Python程序设计-大作业

班级：2022211304

学号：2022211119

姓名：赵宇鹏

1 作业题目

1.1 数据

gpw-v4-population-count-rev11_2020_30_sec_asc.zip是一个全球人口分布数据压缩文件，解压后包括了8个主要的asc后缀文件，他们是全球网格化的人口分布数据文件，这些文件分别是：

- gpw-v4-population-count-rev11_2020_30_sec_1.asc
- gpw-v4-population-count-rev11_2020_30_sec_2.asc
- gpw-v4-population-count-rev11_2020_30_sec_3.asc
- gpw-v4-population-count-rev11_2020_30_sec_4.asc
- gpw-v4-population-count-rev11_2020_30_sec_5.asc
- gpw-v4-population-count-rev11_2020_30_sec_6.asc
- gpw-v4-population-count-rev11_2020_30_sec_7.asc
- gpw-v4-population-count-rev11_2020_30_sec_8.asc

这些文件分布对应地球不同经纬度的范围。

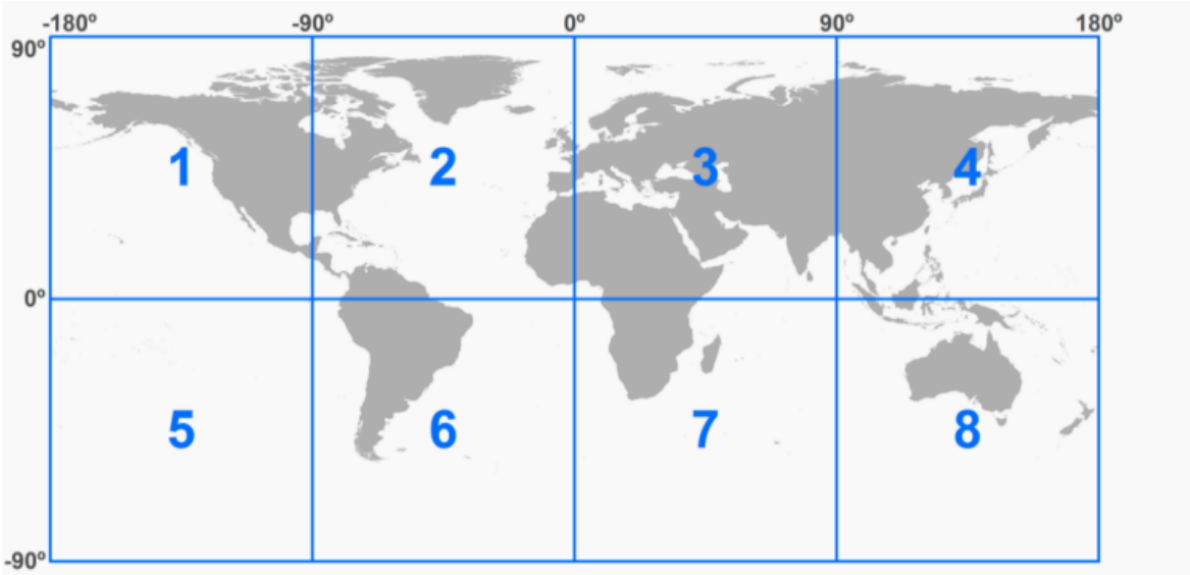


Figure 1. Tiling of 30 arc-second ASCII rasters.

压缩文件下载网页：https://sedac.ciesin.columbia.edu/data/set/gpw-v4-population-count-rev11/data-download

1.2 服务端

压缩文件（gpw-v4-population-count-rev11_2020_30_sec_asc.zip）是一个全球人口分布数据。基于Sanic实现一个查询服务，服务包括：

- 按给定的经纬度范围查询人口总数，查询结果采用JSON格式。
- 不可以采用数据库，只允许使用文件方式存储数据。

- 可以对现有数据进行整理以便加快查询速度，尽量提高查询速度。

查询参数格式 采用GeoJSON（https://geojson.org/）的多边形（每次只需要查询一个多边形范围，只需要支持凸多边形）

```
{
    "type": "Polygon",
    "coordinates": [
        [[30.0, 10.0], [40.0, 40.0], [20.0, 40.0], [10.0, 20.0], [30.0, 10.0]]
    ]
}
```

1.3 客户端

针对上面的查询服务，实现一个服务查询客户端，数据获取后使用Matplotlib散点图（Scatter）进行绘制。

- 横坐标（x轴）为经度。
- 纵坐标（y轴）为维度。

2 服务端代码

首先是数据预处理代码：

```python
import numpy as np
import os

# 输入和输出目录
input_dir = r'C:\Codefield\python\gpw-v4-population-count-rev11_2020_30_sec_asc'
output_dir = r'C:\Codefield\python\processed_data'

os.makedirs(output_dir, exist_ok=True)

# 每度的格网数量
factor = 120

def process_asc_file(file_path):
    """处理单个ASC文件，返回数据及经纬度范围"""
    with open(file_path) as f:
        # 读取头信息
        ncols = int(f.readline().split()[1])
        nrows = int(f.readline().split()[1])
        xllcorner = float(f.readline().split()[1])
        yllcorner = float(f.readline().split()[1])
        cellsize = float(f.readline().split()[1])
        nodata_value = float(f.readline().split()[1])

    # 加载数据并处理无效值
    data = np.genfromtxt(file_path, skip_header=6)
    data[data == nodata_value] = np.nan

    # 计算经纬度范围
    lon_min = xllcorner
```

```
    lon_max = xllcorner + ncols * cellsize
    lat_min = yllcorner
    lat_max = yllcorner + nrows * cellsize

    return data, (lon_min, lon_max, lat_min, lat_max)

# 初始化索引列表
index = []

# 遍历目录下的所有ASC文件
for filename in sorted(os.listdir(input_dir)):
    if filename.endswith('.asc'):
        file_path = os.path.join(input_dir, filename)
        print(f"Processing {filename}...")

        data, (lon_min, lon_max, lat_min, lat_max) =
process_asc_file(file_path)

        # 将数据保存为独立的 NPY 文件
        npy_path = os.path.join(output_dir, f"{os.path.splitext(filename)
[0]}.npy")
        npy_filename = f"{os.path.splitext(filename)[0]}.npy"
        np.save(npy_path, data)
        print(f"Saved to {npy_path}")

        # 记录当前文件的索引信息
        index.append({
            'filename': npy_filename,
            'lon_min': lon_min,
            'lon_max': lon_max,
            'lat_min': lat_min,
            'lat_max': lat_max,
            'shape': data.shape
        })
# 保存索引文件
index_file = os.path.join(output_dir, 'population_index.npy')
np.save(index_file, index)
print(f"All ASC files have been processed and saved as individual NPY files.
Index file saved to {index_file}.")

```

然后是服务端代码

```
import os
import numpy as np
from sanic import Sanic, response
import asyncio
from shapely.geometry import shape, box


app = Sanic("PopulationQueryService")

GRID_DATA_DIR = r"C:\Codefield\python\processed_data"
INDEX_FILE_PATH = os.path.join(GRID_DATA_DIR, "population_index.npy")
CELL_SIZE = 1 / 120  # 每个栅格单元的大小
```

```python

# 加载索引文件
INDEX_DATA = np.load(INDEX_FILE_PATH, allow_pickle=True).tolist()

# 设置请求和响应超时时间
app.config.REQUEST_TIMEOUT = 300
app.config.RESPONSE_TIMEOUT = 300

@app.post("/query_population")
async def query_population(request):
    """
    查询多边形人口总数
    """
    try:
        # 从请求中解析GeoJSON
        geojson = request.json
        polygon = shape(geojson["geometry"])  # 转换为Shapely Polygon

        # 获取多边形边界
        bounds = polygon.bounds  # (min_lon, min_lat, max_lon, max_lat)
        min_lon, min_lat, max_lon, max_lat = bounds

        # 筛选出相关的二进制文件
        relevant_files = filter_files_by_bounds(min_lon, max_lon, min_lat,
max_lat)

        # 计算总人口数
        total_population = 0
        details = []
        tasks = []
        for file_info in relevant_files:
            tasks.append(calculate_population_for_grid(file_info, polygon))

        results = await asyncio.gather(*tasks)
        total_population = sum(result[0] for result in results)
        details = [detail for result in results for detail in result[1]]


        return response.json({
            "total_population": total_population,
            "details": details
        })

    except Exception as e:
        return response.json({"error": str(e)}, status=500)


def filter_files_by_bounds(min_lon, max_lon, min_lat, max_lat):
    """
    根据查询边界筛选相关的二进制文件
    """
    relevant_files = []
    for file_info in INDEX_DATA:
        if (
```

```
            file_info["lon_max"] < min_lon or file_info["lon_min"] > max_lon or
            file_info["lat_max"] < min_lat or file_info["lat_min"] > max_lat
        ):
            continue
        relevant_files.append(file_info)
    return relevant_files


async def calculate_population_for_grid(file_info, polygon):
    """
    计算多边形与某个grid文件相交部分的人口
    """
    try:
        grid_path = os.path.join(GRID_DATA_DIR, file_info["filename"])
        grid_data = np.load(grid_path)
        lon_min = file_info["lon_min"]
        lat_max = file_info["lat_max"]

        # 获取查询多边形的边界范围（仅考虑与当前文件范围重叠部分）
        query_lon_min, query_lat_min, query_lon_max, query_lat_max = polygon.bounds

        # 限制遍历范围：通过比较矩形和文件的边界，找到重合部分
        col_start = max(0, int((query_lon_min - lon_min) / CELL_SIZE))
        col_end = min(grid_data.shape[1], int((query_lon_max - lon_min) / CELL_SIZE) + 1)
        row_start = max(0, int((lat_max - query_lat_max) / CELL_SIZE))
        row_end = min(grid_data.shape[0], int((lat_max - query_lat_min) / CELL_SIZE) + 1)

        # 构建裁剪范围
        cell_population = 0
        cell_details = []
        for i in range(row_start, row_end):
            for j in range(col_start, col_end):
                if np.isnan(grid_data[i, j]):
                    continue

                # 单元格的四个角点经纬度
                cell_polygon = box(
                    lon_min + j * CELL_SIZE, lat_max - (i + 1) * CELL_SIZE,  # 左下角
                    lon_min + (j + 1) * CELL_SIZE, lat_max - i * CELL_SIZE  # 右上角
                )

                # 检查单元格是否与查询多边形相交
                if cell_polygon.intersects(polygon):
                    # 计算相交部分的面积
                    intersection = cell_polygon.intersection(polygon)
                    cell_area = intersection.area
                    population = grid_data[i, j] * (cell_area / cell_polygon.area)
```

```
                    cell_population += population
                    cell_details.append({
                        "lon_min": lon_min + j * CELL_SIZE,
                        "lat_max": lat_max - i * CELL_SIZE,
                        "population": population
                    })

        return cell_population, cell_details
    except Exception as e:
        print(f"处理文件时出错: {e}")
        return 0, []
if __name__ == "__main__":
    app.run(host="0.0.0.0", port=8000, access_log=True)
```

2.1 代码说明

2.1.1 数据预处理

对于项目所给的源数据，每个文件保存的是90°*90°大小的地理范围的人口数据，以`asc`格式储存，数据的前六行说明了文件数据的相关信息：行数，列数，数据文件所储存范围的左下角的点的经纬度，单位栅格的大小以及无效值的具体数值，由于八个数据文件数据量非常大，若不进行处理，则单是在数据加载时就会花费非常多的时间，时间开销很大，所以我在数据预处理中，将`-9999`的值记为无效数据，再分别将这八个数据文件转换成相应的二进制`npy`文件，并利用文件头部的数据信息来建立一个索引表，查询数据时可以通过查询索引表来确定需要载入哪些数据文件，从而大大减少时间开销，经统计，加载一个数据文件的时间由三分钟左右减少到两秒钟左右，数据加载的时间开销已经大大减小。

2.1.2 服务端代码说明

服务端函数说明如下：

`query_population(request)`

- **功能**：处理客户端发送的查询请求，计算多边形内的总人口数及其详细分布。
- 主要逻辑：
  1. 解析客户端发送的 GeoJSON 数据，并将其中的几何数据转换为 Shapely Polygon 对象。
  2. 提取多边形的边界（`bounds`），用于筛选可能包含相关人口数据的文件。
  3. 调用 `filter_files_by_bounds` 筛选相关文件。
  4. 使用异步协程 `asyncio.gather` 调用 `calculate_population_for_grid` 并行处理每个文件的人口数据。
  5. 汇总计算结果，包括多边形内的总人口数和详细分布数据。
- 返回值：
  - JSON 格式的结果：
    - `"total_population"`：多边形内的人口总数。
    - `"details"`：包含每个与多边形相交单元格的人口详情

`filter_files_by_bounds(min_lon, max_lon, min_lat, max_lat)`

- **功能**：根据查询多边形的边界，筛选可能包含相关人口数据的栅格文件。
- 主要逻辑：
  1. 遍历预加载的索引文件（`INDEX_DATA`）。
  2. 对比栅格文件的经纬度范围（`file_info` 中的 `lon_min`, `lon_max`, `lat_min`, `lat_max`）和查询范围。

3. 返回与查询范围重叠的文件列表。

- 输入参数：
  - `min_lon`，`max_lon`：多边形的经度边界。
  - `min_lat`，`max_lat`：多边形的纬度边界。
- 返回值：
  - 索引表中对相应索引的记录。

`calculate_population_for_grid(file_info, polygon)`

- **功能**：计算给定多边形与单个栅格文件的交集部分，并统计人口。
- 主要逻辑：
  1. **加载栅格文件**：通过文件路径加载二进制人口数据。
  2. **计算栅格范围**：提取文件的经纬度边界，获取其对应的行列索引范围。
  3. **遍历栅格单元**：
     - 遍历文件中与查询多边形重叠的栅格区域。
     - 每个栅格单元的边界由 `CELL_SIZE` 计算得到。
     - 使用 Shapely 的 `intersects` 和 `intersection` 方法判断栅格单元与多边形的交集，并计算交集面积。
  4. 加权计算人口：
     - 按照交集面积占栅格单元面积的比例，对栅格内人口进行加权。
  5. 记录人口数据：
     - 累加总人口数。
     - 记录每个单元格的详细人口信息（经纬度范围和人口数）。
- 输入参数：
  - `file_info`：栅格文件的索引信息，包括路径和边界范围。
  - `polygon`：查询的多边形（Shapely Polygon 对象）。
- 返回值：
  - 总人口数。
  - 每个相交单元格的详细人口信息列表。

本程序的查询参数为采用 `GeoJSON` 格式的多边形数据，坐标单位为角秒，服务端接受到客户端发送来的查询请求后，利用 `shapely` 库的内置方法，根据查询参数创建多边形，并求出经纬度的边界值。根据获得的经纬度边界值确定需要查询的人口数据块。对于每个包含多边形的栅格，遍历该栅格中与多边形重合的经纬度，对于其中每个数据单元，计算它和多边形重合的面积，并将其与栅格的面积的比值乘上该栅格的人口数得到多边形在该栅格查询的人口数，并进行相加。

2.1.3 查询优化

首先是基本的查找操作的优化，由于初步判断查找多边形的范围比较困难，所以我采用取多边形的边界的方法，得到多边形最左、右、上、下点的坐标，并通过这四个点画出一个刚好包住多边形的矩形，这样我们在寻找查找范围时的遍历范围就从整个数据文件变成一个比多边形略大的矩形，大大缩小了查找范围；同时，对于跨越多个数据块的多边形，则采用并行查找的策略，各个块的数据计算同步进行，缩短查询时间。

3 客户端代码

客户端代码如下：

```python
import sys
import requests
import json
from PyQt5.QtWidgets import (
    QApplication, QWidget, QVBoxLayout, QPushButton, QLabel,
    QDialog, QLineEdit, QHBoxLayout, QMessageBox, QListWidget
)
from PyQt5.QtCore import Qt
from matplotlib.backends.backend_qt5agg import FigureCanvasQTAgg as FigureCanvas
import matplotlib.pyplot as plt
from shapely.geometry import Polygon
from shapely.geometry.polygon import orient
from matplotlib.colors import LinearSegmentedColormap
from mpl_toolkits.basemap import Basemap

SERVER_URL = "http://127.0.0.1:8000/query_population"

class CoordinateInputDialog(QDialog):
    """
    弹出窗口：用于输入坐标点
    """
    def __init__(self):
        super().__init__()
        self.setWindowTitle("新增坐标点")
        self.setFixedSize(300, 100)
        layout = QVBoxLayout()

        self.coord_input = QLineEdit()
        self.coord_input.setPlaceholderText("请输入经纬度 (例如: -20, 70)")
        layout.addWidget(self.coord_input)

        button_layout = QHBoxLayout()
        self.ok_button = QPushButton("OK")
        self.cancel_button = QPushButton("Cancel")
        button_layout.addWidget(self.ok_button)
        button_layout.addWidget(self.cancel_button)

        layout.addLayout(button_layout)
        self.setLayout(layout)

        self.ok_button.clicked.connect(self.accept)
        self.cancel_button.clicked.connect(self.reject)

    def get_coordinates(self):
        """
        返回用户输入的经纬度
        """
        return self.coord_input.text()

class PopulationQueryApp(QWidget):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("全球人口分布查询系统")
        self.resize(800, 600)
```

```
        # 初始化组件
        self.coord_list = []  # 存储坐标列表
        self.coordinate_display = QListWidget()

        self.new_button = QPushButton("新增坐标点")
        self.delete_button = QPushButton("删除选中点")
        self.query_button = QPushButton("查询")
        self.plot_canvas = FigureCanvas(plt.figure())

        # 布局
        layout = QVBoxLayout()
        layout.addWidget(QLabel("当前坐标列表:"))
        layout.addWidget(self.coordinate_display)
        button_layout = QHBoxLayout()
        button_layout.addWidget(self.new_button)
        button_layout.addWidget(self.delete_button)
        button_layout.addWidget(self.query_button)
        layout.addLayout(button_layout)
        layout.addWidget(self.plot_canvas)

        self.setLayout(layout)

        # 绑定事件
        self.new_button.clicked.connect(self.add_new_coordinate)
        self.delete_button.clicked.connect(self.delete_selected_coordinate)
        self.query_button.clicked.connect(self.perform_query)

        # 绘制初始世界地图
        self.draw_world_map()

    def draw_world_map(self):
        """
        绘制全局世界地图，标注经纬度坐标。
        """
        self.plot_canvas.figure.clear()
        ax = self.plot_canvas.figure.add_subplot(111)

        m = Basemap(projection='mill',
                    llcrnrlon=-180, llcrnrlat=-90,
                    urcrnrlon=180, urcrnrlat=90,
                    resolution='c', ax=ax)

        m.drawcoastlines()
        m.drawcountries()
        m.drawparallels(range(-90, 91, 30), labels=[1, 0, 0, 0])
        m.drawmeridians(range(-180, 181, 60), labels=[0, 0, 0, 1])

        ax.set_title("World Map")
        self.plot_canvas.draw()

    def add_new_coordinate(self):
        """
        新增坐标点
        """
```

```
        dialog = CoordinateInputDialog()
        if dialog.exec_() == QDialog.Accepted:
            coord_text = dialog.get_coordinates()
            try:
                lon, lat = map(float, coord_text.split(","))
                self.coord_list.append((lon, lat))
                self.coordinate_display.addItem(f"经度: {lon}, 纬度: {lat}")
            except ValueError:
                QMessageBox.warning(self, "输入错误", "请输入正确的经纬度格式(例如: -20, 70)")

    def delete_selected_coordinate(self):
        """
        删除选中的坐标点
        """
        selected_items = self.coordinate_display.selectedItems()
        if not selected_items:
            QMessageBox.warning(self, "删除错误", "请先选择一个点进行删除！")
            return

        for item in selected_items:
            index = self.coordinate_display.row(item)
            self.coord_list.pop(index)
            self.coordinate_display.takeItem(index)

    def perform_query(self):
        """
        执行查询：检查坐标是否构成多边形，并查询人口数据
        """
        if len(self.coord_list) < 3:
            QMessageBox.warning(self, "错误", "坐标点不足，无法构成多边形！")
            return

        try:
            polygon = Polygon(self.coord_list)
            if not polygon.is_valid or not polygon.is_simple:
                raise ValueError("多边形无效或自相交！")

            # 确保多边形闭合
            if list(polygon.exterior.coords)[-1] != list(polygon.exterior.coords)[0]:
                coords = list(polygon.exterior.coords)[:-1]
            else:
                coords = list(polygon.exterior.coords)

            # 转换为GeoJSON格式
            geojson = {
                "type": "Feature",
                "geometry": {
                    "type": "Polygon",
                    "coordinates": [coords],
                },
                "properties": {}
            }
```

```
            # 发送查询请求
            response = requests.post(SERVER_URL, json=geojson)
            if response.status_code != 200:
                print(f"响应内容: {response.text}")
                raise Exception("查询失败，服务器错误！")
            result = response.json()
            self.display_results(result)
        except Exception as e:
            QMessageBox.critical(self, "查询错误", str(e))

    def display_results(self, data):
        """
        显示查询结果，并绘制散点图
        """
        total_population = data.get("total_population", 0)
        details = data.get("details", [])

        QMessageBox.information(self, "查询成功", f"总人口:
{total_population}")

        # 创建自定义颜色梯度：白 -> 黄 -> 红 -> 黑
        colors = [(1, 1, 1), (1, 1, 0), (1, 0, 0), (0, 0, 0)]  # RGB
        n_bins = 100  # 梯度分隔数量
        cmap_name = "custom_colormap"
        custom_cmap = LinearSegmentedColormap.from_list(cmap_name, colors,
N=n_bins)

        # 绘制地图和散点图
        self.plot_canvas.figure.clear()
        ax = self.plot_canvas.figure.add_subplot(111)

        # 设置 Basemap
        m = Basemap(projection='merc',
                    llcrnrlon=min(lon for lon, lat in self.coord_list) - 1,
 # 左下角经度
                    llcrnrlat=min(lat for lon, lat in self.coord_list) - 1,
 # 左下角纬度
                    urcrnrlon=max(lon for lon, lat in self.coord_list) + 1,
 # 右上角经度
                    urcrnrlat=max(lat for lon, lat in self.coord_list) + 1,
 # 右上角纬度
                    resolution='i', ax=ax)

        m.drawcoastlines()
        m.drawcountries()
        m.drawparallels(range(-90, 91, 10), labels=[1, 0, 0, 0])
        m.drawmeridians(range(-180, 181, 10), labels=[0, 0, 0, 1])

        # 绘制蓝色框框表示查询范围
        lons, lats = zip(*self.coord_list)
        lons = list(lons) + [lons[0]]
        lats = list(lats) + [lats[0]]
        x, y = m(lons, lats)
        m.plot(x, y, marker=None, color='blue', linewidth=2, label="查询范
围")
```

```
211
212         if details:
213             scatter_lons = [item["lon_min"] for item in details]
214             scatter_lats = [item["lat_max"] for item in details]
215             populations = [item["population"] for item in details]
216 
217             norm_populations = [p ** 0.5 for p in populations]  # 使用平方根拉伸梯度
218             scatter_x, scatter_y = m(scatter_lons, scatter_lats)
219 
220             scatter = ax.scatter(
221                 scatter_x, scatter_y, c=norm_populations, cmap=custom_cmap, s=50, edgecolor='none'
222             )
223             plt.colorbar(scatter, ax=ax, label='Population (sqrt-scaled)')
224 
225         ax.set_title(f"Total population: {total_population}")
226         self.plot_canvas.draw()
227 
228 if __name__ == "__main__":
229     app = QApplication(sys.argv)
230     main_window = PopulationQueryApp()
231     main_window.show()
232     sys.exit(app.exec_())
233 
```

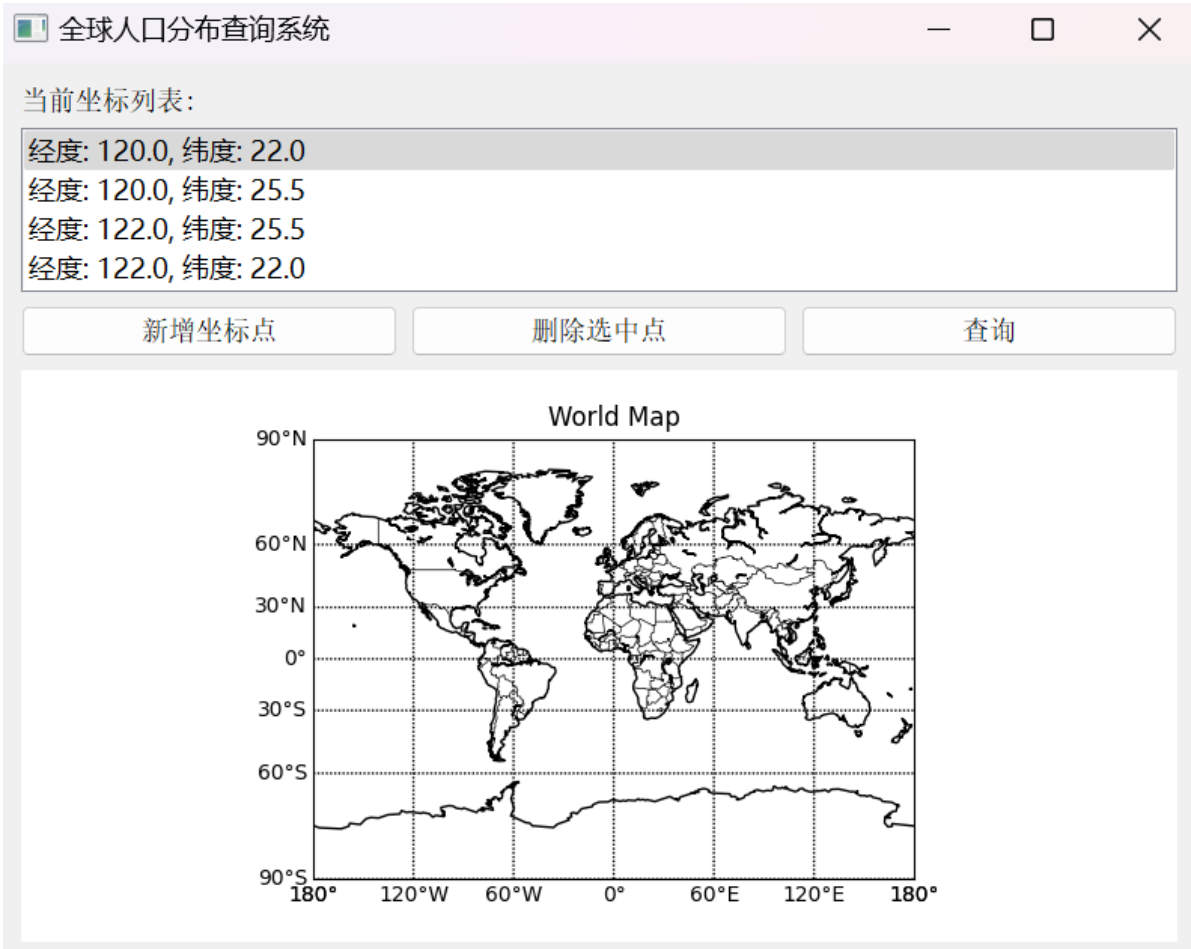
3.1 代码说明

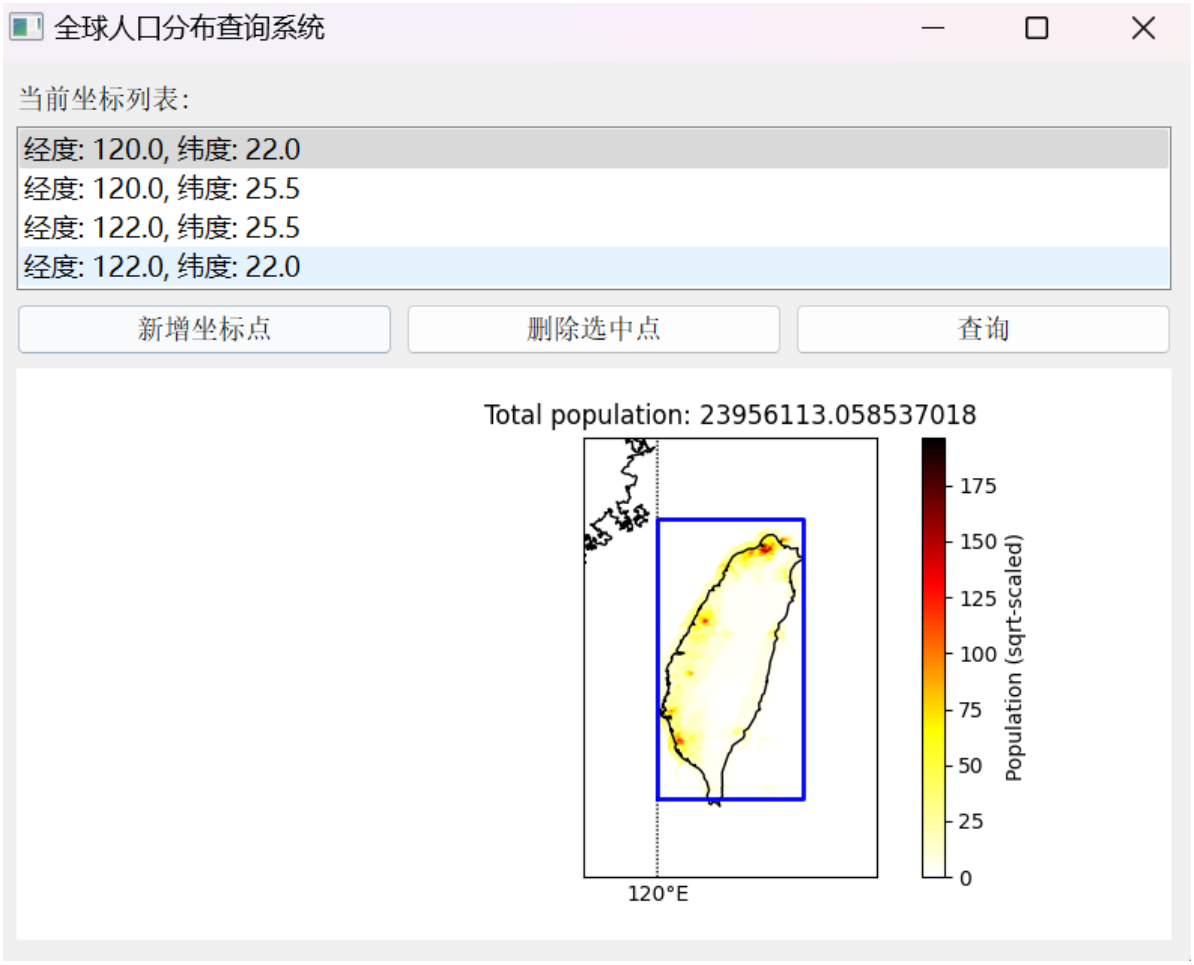
客户端我采用了PyQt5来编写GUI页面，具体实现效果如下：

首先使用`PyQt`以及`basemap`库绘制一个世界地图供用户参考，然后用户需要按顺时针顺序输入需要查询的范围，客户端程序会对输入的范围进行检测，若不是一个正常的凸多边形则会抛出错误报错，若是则会进行查询，查询后服务端的返回值格式如下：

```
{
    "total_population": 42050.22151324356,
    "details": [
        {
            "lon_min": -18.03333333333333,
            "lat_max": 66.56666666666626,
            "population": 9.175664
        },
        {
            "lon_min": -18.025000000000006,
            "lat_max": 66.56666666666626,
            "population": 32.48943
        },
        {
            "lon_min": -18.016666666666666,
            "lat_max": 66.56666666666626,
            "population": 18.73857
        },
        ...
}
```

之后客户端会根据返回的`details`信息来使用`matplotlib`库来绘制查询结果，即人口分布图，以台湾省大致范围为例，结果如下：





可以看到查询结果比较清晰地被绘制出来，同时查找的范围也被用蓝色线条绘制出来，若需要再次查找或对查找范围进行修改，只需要删除原有坐标点即可。