站点聚类实验报告

组队：顾天阳 曾世茂 韩世民

学号：2019211539 2019211532 2019211496

班级：2019211308

指导老师：潘维民 邓芳

目录

[一、 运行环境与操作指令 1](#_Toc90327562)

[1. 运行环境： 1](#_Toc90327563)

[2. 操作指令： 2](#_Toc90327564)

[二、 代码运行截图 4](#_Toc90327565)

[1. 运行时间戳 4](#_Toc90327566)

[2. 绘制散点图 6](#_Toc90327567)

[3. 并行计算 7](#_Toc90327568)

[三、预处理和算法分析 8](#_Toc90327569)

[1、预处理 8](#_Toc90327570)

[2、算法分析 9](#_Toc90327571)

[四、结果分析 14](#_Toc90327572)

1. 运行环境与操作指令
2. 运行环境：
3. 操作系统：Windows 10 家庭中文版 20H2 19042.1348
4. 硬件信息：

CPU：Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz 2.21 GHz

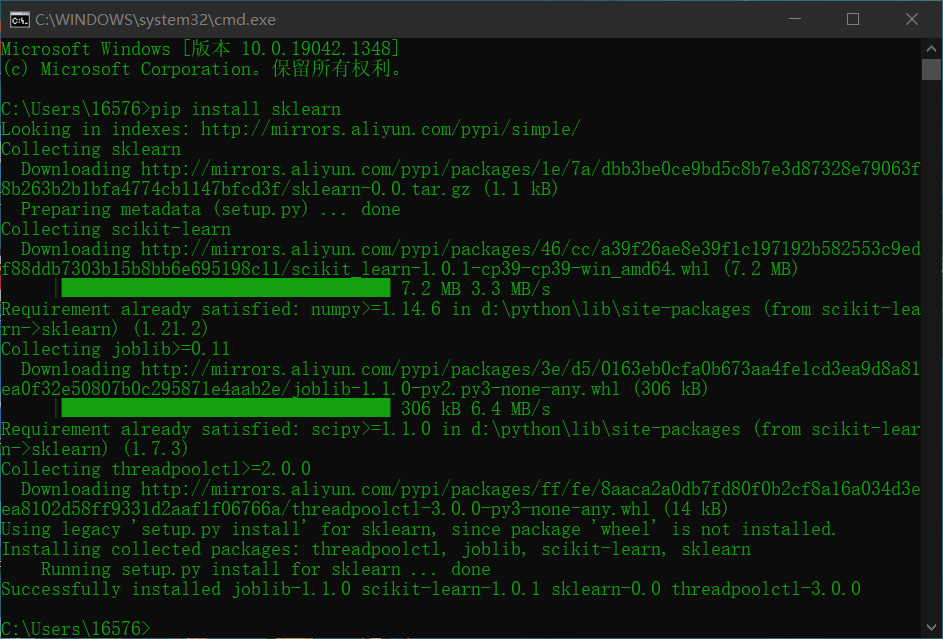
RAM：8.00 GB (7.88 GB 可用)



1. 操作指令：
2. 依赖包安装：

通过cmd命令安装kmeans算法包sklearn命令如下：

pip install sklearn

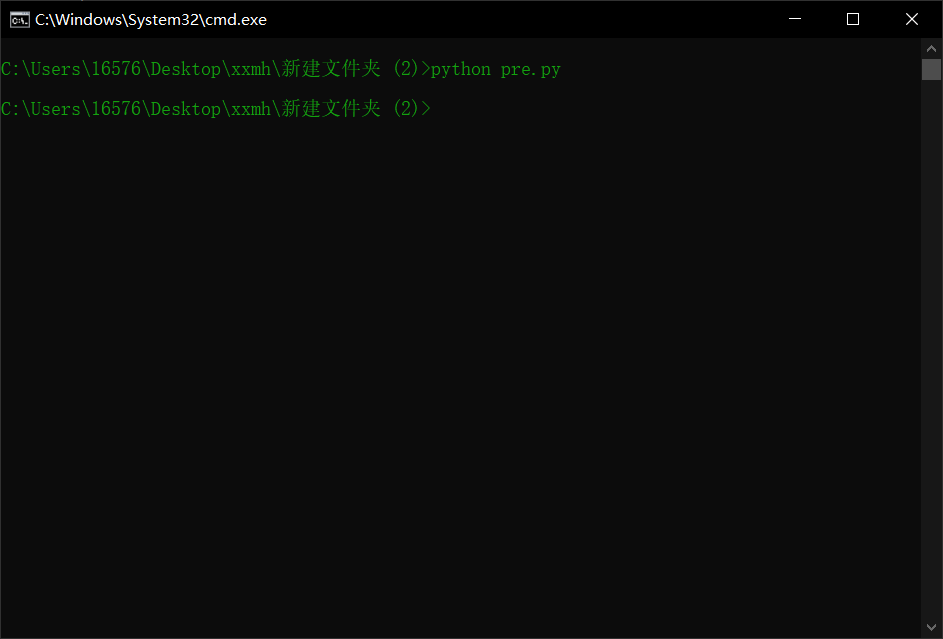


1. 预处理数据：

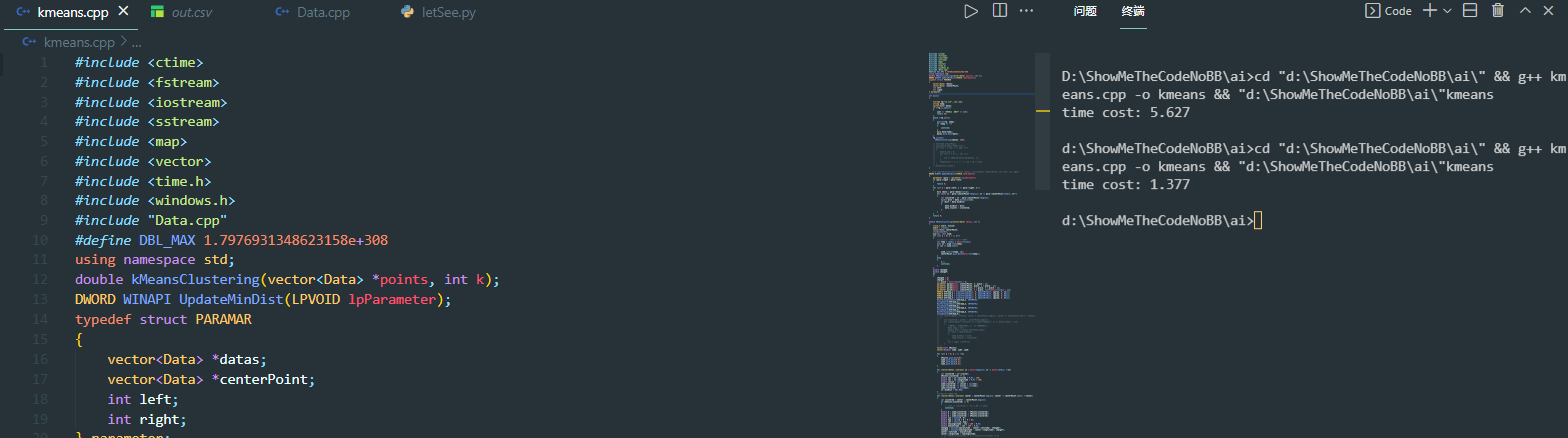
编写python脚本pre.py对数据进行预处理，删除错误数据，规范数据集。

通过cmd命令运行python脚本，命令如下:

python pre.py

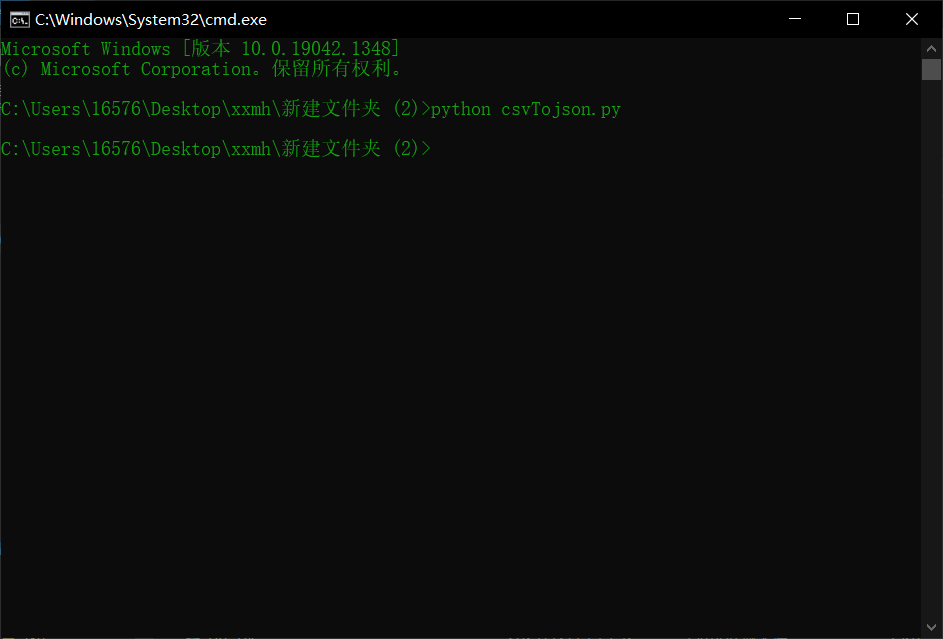


1. 使用自己实现的kmeans算法对预处理后的数据聚类,之前安装的kmeans算法依赖包可以用于检验结果:

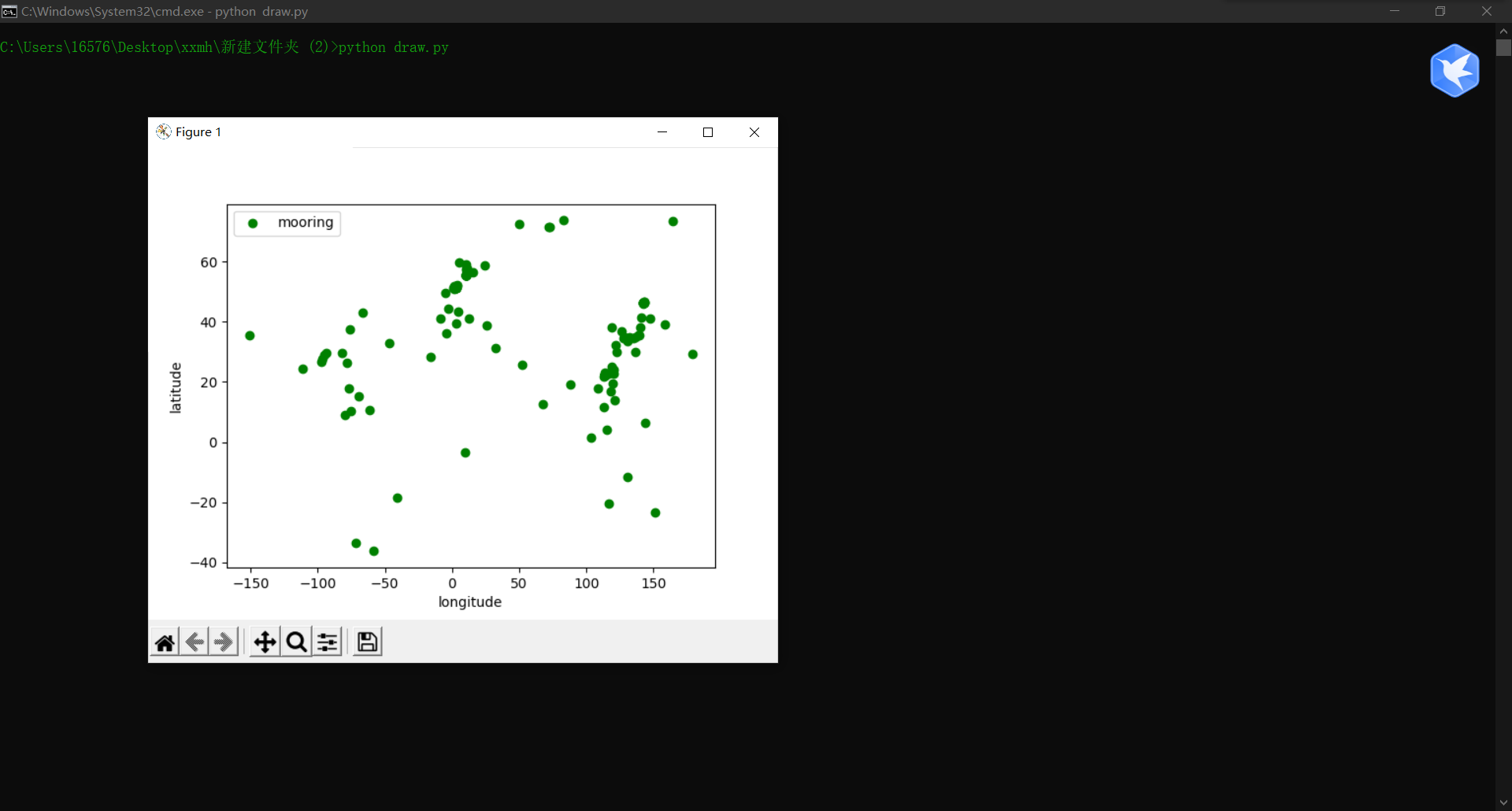


1. 使用python脚本对数据进行格式化处理，并制作相应的图表：

将结果转为json数组：python csvTojson.py

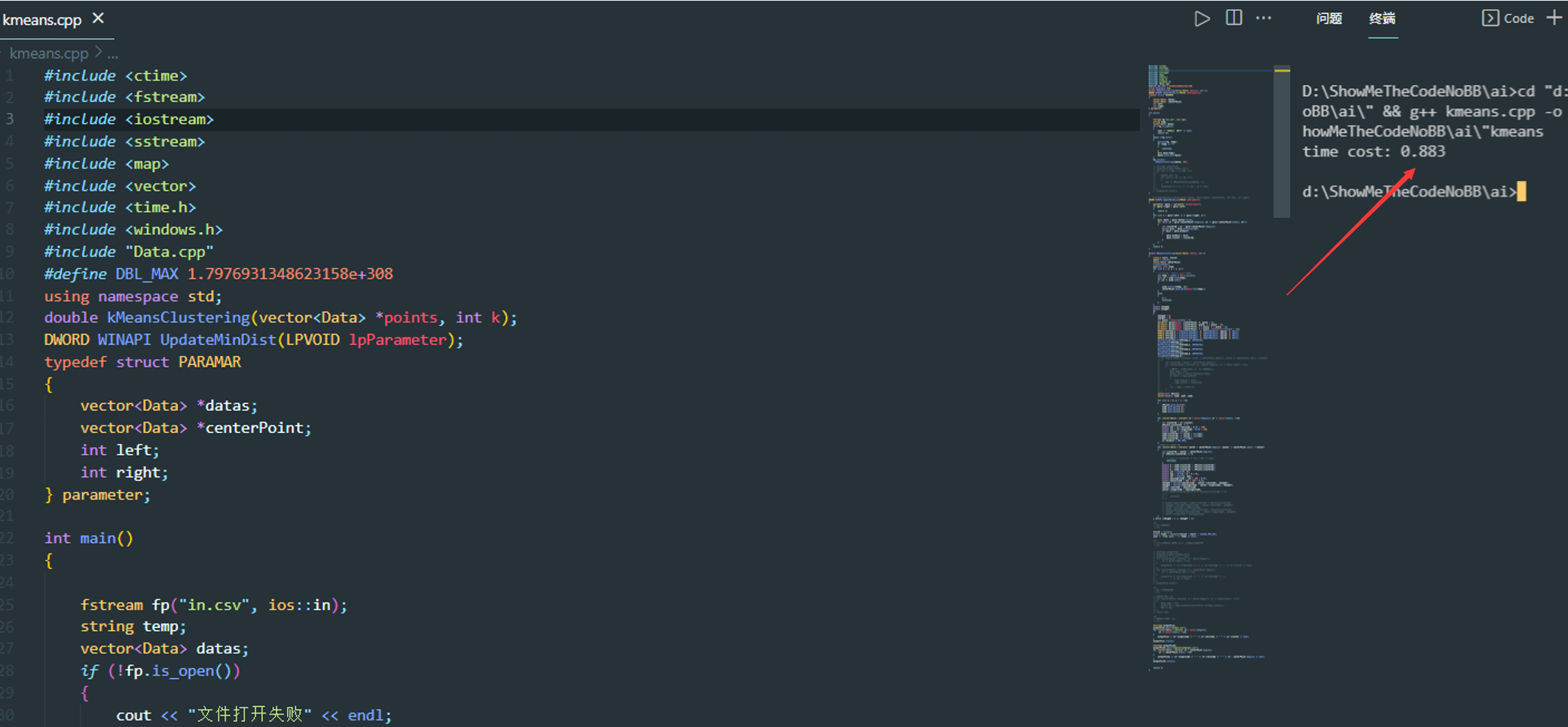


绘制图表：python draw.py

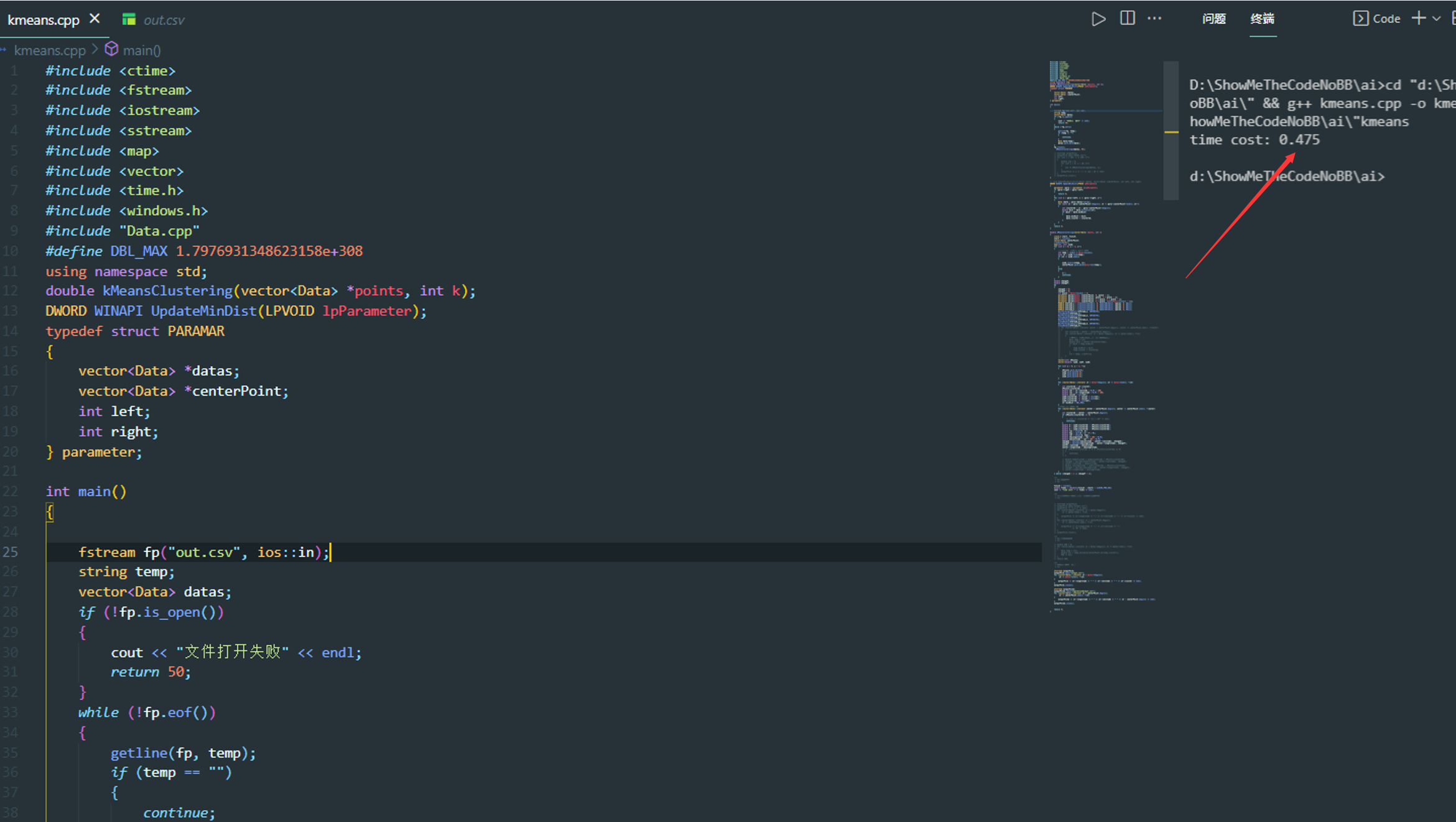


1. 代码运行截图
2. 运行时间戳

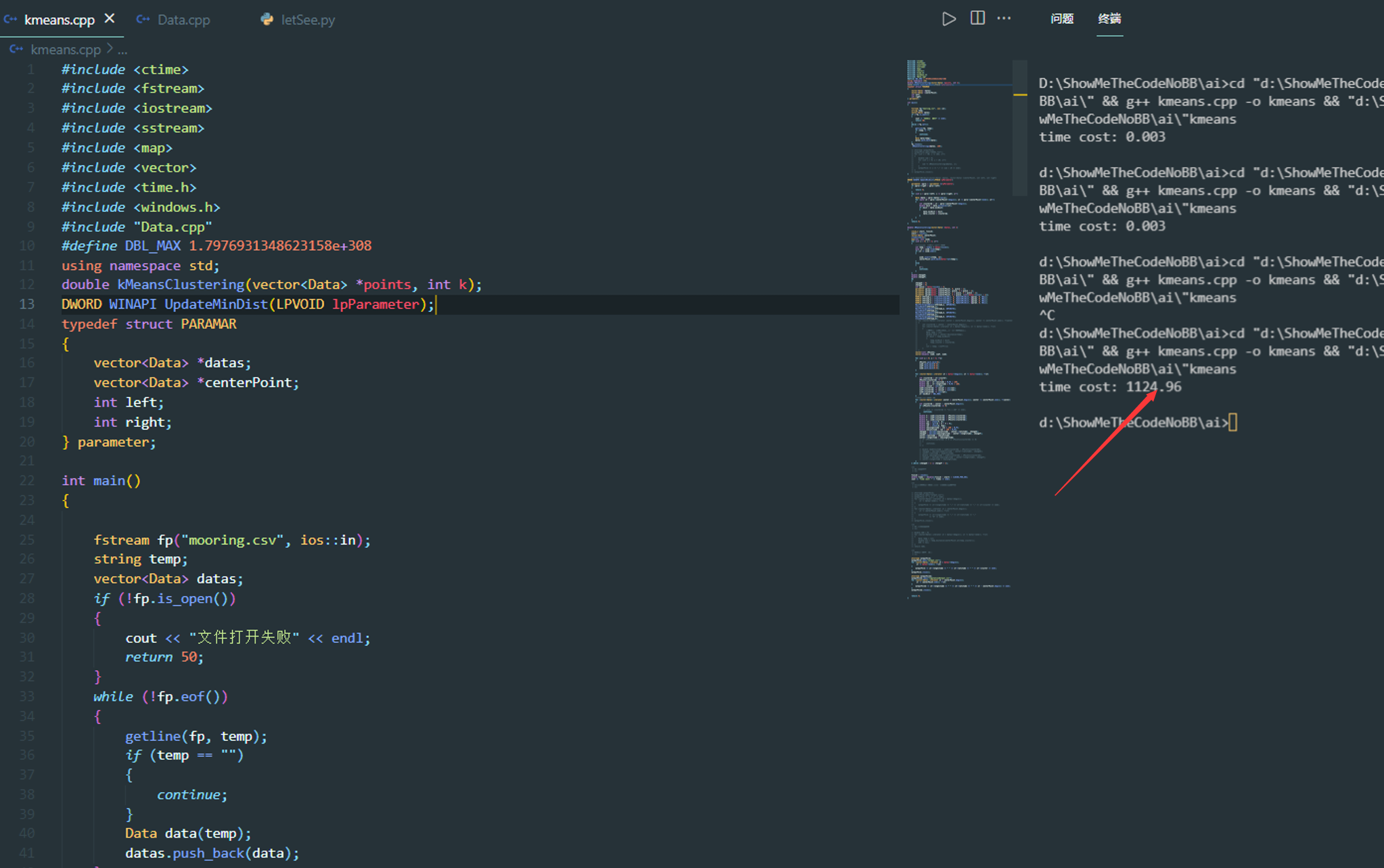
入站：



出站：

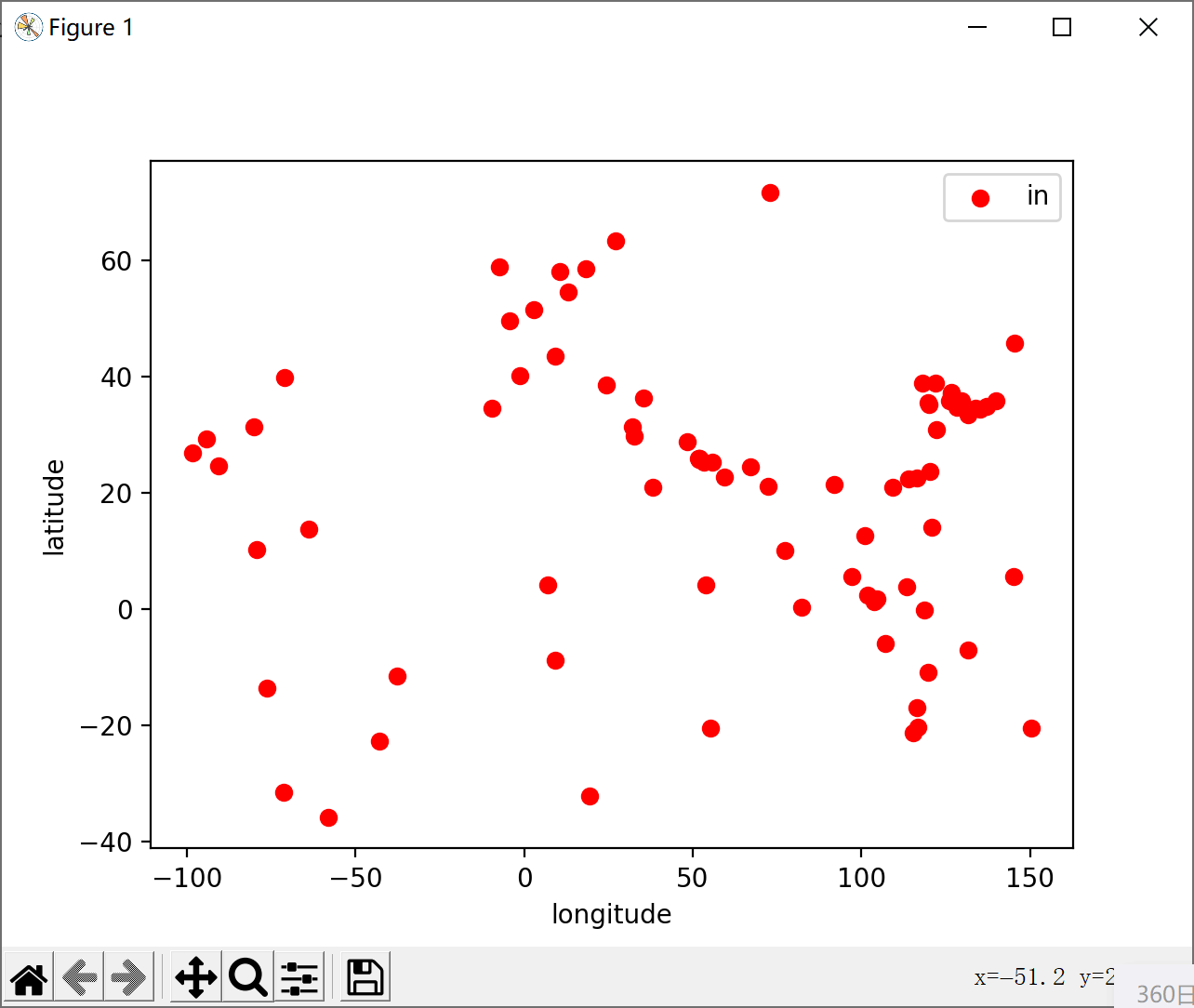


锚地：

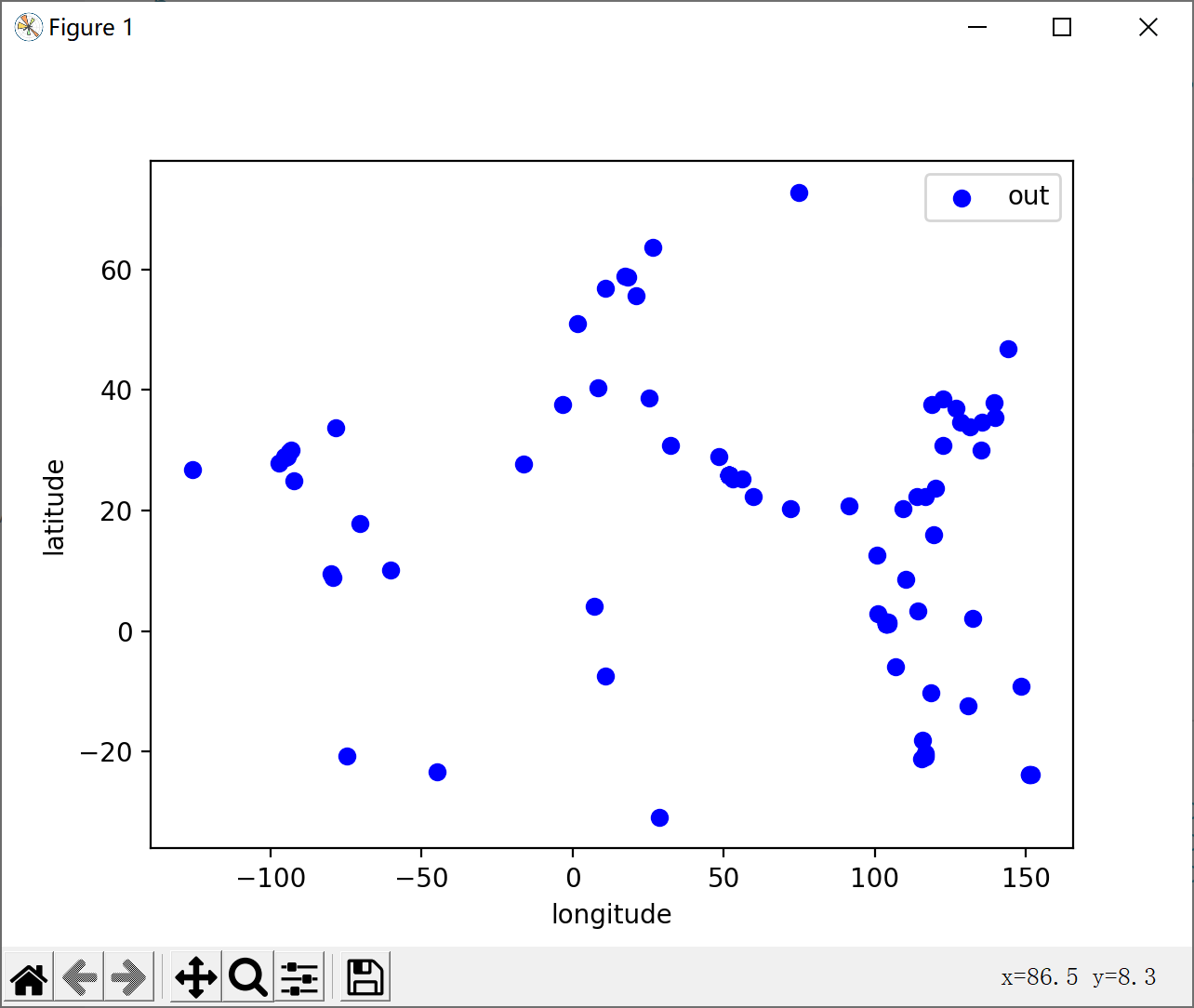


1. 绘制散点图

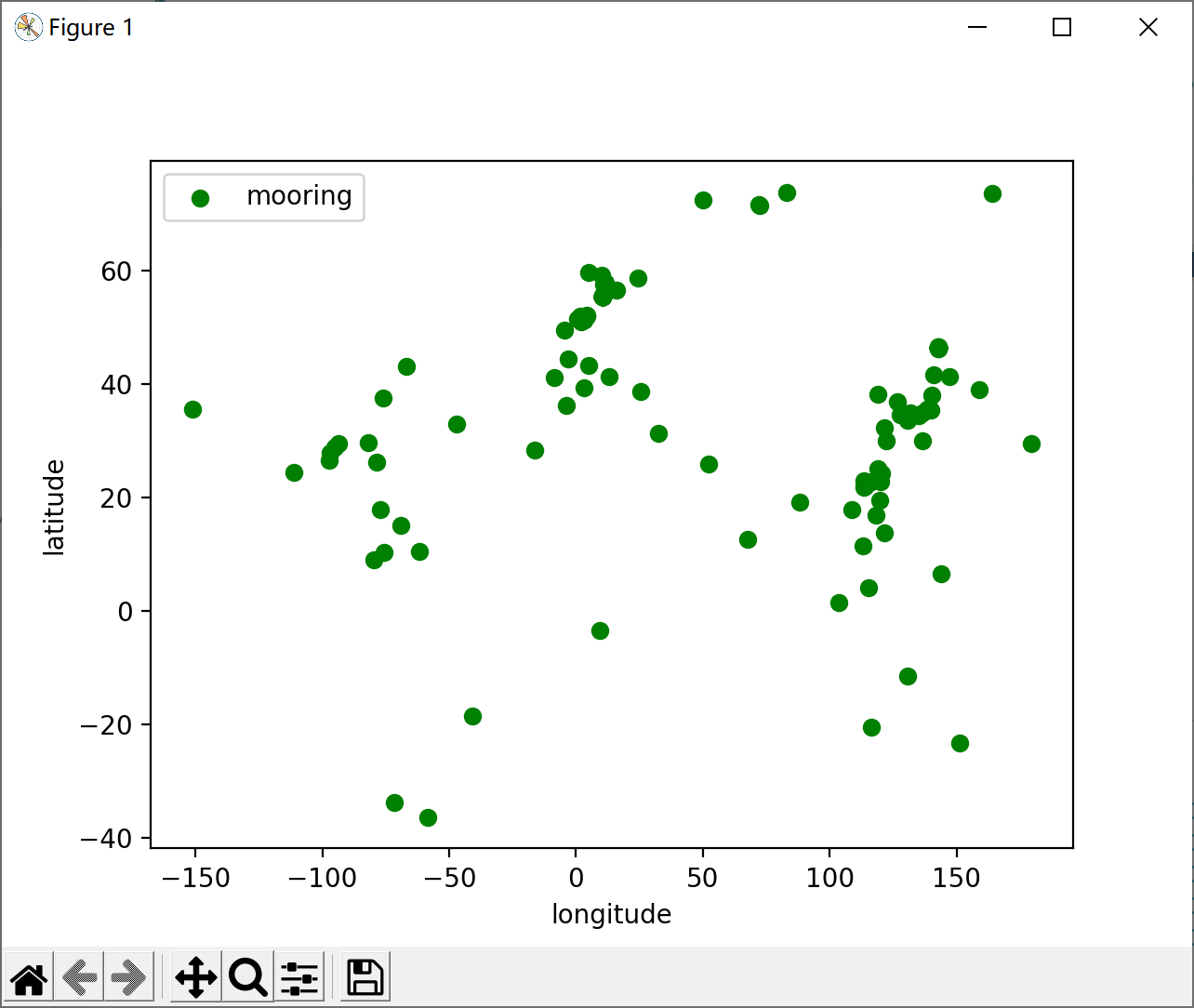
入站：



出站：



锚地：



1. 并行计算

大多数程序运行时间都消耗更新每个点最近的中心点的计算中。由于该计算需要遍历所有的点，每个点均需遍历所有的中心并计算与该中心的距离，找出距离最小的中心点并将自己的类别设置成该中心所属的类。故该过程的时间复杂度为O(NK)。

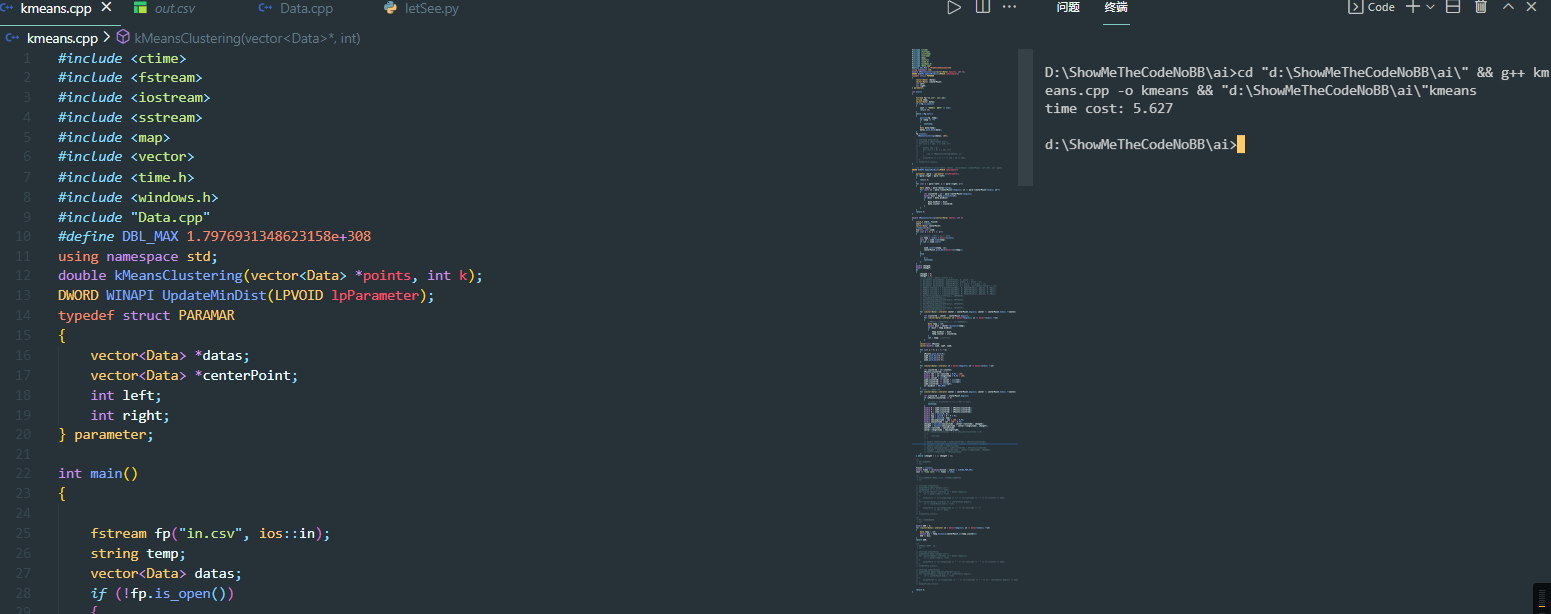
而在比较各个点与各个中心最小距离的时候，各个点的计算过程是相互独立的，所以可以并行计算以加快处理过程。在该程序中，小组结合所使用设备cpu最大核心数为4的特点，使用了四线程并行计算处理该过程。成功将平均运行时间缩短了4倍。

运行时间截图：

数据：LNG入站点

数据量1w

分类数：180

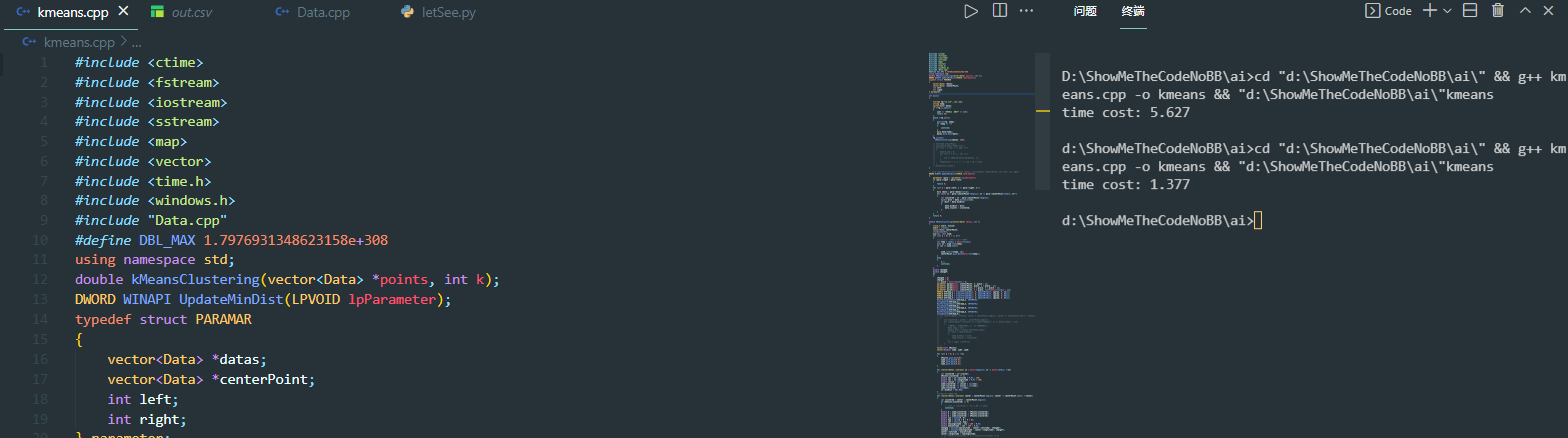


不使用并行计算加速：

运行时间5.627s

使用4线程并行计算加速：

运行时间：1.377s



加速比：5.627/1.377 ≈ 4

与预想中的加速效果以及使用线程数一致，也印证了该算法主要时间消耗点位于 计算更新中心点处，将多线程加速用于该处理过程是正确的选择。

三、预处理和算法分析

1、预处理

1. # 读取lng2.csv中的数据，经过预处理分类输出到各个文件中
2. f = open('lng2.csv', 'r+')
3. f1 = open('in.csv', 'w+')
4. f2 = open('out.csv', 'w+')
5. f3 = open('mooring.csv', 'w+')
7. is\_first = True  # 是否为第一个有效数据
8. draft, last\_draft = 0, 0  # 当前吃水深度和上一个吃水深度
9. **for** line **in** f.readlines():
10. line = line.split()
11. draft = int(line[6])
12. **if** draft == 0 **or** draft > 300:
13. **continue**
15. **if** is\_first == True:
16. is\_first = False
17. f3.write(' '.join(line) + '\n')
18. # 如果小于上一个吃水深度，则加入入站
19. **elif** draft < last\_draft:
20. f1.write(' '.join(line) + '\n')
21. # 如果大于上一个吃水深度，则加入出战
22. **elif** draft > last\_draft:
23. f2.write(' '.join(line) + '\n')
24. # 如果等于上一个吃水深度，则加入锚地
25. **elif** draft == last\_draft:
26. f3.write(' '.join(line) + '\n')
27. # 更新上一个吃水深度
28. last\_draft = draft
30. f.close()
31. f1.close()
32. f2.close()
33. f3.close()

结果中锚地数据约为百万级别，入站和出战为万级别，符合实际中锚地较大导致站点个数过多的事实。

2、算法分析

（1）算法步骤

kmeans算法的主要核心思想很简单。主要的步骤如下：

1、首先根据给出的簇数目k选出k个初始中心点。在该实验中，小组选择了随机设置初始中心点。

2、随后遍历每一个点，找出与其距离最短的中心点，并将自己所属的类设为该中心点的类。

3、遍历完所有的点之后，每个中心点的坐标更改成自己所在类成员的中心点。

4、重复步骤2和3，直到满足退出条件

在该实验中，由于考虑到各个站点之间的距离不会靠的太接近，故小组将退出条件设置成更新完中心点的坐标后，每个新坐标与揂坐标的经度纬度变化均不超过1，则认为所有中心点已经稳定，聚类结束退出。

（2）算法实现：

全局变量以及类定义：

类定义：

1. **class** Data
2. {
3. string msi;
4. **long** **long** **int** time;
5. **int** state;
6. **int** volicity;
7. **int** load;
8. **double** latitude;   //纬度
9. **double** longtitude; //经度
10. **int** cluster;  //所属的类
11. **double** minDist;  //到所属中心的距离
12. }

结构体定义：

1. **typedef** **struct** PARAMAR  //用于多线程传递参数
2. {
3. vector<Data> \*datas;  //所有数据的集合
4. vector<Data> \*centerPoint;  //所有中心点的集合
5. **int** left;  //该线程的任务划分起始下标
6. **int** right;  //该线程任务划分结束下标，左闭右闭
7. } parameter;

全局变量定义：

1. #define DBL\_MAX 1.7976931348623158e+308 // double最大值
2. #define EARTH\_RADIUS 6378.137  //地球半径
3. #define M\_PI 3.14159265358979323846  //圆周率

Kmeans算法

1. **void** kMeansClustering(vector<Data> \*datas, **int** k)//输入样本集以及k值
2. {
3. vector<Data> centerPoint;
4. **for** (**int** i = 0; i < k; i++)
5. {
6. ramdonlySelectKpoint();
7. }
8. **double** changeX,changeY;//更新中心点后每个中心点的经纬度变化最大值
9. **do**
10. {
11. changeX = 0;
12. changeY = 0;
13. updateMinDist();
14. vector<**int**> pointsNum;
15. vector<**double**> sumX, sumY, sumZ;
16. //改变每个中心点位置
17. **for** (vector<Data>::iterator center = centerPoint.begin(); center != centerPoint.end(); ++center)
18. {
19. **int** clusterId = center - centerPoint.begin();
20. **if** (pointsNum[clusterId] == 0)
21. {
22. //说明该类出现了空簇
23. **continue**;
24. }
25. calculateNewCenterPoint();
26. changeX = max(abs(newLatitude - center->latitude), changeX);
27. changeY = max(abs(newLongtitude - center->longtitude), changeY);
28. center->latitude = newLatitude;
29. center->longtitude = newLongtitude;
30. }
31. } **while** (changeX > 1 || changeY > 1);//只要有一个中心经纬度变化超过1则继续聚类
32. }

计算每个点到每个中心的最短距离，并更新所属的类

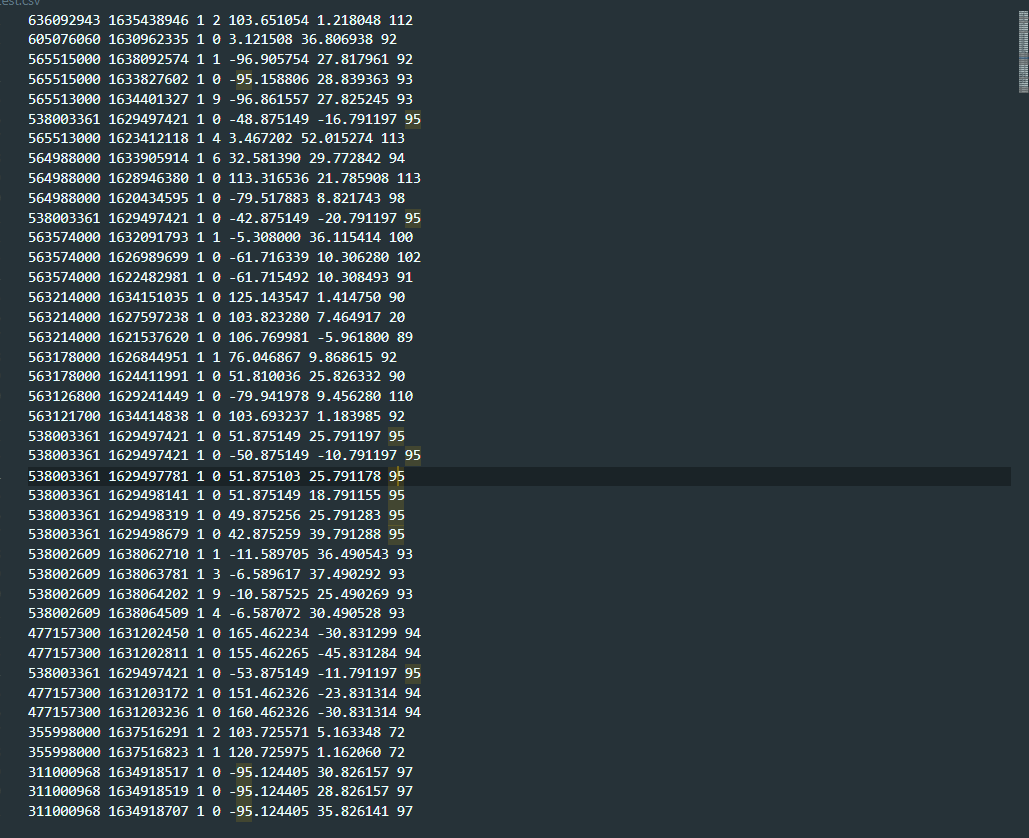
1. **DWORD** WINAPI UpdateMinDist(**LPVOID** lpParameter)  //多线程实现
2. {
3. /\* 原包含参数：
4. vector<Data> \*datas; 数据集
5. vector<Data> \*centerPoint;//中心集
6. int left;//该线程的任务划分起始下标
7. int right;//该线程任务划分结束下标，左闭右闭类型
8. \*/
9. parameter \*para = (parameter \*)lpParameter;
10. **if** (para->right < para->left)
11. {
12. **return** 0;
13. }
14. **for** (**int** i = para->left; i <= para->right; i++)
15. {
16. Data &data = para->datas->at(i);//遍历每一个该线程的划分节点
17. **for** (auto it = para->centerPoint->begin(); it != para->centerPoint->end(); it++)//遍历每一个中心
18. {
19. **int** clusterId = it - para->centerPoint->begin();//得到中心id
20. **double** dist = data.distance(\*it);//计算距离
21. **if** (dist < data.minDist)
22. {
23. data.minDist = dist;
24. data.cluster = clusterId;//更新中心
25. }
26. }
27. }
28. **return** 0;
29. }

根据经纬度计算两点距离

1. **double** Data::GetDistance2(**double** lng1, **double** lat1, **double** lng2, **double** lat2)
2. {
3. **double** radLat1 = rad(lat1);
4. **double** radLat2 = rad(lat2);
5. **double** radLng1 = rad(lng1);
6. **double** radLng2 = rad(lng2);
7. **double** s = acos(cos(radLat1) \* cos(radLat2) \* cos(radLng1 - radLng2) + sin(radLat1) \* sin(radLat2));
8. s = s \* EARTH\_RADIUS;
9. s = round(s \* 1000 \* 1000000) / 1000000;
10. **return** s;
11. }
13. **double** Data::rad(**double** LatOrLon)
14. {
15. **return** LatOrLon \* M\_PI / 180.0;
16. }

（3）运行测试以及效果截图：

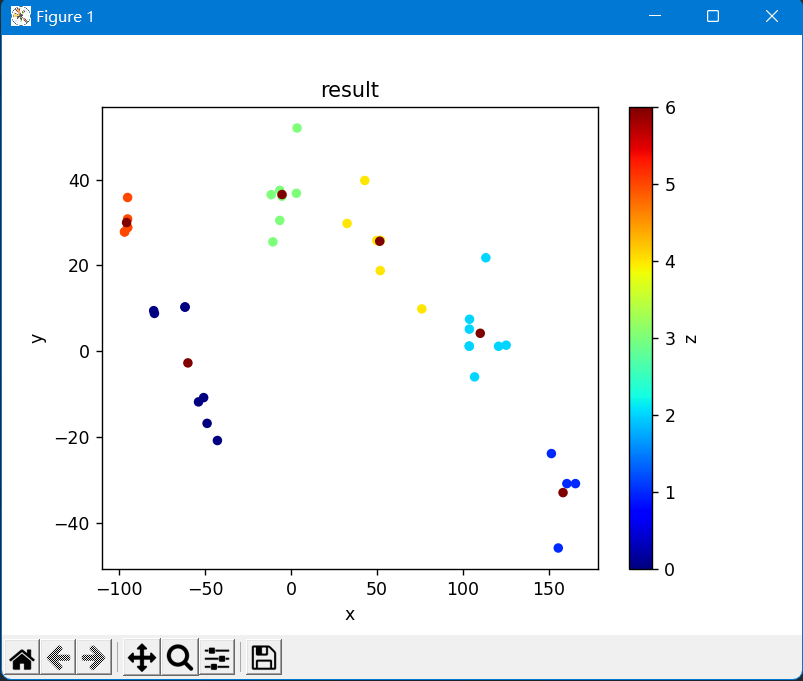
数据源：随机挑选的41条数据



聚类k数：6

聚类结果：（不同颜色代表不同的簇，深红色的点代表该簇的中心位置）

运行时间：0.003s



聚类结果分析：

分析运行结果图可以看出，各个簇之间形状较为规整，没有与其他簇相互交叉，各个簇的成员间距离也较为接近，每个簇的质心均位于所在簇的中心处，由以上分析可得，该算法的聚类效果较好。

四、结果分析

将k值从30~230分别运行kmeans，每次运行都计算该次运行结果的误差平方和SSE，SSE由以下公式给出：其中k为聚类数量，p为每一个样本，mk为第k个聚类的中心点。SSE随着k增大而减少，说明样本聚合程度随着k增大而增高。

当k小于真实聚类数时，由于k的增大会大幅增加每个簇的聚合程度，故SSE的下降幅度会很大，而当k到达真实聚类数时，再增加k所得到的聚合程度回报会迅速变小，所以SSE的下降幅度会骤减，然后随着k值的继续增大而趋于平缓，这个最先趋于平缓的点就是合适的K值。

在该实验中，为了避免由于Kmeans初始选择中心引起的SSE偶然性突变，每个k均运行10次，并求平均值得出该k值对应的SSE。结果如下：

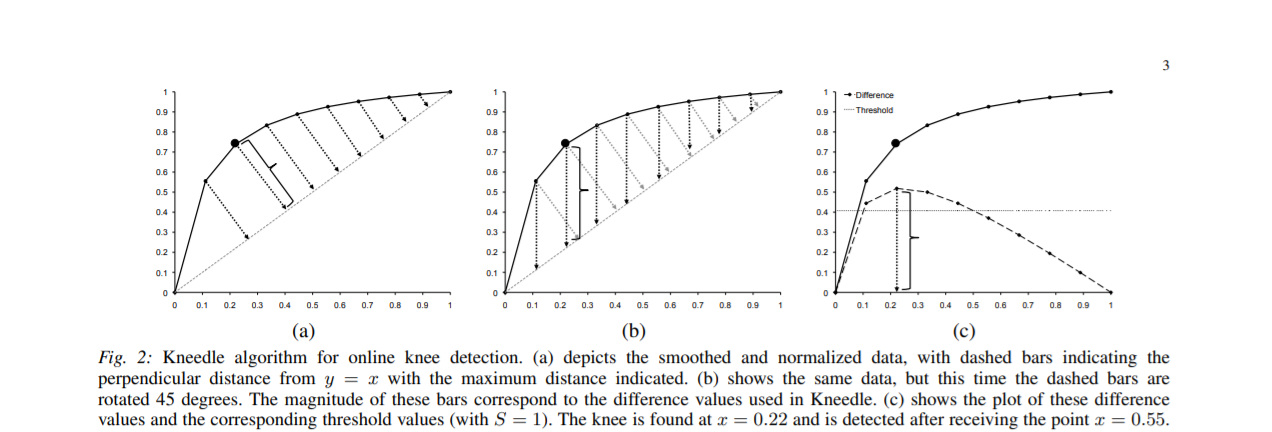
LNG入站点：

得到的拟合图像函数为

LNG出站点：

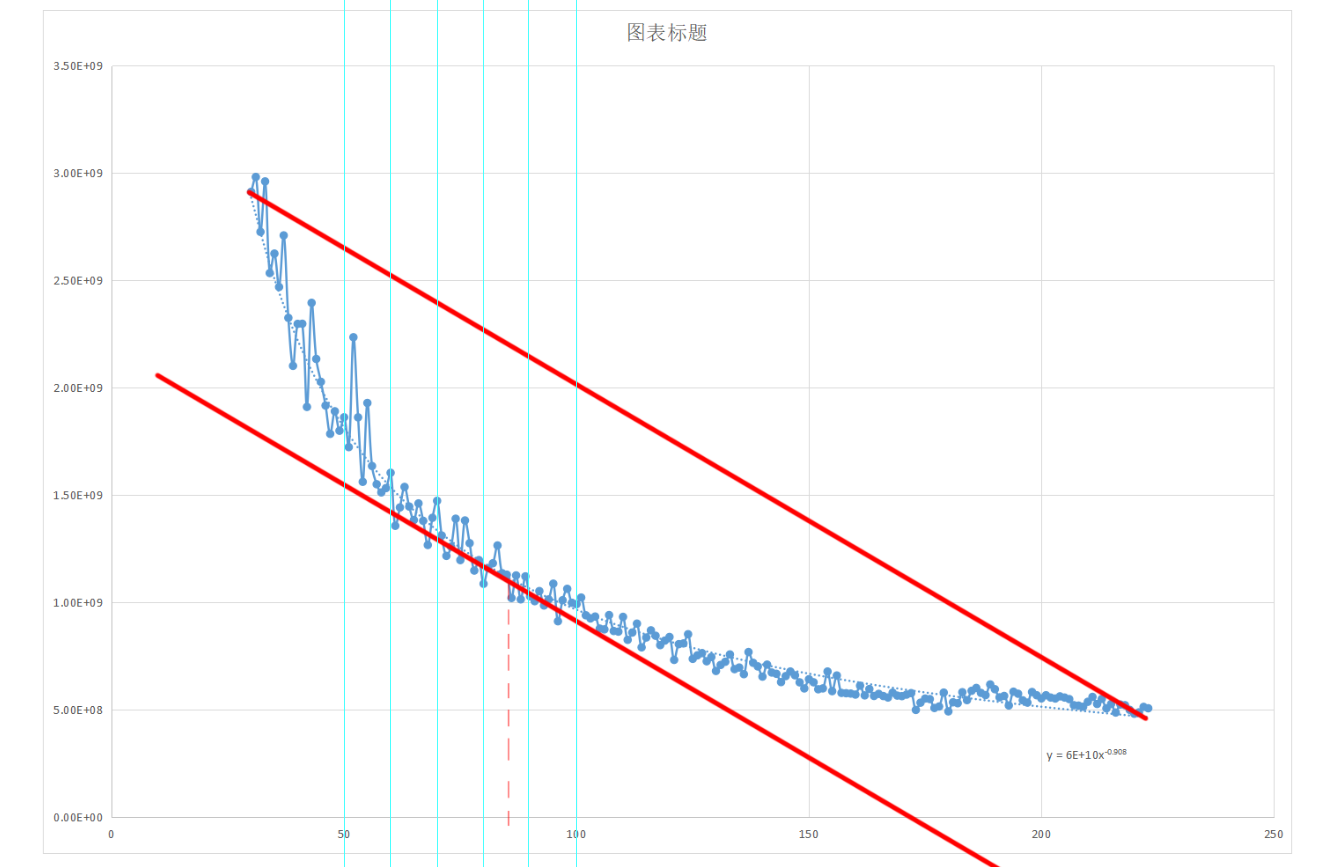
得到的拟合图像函数为

于是这两个图像的肘点--即最大曲率点对应的k值即为对应的最佳k值。《Finding a “Kneedle” in a Haystack: Detecting Knee Points in System Behavior》[[1]](#endnote-1) 中指出了一种快速找出k值的方法：将曲线首尾相连得到一条直线，则与该直线距离最大的点即为肘点的一个近似取值。



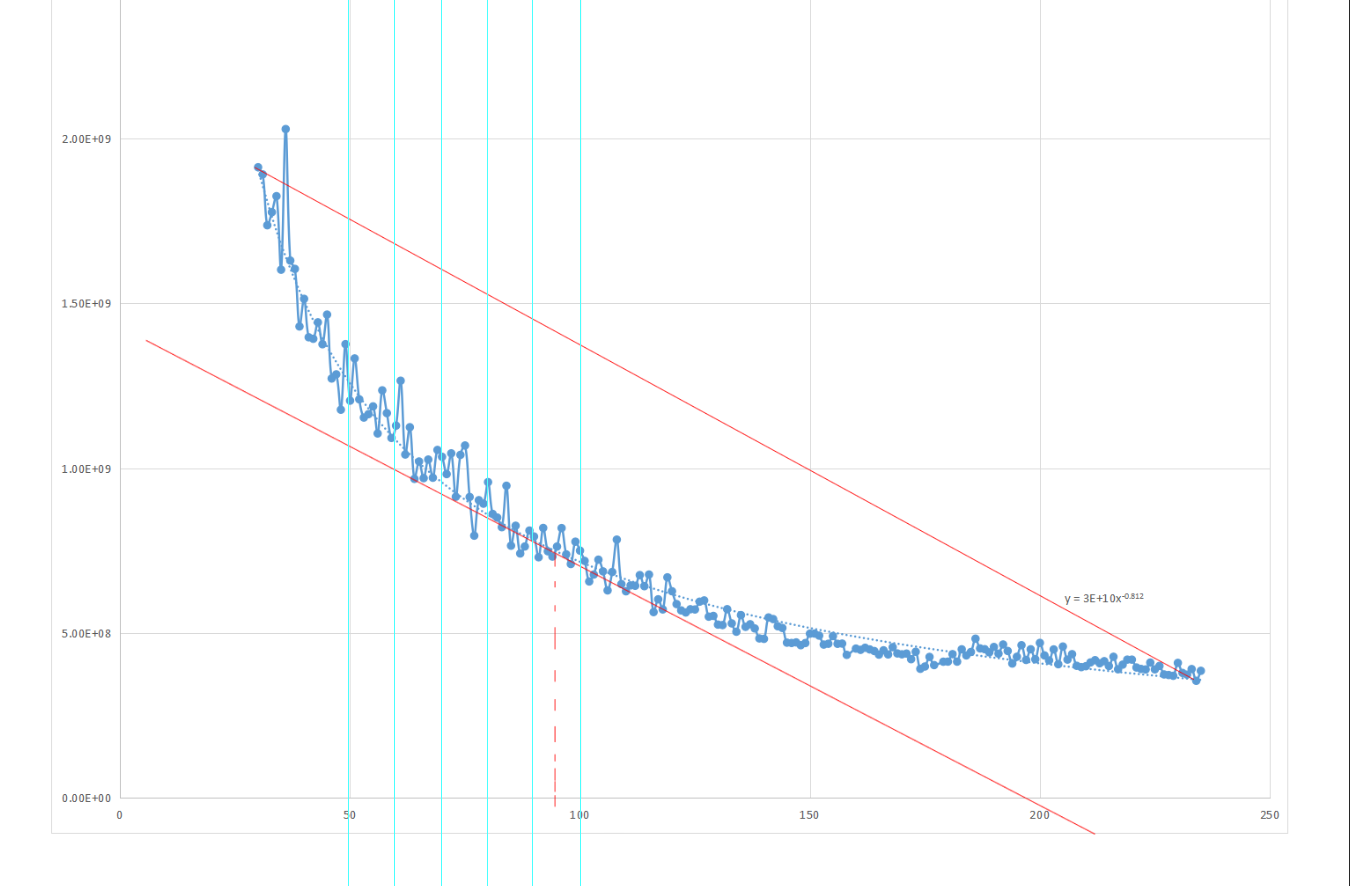
结果如下所示：

LNG入站点：



相交处对应的K值约为85，故在LNG入站点聚类中，选择K值85。

LNG出站点：



相交处对应的K值为95，故在LNG入站点聚类中，选择k值为95。出入站点之和为180，与给出的182个站点相吻合

1. Satopaa V , Albrecht J , D Irwin, et al. Finding a Kneedle in a Haystack: Detecting Knee Points in System Behavior[C]// International Conference on Distributed Computing Systems Workshops. IEEE Computer Society, 2011. [↑](#endnote-ref-1)