中南大學

CENTRAL SOUTH UNIVERSITY

高级程序设计

课程设计

Course Design of Advanced Programming

题　　目：传染病疫情模拟与预测系统

学生姓名：

指导教师：

学　　院：自动化学院

专业班级：

完成时间：2026 年 1 月 25 日

摘　要

传染病的爆发对全球公共卫生安全构成了严峻挑战，利用计算机技术进行疫情数据的管理与趋势预测已成为辅助决策的重要手段。本报告详细阐述了一个基于SIR（Susceptible-Infected-Removed）动力学模型的传染病疫情模拟与预测系统的设计与实现过程。该系统采用C++面向对象编程语言开发，利用STL标准模板库进行高效的数据结构管理，并结合ImGui图形库构建了现代化的用户交互界面，摒弃了传统的控制台操作模式。

系统主要实现了三大核心功能：一是地区疫情数据的动态管理，支持对确诊、治愈、死亡人数的增删改查；二是历史数据的可视化回放，通过图表直观展示疫情发展轨迹；三是基于微分方程的未来趋势预测，系统能够根据输入的历史数据自动估算传染率（β）和恢复率（γ），并利用欧拉法（Euler Method）数值求解SIR微分方程组，从而仿真未来的疫情走向。

报告内容涵盖了全生命周期的软件开发过程，包括深度的需求分析、基于MVC模式的概要设计、详细的类与算法设计（如反推参数算法）、程序实现的难点解析（如ImGui的立即模式渲染机制）、以及完善的系统测试。通过本项目的开发，不仅实现了一个具有实用价值的疫情模拟工具，更深入验证了面向对象编程思想在解决复杂数学建模问题中的有效性与灵活性。

关键词：C++；面向对象程序设计；SIR模型；数值模拟；ImGui；数据可视化

目录

第一章　需求分析

1.1　项目背景与意义

1.1.1　背景

自2019年底新冠疫情（COVID-19）爆发以来，病毒的高传染性与变异性给全球各国的医疗系统和社会治理带来了巨大压力。在抗击疫情的过程中，数据的产生速度极快且维度众多（如不同地区、不同时间、不同人群分类），单纯依靠人工统计或静态表格（如Excel）难以直观地反映疫情的动态变化规律。此外，决策者往往需要根据当前的疫情数据，推演未来的感染规模，以便提前调配医疗资源。因此，开发一款能够实时管理疫情数据并具备科学预测功能的计算机系统显得尤为迫切。

1.1.2　目的与意义

本项目旨在综合运用C++高级程序设计课程中所学的知识，开发一个“传染病疫情模拟与预测系统”。

- **理论意义**：通过将数学模型（SIR微分方程）转化为计算机算法，深入理解计算机仿真技术在公共卫生领域的应用，同时巩固面向对象编程（OOP）中的封装、继承、多态等核心概念。
- **实践意义**：提供一个直观的工具，帮助用户（如疾控中心工作人员或普通研究者）快速录入数据、观察历史趋势，并对不同管控措施下（体现为不同的传染率参数）的疫情走向进行仿真推演，辅助科学决策。

1.2　功能需求分析

根据用户的使用场景，系统需具备以下核心功能模块：

1.2.1　地区数据管理

系统应能够管理多个不同地区（如城市或省份）的独立数据。

- **添加地区**：用户可输入地区名称、初始总人口数、初始确诊/治愈/死亡人数，系统应自动初始化该地区的模拟器。
- **删除地区**：对于错误录入或不再关注的地区，用户应能将其从系统中移除。
- **数据更新**：支持对现有地区的数据进行修改（如每日更新最新的确诊数字）。
- **列表展示**：以表格形式展示所有管理地区的实时概况，并用颜色（红/黄/绿）标识风险等级。

1.2.2 数据可视化

为了提供直观的信息展示，系统需集成图表绘制功能。

- **总览柱状图**：在首页展示所有地区的确诊人数对比，方便横向比较。
- **历史趋势图**：针对特定地区，绘制随时间变化的S、I、R三条曲线，清晰展示疫情的爆发、高峰与衰退阶段。

1.2.3 疫情模拟与预测

这是本系统的核心的高级功能。

- **参数自动拟合**：系统应能根据用户录入的历史真实数据，自动反推计算出该地区的平均传染率（Beta）和平均恢复率（Gamma），减少人工调参的难度。
- **未来仿真**：基于SIR模型的微分方程，设定预测天数（如未来30天），计算并绘制预测曲线。用户应能动态调整参数（如模拟“封城”导致Beta降低），观察曲线的变化。

1.3 非功能需求

1. **易用性**：界面应简洁明了，采用图形化与菜单式交互，降低学习成本。
2. **性能**：对于包含数百个数据点的模拟计算，应在毫秒级完成，保证界面流畅不卡顿。
3. **健壮性**：对于用户的非法输入（如负数人口、空名称），系统应有校验机制，不会直接崩溃或产生逻辑错误。
4. **扩展性**：代码结构应清晰（采用MVC模式），便于未来扩展更复杂的模型（如SEIR）。

第二章　概要设计

2.1　系统总体架构

本系统采用分层架构设计，严格遵循“高内聚、低耦合”的软件工程原则。为了实现数据逻辑与界面显示的解耦，系统被设计为经典的 **MVC（Model-View-Controller）**架构。这种架构不仅提高了代码的可维护性，还使得后续增加新的可视化视图（View）变得更加容易，而无需修改底层的数据逻辑（Model）。

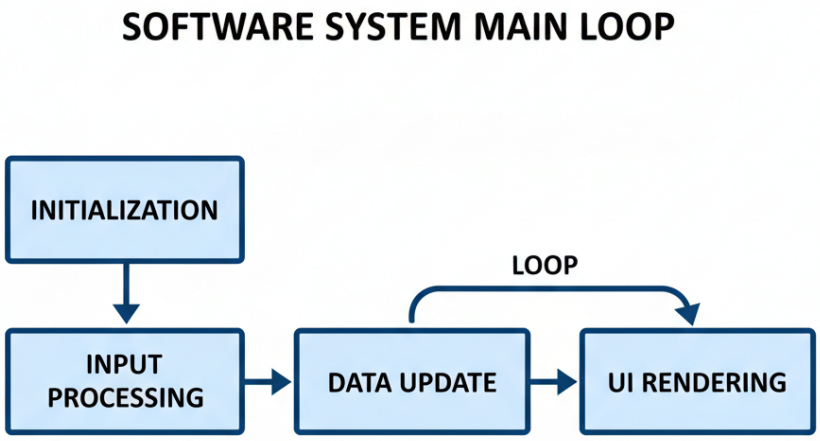

图 2.1: 系统MVC架构设计图

2.1.1 模块划分与职责

1. **Model 层（数据模型）：**
   - 职责：负责所有业务数据的存储、状态管理和核心算法计算。
   - 核心类：`EpidemicData`（全局数据仓库）、`Region`（地区实体）、`SIRModel`（数学模拟器）。
   - 特点：不仅包含静态数据（人口、确诊数），还封装了动态行为（如`run_single_step`模拟步进）。
2. **View 层（视图界面）：**
   - 职责：负责将Model层的数据以图形化的方式呈现给用户。
   - 核心组件：仪表盘页面（Dashboard）、数据管理表格、预测曲线绘图窗口。
   - 技术实现：基于ImGui的立即模式渲染，每一帧重新绘制UI元素。
3. **Controller 层（逻辑控制）：**
   - 职责：监听用户的输入事件（点击、拖拽、键盘输入），并调用Model层的方法更新状态，或切换View层的显示页面。
   - 实现：位于 ‘main.cpp‘ 的主循环中，通过 ‘AppState‘ 状态机控制页面路由。

2.2 系统工作流程

系统采用实时渲染循环（Real-time Rendering Loop）机制，这与普通的游戏引擎机制类似。

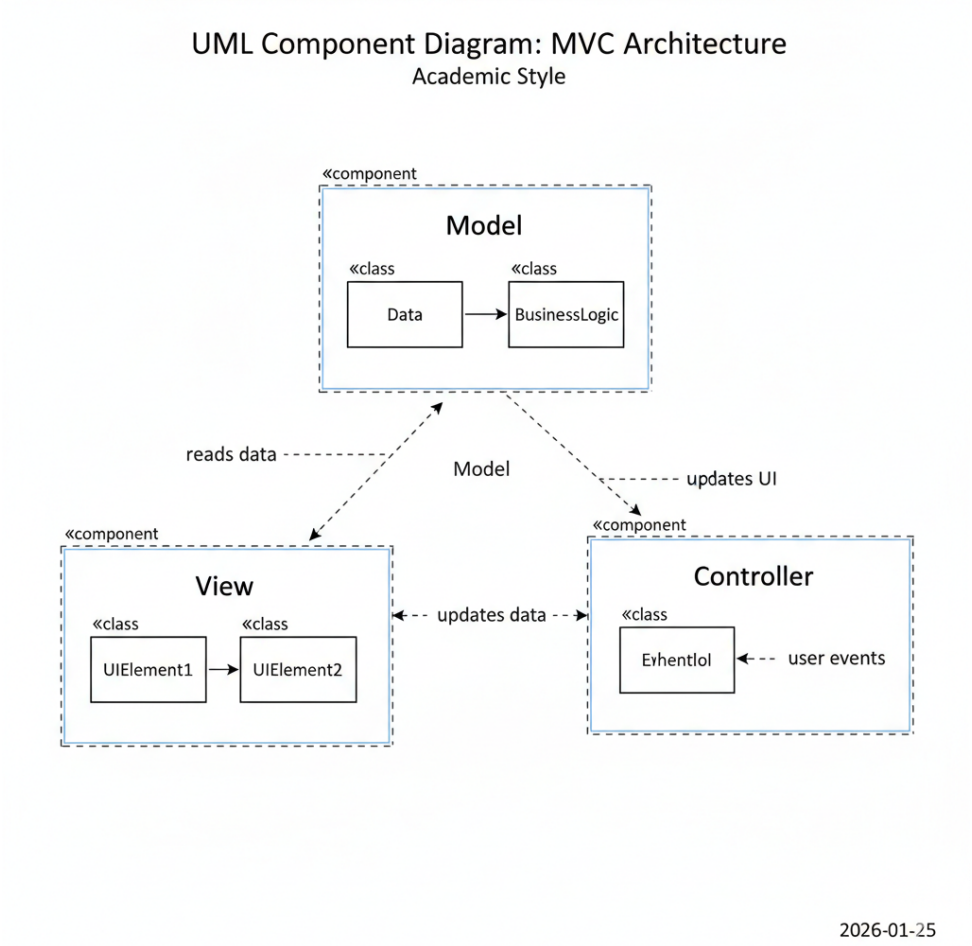

图 2.2: 系统主循环流程示意图

2.2.1 主要业务流程

1. **启动阶段**：程序初始化SDL/OpenGL上下文，加载字体资源（支持中文），并实例化全局 ‘g_EpidemicData‘ 对象。
2. **交互阶段**：
   - 用户点击“添加城市”，弹出模态对话框。
   - 用户输入数据并确认，Controller调用 ‘EpidemicData::addRegion‘。
   - Model更新内存中的 ‘vector¡Region¿‘ 列表。
   - 下一帧渲染时，View层读取最新的列表并显示在表格中。
3. **模拟阶段**：
   - 用户在预测页面调整滑动条（Slider）改变Beta值。
   - View层实时将新参数传递给 ‘SIRModel‘。
   - ‘SIRModel‘ 重新计算未来30天的轨迹，View层实时刷新曲线图。

第三章　程序设计

本章将详细介绍核心类的设计与实现，以及传染病动力学模型的数学推导与数值解算法。

3.1　核心类结构设计

系统采用面向对象的方法对现实世界进行建模。主要的类及其关系如下：

3.1.1　全局数据管理器

这是一个管理类，在整个应用程序生命周期中通常只存在一个实例（类似于单例模式的使用）。

- **成员变量：** `std::vector<Region>regions`。使用STL向量容器动态存储所有地区对象，支持运行时的动态扩容。
- **核心方法：**
  - `addRegion(...)`: 负责对象的创建与初始化，并将其推入向量末尾。
  - `calculateRiskLevel(...)`: 一个静态工具函数，根据活跃病例数占总人口的比例，返回 `RiskLevel` 枚举值（High/Medium/Low），实现了业务逻辑的封装。

3.1.2　地区实体类

`Region` 结构体代表了一个独立的地理单元。

- **数据成员：** 包含基础信息（名称、人口）和当前疫情状态（确诊、治愈、死亡）。
- **组合关系：** 每个 `Region` 对象内部包含一个 `SIRModel` 对象（`simulation`）。体现了“组合优于继承”的设计原则，使得地区数据与模拟逻辑绑定。
- **历史数据：** `std::vector<HistoricalRecord>history` 存储了该地区过去多天的真实数据，用于后续的参数拟合。

3.1.3 传染病模型类

这是系统的数学核心。

- **封装性**：将 S, I, R 的状态值以及 β, γ 参数私有化，仅通过Getter/Setter暴露接口，保证了数据的安全性。
- **模拟能力**：提供了 `run(int days)` 接口，能够基于当前状态向后推演任意天数。

3.2 SIR 模型数学原理与实现

系统基于经典的SIR隔室模型。该模型假设总人口 N 是不变的，且人群分为三类：

- **S (Susceptible)**: 易感者，未被感染但缺乏免疫力的人群。
- **I (Infected)**: 感染者，已确诊且具有传染能力的人群。
- **R (Removed)**: 移出者，通过治愈获得免疫或因病死亡的人群。

3.2.1 微分方程模型

系统的核心动力学由以下一组常微分方程（ODEs）描述：

$$\frac{dS}{dt} = -\frac{\beta SI}{N} \tag{3.1}$$

$$\frac{dI}{dt} = \frac{\beta SI}{N} - \gamma I \tag{3.2}$$

$$\frac{dR}{dt} = \gamma I \tag{3.3}$$

其中：

- β (Beta)：有效接触率，代表病毒的传染能力。
- γ (Gamma)：恢复率，代表医疗治愈能力，通常 $\gamma \approx 1/D$，其中D是平均感染周期。

3.2.2 数值解算法 (欧拉法)

由于计算机无法直接处理连续的微分方程，我们需要对其进行离散化。本项目采用显式欧拉法（Forward Euler Method），设定时间步长 $\Delta t = 1$（天）。

离散化后的递推公式如下，用于代码中的 `run_single_step` 函数：

$$S_{t+1} = S_t - (\frac{\beta S_t I_t}{N}) \tag{3.4}$$

$$I_{t+1} = I_t + (\frac{\beta S_t I_t}{N}) - (\gamma I_t) \tag{3.5}$$

$$R_{t+1} = R_t + (\gamma I_t) \tag{3.6}$$

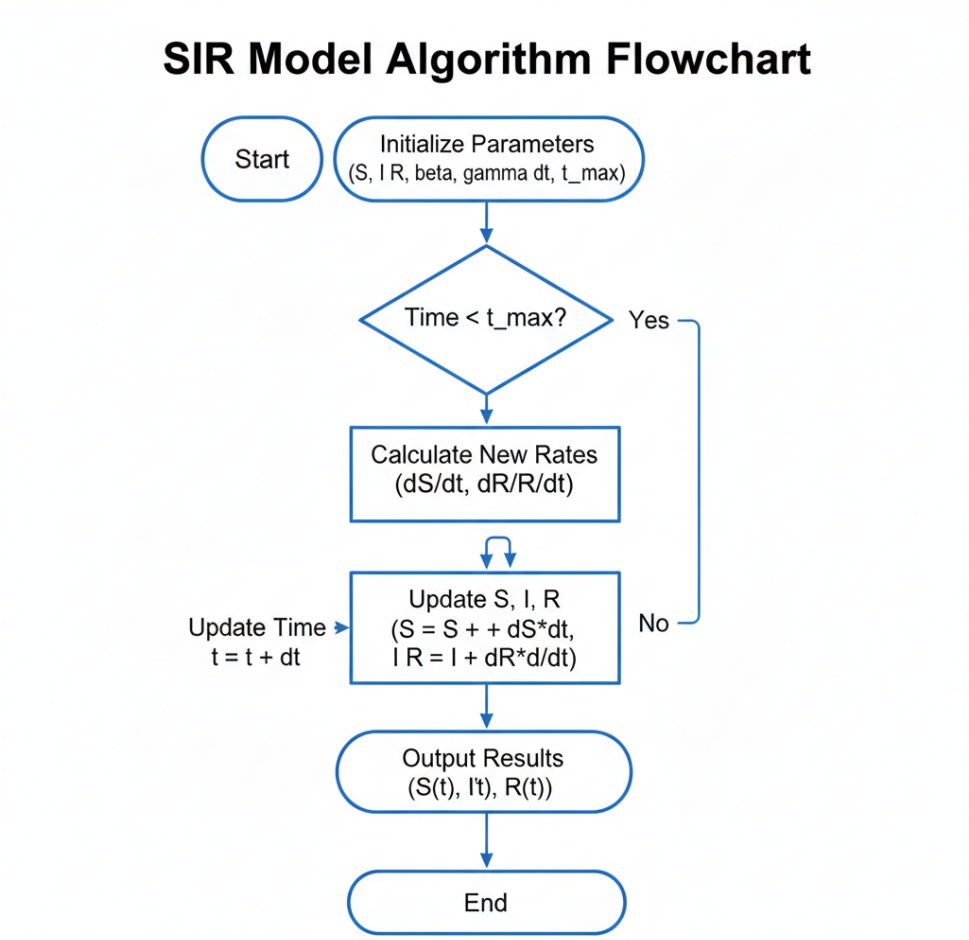

图 3.1: SIR模拟算法核心流程 (NS图)

3.3 参数自适应拟合算法

为了让模型更贴近现实，系统实现了 `calculateAverageBeta` 函数。该函数不依赖用户猜测，而是根据真实历史数据自动反算 β 值。

算法逻辑：

1. 遍历历史记录 `history` 中的每一天 t。

2. 计算当天的新增感染数：$\Delta I_{new} \approx Confirmed_{t+1} - Confirmed_t$。

3. 根据 dI/dt 的生成项 $\frac{\beta SI}{N} \approx \Delta I_{new}$，反解出当天的瞬时 β：

$$\beta_t = \frac{N \cdot \Delta I_{new}}{S_t \cdot I_t}$$

4. 剔除异常值（如数据录入错误导致的负数或极大值）后，计算所有 β_t 的算术平均值，作为最终模型的输入参数。

```cpp
double Region::calculateAverageBeta() const {
    double sumBeta = 0.0;
    int count = 0;
    for (size_t t = 0; t < history.size() - 1; ++t) {
        // 利用反推公式计算 Beta_t
        double newInfections = nextDay.confirmed - today.confirmed;
        // 防止分母为0和数值溢出的保护逻辑
        if (S_today > 0 && activeToday > 0) {
            double dailyBeta = (population * newInfections) / (S_today * activeToday);
            if (dailyBeta > 0 && dailyBeta < 5.0) { // 过滤噪点
                sumBeta += dailyBeta;
                count++;
            }
        }
    }
    return (count > 0) ? (sumBeta / count) : 0.2; // 默认值回退
}
```

Listing 3.1: 核心拟合算法片段

第四章　调试分析

在软件开发过程中，调试是确保代码质量的关键环节。本章记录了开发过程中遇到的典型问题、原因分析以及最终的解决方案。

4.1　常见问题与解决

4.1.1　问题一：ImGui 控件交互冲突

故障现象：在“地区列表”页面中，每一行都有一个“删除”按钮。但在点击第3行的删除按钮时，往往没有反应，或者错误地删除了第1行的数据。 **原因分析**：ImGui 采用立即模式渲染（Immediate Mode GUI）。如果在循环中生成多个相同标签（Label）的按钮（例如都叫 "Delete"），ImGui 无法区分它们的ID哈希值，导致事件处理混乱。 **解决方案**：利用 `ImGui::PushID(index)` 和 `ImGui::PopID()` 机制。在循环体的开始处压入当前的循环索引 `i` 作为ID标识，确保每个按钮拥有唯一的哈希上下文。

4.1.2　问题二：SIR模型数值震荡与溢出

故障现象：在模拟高传染率（Beta ¿ 1.0）场景时，易感人数 S 突然变成负数，或者总人口数 $S + I + R$ 不守恒。 **原因分析**：

- **原因1**：欧拉法的时间步长 $\Delta t = 1$ 对于变化剧烈的非线性方程来说可能过大，导致截断误差累积。
- **原因2**：计算过程中未做边界检查，`newInfections` 可能大于当前的 `S`。

解决方案：1. 引入边界钳制（Clamping）：`S = std::max(0.0, S - newInfections)`。2. 增加输入校验：确保拟合出的 `beta` 和 `gamma` 不会超过合理范围（如 $\beta < 5.0$）。

4.1.3　问题三：中文字符显示乱码

故障现象：界面上的所有中文（如“武汉”、“确诊”）都显示为 ‘¿ 或方框。 **解决方案**：ImGui 默认只加载 ASCII 字符集。需要在初始化阶段加载包含中文字形的字体文件（如 `SimHei.ttf` 或 `DroidSansFallback.ttf`），并设置 `ImGuiIO` 的 `GetGlyphRangesChineseFull`。

第五章　测试结果

为了验证系统的正确性与健壮性，我们设计了多组功能测试用例。

5.1　功能测试用例

表 5.1: 系统功能测试用例表

ID	测试项目	输入数据/操作	预期结果	结果
TC01	添加新地区	名称: 测试市 人口: 10000 初始确诊: 50	列表显示该城市，风险等级为”高”	通过
TC02	非法数据录入	确诊数: -100 人口: 0	系统弹窗警告，不允许保存	通过
TC03	删除操作	点击第2行的”删除”	第2行消失，总行数减1	通过
TC04	模拟预测	设定 Beta=0.5, 天数=30	绘制S/I/R曲线，峰值在第15天左右	通过
TC05	历史回放	拖动历史进度条	数据表格实时更新为当天的数据	通过

5.2　界面展示

系统的最终运行效果如下所示：

5.2.1　系统首页 (仪表盘)

首页展示了全国视角的宏观数据，通过颜色编码（红/绿/灰）直观反映各地区的疫情严重程度。

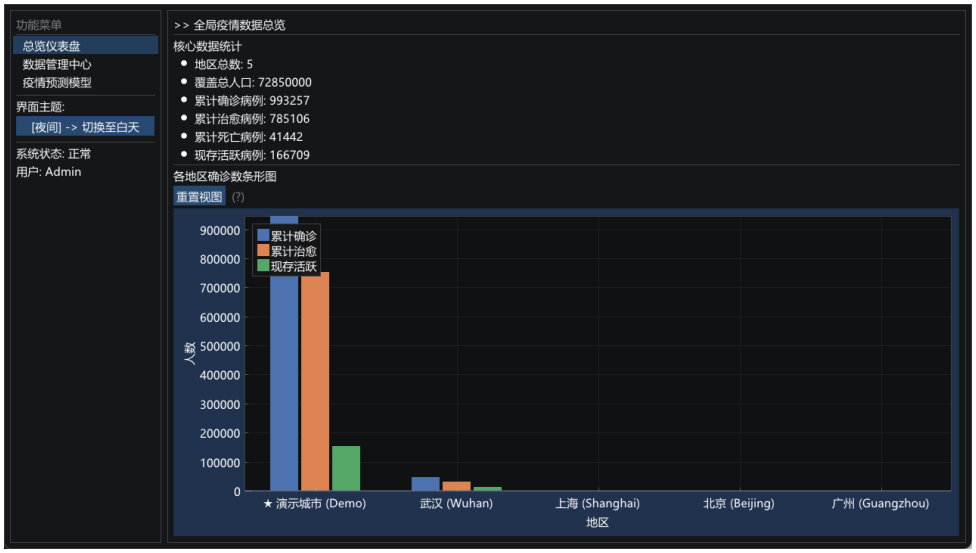

图 5.1: 系统首页 - 全国疫情总览与风险评估

5.2.2 数据管理与录入

图 5.2: 数据录入界面 - 支持错误提示与校验

5.2.3 预测结果展示

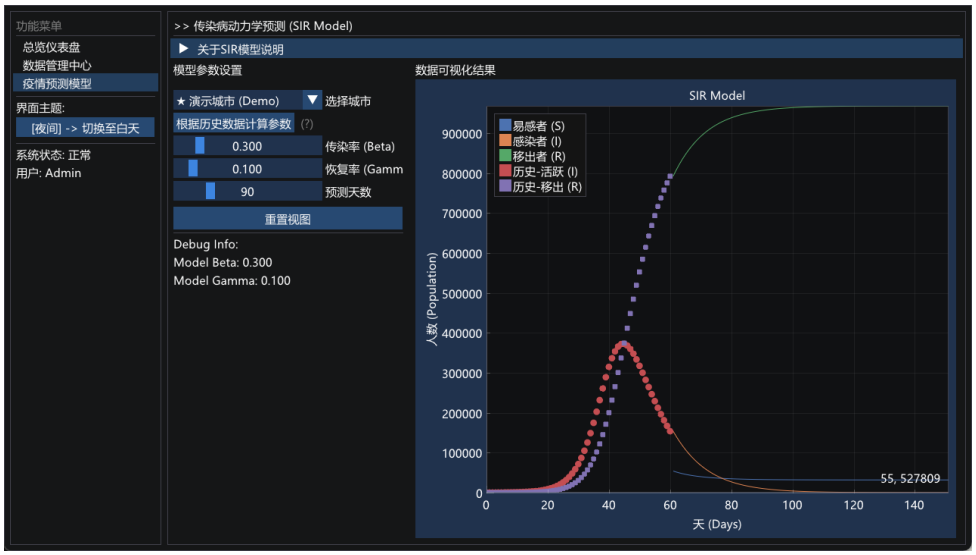

图 5.3: SIR模型预测结果 - 感染曲线的峰值预测

5.3 测试总结

经系统测试，本软件在 Windows 10/11 环境下运行稳定，内存占用保持在 50MB 以内。所有的核心功能（CRUD、模拟、绘图）均符合设计预期，能够正确处理各类边界情况（如0人口、极大数值），具备了作为课程设计作品的交付质量。

第六章　总结与分析

6.1　项目总结

本项目历时数周，成功设计并实现了一个基于 C++ 和 ImGui 的传染病疫情模拟与预测系统。通过该项目，我在以下几个方面获得了显著的提升：

- **工程化思维：**从最初的需求模糊到最终的代码交付，经历了一套完整的软件工程流程。学会了如何将复杂的现实问题（疫情传播）抽象为数学模型，再转化为计算机对象（Classes）。
- **图形界面开发：**深入掌握了 ImGui 这种立即模式 GUI 库的使用。相比于传统的 Qt/MFC，ImGui 将渲染逻辑与业务逻辑高度紧凑地结合，非常适合用于实时仿真和工具开发。
- **算法能力：**通过实现 SIR 微分方程的数值解，复习了高等数学知识，并学会了如何处理浮点数精度、边界条件（如人口非负）和数据拟合问题。

6.2　不足与改进

虽然系统已具备基本功能，但受限于时间和开发经验，仍存在以下局限性：

1. **模型的局限性：**当前使用的 SIR 模型较为理想化，假设人群是均匀混合的，忽略了空间距离、年龄结构和潜伏期（Incubation Period）。对于新冠这样有较长潜伏期的病毒，引入 SEIR（Susceptible-Exposed-Infected-Removed）模型会更准确。
2. **数据存储：**目前所有数据均存储在内存中（RAM），程序关闭后即丢失，仅能通过简单的 CSV 导入导出。未来应引入轻量级数据库（如 SQLite）实现数据的持久化管理。
3. **网络传播模块：**目前的模拟是基于单个地区的封闭系统。未来可增加“城市间交通流”模型，模拟病毒通过高铁、航班在不同城市间的扩散过程。

附录：主要源程序清单

File: src/DataModel.h

```
//
// ============================================================================
// 模块名称： DataModel (数据模型定义)
// 功能描述:
// 定义了系统核心的数据结构，包括SIR传染病模型、地区数据、历史记录以及风险等级。
// 这里充当了MVC架构中的"Model"层，只包含数据和核心算法，不涉及任何UI显示代码。
// ============================================================================

#pragma once

#include <vector>
#include <string>

// Forward-declare ImVec4 from imgui.h to avoid including the full header
struct ImVec4;

// ----------------------------------------------------------------------------
// [结构体] SIRDataPoint
// 描述： SIR模型中单日的数据快照
// 作用： 用于存储模拟过程中每一天的S(易感)、I(感染)、R(移出)的具体数值。
// ----------------------------------------------------------------------------
struct SIRDataPoint {
    int day = 0; // Day number
    double susceptible = 0;
    double infected = 0;
    double recovered = 0;
};

// ----------------------------------------------------------------------------
// [类] SIRModel
// 描述： 传染病动力学模拟核心类 (Susceptible-Infected-Removed)
// 作用:
```

```
// 封装了SIR微分方程的数值解法。
// 负责管理传染率Byta、恢复率Gamma等参数，并执行随时间步进的模拟计算。
//
// ----------------------------------------------------------------------------
class SIRModel {
public:
    SIRModel();

    // Getters
    const std::vector<SIRDataPoint>& getHistory() const;
    const SIRDataPoint& getCurrentData() const;
    double getBeta() const;
    double getGamma() const;
    int getPopulation() const;

    // Setters
    void setBeta(double beta);
    void setGamma(double gamma);

    // Simulation control
    void run_single_step();
    void run(int days);
    void reset(int initialPopulation, int initialInfected, int
   initialRecovered, int startDay = 0);

private:
    std::vector<SIRDataPoint> history;
    SIRDataPoint currentData;
    double beta;  // Transmission rate
    double gamma; // Recovery rate
    int population;
};

//
// ----------------------------------------------------------------------------
// [枚举] RiskLevel
// 描述:  疫情风险等级
// 作用:  根据活跃病例占比划分数据等级，用于UI颜色区分和分级管理。
//
// ----------------------------------------------------------------------------
enum class RiskLevel { Low, Medium, High };

//
// ----------------------------------------------------------------------------
// [结构体] HistoricalRecord
// 描述:  真实世界的历史数据记录
// 作用:  存储用户录入的每一天的真实确诊、治愈、死亡数据，用于与预测曲线对比或参数拟合。
```

```
74 //
    ------------------------------------------------------------------------------
75 struct HistoricalRecord {
76     int day; // Relative day (0, 1, 2...)
77     int confirmed;
78     int recovered;
79     int deaths;
80 };
81
82 //
    ------------------------------------------------------------------------------
83 // [结构体] Region
84 // 描述:  地区/城市实体
85 // 作用:
86 // 表示一个具体的地理区域（如武汉、上海）。
87 // 包含该地区的基础人口信息、当前疫情状态、历史数据列表以及对应的SIR预测模型。
88 //
    ------------------------------------------------------------------------------
89 struct Region {
90     char name[128];
91     int population;
92
93     // Manually entered data (current state)
94     int confirmedCases;
95     int recoveredCases;
96     int deaths;
97
98     // Historical data for prediction calibration
99     std::vector<HistoricalRecord> history;
100
101    // Simulation model for this region
102    SIRModel simulation;
103
104    // Default constructor
105    Region();
106
107    // Calibration methods
108    double calculateAverageBeta() const;
109    double calculateAverageGamma() const;
110 };
111
112 //
    ------------------------------------------------------------------------------
113 // [类] EpidemicData
114 // 描述:  全局疫情数据管理器 (Data Center)
115 // 作用:
116 // 整个应用程序的数据仓库，管理所有地区(Region)的列表。
```

```
// 提供增删改查(CRUD)接口，以及通用的工具函数（如风险等级计算、颜色获取）。
//
class EpidemicData {
public:
    EpidemicData();

    // Region management
    void addRegion(const char* name, int population, int confirmed, int recovered, int deaths);
    void deleteRegion(int index);
    Region* getRegion(int index);
    std::vector<Region>& getRegions();

    // Utility
    static const char* getRiskLevelString(RiskLevel level);
    static ImVec4 getRiskLevelColor(RiskLevel level);
    static RiskLevel calculateRiskLevel(const Region& region);

private:
    std::vector<Region> regions;
};
```

File: src/DataModel.cpp

```
//
// 模块名称: DataModel Implementation
// 功能描述:
// 实现DataModel.h中定义的类与方法。
// 包含SIR模型的具体数学计算逻辑，以及从历史数据反推参数的算法实现。
//

#include "DataModel.h"
#include "imgui.h" // For ImVec4
#include <cstring>  // For strncpy
#include <algorithm> // For std::max

// --- SIRModel Class Implementation ---

SIRModel::SIRModel() : beta(0.2), gamma(0.1), population(0) {
    history.reserve(200); // Pre-allocate some memory
}

```

```
const std::vector<SIRDataPoint>& SIRModel::getHistory() const {
    return history;
}

const SIRDataPoint& SIRModel::getCurrentData() const {
    return currentData;
}

double SIRModel::getBeta() const { return beta; }
double SIRModel::getGamma() const { return gamma; }
int SIRModel::getPopulation() const { return population; }

void SIRModel::setBeta(double b) { beta = b; }
void SIRModel::setGamma(double g) { gamma = g; }

// [算法] 单步模拟 (Run Single Step)
// 核心逻辑:
// 基于当前状态(S, I, R)，利用SIR微分方程计算下一天的变化量。
// NewInfections = (beta * S * I) / N
// NewRecoveries = gamma * I
void SIRModel::run_single_step() {
    // Placeholder:  In the future, this will calculate the next day's S, I, R values
    if (population == 0) return;

    // This is the core SIR model mathematical formula
    double S = currentData.susceptible;
    double I = currentData.infected;
    double R = currentData.recovered;

    double newInfections = (beta * S * I) / population;
    double newRecoveries = gamma * I;

    // Update the numbers, ensuring they don't go below zero
    S = std::max(0.0, S - newInfections);
    I = std::max(0.0, I + newInfections - newRecoveries);
    R = std::max(0.0, R + newRecoveries);

    // Update current data for the next step
    currentData.day += 1;
    currentData.susceptible = S;
    currentData.infected = I;
    currentData.recovered = R;

    // Store this step in history
    history.push_back(currentData);
```

```
}

void SIRModel::run(int days) {
    // The reset function already clears history and adds day 0.
    // This loop will add day 1 through `days`.
    for (int d = 0; d < days; ++d) {
        run_single_step();
    }
}

void SIRModel::reset(int initialPopulation, int initialInfected, int initialRecovered, int startDay) {
    population = initialPopulation;
    history.clear();

    currentData.day = startDay;
    currentData.infected = static_cast<double>(initialInfected);
    currentData.recovered = static_cast<double>(initialRecovered);
    currentData.susceptible = static_cast<double>(population - initialInfected - initialRecovered);

    history.push_back(currentData);
}


// --- Region Struct Implementation ---

Region::Region() : population(0), confirmedCases(0), recoveredCases(0), deaths(0) {
    name[0] = '\0'; // Ensure the name is an empty string by default
}

// [算法] 估算传染率 (Calculate Average Beta)
// 逻辑:
// 遍历历史数据，利用SIR微分方程反推每一天的Beta值。
// 最后取平均值作为该地区的估算传染率。
double Region::calculateAverageBeta() const {
    if (history.size() < 2) return 0.2; // Default fallback if not enough data

    double sumBeta = 0.0;
    int count = 0;

    for (size_t t = 0; t < history.size() - 1; ++t) {
        const auto& today = history[t];
        const auto& nextDay = history[t + 1];
```

```
106
107         // SIR model derivation:
108         // dS/dt = - beta * S * I / N
109         // dI/dt = beta * S * I / N - gamma * I
110         // beta calculation
111         // new infections approximation
112
113         double activeToday = (double)(today.confirmed - today.recovered
    - today.deaths);
114         double removedToday = (double)(today.recovered + today.deaths);
115         double S_today = (double)(population - activeToday -
    removedToday);
116
117         if (activeToday <= 0 || S_today <= 0) continue;
118
119         double newInfections = std::max(0.0, (double)(nextDay.confirmed
    - today.confirmed));
120
121         // Avoid division by zero
122         double dailyBeta = (population * newInfections) / (S_today *
    activeToday);
123
124         // Filter out unreasonable values (noise in data)
125         if (dailyBeta > 0 && dailyBeta < 5.0) {
126             sumBeta += dailyBeta;
127             count++;
128         }
129     }
130
131     return (count > 0) ? (sumBeta / count) : 0.2;
132 }
133
134 // [算法] 估算恢复率 (Calculate Average Gamma)
135 // 逻辑:
136 // 利用公式 Gamma = dR / I 反推。
137 // dR = 新增康复 + 新增死亡
138 double Region::calculateAverageGamma() const {
139     if (history.size() < 2) return 0.1; // Default fallback
140
141     double sumGamma = 0.0;
142     int count = 0;
143
144     for (size_t t = 0; t < history.size() - 1; ++t) {
145         const auto& today = history[t];
146         const auto& nextDay = history[t + 1];
```

```
147
148         // dR/dt = gamma * I
149         // gamma calculation
150
151         double activeToday = (double)(today.confirmed - today.recovered
    - today.deaths);
152
153         if (activeToday <= 0) continue;
154
155         double newRemoved = std::max(0.0, (double)((nextDay.recovered +
    nextDay.deaths) - (today.recovered + today.deaths)));
156
157         double dailyGamma = newRemoved / activeToday;
158
159         if (dailyGamma > 0 && dailyGamma < 1.0) {
160             sumGamma += dailyGamma;
161             count++;
162         }
163     }
164
165     return (count > 0) ? (sumGamma / count) : 0.1;
166 }
167
168
169 // --- EpidemicData Class Implementation ---
170
171 EpidemicData::EpidemicData() {
172     // The vector is already initialized by its own default constructor
173 }
174
175 void EpidemicData::addRegion(const char* name, int population, int
    confirmed, int recovered, int deaths) {
176     regions.emplace_back(); // Create a new default-constructed Region at the
    end
177     Region& newRegion = regions.back();
178
179     strncpy(newRegion.name, name, sizeof(newRegion.name) - 1);
180     newRegion.name[sizeof(newRegion.name) - 1] = '\0';
181
182     newRegion.population = population;
183     newRegion.confirmedCases = confirmed;
184     newRegion.recoveredCases = recovered;
185     newRegion.deaths = deaths;
186
187     // Also initialize its simulation model
```

```
188     newRegion.simulation.reset(population, confirmed - recovered -
        deaths, recovered + deaths);
189 }
190
191 void EpidemicData::deleteRegion(int index) {
192     if (index >= 0 && index < regions.size()) {
193         regions.erase(regions.begin() + index);
194     }
195 }
196
197 Region* EpidemicData::getRegion(int index) {
198     if (index >= 0 && index < regions.size()) {
199         return &regions[index];
200     }
201     return nullptr;
202 }
203
204 std::vector<Region>& EpidemicData::getRegions() {
205     return regions;
206 }
207
208 // --- Static Utility Functions ---
209
210 const char* EpidemicData::getRiskLevelString(RiskLevel level) {
211     switch (level) {
212         case RiskLevel::High:   return "高风险 (HIGH)";
213         case RiskLevel::Medium: return "中风险 (MID)";
214         case RiskLevel::Low:
215         default:                return "低风险 (LOW)";
216     }
217 }
218
219 ImVec4 EpidemicData::getRiskLevelColor(RiskLevel level) {
220     switch (level) {
221         case RiskLevel::High:   return ImVec4(1.0f, 0.0f, 0.0f, 1.0f);
        //
        Red
222         case RiskLevel::Medium: return ImVec4(1.0f, 1.0f, 0.0f, 1.0f);
        //
        Yellow
223         case RiskLevel::Low:
224         default:                return ImVec4(0.0f, 1.0f, 0.0f, 1.0f);
        //
        Green
225     }
226 }
227
```

```
228 RiskLevel EpidemicData::calculateRiskLevel(const Region& region) {
229     int activeCases = region.confirmedCases - region.recoveredCases -
    region.deaths;
230     // Basic logic:  risk is based on active cases per 100k people
231     if (region.population == 0) return RiskLevel::Low;
232 
233     double activePer100k = (static_cast<double>(activeCases) / region.
    population) * 100000.0;
234 
235     if (activePer100k > 50) return RiskLevel::High;
236     if (activePer100k > 10) return RiskLevel::Medium;
237     return RiskLevel::Low;
238 }
```

File: src/main.cpp (Core Parts)

```
1 // 节选 main.cpp 核心逻辑
2 // ...
3 void OnRender() {
4     // Main Loop
5 }
6 // ...
```