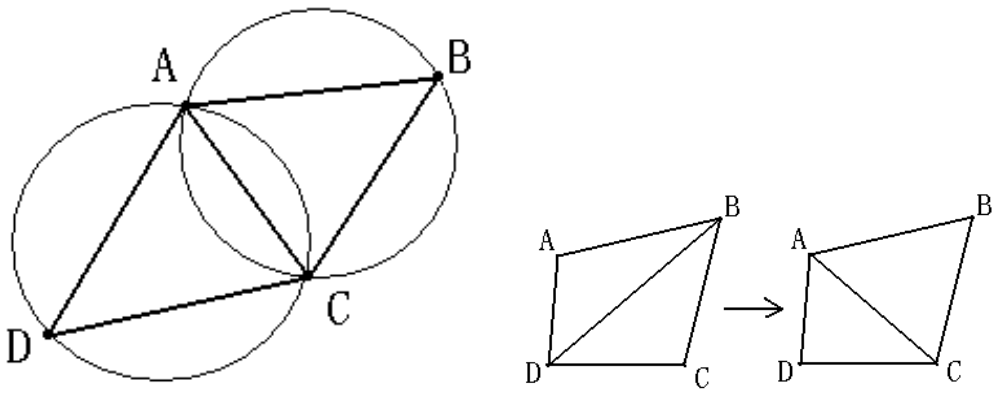
1.1 Delaunay 三角形网的构建

1.1.1 Delaunay 三角形的定义

从百度百科中，我们得到 Dealunay 三角形的定义：Dealunay 三角形是一系列相连的但不重叠的三角形的集合, 而且这些三角形的外接圆不包含这个面域的其他任何点。

要满足 Delaunay 三角剖分的定义，必须符合两个重要的准则：

- 每个德洛内 (Delaunay) 三角形的外接圆不包含面内的其他任何点, 称之为德洛内 (Delaunay) 三角网的空外接圆性质。如图1(a)所示。
- 在散点集可能形成的三角剖分中，Delaunay 三角剖分所形成的三角形的最小角最大。每两个相邻的三角形构成的凸四边形的对角线，在相互交换后，六个内角的最小角不再增大。如图1(b)所示。



(a) Delaunay triangulation Criteria1 (b) Delaunay triangulation Criteria2

图 1: Delaunay triangulation Criteria

1.1.2 构建 Delaunay 三角形的一般步骤

通过查询百度百科中的Delaunay 三角剖分算法，我们了解到该算法的主要步骤如下所示：

1. 遍历所有散点，求出点集的包容盒，得到作为点集凸壳的初始三角形并放入三角形链表。
2. 将点集中的散点依次插入，在三角形链表中找出其外接圆包含插入点的三角形（称为该点的影响三角形），删除影响三角形的公共边，将插入点同影响三角形的全部顶点连接起来，从而完成一个点在 Delaunay 三角形链表中的插入。
3. 根据优化准则对局部新形成的三角形进行优化（如互换对角线等）。将形成的三角形放入 Delaunay 三角形链表。
4. 循环执行上述第 2 步，直到所有散点插入完毕。

看到这里我们仍然不是很清楚到底应该如何实现 Dealunay 三角刨分算法，于是我们查询对应的相关论文得到以下可选算法。

1.1.3 构建 Dealunay 三角形的算法

有关于构建 Dealunay 三角形的算法，有翻边算法、分割合并算法、逐点插入算法和 Bowyer-Watson 算法等。在这几种算法中，我们选择比较简单的分割合并算法，因为该算法相对来算比较简单、易懂。

在论文Triangulate Efficient Triangulation Algorithm Suitable for Terrain Modelling中，对分割合并算法进行了分析并给出了对应的相关伪代码，如下所示：

```
subroutine triangulate
input : vertex list
output : triangle list
   initialize the triangle list
   determine the supertriangle
   add supertriangle vertices to the end of the vertex list
   add the supertriangle to the triangle list
   for each sample point in the vertex list
      initialize the edge buffer
      for each triangle currently in the triangle list
         calculate the triangle circumcircle center and radius
         if the point lies in the triangle circumcircle then
            add the three triangle edges to the edge buffer
            remove the triangle from the triangle list
         endif
      endfor
```

```
      delete all doubly specified edges from the edge buffer
         this leaves the edges of the enclosing polygon only
      add to the triangle list all triangles formed between the point
         and the edges of the enclosing polygon
   endfor
   remove any triangles from the triangle list that use the supertriangle vertices
   remove the supertriangle vertices from the vertex list
end
```

虽然上面的伪代码可以实现三角化，但是实际上效率不是很高。目前大家普遍采用的算法是基于上述伪代码并且在其中进行了一次排序优化，从而使得代码的运行速度得到了提高。该伪代码如下所示：

```
input: 顶点列表(vertices)                                    // vertices为外部生成的随机或乱序顶点列表 output:input: 顶点列表(vertices)
  //vertices为外部生成的随机或乱序顶点列表
output:已确定的三角形列表(triangles)
    初始化顶点列表
    创建索引列表(indices = new Array(vertices.length))      //indices数组中的值为0,1,2,3,.....,vertices.length-1
    基于vertices中的顶点x坐标对indices进行sort              //sort后的indices值顺序为顶点坐标x从小到大排序（也可对y坐标，本例中针对x坐标）
    确定超级三角形
    将超级三角形保存至未确定三角形列表（temp triangles）
    将超级三角形push到triangles列表
    遍历基于indices顺序的vertices中每一个点                 //基于indices后，则顶点则是由x从小到大出现
        初始化边缓存数组（edge buffer）
        遍历temp triangles中的每一个三角形
            计算该三角形的圆心和半径
            如果该点在外接圆的右侧
                则该三角形为Delaunay三角形，保存到triangles
                并在temp里去除掉
                跳过
            如果该点在外接圆外（即也不是外接圆右侧）
                则该三角形为不确定                           //后面会在问题中讨论
                跳过
            如果该点在外接圆内
                则该三角形不为Delaunay三角形
                将三边保存至edge buffer
                在temp中去除掉该三角形
        对edge buffer进行去重
        将edge buffer中的边与当前的点进行组合成若干三角形并保存至temp triangles中
    将triangles与temp triangles进行合并
    除去与超级三角形有关的三角形
end
```

上述伪代码实际上来自 Github 中的delaunay-fast，作者使用 JavaScript 实现了快速构建 Delaunay 三角刨分算法。

1.1.4 构建 Dealunay 三角形的例子

其实我看了上面的代码也并不是很懂到底要如何去做，才能实现 Delaunay 三角刨分算法，从给定的离散点中构建出 Dealunay 三角形。于是我在网上找了相关的案例，如下所示：我们使用 A、B、C 和 D 四点举例来构建 Delaunay 三角形。

首先构建一个超级三角形 PQR，将 A、B、C、D 四点都包含进去。如图2所示

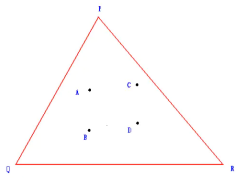


图 2: Step 1: Build A Super Triangle

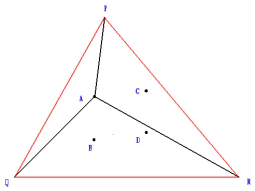


图 3: Step 2: Analyze Point A

接着对 A 点进行分析，因为 A 点在 $\triangle PQR$ 外接圆的内部，所以利用 A 点将三角形 PQR 拆分为三个三角形 $\triangle APQ$、$\triangle APR$ 和 $\triangle ARQ$，如图3所示。

同样，我们对 B 点进行分析，发现 B 点在 $\triangle ABQ$ 的内部，然后我们就将 $\triangle ABQ$ 也分成三个子三角形，如图4所示。

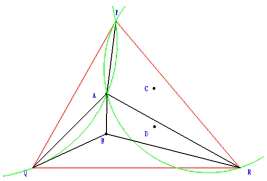


图 4: Step 3: Analyze Point B

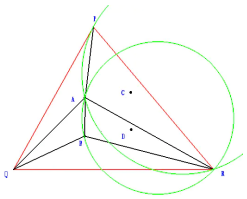


图 5: Step 4: Analyze Point C

在分析完 B 点之后，我们对 C 点进行分析。从图4中我们可以到图中已经有五个三角形，于是我们逐一检查 C 点是否在每个三角形的外接圆内。然后我们发现 C 点在 $\triangle APR$ 和 $\triangle ABR$ 的内部，如图5所示。

出现两个三角形，于是我们将 $\triangle APR$ 和 $\triangle ABR$ 的公共边长 AR 删除，再将 C 点和 $\triangle APR$ 和 $\triangle ABR$ 的其他点组成四边形，如图所示。

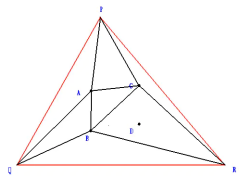


图 6: Step 5: Delete Common Edge AR

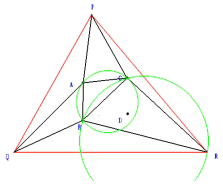


图 7: Step 6: Analyze Point D

最后对 D 点进行分析，这个过程和 C 点很类似，也是先查看点 D 是否在所有的三角形外接圆内。最后我们发现 D 点在 $\triangle ABC$ 和 $\triangle BCR$ 内，所以应该删除 $\triangle ABC$ 和 $\triangle BCR$ 的公共边长 BC，然后让 D 点和 $\triangle ABC$ 和 $\triangle BCR$ 的其他点组成新的三角形，如图7和8所示。

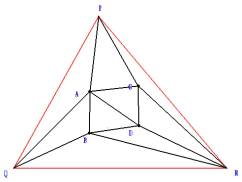


图 8: Step 7: Delete Common Edge BC

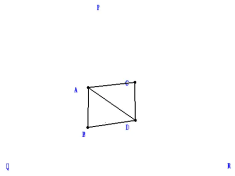


图 9: Step 8: Delete Super Triangle ABC

最后将含有超级 $\triangle ABC$ 的顶点的三角形全部删除，就得到了该四个离散点的三角刨分，如图9所示。

以上就是使用离散点构建 Dealunay 三角形的例子, 下面我们讨论如何使用 Python 实现 Dealunay 三角刨分算法。

1.2 Python 实现 Dealunay 三角刨分算法

1.2.1 代码实现

在本次实现中，我们主要是使用了 Python 中的 matplotlib、numpy、copy、math 库文件。实际上我们代码的核心思想是来源于第1.1.3小节中的伪代码，只是使用 python 实现了而已。由于代码过长，而且有很多自己写的内置的函数，我们这里只是介绍了主要的框架和主要函数的使用，具体的内置函数的作用和实现在附录的代码1中给出的很清楚。

首先我们按照为代码中先创建一个超级三角形，将所有的离散点都可以包住。具体代码如下所示：

```
maxTriangle = np.array([[(xmin+xmax)/2 -(xmax-xmin)*1.5, ymin-(xmax-xmin)*0.5],
                        [(xmin+xmax)/2,ymax+(ymax-ymin)+(xmax-xmin)*0.5], [(xmin+xmax)
    /2+(xmax-xmin)*1.5,ymin-(xmax-xmin)*0.5]])
```

然后使用 numpy 中的 concatenate 函数将超级三角形的三个点放到点集 dots 中，同时还设置了所有边的集合。点集中所有的元素包含两个值代表 x、y 的值，边集中包含四个值分别表示组成边的两个点的序号和值。然后和第1.1.3小节中的伪代码所述的相同，开始遍历所有的点。如果点在一个三角形的外接圆内，就和该三角形的三个点形成三个子三角形；如果点在两个三角形的外接圆内，就去除三角形的公共边，然后和三角形的所有点形成多个三角形。

具体实现如下面的代码所示 (由于代码过长，这里我只是展示了关键函数和步骤)：

```
for i in range(3, self.N + 3):
            tempPoint = self.dots[i, :] # 初始化第一个点
            tempDel = [] # 初始化要删除的
            tempEdgeMat = np.empty(shape=[0, 6], dtype=np.int) # 初始化临时边
            # print("tempEdgeMat.shape: ", tempEdgeMat.shape)
            for j in range(tempTriangleMat.shape[0]):
                # print(tempTriangleMat.shape[0])
                # print(tempTriangleMat[j, 0])
                mask = self._locationPoint(self.dots[tempTriangleMat[j, 0], :], self.dots[
    tempTriangleMat[j, 1], :], self.dots[tempTriangleMat[j, 2], :], tempPoint)
                if mask == 2:
                    # 点在三角形外接圆的外部 说明该三角形就是Delaunay
                    # 将该三角形添加到正式的三角形中 axis=0表示行添加
                    triangleMat = np.concatenate([triangleMat, tempTriangleMat[j, :].reshape
    (1, -1)], axis=0)
                    tempDel.append(j)

                    # 将新的三角形的边添加到edgeMat中
                    edgeMat = np.concatenate([edgeMat, self._makeEdge(tempTriangleMat[j
    , 0], tempTriangleMat[j, 1], tempTriangleMat[j, 2])])
                    # 对edgeMat进行不排序的去重 从而实现去除重复边
                    edgeMat = self._unranked_unique(edgeMat)
                elif mask == 0:
                    # 如果点在三角形外接圆的内部 说明该三角形不是Delaunay三角形
                    # 形成三个顶点和遍历的第i个点的三条线段
                    tempEdge = self._makeTempEdge(tempTriangleMat[j, 0], tempTriangleMat[j
    , 1], tempTriangleMat[j, 2], i)
                    # 将新加入的线段加入到tempEdgeMat中
                    tempEdgeMat = np.append(tempEdgeMat, tempEdge, axis=0)
                    tempDel.append(j)
                else:
                    continue
```

在完成遍历之后，我们去掉含有超级三角形顶点的三角形，最终得到离散点对应的三角刨分。然后将生成的三角形集合和边的集合返回到主函数中，以便生成对应的 Voronoi 图形。

1.2.2 程序运行

在我们定义的 myVoronoi 类中，只要调用函数 plot_figure 就可以生成离散点对应的三角形集，调用的参数就是函数 createDelaulay 生成的三角形集合。如图10所示。

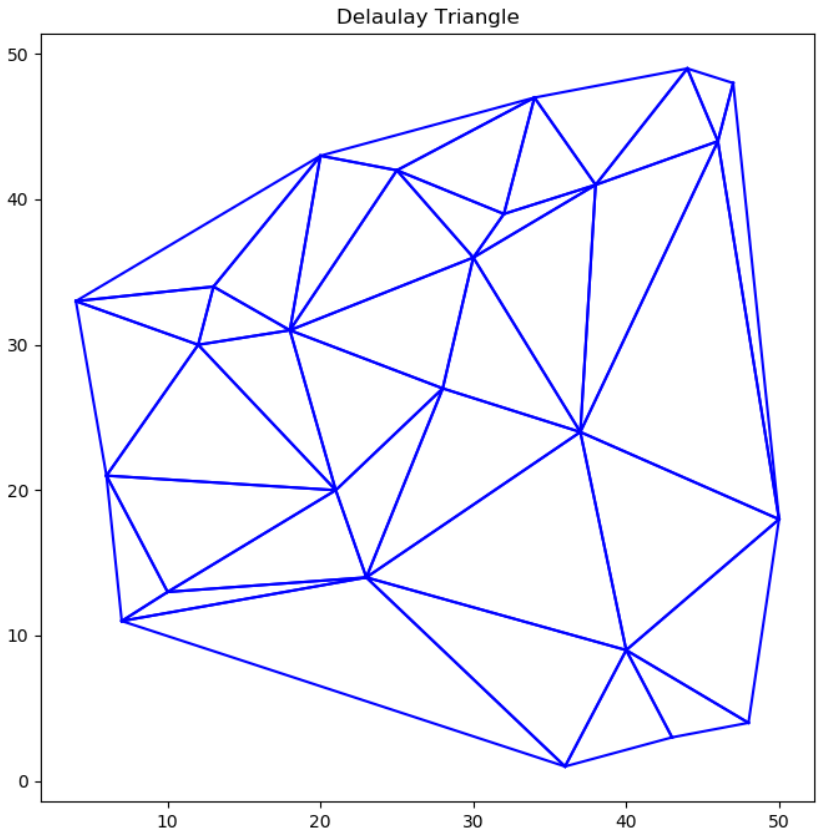


图 10: Delaunay Triangle

1.3 生成 Voronoi 图形

1.3.1 代码实现

在生成了 Delaunay 三角形网络之后，其实生成对应的 Voronoi 图形已经变得比较简单了，我们需要做的工作就是连接生成的 Delaunay 三角形网络中的三角形的外接圆圆心。其实比较麻烦的工作是边缘检测问题，因为总会有些边是要和边缘连接形成区域的。

同样的，我们下面给出生成 Voronoi 图形的主要框架和主要函数的使用，对于内置函数的使用可以查看附录中的代码1。

首先，我们要先得到第1.2小节中生成的三角形网络的外接圆圆心，这里主要是通过内置函数 _getTriangleCenterMat 生成。具体的公式我是参考博客求三点外接圆圆心公式，对应的代码如下所示，其中 a、b 是对应的圆心，r 表示半径。

```
a=((y2-y1)*(y3*y3-y1*y1+x3*x3-x1*x1)-(y3-y1)*(y2*y2-y1*y1+x2*x2-x1*x1))/(2.0*((x3-x1)*(y2-y1)-(x2-x1)*(y3-y1)))
b=((x2-x1)*(x3*x3-x1*x1+y3*y3-y1*y1)-(x3-x1)*(x2*x2-x1*x1+y2*y2-y1*y1))/(2.0*((y3-y1)*(x2-x1)-(y2-y1)*(x3-x1)))
r=math.sqrt((x1-a)*(x1-a)+(y1-b)*(y1-b))
```

然后我们通过函数 _makeBorderTriangle 得到边缘三角形和边缘点，方便我们后面计算那些点需要和边缘组成边缘图形，以进行可视化。

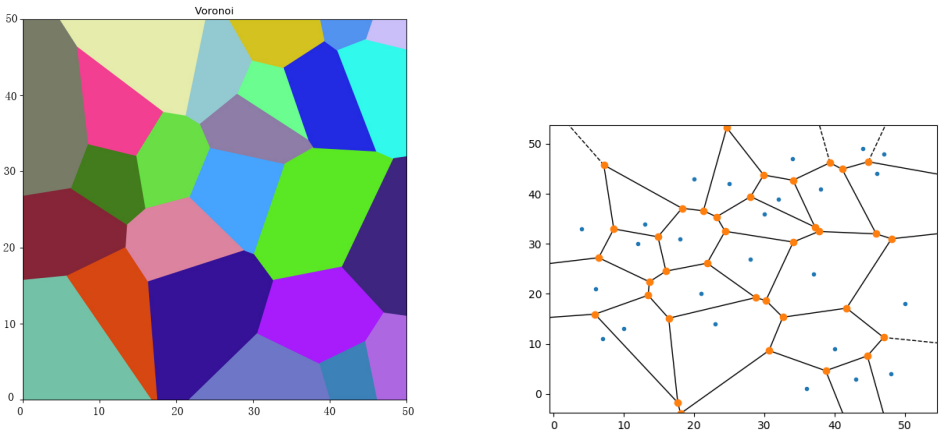
然后我们遍历 Dealunay 三角形集合，做出中心点由离散点形成的三角形，并且判断是否需要进行边缘检测。这个过程我们主要是使用函数 _findPointTriangle 来实现。如果判断不需要进行边缘化处理的话，那我们就可以直接进行可视化，如以下代码所示。

```
# 进行边缘检测
for i in range(3, self.dots.shape[0]):
    tempTriangle, consTempTriangle, mask = self._findPointTriangle(i, triangleMat)
    # print("consTempTriangle:", consTempTriangle, type(consTempTriangle))

    plt.scatter(triangleCenterMat[consTempTriangle, 0], triangleCenterMat[consTempTriangle, 1], color="r", s=20)

    if mask:
        plt.fill(triangleCenterMat[consTempTriangle, 0], triangleCenterMat[consTempTriangle, 1], color=randomcolor())
```

如果判断需要进行边缘检测的话，那就比较麻烦，我们需要得到边缘化的两条线段，在代码中我们将这两条线段定义为 tempBorderDot1 和 tempBorderDot2，并将两点加入我们可视化边中。并且如果这两个点不是相同的话，我们还需要添加一个临界点，这样做的目的就是为了让图形填满整个画图 (我尝试过不做边缘处理结果绘制出来的泰森多边形很难看)，函数 _selectEdge 的作用就是添加边缘点。具体的实现代码可以查看附录中的代码1。

1.3.2 代码运行结果

在我们定义的 myVoronoi 类中，只要调用函数 createVoronoi 就可以生成离散点对应的三角形集，调用的参数就是函数 createDelaulay 生成的三角形集合。如图11(a)所示。



(a) Drawing Voronoi Graph by Own Program (b) Drawing Voronoi Graph by Calling API

图 11: Comparison of Voronoi figures generated by two programs

此外，我们还调用了 scipy.spatial 库文件中的函数 Voronoi 和 voronoi_plot_2d，用来检测我们得出的结果是否是正确的，如图11(b)所示为我们调用官方给出的 API 绘制出来的 Voronoi 图形。绘制出标准的 Voronoi 图的详细代码可以参考附录中的代码

1.4 声明和总结

本次 python 代码只是用了最基础的库文件，如 numpy、math、random、os、matplotlib 和 copy，采用的思想其实上文已经已经写得很清楚了，有些具体的细节，例如在第1.3小节关于边缘检测的处理方式并没有很详细地展开，因为篇幅所限，详细的可以参考代码，里面有比较详细的注释。有关于代码实现，这几篇博客和论文对我帮助很大，在这里具体列出。

1. 绘制 Delaunay 三角网和泰森多边形
2. Delaunay 三角剖分
3. Delaunay 三角剖分的几种算法综述
4. voronoi 图

2 GJK 算法

2.1 GJK 算法原理

GJK 算法的结论是：如果两个多边形相交，那么这两个多边形构成的闵可夫斯基差集 (Minkowski Difference)，必然会包含原点。注意 GJK 算法只能适用于凸包之间，我们这里

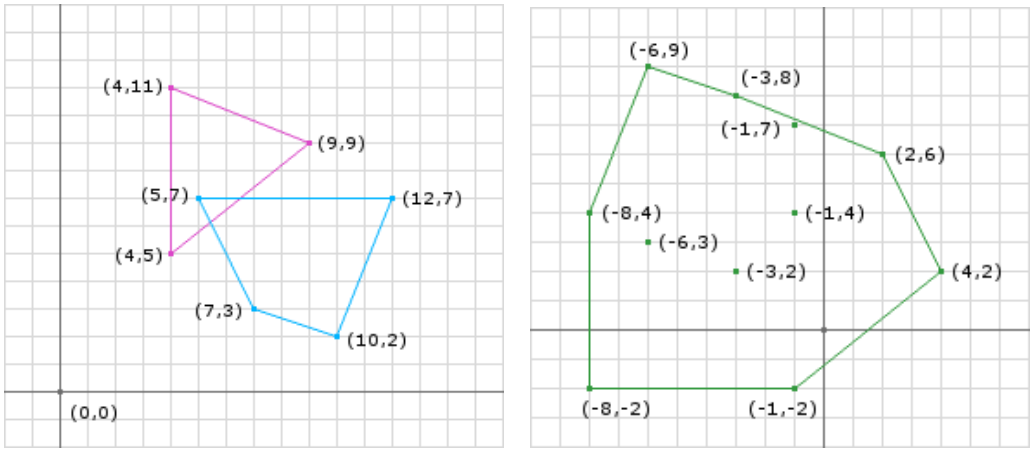
说的多边形其实就是特指凸包，下面我们介绍什么是闵可夫斯基差。

2.2 闵可夫斯基差

如公式 (1) 所示，使用用多边形 A 的所有点，减去多边形 B 中所有的点得到的一个点集合，我们就成这个集合为闵可夫斯基差。

$$A - B = a - b | a \in A, b \in B \tag{1}$$

为了让上面的公式更加形象化，我们给出图片的例子，如图12所示。图12(a)展示的是两个相交的凸包，图12(b)展示的是两个凸包之间的闵可夫斯基差。可以看到两个相交的凸包的闵可夫斯基差一定是经过原点的，计算公式就是公式 (1)。



(a) Two Convex Shapes Intersect (b) Minkovsky Difference

图 12: Minkovsky Difference of Two Convex Hull

我们的代码使用函数 pickTetrahedron 来判断生成的三维单纯形是否包含原点，因为函数 pickTetrahedron 的内容过长，所以我们将内容放到目录中，可以查看目录中的代码3。

2.3 单纯形

计算小节2.2中计算两个凸包的闵可夫斯基差是一件很麻烦的事情，想象一下如果两个凸包的点的数量非常多的话。但是实际上我们并不需要计算出闵可夫斯基差，我们所关心的事情仅仅是两个凸包之间是否发生了碰撞，也就是两个凸包的闵可夫斯基差是否包含原点。所以问题就变成了计算闵可夫斯基差中的单纯形是否包含原点的问题，那这个问题就变得简单多了，只需要我们迭代进行结算就行了。我们先介绍什么是单纯形，从百度百科单纯形中我们知道：1 维单纯形就是线段；2 维单纯形就是三角形；三维单纯形就是四面体。

如图13所示。

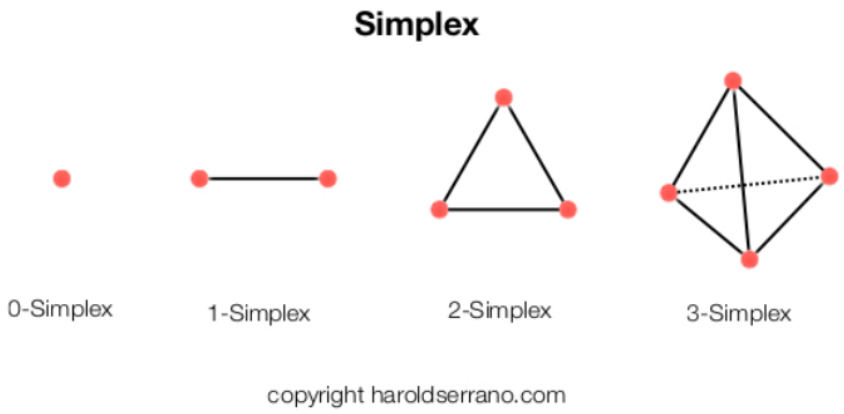


图 13: Simplex

下面给出两个凸包的闵可夫斯基差对应的单纯形在我们代码中是如何构建的，由于代码比较长，我们这里只是给出一定框架，如下所示：

```
function flag = GJK(shape1, shape2, iterations)
% GJK 碰撞算法
% 只适用于两个凸包物体
%
% Input:
%   shape1:
%   必须是三维的，我们的算法是基于三维的
%   判断。因此，输入形状必须是 N* 3 维的。
%
%   shape2:
%   与shape1相同，它必须是由三维几何图形组成的一组点
%
%   iterations:
%   该算法试图构造一个3-simplex的四面体。通过一定的迭代次数，来判断物体是否发生碰撞。
%   比较低的迭代次数，算法的运算时间也会减少。而且随着两个物体之间重叠层度的增加，
%   算法需要的迭代次数也在减少。所以需要我们在迭代次数中做出权衡。
% outputs:
%   flag:
%   true: 物体发生碰撞
%   false: 物体没有发生碰撞

v = [0.8  0.5  1]; % 方向向量
% 先选择两个点
[a,b] = pickLine(v,shape2,shape1);

% 选择第三个点 从而构建三角形(单纯性)
```

```
[a, b, c, flag] = pickTriangle(a ,b, shape2, shape1, iterations);

% 选择第四个点 构建四面体
if (flag == 1)  % 如果找到三角形 就可以尝试开始构建四面体 如果没找到三角形就直接结束
    [a,b,c,d,flag] = pickTetrahedron(a,b,c,shape2,shape1,iterations);
end

end
```

由于我们检测的物体是三维的，所以我们需要构建 3 维单纯形，也就是四面体。从上面算法中，我们先尝试选择两个点，然后构建 2 维单纯形，如果 2 维单纯形存在的话，我们再尝试构建 3 维单纯形。

2.4 Support 函数

从第2.3小节中我们知道只要从两个凸包形成的闵可夫斯基差中构建单纯形，然后判断单纯形是否包含原点就可以了。那么问题来了如何构建单纯形呢，就是利用 support 函数。support 函数的作用就是给定两个凸包，该函数返回这两个凸包的闵可夫斯基差中的一个点。

下面我们直接给出该函数的 MATLAB 代码实现形式，如下所示：

```
function point = support(shape2, shape1, v)
% 得到 Minkowski 差
% 分别得到在给定 v 和 -v方向上最远的两个点
point1 = getFartherPoint(shape1, v);
point2 = getFartherPoint(shape2, -v);
point = point1 - point2;
end

function point = getFartherPoint(shape, v)
% 找到在给定方向上，该几何体投影的点，也就是最远的点
x = shape(:, 1);
y = shape(:, 2);
z = shape(:, 3);
% 这一步操作相当于该几何体上所有的点和原点组成的向量和方向向量v点乘
% 然后计算出点乘之后的最大值 就是该几何体上沿着该方向向量最远的点
dotted = x*v(1) + y*v(2) + z*v(3);
[~, maxIdx] = max(dotted);
% 获得该几何体在该方向向量上最远的点
point = [x(maxIdx), y(maxIdx), z(maxIdx)];
end
```

代码中的思想就是我们从凸包 shape1 中获取一个点，同样的在凸包 shape2 中获取另外一个点，然后求出两个点在给定的相反方向上最远的两个点，最后讲求出的最远的两个

点相减，就得到了两个凸包的闵可夫斯基差中的一个点。关于函数 getFartherPoint 中，我们使用向量点积的方式求出最大值，进而得到在给定方向上，几何体投影最远的点。

2.5 GJK 算法运行结果

在做好上面所说的准备工作之后，我们下面查看 GJK 算法的运行效果，如图14所示。算法判断 A 和 B 之间是有碰撞的，B 和 C 以及 A 和 C 之间是没有碰撞的。对应的代码可以查看附录中的代码3。

```
ans =

    'There is a collision between object A and object B'


ans =

    'There is no collision between object B and object C'


ans =

    'There is no collision between object C and object A'
```

图 14: Running Results of GJK Algorithm

3 包围盒算法

由于包围盒算法比较简单，在做的时候顺便实现了两个算法：基于包围球的碰撞检测算法和基于 AABB 包围盒的碰撞检测算法，下面我们将两个算法都列出来。

3.1 基于包围球的碰撞检测算法

3.1.1 包围球的定义

包围球就是包含该对象的最小球体。

3.1.2 如何创建包围球

包围球的创建主要可以分成以下两个步骤：

1. 计算物体中所有元素的顶点的 x、y、z 坐标，找到三个坐标轴最大值和最小值，然后将最大值和最小值取平均就得到了包围球的球心坐标。
2. 计算球心与三个最大值坐标所确定的点间的距离，就得到了包围球的半径。

具体公式如下所示:

$$o_x = \frac{1}{2}(x_{max} + x_{min}), o_y = \frac{1}{2}(y_{max} + y_{min}), o_z = \frac{1}{2}(z_{max} + z_{min}) \tag{2}$$

$$r = \frac{1}{2}\sqrt{(x_{max} - x_{min})^2 + (y_{max} - y_{min})^2 + (z_{max} - z_{min})^2} \tag{3}$$

在代码中，我们主要是使用函数 getSphereCenter 实现，如下所示。

```
function [centerDot, r] = getSphereCenter(dots)
% 输入三维几何题的坐标 返回该三维几何体球心坐标和球的半径
% 输入参数：
%   dots： 输入凸包的三维坐标
xmax = max(dots(:, 1)); xmin = min(dots(:, 1));
ymax = max(dots(:, 2)); ymin = min(dots(:, 2));
zmax = max(dots(:, 3)); zmin = min(dots(:, 3));

% 求解球心的坐标
% 球心的坐标就是最大点和最小点的中间
x = mean([xmax, xmin]); y = mean([ymax, ymin]); z = mean([zmax, zmin]);
centerDot = [x, y, z];
r = pdist2([xmax, ymax, zmax], [xmin, ymin, zmin]) / 2; % 计算两个点之间的距离除以2

end
```

3.1.3 包围球间的相交检测

在构建了两个凸包的包围球之后，我们做两个包围球间的相交检测。如果两球心距离小于半径之和，那么这两个凸包之间相交的关系。

例如，对于两个包围球 (c_1, r_1) 和 (c_2, r_2)。如果满足公式 (4)，就说明这两个包围球之间是相交的，也就说明了对应的凸体之间是相交的。

$$|c_1 - c_2| \leq r_1 + r_2 \tag{4}$$

在代码中我们的实现如下所示，其实原理都是按照上面解释的来的。

```
function mask = judgeCollision(center1, r1, str1, center2, r2, str2)
% 输入两个三维几何题的坐标 判断两个三维几何题是否会发生碰撞
% 输入参数:
%   center1 center2: 两个包围球的球心坐标
%   r1 r2: 两个包围球的球心半径
%   str1 str2: 说明是那个几何体
% 输出参数:
%   mask:
%   1 表示没有发生碰撞
%   0 表示发生碰撞

% 比较两个圆心之间的距离
% 如果两个圆心之间的距离大于两个圆的半径总和说明没有发生碰撞
if pdist2(center1, center2) > (r1 + r2)
    mask = 1;
else
    mask = 0;
end

% 输出判断
outputDescription(str1, str2, mask);

end
```

3.1.4 包围球的碰撞检测算法的运行结果

我们下面查看包围球的碰撞检测算法的运行效果，如图14所示。算法判断 A 和 B 之间是有碰撞的，B 和 C 以及 A 和 C 之间是没有碰撞的。对应的代码可以查看附录中的代码4。

```
ans =

    'There is a collision between object A and object B'


ans =

    'There is no collision between object B and object C'


ans =

    'There is no collision between object C and object A'
```

图 15: Running Results of Bounding Sphere Collision Detection Algorithm

3.2 基于 AABB 包围盒的碰撞检测算法

3.2.1 AABB 包围盒的定义

包围球就是包含该碰撞体，且边平行于坐标轴的最小六面体。

3.2.2 如何创建 AABB 包围盒

包围球的创建：在物体中所有元素的顶点的 x、y、z 坐标，找到三个坐标轴最大值和最小值，然后这两个点就是描述该六面体的两个坐标，该凸包上面的点满足公式5。

$$x_{min} \leq x \leq x_{max}, y_{min} \leq y \leq y_{max}, z_{min} \leq z \leq z_{max} \tag{5}$$

我们通过函数 getMinMaxDots 得到每个凸包的最大点和最小点，代码如下所示

```
function [mindots, maxdots] = getMinMaxDots(dots)
% 获取传入的多边形的最小顶点和最大顶点
% 输入参数:
%   dots:  输入凸包的三维坐标

xmax = max(dots(:, 1)); xmin = min(dots(:, 1));
ymax = max(dots(:, 2)); ymin = min(dots(:, 2));
zmax = max(dots(:, 3)); zmin = min(dots(:, 3));

mindots = [xmin, ymin, zmin];
maxdots = [xmax, ymax, zmax];
end
```

3.2.3 AABB 包围盒间的相交检测

这里介绍二维场景中的 AABB 碰撞检测，因为三维的只不过是将几何体在 xy、yz、xz 方向上做二维的碰撞检测。对于二维的物体 A 和物体 B 分别沿两个坐标轴做投影，当在两个坐标轴都发生重叠的情况下，就说明物体之间发生了碰撞。

在代码中我们主要通过函数 judgeTwoDim 来实现二维场景中的 AABB 碰撞检测，然后对几何体在坐标轴上进行遍历，实现如下所示:

```
function flag = judgeTwoDim(minDot1, maxDot1, minDot2, maxDot2)
% 对应坐标轴上的多边形是否发生重叠 进而判断是否发生碰撞
% 输入参数:
%   minDot1 minDot2:  表示两个凸包在两个坐标轴上最小的点
%   maxDot1 maxDot2:  表示两个凸包在两个坐标轴上最大的点
% 输出参数:
```

```
%   flag:
%   1 表示没有发生碰撞
%   0 表示发生了碰撞

flag = 1; % 1 表示没有发生碰撞
if (maxDot1(1) >= minDot2(1)) && (maxDot2(1) >= minDot1(1)) &&...
                    (maxDot1(2) >= minDot2(2)) && (maxDot2(2) >= minDot1(2))
    flag = 0; % 如果满足碰撞的条件 也就是对应坐标轴上的投影发生重叠
else
    flag = 1;
end

end
```

3.2.4 AABB 包围盒的碰撞检测算法的运行结果

AABB 包围盒的碰撞检测算法的运行效果如图16所示。算法判断 A 和 B 之间是有碰撞的，B 和 C 以及 A 和 C 之间是没有碰撞的。对应的代码可以查看附录中的代码5。

```
ans =

    'There is a collision between object A and object B'


ans =

    'There is no collision between object B and object C'


ans =

    'There is no collision between object C and object A'
```

图 16: Running Results of AABB Bounding Box Collision Detection Algorithm

4 附录

4.1 Drawing Voronoi Graph by Own Program

说明：由于是完全实现给定离散点生成对应的 Voronoi 图形，所以代码很长，本来想将生成 Delaulay 的代码放到另外一个文件，但是时间有限，该工作就没做。下面给出主要的函数说明，辅助函数的说明几乎在注释中都写得很清楚：

createDelaulay: 生成离散点对应的 Delaulay 三角形网，函数将会返回三角形和边的。

plot_figure: 绘制离散点对应的 Delaulay 三角形网，函数没有返回值。

createVoronoi: 通过离散点对应的 Delaulay 三角形网，连接这些三角形外接圆的圆心。同时为了图像的美观，我们还做了边缘处理，让图像能够铺满画布。

Listing 1: **Giving Discrete Points, Generating the Corresponding Vonoroi Graph**

```
import numpy as np
import math
import random
import os
import matplotlib.pyplot as plt
import copy

def getColor():
    color:  int
    color1 = ri(16, 255)
    color2 = ri(16, 255)
    color3 = ri(16, 255)
    color1 =  hex(color1)
    color2 =  hex(color2)
    color3 =  hex(color3)
    ans = "#" + color1[2:] + color2[2:] + color3[2:]
    return ans

def randomcolor():
    colorArr = ['1','2','3','4','5','6','7','8','9','A','B','C','D','E','F']
    color = ""
    for i in  range(6):
        color += colorArr[random.randint(0,14)]
    return "#"+color

class myVoronoi:

    def __init__(self, x=None, y=None):
```

```
29          # np.random.seed(0)
30          # self.N = 4
31          # self.dots = np.random.randn(self.N, 2)
32
33          self.lower = 0
34          self.upper = 50
35          self.dots = np.concatenate([np.array(x).reshape(-1, 1), np.array(y).reshape(-1, 1)],
        axis=1)
36          self.N = self.dots.shape[0]
37
38          self.based_path = os.path.abspath(os.path.dirname(__file__)) # 获取代码运行的基本路
        径
39
40          # 如果存储图片的文件夹不存在的话，就创建该文件夹
41          if not os.path.exists(os.path.join(self.based_path, "./figures")):
42              os.makedirs(os.path.join(self.based_path, "./figures"))
43
44          triangleMat, edgeMat = self.createDelaulay()
45
46          self.plot_figure(triangleMat)
47
48          self.createVoronoi(triangleMat)
49
50      def createVoronoi(self, triangleMat):
51
52          triangleCenterMat = self._getTriangleCenterMat(triangleMat)
53
54          borderTriangleMat, borderPoint, triangleTempMat = self._makeBorderTriangle(
        triangleMat)
55          # print("borderPoint", borderPoint)
56          # print("*"*30)
57
58          # 绘制散点图
59          plt.figure(num = 2, figsize=(8, 8), facecolor='w')
60
61          # 进行边缘检测
62          for i in  range(3, self.dots.shape[0]):
63              tempTriangle, consTempTriangle, mask = self._findPointTriangle(i, triangleMat)
64              # print("consTempTriangle:", consTempTriangle, type(consTempTriangle))
65
66
67              plt.scatter(triangleCenterMat[consTempTriangle, 0], triangleCenterMat[
```

```
    consTempTriangle, 1], color="r", s=20)
68
69              if mask:
70                  plt.fill(triangleCenterMat[consTempTriangle, 0], triangleCenterMat[
    consTempTriangle, 1], color=randomcolor())
71              # 需要做边缘化处理
72              else:
73                  tempTriangle, consTempTriangle, delLeft, delRight = self._selectTempTriangle
    (tempTriangle, consTempTriangle, triangleCenterMat)
74                  # print(delLeft)
75                  flag = True
76                  # 获取边缘线段
77                  tempBorderEdge = triangleCenterMat[consTempTriangle, :]
78                  # 边缘延长线1
79                  if delLeft == 0:
80                      tempEdge = tempTriangle[0, [0, 1]]
81                      tempBorderDot1 = self._edgePointFind(tempEdge, triangleCenterMat[
    consTempTriangle[0], :], borderPoint)
82                      if tempBorderDot1.shape[0] == 0: flag = False # 由于边缘点超出就不再画出
83                  else:
84                      tempBorderDot1 = self._edgePointFindrd(triangleCenterMat[
    consTempTriangle[0], :], triangleCenterMat[delLeft, :])
85                  tempBorderEdge = np.append(tempBorderDot1.reshape(1, -1), tempBorderEdge,
    axis=0) # 按照行连接
86
87                  # 边缘延长线2
88                  if delRight == 0:
89                      tempEdge = tempTriangle[-1, [0, 2]]
90                      tempBorderDot2 = self._edgePointFind(tempEdge, triangleCenterMat[
    consTempTriangle[-1], :], borderPoint)
91                      if tempBorderDot2.shape[0] == 0: flag = False # 边缘点超出不在画出
92                  else:
93                      tempBorderDot2 = self._edgePointFindrd(triangleCenterMat[
    consTempTriangle[-1], :], triangleCenterMat[delRight, :])
94                  tempBorderEdge = np.append(tempBorderEdge, tempBorderDot2.reshape(1, -1),
    axis=0) # 按照行连接
95
96                  # 绘制边缘图形
97                  if flag:
98                      if tempBorderDot1[0] != tempBorderDot2[0] and tempBorderDot1[1] !=
    tempBorderDot2[1]:
99                          # 求交点三
```

```
100                         tempBorderEdge = np.append(tempBorderEdge, self._makeTempBorder(
    tempBorderDot1, tempBorderDot2).reshape(1, -1), axis=0)
101
102                     # 绘制图形
103                     plt.fill(tempBorderEdge[:, 0], tempBorderEdge[:, 1], color=randomcolor()
    )
104
105
106         plt.xlim([0, self.upper])
107         plt.ylim([0, self.upper])
108         plt.title("Voronoi")
109         plt.show()
110
111
112     def _makeTempBorder(self, tempBorderPoint1, tempBorderPoint2):
113         tempBorderPoint3 = np.zeros(shape=[2])
114
115         # 检测边缘点
116         for i in  range( len(tempBorderPoint1)):
117             if tempBorderPoint1[i] == 0 or tempBorderPoint1[i] == self.upper:
118                 tempBorderPoint3[i] =  tempBorderPoint1[i]
119
120             # 对第二个点也是相同的道理
121             if tempBorderPoint2[i] == 0 or tempBorderPoint2[i] == self.upper:
122                 tempBorderPoint3[i] =  tempBorderPoint2[i]
123
124         return tempBorderPoint3
125
126
127
128     def _edgePointFindrd(self, xy1, xy2):
129         xy3, xy4 = self._selectEdge(xy1, xy2)
130         tempBorderDot1 = self._cross_point(xy1, xy2, xy3, xy4)
131
132         return tempBorderDot1
133
134
135     def _edgePointFind(self, tempEdge, xy, borderPoint):
136         # 做出一点对边缘的中垂线，得到边缘点坐标
137
138         x = xy[0]
139         y = xy[1]
```

```
140         # 判断中心点是否在大图形中
141         dpoint = np.array([borderPoint[0, 0]])
142         dpoint = np.hstack((dpoint, borderPoint[:, 1]))
143 
144         # dpoint.append(list(borderPoint[:, 1]))
145         mask = self._pointInPoly(dpoint, xy)
146 
147         # 求边的中心点
148         xz = (self.dots[tempEdge[0], 0] + self.dots[tempEdge[1], 0]) / 2
149         yz = (self.dots[tempEdge[0], 1] + self.dots[tempEdge[1], 1]) / 2
150 
151         flag = 1
152         if mask:
153             # 如果点在内部 做中心到边缘延长线即可
154             xy1 = copy.deepcopy(xy)
155             xy1 += [ i * 2 / math.sqrt((xz - x) ** 2 + (yz - y) ** 2) for i in [xz - x, yz -
     y] ]
156         else:
157             # 点在外部 但是没有超过边界
158             if not ((x < 0) or (x > self.upper) or (y < 0) or (y > self.upper)):
159                 xy1 = copy.deepcopy(xy)
160                 xy1 -= [ i * 2 / math.sqrt((xz - x) ** 2 + (yz - y) ** 2) for i in [xz - x,
    yz - y] ]
161             else:
162                 # 在外界
163                 flag = False
164                 tempBorderdot = np.empty(shape=(1, 1), dtype=np. int)
165 
166         if flag:
167             # 判断4个边是哪个
168             xy2, xy3 = self._selectEdge(xy, xy1)
169             # 求两点求交运算
170             tempBorderdot = self._cross_point(xy, xy1, xy2, xy3)
171 
172         return tempBorderdot
173 
174 
175     def _cross_point(self, xy, xy1, xy2, xy3):
176         # 计算两点之间的交点
177         x1, y1 = xy[0], xy[1]
178         x2, y2 = xy1[0], xy1[1]
179         x3, y3 = xy2[0], xy2[1]
```

```
180         x4, y4 = xy3[0], xy3[1]
181
182         k1=(y2-y1)*1.0/(x2-x1)#计算k1,由于点均为整数，需要进行浮点数转化
183         b1=y1*1.0-x1*k1*1.0#整型转浮点型是关键
184         if (x4-x3)==0:#L2直线斜率不存在操作
185             k2=None
186             b2=0
187         else:
188             k2=(y4-y3)*1.0/(x4-x3)#斜率存在操作
189             b2=y3*1.0-x3*k2*1.0
190         if k2==None:
191             x=x3
192         else:
193             x=(b2-b1)*1.0/(k1-k2)
194         y=k1*x*1.0+b1*1.0
195         return np.array([x,y])
196
197
198     def _selectEdge(self, xy, xy1):
199         # 判断射线会和那条边相交
200         deg1 = np.angle( complex(0-xy[0], 0-xy[1]))
201         deg2 = np.angle( complex(self.upper-xy[0], 0-xy[1]))
202         deg3 = np.angle( complex(self.upper-xy[0], self.upper-xy[1]))
203         deg4 = np.angle( complex(0-xy[0], self.upper-xy[1]))
204         deg0 = np.angle( complex(xy1[0]-xy[0], xy1[1]-xy[1]))
205
206         # 使用np.hstach将一维数组连接起来
207         idx = np.argsort(np.hstack([deg0, deg1, deg2, deg3, deg4]))
208         k = np.where(idx == 0)[0]
209
210
211
212         if k == 0:
213             xy2 = np.array([0, 0])
214             xy3 = np.array([0, self.upper])
215         elif k == 1:
216             xy2 = np.array([0, 0])
217             xy3 = np.array([self.upper, 0])
218         elif k == 2:
219             xy2 = np.array([self.upper, 0])
220             xy3 = np.array([self.upper, self.upper])
221         elif k == 3:
```

```
222             xy2 = np.array([0, self.upper])
223             xy3 = np.array([self.upper, self.upper])
224         else:
225             xy2 = np.array([0, 0])
226             xy3 = np.array([0, self.upper])
227 
228 
229         return xy2, xy3
230 
231 
232     def _pointInPoly(self, dpoint, xy):
233         Ndot = self.dots[dpoint, :]
234         PN = Ndot - xy
235         TN = np.zeros(shape=( len(Ndot), 1))
236         for i in  range( len(Ndot) - 1):
237             TN[i] = self._crossDot(PN[i, :], PN[i+1, :])
238 
239         if np. abs(np. sum(np.sign(TN))) ==  len(Ndot) - 1:
240             mask = 1
241         else:
242             mask = 0
243         return mask
244 
245     def _crossDot(self, point1, point2):
246         # 两个向量之间做差积
247         x1, y1 = point1[0], point1[1]
248         x2, y2 = point2[0], point2[1]
249 
250         return (x1 * y2 - y1 * x2)
251 
252     def _selectTempTriangle(self, tempTriangle, consTempTriangle, triangleCenterMat):
253         """
254         删除所有中心点超出边界的三角形
255         """
256 
257         delLeft = 0
258         delRight = 0
259         for i in  range(tempTriangle.shape[0]):
260             centerDots = triangleCenterMat[consTempTriangle[i], :]
261             if centerDots[0] > 0 and centerDots[0] < self.upper and centerDots[1] < self.
    upper and centerDots[1] > 0:
262                 break
```

```
263
264         if i != 0:
265             delLeft = consTempTriangle[i-1]
266             tempTriangle = np.delete(tempTriangle,  list( range(i)), axis=0)
267             consTempTriangle = np.delete(consTempTriangle,  list( range(i)))
268             # del consTempTriangle[list(range(i-1))]  # 删除对应的下标
269 
270         # 将矩阵倒着来一遍
271         tempTriangle = np.flipud(tempTriangle)
272         consTempTriangle = np.flipud(consTempTriangle)
273 
274         for j in  range(tempTriangle.shape[0]):
275             centerDots = triangleCenterMat[consTempTriangle[j], :]
276             if centerDots[0] > 0 and centerDots[0] < self.upper and centerDots[1] < self.
    upper and centerDots[1] > 0:
277                 break
278 
279         if j != 0:
280             delRight = consTempTriangle[j-1]
281             tempTriangle = np.delete(tempTriangle,  list( range(j)), axis=0)
282             consTempTriangle = np.delete(consTempTriangle,  list( range(j)))
283 
284         # 将三角形数组转置回来
285         tempTriangle = np.flipud(tempTriangle)
286         consTempTriangle = np.flipud(consTempTriangle)
287 
288         return tempTriangle, consTempTriangle, delLeft, delRight
289 
290 
291     def _findPointTriangle(self, k, triangleMat):
292         consTempTriangle = np.arange(triangleMat.shape[0])
293         mask = True
294         idx = self._findANumber(triangleMat, k)
295         tempTriangleMat = triangleMat[idx, :]
296         consTempTriangle = np.array(idx)
297 
298         # 将k放到每行第一个
299         for m in  range(tempTriangleMat.shape[0]):
300             if tempTriangleMat[m][0] != k:
301                 tempIndex =  list(np.where(tempTriangleMat[m] == k)[0])[0]
302                 tempTriangleMat[m, [0, tempIndex]] = tempTriangleMat[m, [tempIndex, 0]]
303         # 如果有一个点只出现过一次，把这个点包含的三角形放到第一行
```

```
304         nums = self._findNumsAppearOnce(tempTriangleMat.reshape(-1))
305 
306         if  len(nums) != 0:
307             num = np.sort(nums)[0]
308             idx = self._findANumber(tempTriangleMat, num)[0] # 只能找到一个数字 所以直接取值
309             tempTriangleMat[[0, idx], :] = tempTriangleMat[[idx, 0], :]
310             consTempTriangle[[0, idx]] = consTempTriangle[[idx, 0]]
311             mask = False
312             if tempTriangleMat[0, 2] == num:
313                 tempTriangleMat[0, [1, 2]] = tempTriangleMat[0, [2, 1]]
314 
315         # 然后将首尾排序
316         for i in  range(1, tempTriangleMat.shape[0]):
317             idx = self._findANumber(tempTriangleMat, tempTriangleMat[i-1, 2])
318             idx.remove(idx[0])
319             idx = idx[0]
320             tempTriangleMat[[idx, i], :] = tempTriangleMat[[i, idx], :]
321             if tempTriangleMat[i-1, 2] != tempTriangleMat[i, 1]:
322                 tempTriangleMat[i, [1, 2]] = tempTriangleMat[i, [2, 1]]
323             consTempTriangle[[i, idx]] = consTempTriangle[[idx, i]]
324 
325         return tempTriangleMat, consTempTriangle, mask
326 
327     def _findNumsAppearOnce(self, array):
328         # write code here
329         dic = {}
330         for i in array:
331             if i in dic:
332                 dic[i] += 1
333             else:
334                 dic[i] = 1
335         res = []
336         for i in dic:
337             if dic[i] == 1:
338                 res.append(i)
339         return res
340 
341     def _findANumber(self, arr, k):
342         """
343         返回k在arr中的下标
344         输入参数:
345             arr: 数组 注意是二维数组！！
```

```
346         输出参数:
347             idx 下标数组
348         """
349         idx = []
350         # print(arr.shape[1])
351         # print(arr)
352         # print(arr[0])
353         for i in  range(arr.shape[0]):
354             if k in  list(arr[i]):
355                 idx.append(i)
356         return idx
357 
358     def _getTriangleCenterMat(self, triangleMat):
359         """
360         获取三角形的外接圆的圆心位置
361         输入参数:
362             triangleMat: 三角形
363         输出参数:
364             triangleCenterMat: 每个三角形的外接圆的圆心位置
365         """
366         triangleCenterMat = np.empty(shape=[0, 2], dtype=np. int) # 三角形外接圆的圆心位置
367         for i in  range(triangleMat.shape[0]):
368             # 调用函数_getCircle生成三角形的外接圆 返回圆心的位置和半径
369             a, b, _ = self._getCircle(self.dots[triangleMat[i, 0], :], self.dots[triangleMat
    [i, 1], :],
370                                                 self.dots[triangleMat[i, 2], :])
371             triangleCenterMat = np.append(triangleCenterMat, np.array([a, b]).reshape(1, -1)
    , axis=0)
372 
373         return triangleCenterMat
374 
375     def _makeBorderTriangle(self, triangleMat):
376 
377         borderTriangleMat = np.empty(shape=[0, triangleMat.shape[1]], dtype=np. int)
378         borderPoint = np.empty(shape=[0, 2], dtype=np. int)
379         triangleTempMat = np.zeros(triangleMat.shape)
380 
381         for i in  range(triangleMat.shape[0]):
382             tempTriangle = triangleMat[i, :] # 获取临时三角形
383 
384             # 判断三角形的12 23 13点
385             for j in  range(triangleMat.shape[1]):
```

```
386                 borderPoint, triangleTempMat = self._generateBorderPoint(triangleMat,
    tempTriangle,
387                                                                          borderPoint,
    triangleTempMat, i, j)
388
389             if ~np. all(triangleTempMat[i, :]):
390                 # 如果边缘三角形少于三个就添加
391                 borderTriangleMat = np.append(borderTriangleMat, tempTriangle.reshape(1, -1)
    , axis=0)
392
393         # 对得到的borderPoint进行排序输出
394         for i in  range(borderPoint.shape[0]-1):
395             idx = self._returnIndex(borderPoint, i)
396             borderPoint[[i+1, idx], :] = borderPoint[[idx, i+1], :] # 交换顺序
397             if borderPoint[i, 1] == borderPoint[i+1, 1]:
398                 borderPoint[i+1, [0, 1]] = borderPoint[i+1, [1, 0]] # 交换下一点的顺序
399         return borderTriangleMat, borderPoint, triangleTempMat
400
401     def _returnIndex(self, borderPoint, k):
402         idx = []
403         for i in  range(borderPoint.shape[0]):
404             if k != i and borderPoint[k, 1] in borderPoint[i]:
405                 idx.append(i)
406         if  len(idx) == 0:
407             return idx
408         else:
409             return idx[0]
410         # return idx
411
412     def _generateBorderPoint(self, triangleMat, triangle, borderPoint, triangleTempMat, m, k
    ):
413
414         """
415         找到三角形中的找到临时边，也就是不和其他三角形连接的边
416         输入参数:
417             triangleMat: 总三角形的集合
418             triange: 需要判断的三角形，也就是我们遍历的三角形
419             borderPoint: 边界点
420             triangleTempMat: 临时三角形的集合
421             m: 遍历中的第几个三角形
422             k: 需要判断的三角形的第k个点
423         输出参数:
```

```
424             borderPoint: 临界边上的点
425             triangleTempMat
426             调试使用
427         """
428         idx = []
429         for i in  range(triangleMat.shape[0]):
430             if i != m and triangle[k] in triangleMat[i] and triangle[(k+1) % triangle.shape
    [0]] in triangleMat[i]:
431                 idx.append(i)
432         if  len(idx) == 0:
433             # 如果这条边没有和别的三角形相连接 就说明这条边是临时边
434             borderPoint = np.append(borderPoint, np.array([triangle[k], triangle[(k + 1) %
    triangle.shape[0]]]).reshape(1, -1), axis=0)
435         elif  len(idx) == 1:
436             triangleTempMat[m, k] = idx[0]
437 
438         return borderPoint, triangleTempMat
439 
440     def createDelaulay(self):
441         # 创建Delaulay三角形
442 
443         # 对二位数组进行自定义排序 按照行排序一列、二列依次排序
444         # 这里千万不能使用np.sort 后悔一生
445         self.dots = np.array( sorted(self.dots.tolist(), key=lambda x: (x[0], x[1])))
446         # 找出最大包含的三角形
447         # axis-0 表示按照列返回最大值或者最小值
448         xmin, ymin = np. min(self.dots, axis=0)
449         xmax, ymax = np. max(self.dots, axis=0)
450 
451         # print(xmin, xmax)
452 
453         # 创建最大的三角形，这个三角形正好能包住所有点
454         maxTriangle = np.array([[(xmin+xmax)/2 -(xmax-xmin)*1.5, ymin-(xmax-xmin)*0.5],
455                                 [(xmin+xmax)/2,ymax+(ymax-ymin)+(xmax-xmin)*0.5],  [(xmin+xmax)
    /2+(xmax-xmin)*1.5,ymin-(xmax-xmin)*0.5]])
456         # print(maxTriangle)
457         self.dots = np.concatenate([maxTriangle, self.dots], axis=0) # axis=0表示按照行进行
    拼接
458         # print(self.dots)
459 
460         # 点的集合获得最大三角形的三个点
461         edgeMat = np.array([self.__convertData(0, 1, self.dots[0, :], self.dots[1, :]),
```

```
462                         self.__convertData(1, 2, self.dots[1, :], self.dots[2, :]),
463                         self.__convertData(0, 2, self.dots[0, :], self.dots[2, :])])
464         # print(edgeMat)
465         triangleMat = np.array([0, 1, 2]).reshape(1, -1) # 三角集合 包含三个点
466         tempTriangleMat = np.array([0, 1, 2]).reshape(1, -1) # 临时三角形
467 
468         # 开始遍历 因为self.dots前三个点是最大三角形的三个点
469         for i in  range(3, self.N + 3):
470             tempPoint = self.dots[i, :] # 初始化第一个点
471             tempDel = [] # 初始化要删除的
472             tempEdgeMat = np.empty(shape=[0, 6], dtype=np. int) # 初始化临时边
473             # print("tempEdgeMat.shape: ", tempEdgeMat.shape)
474             for j in  range(tempTriangleMat.shape[0]):
475                 # print(tempTriangleMat.shape[0])
476                 # print(tempTriangleMat[j, 0])
477                 mask = self._locationPoint(self.dots[tempTriangleMat[j, 0], :], self.dots[
    tempTriangleMat[j, 1], :],
478                                            self.dots[tempTriangleMat[j, 2], :],
    tempPoint)
479                 # print("mask: ", mask)
480                 if mask == 2:
481                     # 点在三角形外接圆的外部 说明该三角形就是Delaunay
482 
483                     # 将该三角形添加到正式的三角形中 axis=0表示行添加
484                     # print("tempTriangleMat[j, :]", tempTriangleMat[j, :])
485                     # print("triangleMat", triangleMat)
486                     triangleMat = np.concatenate([triangleMat, tempTriangleMat[j, :].reshape
    (1, -1)], axis=0)
487                     # tempDel = np.concatenate([tempDel, j], axis=0)
488                     tempDel.append(j)
489 
490                     # 将新的三角形的边添加到edgeMat中
491                     edgeMat = np.concatenate([edgeMat, self._makeEdge(tempTriangleMat[j, 0],
     tempTriangleMat[j, 1], tempTriangleMat[j, 2])])
492                     # print("edgeMat:\n", edgeMat)
493                     # 对edgeMat进行不排序的去重 从而实现去除重复边
494                     edgeMat = self._unranked_unique(edgeMat)
495                 elif mask == 0:
496                     # 如果点在三角形外接圆的内部 说明该三角形不是Delaunay三角形
497 
498                     # 形成三个顶点和遍历的第i个点的三条线段
499                     tempEdge = self._makeTempEdge(tempTriangleMat[j, 0], tempTriangleMat[j,
```

```
    1], tempTriangleMat[j, 2], i)
500                 # print("tempEdge: ", tempEdge)
501                 # 将新加入的线段加入到tempEdgeMat中
502                 tempEdgeMat = np.append(tempEdgeMat, tempEdge, axis=0)
503                 # print("tempEdgeMat:", tempEdgeMat)
504                 # print("***"*30)
505                 # tempDel = np.concatenate([tempDel, j], axis=0)
506                 tempDel.append(j)
507             else:
508                 continue
509
510         # 移除加入的临时三角形
511         tempTriangleMat = np.delete(tempTriangleMat, tempDel, axis=0) # 删除要去掉的行
512
513         # 检查每一个行中是否为全0元素
514         flags = np. any(tempTriangleMat, axis=1).tolist()
515         idxArr = [i for i, x in  enumerate(flags) if x == False]
516         tempTriangleMat = np.delete(tempTriangleMat, idxArr, axis=0)
517
518         tempTriangleMat = self._detectArray(tempTriangleMat, dim=3)
519         # print("tempTriangleMat:", tempTriangleMat)
520         # 对tempEdgeMat进行去重并且保持原先的顺序不变
521         tempEdgeMat = self._unranked_unique(np.array(tempEdgeMat))
522         # print("tempEdgeMat", tempEdgeMat)
523         # print("tempEdgeMat.shape", tempEdgeMat.shape)
524         # print("***"*30)
525         tempTriangleMat = np.append(tempTriangleMat, self._makeTempTriangle(np.array(
    tempEdgeMat), i), axis=0)
526         # print("tempTriangleMat", tempTriangleMat)
527
528     # 遍历完成之后 合并三角形的集合
529     triangleMat = np.concatenate([triangleMat, tempTriangleMat], axis=0)
530     edgeMat = np.concatenate([edgeMat, np.array(tempEdgeMat)], axis=0) # 合并边的集合
531     # print("triangleMat", triangleMat)
532     # print("edgeMat", edgeMat.shape)
533     # 删除一开始初始化的最大的三角形maxtriangle
534     tempDel = []
535     for k in  range(triangleMat.shape[0]):
536         if 0 in triangleMat[k, :] or 1 in triangleMat[k, :] or 2 in triangleMat[k, :] :
537             tempDel.append(k)
538     # print(tempDel)
539     triangleMat = np.delete(triangleMat, tempDel, axis=0) # axis=0 表示删除行
```

```
540         edgeMat = np.concatenate([triangleMat[:, [0, 1]], triangleMat[:, [1, 2]],
    triangleMat[:, [2, 0]]], axis=0)
541         # edgeMat = np.array(sorted(edgeMat.tolist(), key=lambda x: (x[0], x[1])))
542         edgeMat = np.sort(edgeMat, axis=1)
543         edgeMat = self._unranked_unique(edgeMat)
544 
545         return triangleMat, edgeMat
546 
547 
548         # tempEdgeMat = np.empty(shape=[0, 6], dtype=np.int)
549         # tempTriangleMat = np.empty(shape=[0, 3], dtype=np.int)
550 
551     def plot_figure(self, triangleMat, title="Delaulay Triangle", save_path="./figures/
    Delaunay_Triangle.png"):
552         plt.figure(figsize=(8, 8), facecolor='w')
553         for i in  range(triangleMat.shape[0]):
554             # for j in range(2)
555                 # plt.plot()
556             # point1 = [self.dots[triangleMat[i, 0], 0], self.dots[triangleMat[i, 0], 0]]
557             plt.plot([self.dots[triangleMat[i, 0], 0], self.dots[triangleMat[i, 1], 0]],
558                      [self.dots[triangleMat[i, 0], 1], self.dots[triangleMat[i, 1], 1]],
    "b-")
559             plt.plot([self.dots[triangleMat[i, 0], 0], self.dots[triangleMat[i, 2], 0]],
560                      [self.dots[triangleMat[i, 0], 1], self.dots[triangleMat[i, 2], 1]],
    "b-")
561             plt.plot([self.dots[triangleMat[i, 1], 0], self.dots[triangleMat[i, 2], 0]],
562                      [self.dots[triangleMat[i, 1], 1], self.dots[triangleMat[i, 2], 1]],
    "b-")
563 
564         plt.title(title)
565         plt.savefig(os.path.join(self.based_path, save_path))
566         plt.show()
567 
568     def _makeTempTriangle(self, tempEdgeMat, i):
569         """
570         将输入的边和目标点组成三角形，最终返回组成的三角形的下标
571         输入参数:
572             tempEdgeMat: 输入的边的集合 N * 6 分别表示组成该边两个点的下标 两个点对应的x y值
573             i: 表示目标点在self.dots中的下标
574         输出参数:
575             tempTriangle: N * 3 3表示组成该三角形的顶点的在dots中的下标
576         """
```

```
577         pointNum = tempEdgeMat[:, [0, 1]] # 得到所有顶点的坐标
578         # print("pointNum", pointNum)
579         pointLine = pointNum[pointNum != i].astype(np. int) # 得到所有的不是该顶点的point
580 
581         N =  len(pointLine) # 边的个数其实就是pointLine的个数
582         points = self.dots[pointLine, :]
583         complexPoints = [ complex(point[0], point[1]) for point in points]
584         # print(complexPoints)
585         diffComplexPoints = [complexPoint -
    complex(self.dots[i, 0], self.dots[i, 1]) for complexPoint in complexPoints]
586         # print(diffComplexPoints)
587         angle = np.angle(diffComplexPoints, deg=False) # 返回每个复数的角度 然后对复数进行排
    序
588         index = np.argsort(angle).tolist()
589         index.append(index[0]) # 这里将最开始的下标添加进去方便遍历
590         # print(index)
591         tempTriangle = np.empty(shape=[0, 3], dtype=np. int)
592         for k in  range(N):
593             tempTriangle = np.append(tempTriangle, np.array([pointLine[index[k]], pointLine[
    index[k+1]], i]).reshape(1, -1), axis=0)
594         return tempTriangle
595 
596     def __convertData(self, dot1, dot2, arr1, arr2, flag=True):
597         """
598         转化数据类型
599         输入参数:
600             dot1 dot2 表示两个数值
601             arr1 arr2 表示两个数组
602             flag: True 表示默认转为list 否则则是array
603         输出参数:
604             表示将dot1 dot2 arr1 arr2连接的list
605         """
606         if flag:
607             return np.concatenate([np.array([dot1, dot2]), arr1, arr2]).tolist()
608         else:
609             return np.concatenate([np.array([dot1, dot2]), arr1, arr2])
610 
611     def _unranked_unique(self, nparray):
612         """
613         输出二维list的去重不排序的结果
614         输出参数:
615             nparray: 数组
```

```
616         输出参数:
617             T: 不自动排序但是去重的数组
618         """
619         # print("nparray", nparray)
620         T = np.empty(shape=[0, nparray.shape[1]], dtype=np. int)
621         for i in nparray:
622             mask = True
623             for j in T:
624                 if (i == j). all():
625                     mask = False
626             if mask:
627                 T = np.append(T, i.reshape(1, -1), axis=0)
628         return T
629 
630     def _makeEdge(self, dot1, dot2, dot3):
631         """
632         将dot1 dot2 dot3 这三个点构成三条边
633         输入参数:
634             dot1 dot2 dot3: 三个点在dots中的序号
635         输出参数:
636             edgeMat: 三个点构成的三条边的集合
637         """
638         edge1 = self._makeOneEdge(dot1, dot2)
639         edge2 = self._makeOneEdge(dot1, dot3)
640         edge3 = self._makeOneEdge(dot2, dot3)
641         edgeMat = np.concatenate([edge1, edge2, edge3], axis=0)
642         return edgeMat
643         # print("edge1:", edge1)
644 
645     def _makeTempEdge(self, dot1, dot2, dot3, targetdot):
646         """
647         将dot1 dot2 dot3 这三个点和targetdot构成三条边
648         输入参数:
649             dot1 dot2 dot3: 三个点在dots中的序号
650             targetdot: 目标点
651         输出参数:
652             edgeMat: 这三个点和targetdot构成三条边
653         """
654 
655         edge1 = self._makeOneEdge(dot1, targetdot)
656         edge2 = self._makeOneEdge(dot2, targetdot)
657         edge3 = self._makeOneEdge(dot3, targetdot)
```

```
658         edgeMat = np.concatenate([edge1, edge2, edge3], axis=0)
659         # print("edgeMat", edgeMat)
660         return edgeMat
661 
662     def _makeOneEdge(self, dot1, dot2):
663         """
664         利用dot1 和 dot2 生成一条边返回 辅助函数_makeEdged的使用
665         输入参数:
666             dot1 dot2: 两个点在dots中的位置
667         输出参数:
668             edge: 顶点+值的组合 输出为一行内容四个值
669 
670         """
671         if self.dots[dot1, 0] < self.dots[dot2, 0]:
672             edge = self.__convertData(dot1, dot2, self.dots[dot1, :], self.dots[dot2, :],
    flag=False)
673         elif self.dots[dot1, 0] == self.dots[dot2, 0]:
674             # 如果两个点的x相等的话
675             if self.dots[dot1, 1] < self.dots[dot2, 1]:
676                 # 判断两个点的y的大小
677                 edge = self.__convertData(dot1, dot2, self.dots[dot1, :], self.dots[dot2,
    :], flag=False)
678             else:
679                 edge = self.__convertData(dot2, dot1, self.dots[dot2, :], self.dots[dot1,
    :], flag=False)
680         else:
681             edge = self.__convertData(dot2, dot1, self.dots[dot2, :], self.dots[dot1, :],
    flag=False)
682 
683         return edge.reshape(1, -1)
684 
685     def _detectArray(self, inputs, dim=3):
686         if inputs.size == 0:
687             # print("+++"*40)
688             return np.empty(shape=[0, dim], dtype=np. int)
689         else:
690             return inputs
691 
692     def _locationPoint(self, dot1, dot2, dot3, tempPoint):
693         """
694         判断点在三角形外接圆的那个部分
695         输入参数:
```

```
696             dot1 dot2 dot2: 三角形的三个顶点
697             tempPoint: 需要判断的点
698         输出参数:
699             mask: 该点相对元三角形外接圆的相对位置
700             0表示外侧 1表示内部 2表示右侧
701         """
702         # print(dot1, dot2, dot3, tempPoint)
703         x0, y0, r = self._getCircle(dot1, dot2, dot3)
704         x, y = tempPoint
705         if x0 + r < x:
706             # 如果点在三角形外接圆的右侧
707             mask = 2
708         elif math.sqrt((x - x0) ** 2 + (y - y0) ** 2) < r:
709             # 如果点在三角形外接圆的内部
710             mask = 0
711         else:
712             # 点在三角形外接圆的外部 跳过
713             mask = 1
714         return mask
715 
716     def _getCircle(self, dot1, dot2, dot3):
717 
718         """
719         给定三角形的三个点 返回三角形的外接圆的x y 和半径
720         输入参数:
721             dot1 dot2 dot3 表示三角形的三个点
722         输出参数:
723             a b:  表示三角形外接圆的x y值
724             r:  表示三角形的外接圆的半径
725         """
726 
727         x1, y1 = dot1
728         x2, y2 = dot2
729         x3, y3 = dot3
730 
731         a=((y2-y1)*(y3*y3-y1*y1+x3*x3-x1*x1)-(y3-y1)*(y2*y2-y1*y1+x2*x2-x1*x1))/(2.0*((x3-x1
    )*(y2-y1)-(x2-x1)*(y3-y1)))
732         b=((x2-x1)*(x3*x3-x1*x1+y3*y3-y1*y1)-(x3-x1)*(x2*x2-x1*x1+y2*y2-y1*y1))/(2.0*((y3-y1
    )*(x2-x1)-(y2-y1)*(x3-x1)))
733         r=math.sqrt((x1-a)*(x1-a)+(y1-b)*(y1-b))
734 
735         return a, b, r
```

```
736
737
738 if __name__ == "__main__":
739     x = [43, 20, 34, 18, 12, 32, 40, 4, 44, 30, 6, 47, 23, 13, 38, 48, 36, 46, 50, 37, 21,
        7, 28, 25, 10]
740     y = [3, 43, 47, 31, 30, 39, 9, 33, 49, 36, 21, 48, 14, 34, 41, 4, 1, 44, 18, 24, 20, 11,
         27, 42, 13]
741     # x = [0.814723686393179, 0.905791937075619, 0.126986816293506]
742     # y = [0.913375856139019, 0.632359246225410, 0.0975404049994095]
743     # x = [0.814723686393179,
744     #      0.905791937075619,
745     #      0.126986816293506,
746     #      0.913375856139019]
747     # y = [0.632359246225410,
748     #      0.0975404049994095,
749     #      0.278498218867048,
750     #      0.546881519204984]
751
752     voronoi = myVoronoi(x, y)
753     # voronoi.createDelaulay()
```

4.2 Drawing Voronoi Graph by Calling API

Listing 2: **Giving Discrete Points, Drawing Voronoi Graph by Calling API**

```
1  import pandas as pd
2  import numpy as np
3  import matplotlib.pyplot as plt
4  import os
5  from scipy.spatial import Voronoi, voronoi_plot_2d
6
7
8  if __name__ == "__main__":
9
10     x = [43, 20, 34, 18, 12, 32, 40, 4, 44, 30, 6, 47, 23, 13, 38, 48, 36, 46, 50, 37, 21,
        7, 28, 25, 10]
11     y = [3, 43, 47, 31, 30, 39, 9, 33, 49, 36, 21, 48, 14, 34, 41, 4, 1, 44, 18, 24, 20, 11,
         27, 42, 13]
12
13     points = [ list(i) for i in  zip(x, y)]
14     vor = Voronoi(points=points)
```

```
15
16
17      # plt.figure(figsize=(8, 8), facecolor='w')
18      fig = voronoi_plot_2d(vor=vor)
19      based_path = os.path.abspath(os.path.dirname(__file__)) # 获取代码运行的基本路径
20      save_path="./figures/Voronoi2.png"
21      plt.savefig(os.path.join(based_path, save_path))
22      plt.show()
```

4.3 GJK

Listing 3: **GJK Algorithm**

```
1  close all; clc; clear;
2
3  Adots = [[0, 0, 0]; [1, 0, 0]; [1, 1, 0]; [0, 1, 0]; [0, 0, 1]; [1, 0 ,1]; [1, 1, 1]; [0, 1,
       1]];
4  Bdots = [[1, 1, 1]; [2, 1, 1]; [2, 2, 1]; [1, 2, 1]; [1, 1, 2]; [2, 1, 2]; [2, 2, 2]; [1, 2,
       2]];
5  Cdots = [[3, 3, 2]; [5, 3, 2]; [4, 5, 2]; [4, 4, 4]];
6
7  iterations = 10;
8  flag1 = GJK(Adots, Bdots, iterations);
9  flag2 = GJK(Bdots, Cdots, iterations);
10 flag3 = GJK(Cdots, Adots, iterations);
11
12 flag = [flag1; flag2; flag3];
13 dotsStr = ['A', 'B', 'C', 'A'];
14 for i=1:size(flag, 1)
15     outputDescription(dotsStr(i), dotsStr(i+1), flag(i));
16 end
17
18
19 function flag = GJK(shape1, shape2, iterations)
20 % GJK 碰撞算法
21 % 只适用于两个凸包物体
22 %
23 % Input:
24 %   shape1:
25 %   必须是三维的，我们的算法是基于三维的
26 %   判断。因此，输入形状必须是 N* 3 维的。
```

```
27 %
28 %   shape2:
29 %   与shape1相同，它必须是由三维几何图形组成的一组点
30 %
31 %   iterations:
32 %   该算法试图构造一个3-simplex的四面体。通过一定的迭代次数，来判断物体是否发生碰撞。
33 %   比较低的迭代次数，算法的运算时间也会减少。而且随着两个物体之间重叠层度的增加，
34 %   算法需要的迭代次数也在减少。所以需要我们在迭代次数中做出权衡。
35 % outputs:
36 %   flag:
37 %   true: 物体发生碰撞
38 %   false: 物体没有发生碰撞
39
40 v = [0.8 0.5 1]; % 方向向量
41 % 先选择两个点
42 [a,b] = pickLine(v,shape2,shape1);
43
44 % 选择第三个点 从而构建三角形(单纯性)
45 [a, b, c, flag] = pickTriangle(a ,b, shape2, shape1, iterations);
46
47 % 选择第四个点 构建四面体
48 if (flag == 1)  % 如果找到三角形 就可以尝试开始构建四面体 如果没找到三角形就直接结束
49     [a,b,c,d,flag] = pickTetrahedron(a,b,c,shape2,shape1,iterations);
50 end
51
52 end
53
54
55 function [a,b,c,d,flag] = pickTetrahedron(a,b,c,shape1,shape2,iterations)
56
57 % 我们已经成功构建了2D的三角单纯性
58 % 现在我们需要检查原点是否在三维单纯形的内部
59
60 % flag初始化为0 表示没有构建四面体
61 flag = 0;
62
63 ab = b-a;
64 ac = c-a;
65
66 % 垂直于三角形的面
67 abc = cross(ab,ac);
68 ao = -a;
```

```
69
70 % 原点在三角形的上面
71 if dot(abc, ao) > 0
72     d = c;
73     c = b;
74     b = a;
75
76     v = abc;
77     % 四面体的新点
78     a = support(shape2,shape1,v);
79
80 % 原点在三角形的下面
81 else
82     d = b;
83     b = a;
84     v = -abc;
85     % 构建四面体的新点
86     a = support(shape2,shape1,v);
87 end
88
89 % 允许迭代最多iterations次数来尝试构建四面体
90 for i = 1:iterations
91     ab = b-a;
92     ao = -a;
93     ac = c-a;
94     ad = d-a;
95
96     % 检查面ABC，ABD和ACD
97     % 垂直于三角形的面
98     abc = cross(ab,ac);
99
100    if dot(abc, ao) > 0 % 面ABC在三角形的上方
101        % continue
102    else
103        acd = cross(ac,ad);% 垂直三角形的面
104
105        if dot(acd, ao) > 0 % 面ACD在三角形的上方
106            % 把这个变成新的底边三角形。
107            b = c;
108            c = d;
109            ab = ac;
110            ac = ad;
```

```
111             abc = acd;
112         elseif dot(acd, ao) < 0
113             adb = cross(ad,ab);% 垂直三角形的面
114 
115             if dot(adb, ao) > 0 % 面ADB在三角形的上方
116                 % 把这个变成新的底边三角形。
117                 c = b;
118                 b = d;
119                 ac = ab;
120                 ab = ad;
121                 abc = adb;
122             else
123                 flag = 1;
124                 break; % 原点在四面体的内部 退出
125             end
126         end
127     end
128 
129     if dot(abc, ao) > 0
130         d = c;
131         c = b;
132         b = a;
133         v = abc;
134         a = support(shape2,shape1,v); % 构建四面体的新点
135     else %below
136         d = b;
137         b = a;
138         v = -abc;
139         a = support(shape2,shape1,v); % 构建四面体的新点
140     end
141 end
142 
143 end
144 
145 function [a, b, c, flag] = pickTriangle(a, b, shape2, shape1, iterations)
146 
147 % flag = 0 表示没有建立三角形
148 flag = 0;
149 
150 % 第一次尝试
151 ab = b-a;
152 ao = -a;
```

```
153
154 % v垂直于ab，指向原点的大致方向。
155 v = cross(cross(ab,ao),ab);
156
157 c = b;
158 b = a;
159 a = support(shape2,shape1,v);
160
161 for i = 1:iterations
162
163     ab = b-a;
164     ao = -a;
165     ac = c-a;
166
167     % 垂直于三角形的面
168     abc = cross(ab,ac);
169
170     % 垂直于AB远离三角形
171     abp = cross(ab,abc);
172     % 垂直于AC远离三角形
173     acp = cross(abc,ac);
174
175     % 首先，确保我们的三角形“包含”二维投影中的原点
176     % 原点是否在AB之上
177     if dot(abp,ao) > 0
178         c = b; % 丢弃最远的一点，在正确的方向得到最新的一个点
179         b = a;
180         v = abp;
181
182         % 原点是否在AC之上
183     elseif dot(acp, ao) > 0
184         b = a;
185         v = acp;
186
187     else
188         flag = 1;
189         break; % 成功构建三角形
190     end
191     a = support(shape2,shape1,v);
192 end
193
194 end
```

```
195
196 function [a, b] = pickLine(v, shape2, shape1)
197 % 构造单纯性的第一条直线
198 % 分别在方向向量v和-v上选择最远的点
199 % 从而可以构造出单纯性的第一条线段
200 b = support(shape2, shape1, v);
201 a = support(shape2, shape1, -v);
202 end
203
204 function point = support(shape2, shape1, v)
205 % 得到 Minkowski 差
206 % 分别得到在给定 v 和 -v方向上最远的两个点
207 point1 = getFartherPoint(shape1, v);
208 point2 = getFartherPoint(shape2, -v);
209 point = point1 - point2;
210 end
211
212 function point = getFartherPoint(shape, v)
213 % 找到在给定方向上，该几何体投影的点，也就是最远的点
214 x = shape(:, 1);
215 y = shape(:, 2);
216 z = shape(:, 3);
217 % 这一步操作相当于该几何体上所有的点和原点组成的向量和方向向量v点乘
218 % 然后计算出点乘之后的最大值 就是该几何体上沿着该方向向量最远的点
219 dotted = x*v(1) + y*v(2) + z*v(3);
220 [~, maxIdx] = max(dotted);
221 % 获得该几何体在该方向向量上最远的点
222 point = [x(maxIdx), y(maxIdx), z(maxIdx)];
223 end
224
225
226 function [] = outputDescription(str1, str2, mask)
227 % 输出说明
228 % 输入mask 输出两个物体是否产生碰撞
229
230 if mask
231     sprintf('There is no collision between object %s and object %s', str1, str2)
232 else
233     sprintf('There is a collision between object %s and object %s', str1, str2)
234
235 end
236 end
```

4.4 Bounding Sphere Collision Detection Algorithm

Listing 4: **Bounding Sphere Collision Detection Algorithm**

```
% 基于包围球的碰撞检测
close all;clear;clc;


Adots = [[0, 0, 0]; [1, 0, 0]; [1, 1, 0]; [0, 1, 0]; [0, 0, 1]; [1, 0 ,1]; [1, 1, 1]; [0, 1, 1]];
Bdots = [[1, 1, 1]; [2, 1, 1]; [2, 2, 1]; [1, 2, 1]; [1, 1, 2]; [2, 1, 2]; [2, 2, 2]; [1, 2, 2]];
Cdots = [[3, 3, 2]; [5, 3, 2]; [4, 5, 2]; [4, 4, 4]];
% dotsList = cat(3, Adots, Bdots, Cdots)
dotsStr = ['A', 'B', 'C', 'A'];

[center1, r1] = getSphereCenter(Adots);
[center2, r2] = getSphereCenter(Bdots);
[center3, r3] = getSphereCenter(Cdots);

centerList = [center1; center2; center3];
rList = [r1; r2; r3];


for i=1:size(dotsStr, 2)-1
    judgeCollision(centerList(i), r1, dotsStr(i), centerList(i+1), r2, dotsStr(i+1));
end

function mask = judgeCollision(center1, r1, str1, center2, r2, str2)
% 输入两个三维几何题的坐标 判断两个三维几何题是否会发生碰撞
% 输入参数:
%   center1 center2:  两个包围球的球心坐标
%   r1 r2:  两个包围球的球心半径
%   str1 str2:  说明是那个几何体
% 输出参数:
%   mask:
%   1 表示没有发生碰撞
%   0 表示发生碰撞

% 比较两个圆心之间的距离
```

```
35 % 如果两个圆心之间的距离大于两个圆的半径总和说明没有发生碰撞
36 if pdist2(center1, center2) > (r1 + r2)
37     mask = 1;
38 else
39     mask = 0;
40 end
41
42 % 输出判断
43 outputDescription(str1, str2, mask);
44
45 end
46
47
48 function [centerDot, r] = getSphereCenter(dots)
49 % 输入三维几何题的坐标 返回该三维几何体球心坐标和球的半径
50 % 输入参数:
51 %   dots:  输入凸包的三维坐标
52 xmax = max(dots(:, 1)); xmin = min(dots(:, 1));
53 ymax = max(dots(:, 2)); ymin = min(dots(:, 2));
54 zmax = max(dots(:, 3)); zmin = min(dots(:, 3));
55
56 % 求解球心的坐标
57 % 球心的坐标就是最大点和最小点的中间
58 x = mean([xmax, xmin]); y = mean([ymax, ymin]); z = mean([zmax, zmin]);
59 centerDot = [x, y, z];
60 r = pdist2([xmax, ymax, zmax], [xmin, ymin, zmin]) / 2; % 计算两个点之间的距离除以2
61
62 end
63
64
65 function [] = outputDescription(str1, str2, mask)
66 %% 输出说明
67 % 输入mask 输出两个物体是否产生碰撞
68
69 if mask
70     sprintf('There is no collision between object %s and object %s', str1, str2)
71 else
72     sprintf('There is a collision between object %s and object %s', str1, str2)
73
74 end
75
76 end
```

4.5 AABB Bounding Box Collision Detection Algorithm

Listing 5: **AABB Bounding Box Collision Detection Algorithm**

```
% 基于AABB的碰撞检测
close all;clear;clc;


Adots = [[0, 0, 0]; [1, 0, 0]; [1, 1, 0]; [0, 1, 0]; [0, 0, 1]; [1, 0 ,1]; [1, 1, 1]; [0, 1, 1]];
Bdots = [[1, 1, 1]; [2, 1, 1]; [2, 2, 1]; [1, 2, 1]; [1, 1, 2]; [2, 1, 2]; [2, 2, 2]; [1, 2, 2]];
Cdots = [[3, 3, 2]; [5, 3, 2]; [4, 5, 2]; [4, 4, 4]];
% dotsList = cat(3, Adots, Bdots, Cdots)
dotsStr = ['A', 'B', 'C', 'A'];


[mindots1, maxdots1] = getMinMaxDots(Adots);
[mindots2, maxdots2] = getMinMaxDots(Bdots);
[mindots3, maxdots3] = getMinMaxDots(Cdots);

minDots = [mindots1; mindots2; mindots3];
maxDots = [maxdots1; maxdots2; maxdots3];
dic = [1, 2, 3, 1];

for i=1:size(dotsStr, 2) - 1
    k = dic(i+1);
    judgeCollision(minDots(i, :), maxDots(i, :), dotsStr(i),...
                                    minDots(k, :), maxDots(k, :), dotsStr(k));
end


function mask = judgeCollision(minDots1, maxDots1, str1, minDots2, maxDots2, str2)
% 输入两个三维几何题的坐标 判断两个三维几何题是否会发生碰撞

% 比较两个圆心之间的距离
% 如果两个圆心之间的距离大于两个圆的半径总和说明没有发生碰撞
rangeList = [1, 2, 3, 1];
res = 0;
for i=1:size(rangeList, 2)-1
```

```
35      res = res + judgeTwoDim(minDots1([rangeList(i), rangeList(i+1)]),...
36                      maxDots1([rangeList(i), rangeList(i+1)]), minDots2([rangeList(i),
        rangeList(i+1)]),...
37                      maxDots2([rangeList(i), rangeList(i+1)]));
38  end
39
40
41  % 只有三个的坐标轴都不发生碰撞 才表明两个三维几何体不会发生碰撞
42  if res == 3
43      mask = 1;
44  else
45      mask = 0;
46  end
47
48  % 输出判断
49  outputDescription(str1, str2, mask);
50
51  end
52
53  function flag = judgeTwoDim(minDot1, maxDot1, minDot2, maxDot2)
54  % 对应坐标轴上的多边形是否发生重叠 进而判断是否发生碰撞
55  % 输入参数:
56  %   minDot1 minDot2:  表示两个凸包在两个坐标轴上最小的点
57  %   maxDot1 maxDot2:  表示两个凸包在两个坐标轴上最大的点
58  % 输出参数:
59  %   flag:
60  %   1 表示没有发生碰撞
61  %   0 表示发生了碰撞
62
63  flag = 1; % 1 表示没有发生碰撞
64  if (maxDot1(1) >= minDot2(1)) && (maxDot2(1) >= minDot1(1)) &&...
65                      (maxDot1(2) >= minDot2(2)) && (maxDot2(2) >= minDot1(2))
66      flag = 0; % 如果满足碰撞的条件 也就是对应坐标轴上的投影发生重叠
67  else
68      flag = 1;
69  end
70
71  end
72
73
74  function [mindots, maxdots] = getMinMaxDots(dots)
75  % 获取传入的多边形的最小顶点和最大顶点
```

```
76 % 输入参数:
77 %   dots:  输入凸包的三维坐标
78
79 xmax = max(dots(:, 1)); xmin = min(dots(:, 1));
80 ymax = max(dots(:, 2)); ymin = min(dots(:, 2));
81 zmax = max(dots(:, 3)); zmin = min(dots(:, 3));
82
83 mindots = [xmin, ymin, zmin];
84 maxdots = [xmax, ymax, zmax];
85 end
86
87
88 function [] = outputDescription(str1, str2, mask)
89 % 输出说明
90 % 输入mask 输出两个物体是否产生碰撞
91
92 if mask
93     sprintf('There is no collision between object %s and object %s', str1, str2)
94 else
95     sprintf('There is a collision between object %s and object %s', str1, str2)
96
97 end
98
99 end
```