OOP_Project_孙铭浩

姓名: 孙铭浩

学号: 3180100729

专业：动物医学

选题：并行K-Means算法实现

小组成员：孙铭浩（个人项目）

1 Kmeans和多线程简介

1.1 K-means

K-means聚类算法是一种聚类分析算法，认为每个类由相似的点组成，而这种相似性由距离来衡量，不同的类间的点应当尽量不相似，每个类都会有一个重心，任意点必然属于某一个类且只属于该类。

K-means作为一种迭代算法，主要的步骤如下：

1. Arbitrarily choose an initial k centers $C = \{c_1, c_2, ..., c_k\}$.
2. For each $i \in \{1, ...k\}$, set the cluster C_i to be the set of points in χ that are closer to c_i than they are to c_j for all $j \neq i$.
3. For each $i \in \{1, ..., k\}$, set c_i to be the center of mass of all points in C_i: $c_i = \frac{1}{|C_i|} \sum_{x \in C_i} x$.
4. Repeat Steps 2 and 3 until C no longer changes or reaching the maximal iteration number.

K-means算法的主要缺陷在于，K值预先给定，而K值的选取难以估计，并且不同的初始条件也可能会导致不同的结果。

1.2 多线程

线程是操作系统能够进行CPU调度的最小单位，可以使用多个线程完成同一个任务，能够加快某些非前后依赖性的程序的运行速度。

在C++多线程并发程序的思路如下：将任务的不同功能交由多个函数分别实现，创建多个线程，每个线程执行一个函数，一个任务就这样同时分由不同线程执行。

通过在C++中引入头文件 `#include <thread>` 管理线程的函数和类在此头文件中的声明，包括std::thread类。例如语句 `std::thread th1(proc1);` 可以创建一个名为 `th1` 的线程，并且该线程开始执行。

在实例化thread对象时，需要传递参数的顺序一次为函数名，函数的第一个参数，函数的第二个参数…，例如在本项目中采用了 `std::thread(&kmeans::n_to_kind, this, i)` 的形式。

若要实现等待调用线程运行结束后线程继续运行，可以采用 `join()` 函数实现进程阻塞，即执行到该位置之后，主函数阻塞，直到调用该 `join()` 的线程运行结束，主函数继续运行。在本项目中采用如下循环实现多个线程之间的同步运行：

```
for (int i = 0; i < k; i++)
        t[i].join();    // 同步点
```

1.3 多线程实现K-means的思路

我们的目标是使用多线程实现K-means算法，主要的思路如下：

`KM_simulation.cpp` 是最主要的程序，这里面并行分为两个部分，kmeans算法的工作流程如下：

1. 首先给出k个点代表k类，n的点代表要分类的数据，重复以下过程：
2. n个点中，对于任何点a，距离k个点中的那个点b近，这个a点就属于b。
3. 根据分类结果重新计算k个点，即，每一类的中心点就是新的类别代表点。

通过对于以上过程的分析，我们可以发现，K-means算法中的2-3步是可以并行运行的，也就是说，对于n个点中的每一个点，可以同时计算这些点离原本k个点的距离，从而计算该n个点属于哪一类。同样地，对于每一个类的计算也可以并行进行。因此，我将采用多线程的方式处理2和3步，处理的方法如下：

问题：n个点分为k类，现在给出k个点的位置。

对于2：可以开k个线程，每个线程计算n个点到k的点中某一个点的距离，并在计算的过程中计算出最小距离。

对于3：先对分好类的点集中起来，方便线程对数据的调用；然后开k个线程处理被分类的点，每一类使用一个线程计算此类点的中心为多少，并更新k。

基于上述分析，我们可以实现多线程计算Kmeans聚类的目标。

2 模板声明以及类的声名

在多个线程访问同一个全局资源的死后，必须确保所有其他线程不在同一个时间访问临界资源，因此我们引入原子操作atomic保证对于临界资源的互斥访问。由于原子操作更加接近底层，因而效率更高。在C++中通过头文件 `#include <atomic>` 引入相关的原子操作头文件。

由于vector无法与atomic联用，因此这里采用一个结构体包裹atomic。

```
typedef vector<double> double_list;
template <typename T> struct atomwrapper {  // vector无法和atomic连用，所以这里使用一个结构体包裹atomic。
  std::atomic<T> _a; // 定义原子操作，执行中不可被中断
  atomwrapper() : _a() {}
  atomwrapper(const std::atomic<T> &a) : _a(a.load()) {}
  atomwrapper(const atomwrapper &other) : _a(other._a.load()) {}
  atomwrapper &operator=(const atomwrapper &other) {
    _a.store(other._a.load());
  }
};
```

对于 `kmeans` 类的实现，首先定义私有变量，n表示生成的n个点，k表示中心， `max_iter` 表示最大的迭代次数，n_kind表示n个点中每个点所属的类别。由于多个线程可能同时访问临界资源（访问同一个内容），因此需要实现互斥操作或者原子操作来实现访问的控制。

```
// kmeans类的私有变量
private:
  int n, k, max_iter;
  atomic<bool> iter_flag;
  double_list x, y, z, kx, ky;
  vector<atomwrapper<int>> n_kind;  // n_kind表示n个点中每个点属于哪一类
                                    // 多个线程可能会访问临界资源这个时候需要1.
                                    // 互斥操作， 2. atomic也可以。
  vector<vector<int>> set_k;        // k类中，每一类有哪些点。
  vector<string> k_color, n_color;  // k类颜色，n个点每个点的颜色
  static double thr;
```

set_k表示k个类别中，每一个类有哪些点，k_color表示k类颜色，n_color表示每一个点的颜色。 `kmeans` 的公共变量包括展示实现结果函数，获取颜色，构造函数，距离计算，点更新函数，颜色更新函数和kmeans迭代函数，将在下面小节进行详细论述。

3 点云显示和着色功能

3.1 展示结果和颜色获取

对于点云模拟程序，我们需要进行图像的可视化操作，以演示kmeans聚类方法。我将展示实验结果功能定义在 `display()` 函数中。对于每一个不同的类别（类别数目用k表示），需要采用不同的颜色进行表示，而具体的颜色采用不同的RGB三种颜色比例进行。

```
public:
  void display(int iter_generation) { // 展示实验结果
    renew_color();            // 首先给每个点上色
    plt::cla();               // 清空画板，否则k点一直存在
    plt::scatter_colored(x, y, n_color);  // 画出n个点
    plt::scatter_colored(kx, ky, k_color, 60);    // 画出k个点
    if (NO_SCREEN) { // 以图片保存结果
      char filename[100];
      sprintf(filename, "Image/kmeans%d.png",iter_generation);
      plt::save(filename); // 保存实验结果文件
    } else
      plt::pause(0.01);     // 展示0.01秒
  }
  string getcolor(double col) {   // 获取颜色. RGB - 256 * 256 * 256
    // 将5000 -- 255 * 255 * 255 - 5000 映射到 0 -- 1之间的小数
    // 其中任何一个数字转换为十六进制，对应某一种颜色
    // int col_ = col / n * (pow(255, 3) - 100000) + 5000;
    int col_ = (double)col * (pow(255, 3) - 100000) + 5000;
    string ans = "#";
    for (int i = 0; i < 6; i++) { // 转换为十六进制
      if (col_ % 16 < 10) {
        ans += (col_ % 16) + '0';
      } else {
        ans += ((col_ % 16) - 10) + 'a';
      }
      col_ /= 16;
    }
    return ans;
  }
```

在 `display()` 函数中，首先调用函数 `renew_color()` 对点进行着色，采用 `matplotlibcpp.h` 库中的 `cla()`，`scatter_colored` 函数来分别清空画板，画出n个点和k个中心等。

```
if (NO_SCREEN) { // 以图片保存结果
      char filename[100];
      sprintf(filename, "Image/kmeans%d.png",iter_generation);
      plt::save(filename); // 保存实验结果文件
    } else
      plt::pause(0.01);     // 展示0.01秒
```

这里根据宏定义 `NO_SCREEN` 的值不同，对于图片结果进行分别处理，若为 `true`，则以图片形式保存结果，进行文件名的建立以及实验结果文件的保存操作。如果为 `false`，则采取直接在窗口动态显示的操作。

对于 `getcolor()` 函数，首先将 [5000, 255 * 255 * 255 - 5000] 映射到 [0, 1]之间的小数，其中任何一个数字转换为十六进制，对应某一种颜色。

转换为十六进制颜色的操作通过如下for循环实现：

```
for (int i = 0; i < 6; i++) { // 转换为十六进制
      if (col_ % 16 < 10) {
        ans += (col_ % 16) + '0';
      } else {
        ans += ((col_ % 16) - 10) + 'a';
      }
      col_ /= 16;
    }
```

3.2 构造函数以及点的输入

对于构造函数 `kmeans()`，采用如下定义：

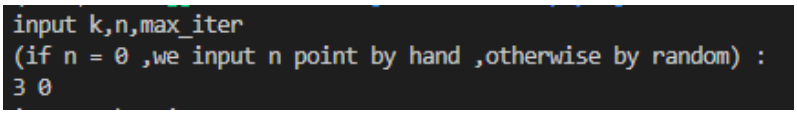
```
kmeans(int k, int max_iter, int n = 0) : k(k), max_iter(max_iter), n(n) {

    //构造函数，用于随机生成n个点或者手动获得一些点。
    double temp_xy[2];
    if (n == 0) {
      printf("input point,Ctrl-D finish input\n");
      while (scanf("%lf%lf", &temp_xy[0], &temp_xy[1]) == 2) {
        x.push_back(temp_xy[0]);
        y.push_back(temp_xy[1]);
        std::atomic<int> a_i(0);
        n_kind.push_back(a_i);
        this->n++;
      }
    } else {
      for (int i = 0; i < n; i++) {
        std::atomic<int> a_i(0);
        n_kind.push_back(a_i);
        x.push_back(((double)1 / (double)n) * (rand() % n));
        y.push_back(((double)1 / (double)n) * (rand() % n));
      }
    }

    // 生成初始的k个点
    set<int> temp_posi;
    int i = 0;
    srand(time(NULL));
    while (i < k) {
      vector<int> temp_set;
      int posi = rand() % this->n;
      if (temp_posi.count(posi) == 1)
```

```
        continue;
      kx.push_back(x[posi]);
      ky.push_back(y[posi]);
      set_k.push_back(temp_set);
      temp_posi.insert(posi);
      k_color.push_back(getcolor(i)); // 将生成的k个点着色
      i++;
    }
    //display(1);
  }
```

构造函数用于一开始随机生成n个点或者采取手动输入的方式输入一些点。



如果提示输入的参数n的值为0时，则采用手动输入的方式进行点的输入，示例的输入格式如下：

```
0.532 0.472
0.634 0.264
0.437 0.211
0.639 0.161
0.359 0.188
0.483 0.312
```

直到进行 `ctrl + D` 操作表示输入结束，开始程序的运行。

由于K-Means算法要求程序最初有k个中心，因此需要在构造函数中生成初始的k个点，并且位置随机，采用 `rand()` 函数归一到平面范围内进行位置的初始化，并且采用 `k_color` 为k个点进行颜色的定义。

4 距离计算和K-Means迭代

4.1 距离度量

通过定义函数 `len` 来计算n个点中第ni个点到k中第ki个点之间的距离。这里采用Euclidean Distance (欧氏距离)的方法进行距离的度量。Euclidean Distance的度量如下：

$\rho = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$, ρ为点(x_2, y_2)到点(x_1, y_1)之间的欧氏距离。

代码如下：

```
double len(int ni, int ki) {

    //用于计算n个点中的第ni个点到k中的ki个点的距离。
```

```
    return sqrt(pow(x[ni] - kx[ki], 2) + pow(y[ni] - ky[ki], 2));
  }
```

根据n个点于各个中心的距离，进行点的分类，将点归为离该点距离近的中心所属的那一类。

```
void n_to_kind(int id) { // 分类，将n个点分为k类
    for (int i = 0; i < n; i++) {
      if (len(i, n_kind[i]._a) > len(i, id)) {
        n_kind[i]._a = id;
      }
    }
  }
```

4.2 点的类别/颜色更新

然后根据n个点分类好的结果重新计算k个中心点的位置：

```
void renew_k(int id) {    // 根据n个点分类好的结果重新计算k个点的位置
    if (set_k[id].size() == 0)
      return;
    double pre_x = kx[id], pre_y = ky[id];
    kx[id] = 0;
    ky[id] = 0;
    for (vector<int>::iterator iter = set_k[id].begin();
         iter != set_k[id].end(); iter++) {
      kx[id] += x[*iter];
      ky[id] += y[*iter];
    }
    kx[id] /= set_k[id].size();
    ky[id] /= set_k[id].size();
    if (sqrt(pow(kx[id] - pre_x, 2) + pow(ky[id] - pre_y, 2)) > this->thr)
      iter_flag = false;
  }
```

并且根据分类结果更新n个点的颜色，便于后面的作图：

```
void renew_color() {
    // 更新n个点的颜色，用于后面的作图。
    n_color.clear();
    for (int i = 0; i < n; i++) {
      n_color.push_back(k_color[n_kind[i]._a]); // n_kind[i]：n个点中的第i个点所属的类别
    }
  }
```

4.3 K-Means迭代

在 `kmeans` 迭代的实现函数中，需要实现：

1. 计算n个点中每个点到k个中心的距离，对n个点进行分类
2. 得到分类之后根据分好类的n个点重新计算k个中心。

这里再对于每一个中心点进行处理的之后，可以采用多线程技术进行并行计算，加快对于每一个类的处理速度。采用 `join()` 函数可以实现点的同步。

```
while (iter_num < max_iter) {
      if (iter_flag)
        break;
      iter_flag = true;
      vector<std::thread> t;
      for (int i = 0; i < k; i++) { // n点分为k类
        set_k[i].clear();
        t.push_back(std::thread(&kmeans::n_to_kind, this, i));  // 开线程处理函数
      }
      for (int i = 0; i < k; i++)
        t[i].join();    // 同步点

      for (int i = 0; i < n; i++) {
        set_k[n_kind[i]._a].push_back(i);
      }
      t.clear();
```

每一轮的迭代过程中自增 `iter_num` 的值，进行迭代次数的计数，并且调用库函数关闭画图界面。

`kmeans` 迭代的函数定义如下：

```
void kmeans_iteration() // kmeans迭代
  {
    /*
     * 1. 计算n个点中每个点到k个点的距离，对n个点进行分类
     * 2. 得到分类之后重新计算k个点。
     */
    int iter_num = 0;
    iter_flag = false;
    while (iter_num < max_iter) {
      if (iter_flag)
        break;
      iter_flag = true;
      vector<std::thread> t;
      for (int i = 0; i < k; i++) { // n点分为k类
        set_k[i].clear();
        t.push_back(std::thread(&kmeans::n_to_kind, this, i));  // 开线程处理函数
      }
      for (int i = 0; i < k; i++)
        t[i].join();    // 同步点

      for (int i = 0; i < n; i++) {
        set_k[n_kind[i]._a].push_back(i);
      }
      t.clear();


      for (int i = 0; i < k; i++) { // 重新计算k
        t.push_back(std::thread(&kmeans::renew_k, this, i));    // 开线程处理函数。
      }
      for (int i = 0; i < k; i++)
        t[i].join();  //同步点
```

```
      display(iter_num);
      iter_num++;
    }
    printf("\niter number is %d\n"
           "finish iter\n", iter_num);
    plt::close();
    exit(0); // zomb
  }
};
```

最后还设定了一个迭代的阈值，用于进行迭代结束的判断。

```
double kmeans::thr = 1e-3;  // 阈值，每个点变化量小于这个值就会停止迭代
```

设置一个常量0.001，如果每次迭代后点的变化量小于该点则会停止迭代过程。

4.4 主函数

主函数中包括输入提示，迭代参数k, n, max_iter的获取，迭代函数的实现和结果的计算等。

```
int main() {
  int n, k, max_iter;
  printf("input k,n,max_iter\n(if n = 0 ,we input n point by hand ,otherwise by random) :\n");
  scanf("%d%d%d", &k, &n, &max_iter);
  kmeans a(k, max_iter, n); // n == 0 从输入中得到点的坐标。
  a.kmeans_iteration();     // 计算结果
  pthread_exit(NULL);
}
```

5 实验结果

5.1 结果运行方式

通过在文件目录下 `make` 产生可执行文件中(具体操作见附录6)，可以运行 `start.sh` 中的脚本进行项目的运行。脚本的内容是首先清除Image文件夹中原有的画图结果，然后将input中的输入重定向到可执行文件 `KM_simulation` 中。

```
#!/bin/bash
rm ./Image/*
./KM_simulation < input
```

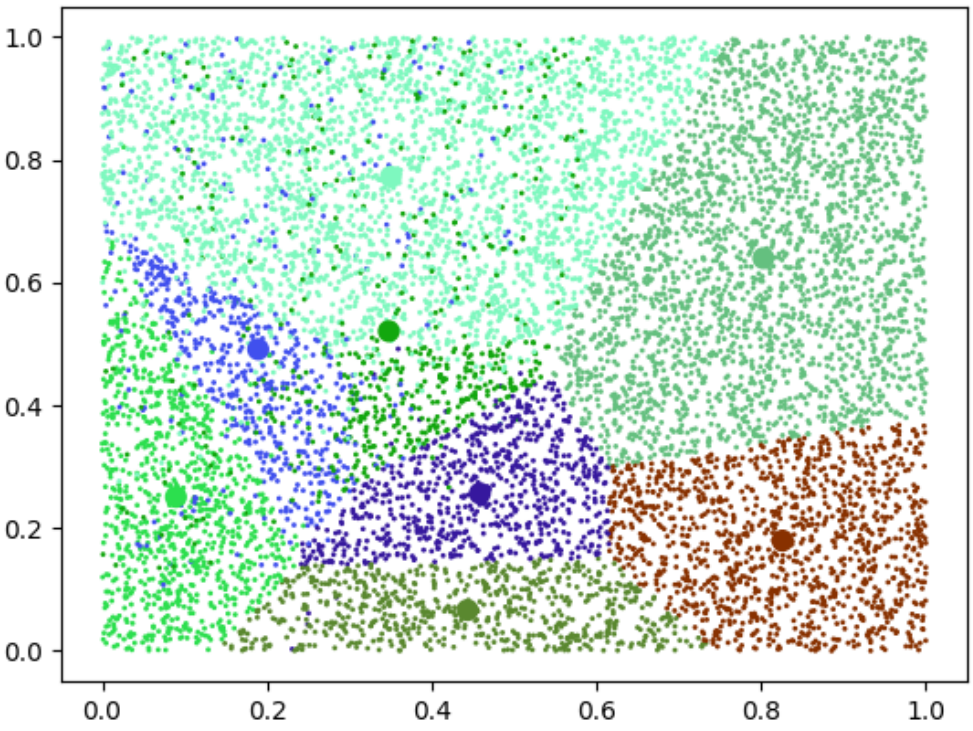
如图所示，在input参数为8 10000 100，即8个中心，10000个点，最大迭代次数为100的设置中，共进行了27次迭代达到收敛。生成的图片在Image文件夹中。

```
g++ KM_simulation.cpp -std=c++11 -g -I./ -I/usr/include/python3.8 -lpython3.8 -lpthread -o KM_simulation
(base) zhuang@Ubuntu211:~/hejindata/smh/oop/project/KMv2.3$ bash start.sh
input k,n,max_iter
(if n = 0 ,we input n point by hand ,otherwise by random) :

iter number is 27
finish iter
(base) zhuang@Ubuntu211:~/hejindata/smh/oop/project/KMv2.3$
```

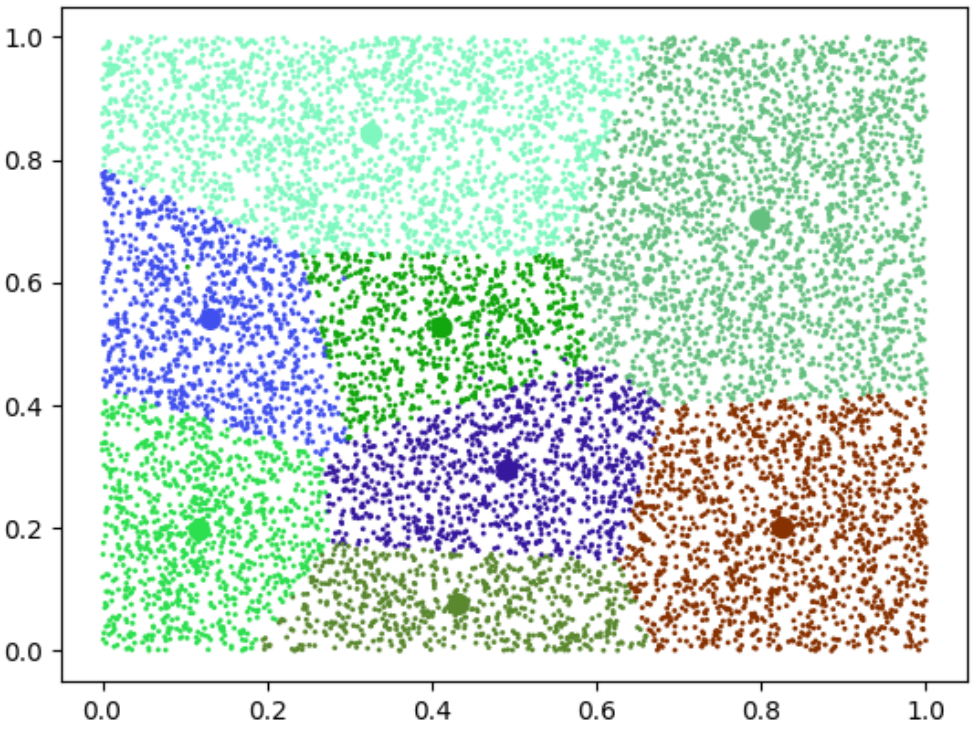
5.2 图片结果分析

采取的参数为8 10000 100，即8个中心，10000个点，最大迭代次数为100的设置中，共产生了27次迭代中的27张图片。我们选取几张典型的图片进行K-Means过程的分析。



kmeans0.png

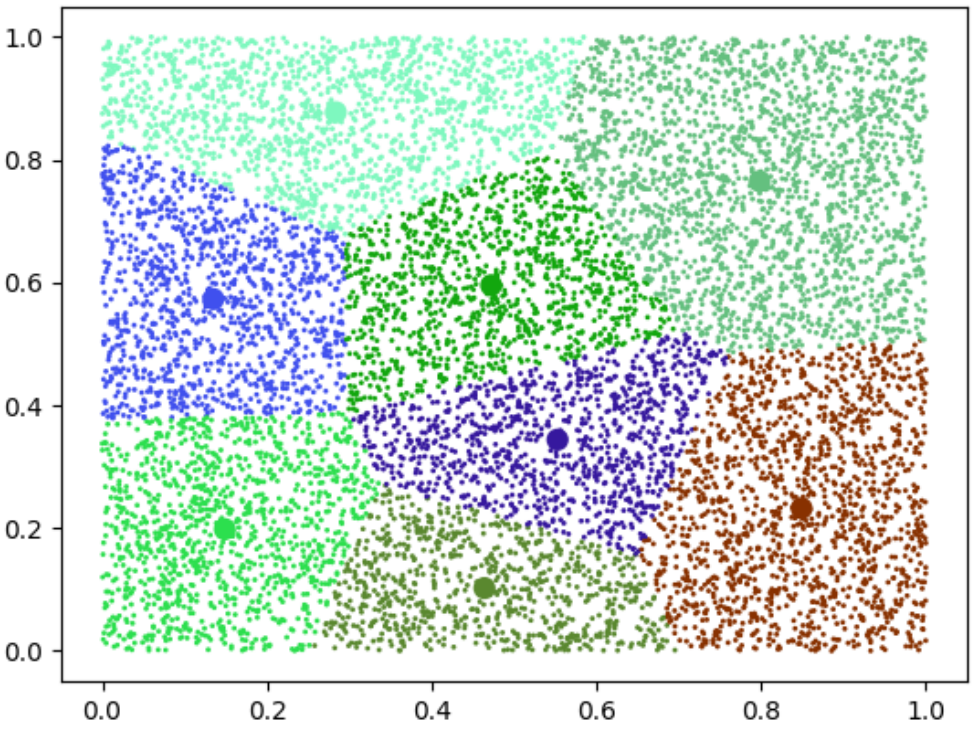
从初始图片中可以看到所有的点共被分为8个类，其中每一个点只能属于一个类。存在一些点散布在不同的类中，例如左上角绿色范围中存在相当数量蓝色的点。



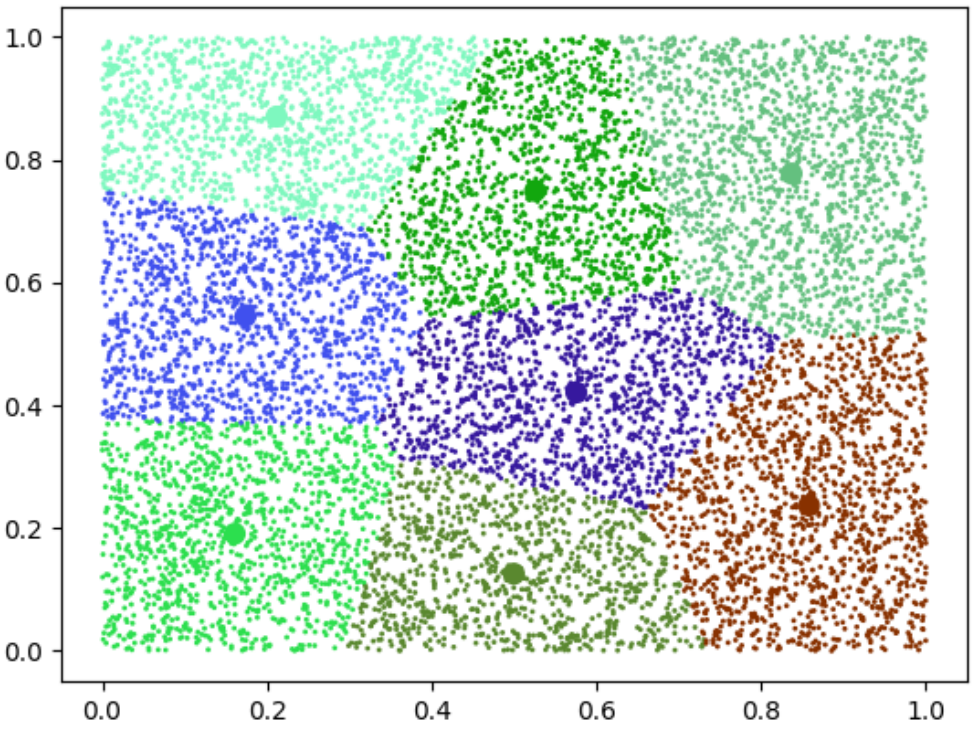
kmeans1.png

经过第一次迭代，已经初步将所有的点归类到8个类中的一个，第一张图片中夹杂在不同类的点已经完全有了分明的界限，并且根据归类好之后的点计算了新的点中心。

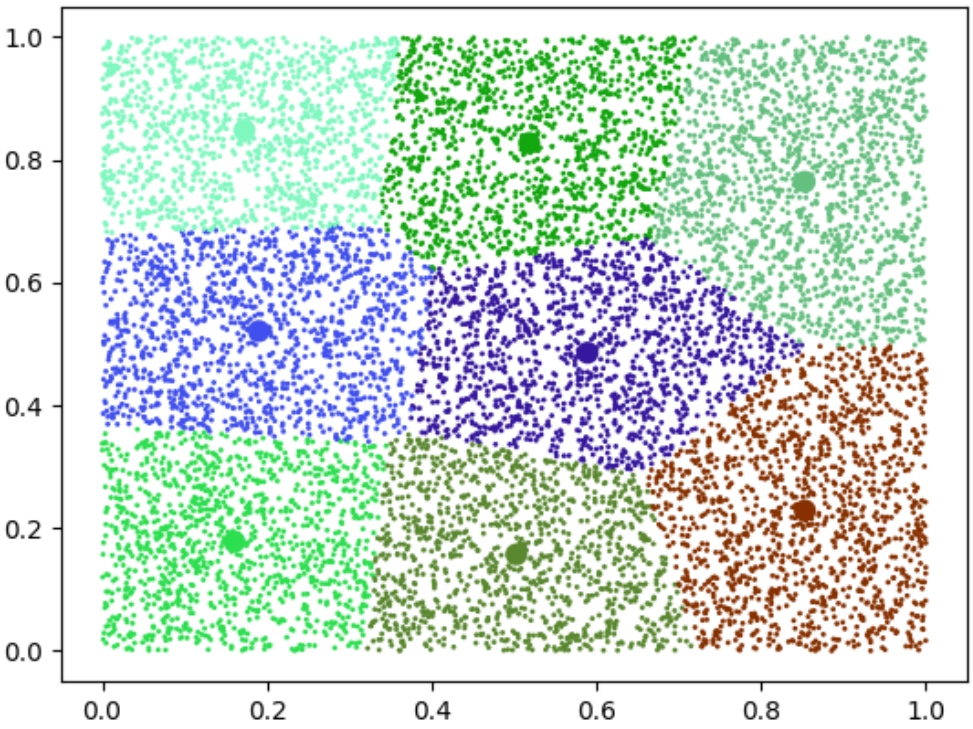
随后继续进行迭代，可以看到每一个类的范围逐渐均匀。



kmeans4.png

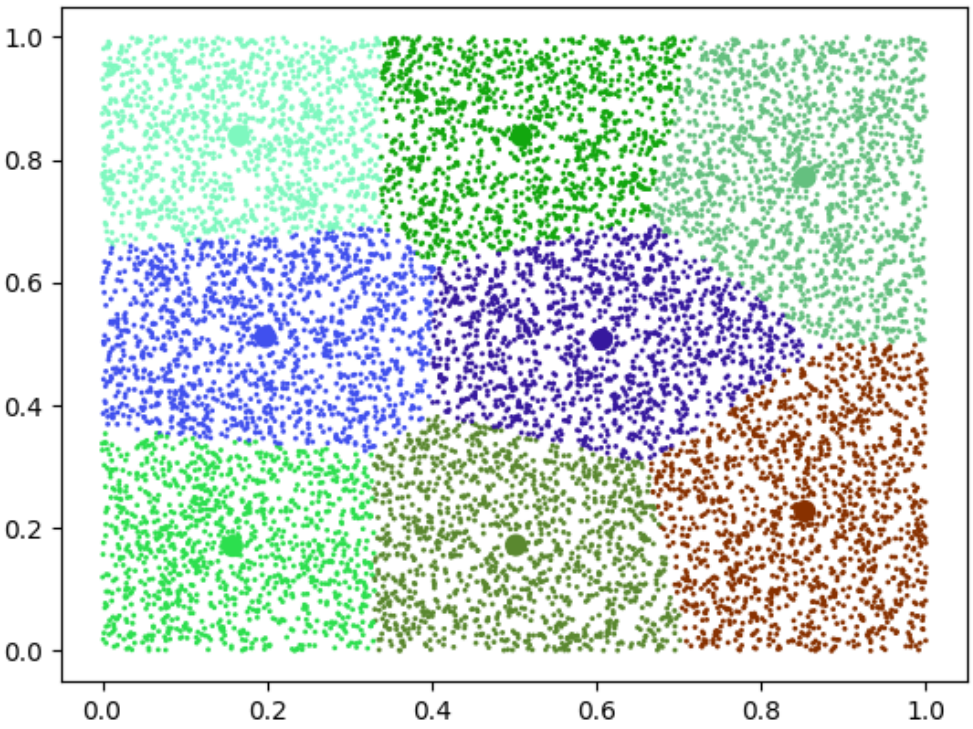


kmeans11.png



kmeans15.png

随后模型在每一次迭代中进行微调，在第16次之后已经基本不再变化，最终达到设定的阈值0.001后，K-Means迭代结束，得到最终的分类结果如 `kmeans26.png` 所示。



kmeans26.png

5.3 聚类结果动态演示

见实验项目文件的KM_RealTime.mp4演示视频文件。

6 附：文件使用说明和注意问题

6.1 文件使用说明

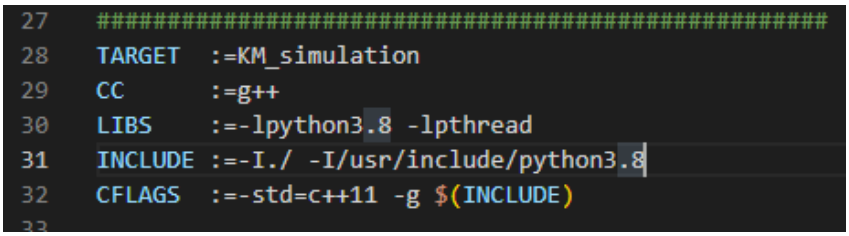
1. 在Linux（我用的是Ubuntu20.04.1服务器）环境中将文件解压。
2. 在/usr/include目录找到python的版本，即 `ls | grep python`.

```
4 x86_64 GNU/Linux
(base) zhuang@Ubuntu211:~/hejindata/smh/oop/project$ ls /usr/include | grep python
python3.8
(base) zhuang@Ubuntu211:~/hejindata/smh/oop/project$
```

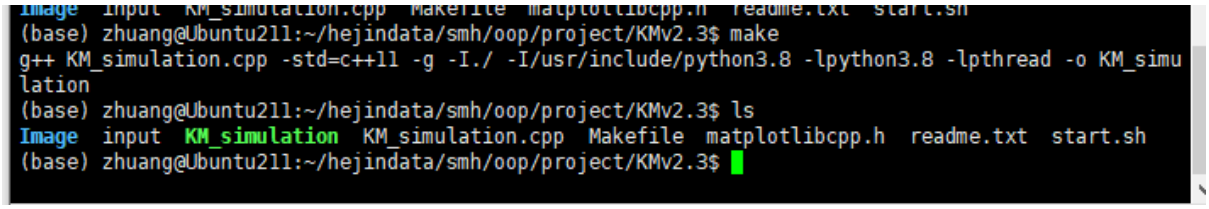
这里我的版本是python3.8。

3. 根据该python版本号修改makefile的内容。其他部分的内容尽量不要改动。修改后保存

```
# Makefile
TARGET  :=KM_simulation
CC      :=g++
LIBS    :=-lpython3.8 -lpthread
INCLUDE :=-I./ -I/usr/include/python3.8
CFLAGS  :=-std=c++11 -g $(INCLUDE)
```



4. 回到makefile文件所在位置，在命令行中输入make对源文件进行编译，生成程序KM_simulation。



4. 采用 `./KM_simulation` 运行程序，按照提示完成输入。或者直接执行 `start.sh` 中的脚本，得到运行结果。通过 `make clean` 可以清除编译的产物。

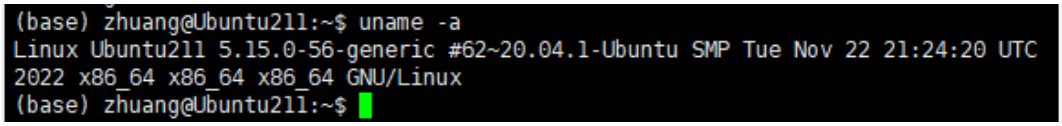
5. `KMv2.3` 中有宏开关NO_SCREEN，其值为false时会实时显示计算结果，为true时会将结果以图片形式保存在Image文件夹中.

6.2 注意问题

在自己的电脑上遇到一些问题：

1. 编译过程中提示<Python.h>找不到的问题（报错信息：**fatal error: Python.h: No such file or directory**）。这里是python没有安装 `python-dev` 的问题。按照这里的第一个答案所示操作即可：https://stackoverflow.com/questions/21530577/fatal-error-python-h-no-such-file-or-directory 注意，对于python3以上的版本要具体到版本编号.

2. 我的运行方式是在Linux服务器环境中运行，版本号如下：



在本地VScode中通过 `Remote-SSH` 插件可以连接到远程服务器，进行代码的debug操作。不过在VScode的terminal中直接运行可能会有一些提示 `matplotlibcpp.h` 相关的warning（在debug的过程中曾出现过），只需要忽略或者到Xshell的那个界面运行即可。VScode窗口编译有时候会报一些奇怪的这些warning。

3. 编译中 `# include <numpy/arrayobject.h>` 报错，是因为使用前需要安装numpy或者版本异常。我的解决方式是 `sudo apt update` 然后重装了numpy后解决。

7 参考资料

C++ Plotting Library: https://github.com/lava/matplotlib-cpp

一个K-means的算法描述及其可能的改进版：Arthur D, Vassilvitskii S. k-means++: The advantages of careful seeding. Stanford; 2006 Jun 7.