课后任务

线性方程组求解系统

设计一个线性方程组求解系统，支持多种求解方法：

- 使用容器类存储矩阵和向量数据
- 使用面向对象多态实现不同的求解算法（高斯消元法、LU 分解法、雅可比迭代法、SOR 迭代法）
- 使用 Lambda 表达式实现自定义的矩阵运算和向量操作
- 使用智能指针管理动态分配的矩阵对象
- 包含输入验证和异常处理机制（如奇异矩阵检测、收敛性判断）

线性方程组求解问题简介

问题描述

线性方程组求解旨在寻找满足 $Ax = b$ 的未知向量 x，其中 A 为 $n \times n$ 系数矩阵，b 为常数向量。
该问题广泛应用于科学计算、工程仿真、数据分析等领域，是数值计算的基础问题之一。

常用求解方法

- **高斯消元法**：直接法，适用于小型稠密矩阵，数值稳定性高。
- **LU 分解法**：将 A 分解为上下三角矩阵，便于多次求解不同右端项。
- **雅可比迭代法**：适合大型稀疏矩阵，易于并行实现。
- **SOR 迭代法**：在雅可比基础上加速收敛，适用于对角占优矩阵。

$Ax = b$——线性方程组简介

- A：$n \times n$ 的系数矩阵，包含所有方程中未知数的系数。每一行对应一个方程，每一列对应一个未知数。例如：

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}$$

- x：n 维未知向量，表示待求解的变量。形式为：

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

- b：n 维常数向量，表示每个方程右端的常数项。形式为：

$$b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

高斯消元法

高斯消元法的基本思想是：通过消元操作，将线性方程组 $Ax = b$ 转化为上三角矩阵，然后通过回代求解。高斯消元法的步骤如下：

- **选主元（可选）**：在每一步消元前，选取当前列绝对值最大的元素作为主元，通过行交换将其移至对角线位置，提高数值稳定性。
- **消元操作**：从第 1 行开始，依次对每一行，将该行以下的各行对应列的系数消为 0。具体做法如下：
  对于第 k 步 $(k = 1, 2, \ldots, n-1)$，对第 k 行以下的每一行 i，用如下公式更新：
  $$a_{ij} = a_{ij} - \frac{a_{ik}}{a_{kk}} a_{kj}, \quad (j = k, k+1, \ldots, n); b_i = b_i - \frac{a_{ik}}{a_{kk}} b_k$$
  其中 a_{ik}/a_{kk} 为消元因子。
- **回代求解**：当矩阵变为上三角后，从最后一行开始，依次向上代入已知的未知数，递推求解出所有未知数的值。回代公式为：
  $$x_i = \frac{1}{a_{ii}} \left(b_i - \sum_{j=i+1}^{n} a_{ij} x_j \right), \quad (i = n, n-1, \ldots, 1)$$

LU 分解法

LU 分解法是一种高效求解线性方程组 $Ax = b$ 的方法，其核心思想是将系数矩阵 A 分解为下三角矩阵 L 和上三角矩阵 U 的乘积，即 $A = LU$，从而将原问题转化为两个更易求解的三角方程组。具体步骤如下：

- **LU 分解**：将 $n \times n$ 矩阵 A 分解为下三角矩阵 L（对角线元素为 1）和上三角矩阵 U，满足 $A = LU$。常用的分解方法有 Doolittle 法（L 主对角线为 1）和 Crout 法（U 主对角线为 1）。分解过程通常通过消元操作实现，类似高斯消元，但将消元因子存入 L 矩阵。
- **前向替换**：将 $Ax = b$ 转化为 $LUx = b$，令 $Ux = y$，先解 $Ly = b$。由于 L 为下三角矩阵，$Ly = b$ 可通过前向替换递推求解：

$$y_i = b_i - \sum_{j=1}^{i-1} l_{ij} y_j, \quad (i = 1, 2, \ldots, n)$$

- **后向替换**：已知 y 后，解 $Ux = y$。U 为上三角矩阵，可通过后向替换递推求解 x：

$$x_i = \frac{1}{u_{ii}} \left(y_i - \sum_{j=i+1}^{n} u_{ij} x_j \right), \quad (i = n, n-1, \ldots, 1)$$

LU 分解法：Doolittle 法详解

目标：找到 L 和 U，使 $A = LU$，其中

$$L = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ l_{21} & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & \cdots & 1 \end{bmatrix}, \quad U = \begin{bmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ 0 & u_{22} & \cdots & u_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & u_{nn} \end{bmatrix}$$

计算公式：对 $k = 1, 2, \ldots, n$，依次计算 U 和 L 的元素：

- 计算 U 的第 k 行：

$$u_{kj} = a_{kj} - \sum_{s=1}^{k-1} l_{ks} u_{sj}, \quad j = k, k+1, \ldots, n$$

- 计算 L 的第 k 列（主对角线 $L_{kk} = 1$）：

$$l_{ik} = \frac{1}{u_{kk}} \left(a_{ik} - \sum_{s=1}^{k-1} l_{is} u_{sk} \right), \quad i = k+1, k+2, \ldots, n$$

分解流程：逐行逐列递推，先算 U 的第 k 行，再算 L 的第 k 列，直到全部元素求出。

雅可比迭代法

雅可比迭代法是一种常用的迭代求解线性方程组 $Ax = b$ 的方法。其基本思想是利用矩阵的分解，将每个未知数的计算与其它未知数的当前近似值分离开来，从而逐步逼近精确解。具体过程如下：

- **矩阵分解**：将系数矩阵 A 分解为对角矩阵 D 和非对角部分 R，即 $A = D + R$，其中 D 为 A 的主对角线元素构成的对角矩阵，$R = A - D$ 为其余元素组成的矩阵。
- **迭代公式推导**：原方程 $Ax = b$ 可写为 $Dx = b - Rx$，从而得到迭代格式：

  $$x^{(k+1)} = D^{-1}(b - Rx^{(k)})$$

  具体到每个分量，有

  $$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1,\, j\neq i}^{n} a_{ij} x_j^{(k)} \right), \quad i = 1, 2, \ldots, n$$

  即每次迭代时，第 i 个分量的新值由其余分量的上一次迭代值计算得到。
- **初始值选择**：给定初始向量 $x^{(0)}$，通常可取零向量或任意猜测值。
- **收敛判据**：迭代过程中，若 $\|x^{(k+1)} - x^{(k)}\|$ 小于预设精度 ε，则认为收敛。

SOR 迭代法

SOR（Successive Over-Relaxation，超松弛迭代）法是在高斯-赛德尔迭代法基础上引入松弛因子 ω 的一种加速收敛的迭代方法，常用于求解大型稀疏线性方程组 $Ax = b$。

- **基本思想**：通过引入松弛因子 ω，调整每次迭代的步长，从而加快收敛速度。$\omega = 1$ 时，SOR 法退化为高斯-赛德尔法；$\omega > 1$ 为超松弛，$\omega < 1$ 为欠松弛。
- **矩阵分解**：将 A 分解为 $A = D + L + U$，其中 D 为对角矩阵，L 为严格下三角部分，U 为严格上三角部分。
- **迭代公式推导**：原方程 $Ax = b$ 可写为

$$(D + L)x^{(k+1)} = b - Ux^{(k)}$$

SOR 法的迭代格式为

$$x^{(k+1)} = (1 - \omega)x^{(k)} + \omega(D + L)^{-1}(b - Ux^{(k)})$$

具体到每个分量，第 i 个分量的迭代公式为

$$x_i^{(k+1)} = (1 - \omega)x_i^{(k)} + \frac{\omega}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k)}\right)$$

其中 ω 为松弛因子，$0 < \omega < 2$。

线性方程组求解系统设计

系统架构

- **矩阵与向量类**：基于 `std::vector` 实现高效存储与运算，支持 Lambda 表达式优化常用操作。
- **求解器基类（SolverBase）**：定义统一的虚接口，便于扩展多种算法，体现面向对象多态。
- **具体求解器实现**：
  - **GaussianSolver**：高斯消元法，支持部分选主元与行交换优化。
  - **LUSolver**：Doolittle LU 分解，前向/后向替代高效实现。
  - **JacobiSolver**：雅可比迭代，内置收敛性检查。
  - **SORSolver**：SOR 加速迭代，支持超松弛参数自定义。
- **系统管理类（LinearSystem）**：封装求解流程，负责输入验证、奇异性检测、精度分析与异常处理。
- **智能指针与 RAII**：采用 `std::unique_ptr` 管理动态对象，保证内存安全与异常安全。

开发步骤

- **1. 设计系统架构**：确定矩阵与向量类、求解器基类和具体求解器实现。
- **2. 实现矩阵与向量类**：基于 `std::vector` 实现高效存储与运算，支持 Lambda 表达式优化常用操作。
- **3. 实现求解器基类**：定义统一的虚接口，便于扩展多种算法，体现面向对象多态。
- **4. 实现具体求解器**：实现高斯消元法、LU 分解法、雅可比迭代法、SOR 迭代法。
- **5. 实现系统管理类**：封装求解流程，负责输入验证、奇异性检测、精度分析与异常处理。

Vector 类

向量类设计

基于 `std::vector` 实现的一维数值向量类
支持 Lambda 表达式优化常用运算
提供完整的向量代数操作接口

核心特性

- **存储结构**：`std::vector<double>` 连续存储
- **构造函数**：默认、数据、移动构造
- **元素访问**：带边界检查的 `at()` 方法
- **标量运算**：`multiply()` 方法
- **范数计算**：`norm()` 方法（2-范数）
- **输出**：格式化的控制台打印

Lambda 优化

- 标量乘法：`std::transform`
- 范数计算：`std::accumulate`
- 打印输出：`std::for_each`
- 性能提升：编译时内联优化

知识点：构造函数

构造函数

构造函数是类的一种特殊成员函数，用于在创建对象时初始化对象。
构造函数可以有参数，也可以没有参数。
构造函数可以有返回值，也可以没有返回值。
构造函数可以有多个，也可以没有多个。
默认构造函数：没有参数的构造函数，用于创建对象时自动调用。
拷贝构造函数：用于创建对象时复制另一个对象的值。
移动构造函数：用于创建对象时移动另一个对象的值。

知识点：C++ 异常

C++ 异常处理机制

C++ 异常处理用于在程序运行时捕获和处理错误，提升程序健壮性。
throw：抛出异常对象，可为内置类型或自定义类型。
try-catch：`try` 块包裹可能出错的代码，`catch` 块捕获并处理异常。
noexcept：用于声明函数不会抛出异常，便于编译器优化。
支持异常的层级捕获与重新抛出，便于精细化错误处理。
建议自定义异常类型继承自 `std::exception`，便于统一管理。

std::out$_{o}f_{r}ange$
std::invalid$_{a}rgument$ $std :: runtime_{e}rror$
std::logic$_{e}rror$ $std :: exception$
std::exception_ptr：用于表示异常的指针。

知识点：std::transform + Lambda 表达式优化

std::transform 详解

std::transform 是 C++ 标准库中的泛型算法，用于将一个或多个输入区间的元素通过指定的操作变换后，输出到目标区间。
常见用法为对容器（如 std::vector）中的每个元素应用某个函数或 Lambda 表达式，实现批量数据处理。
语法示例：std::transform(first, last, d_first, unary_op);，其中 unary_op 可以是函数指针、函数对象或 Lambda 表达式。
也支持二元操作：std::transform(first1, last1, first2, d_first, binary_op);，可用于两个容器的元素逐一运算。
结合 Lambda 表达式，可实现如标量乘法、元素映射、批量转换等高效操作。例如：std::transform(v.begin(), v.end(), v2.begin(), [](double x) return x * 2;);
std::transform 通常比手写循环更简洁且易于编译器优化，推荐在需要元素级变换时优先使用。

知识点：std::move

std::move 详解

`std::move` 是 C++ 标准库中的一个函数，用于将一个对象转换为右值引用，从而可以被移动构造或赋值。
常见用法为将一个对象转换为右值引用，从而可以被移动构造或赋值。
语法示例：`std::move(x);`，其中 `x` 是一个对象。
也支持二元操作：`std::move(x, y);`，可用于两个对象的移动构造或赋值。

右值引用

右值引用是 C++11 引入的一种新的引用类型，用于表示临时对象或右值。
常见用法为将一个对象转换为右值引用，从而可以被移动构造或赋值。
语法示例：`int x = 1;`，其中 `x` 是一个右值引用。

知识点：

是 C++11 引入的一种新的引用类型，用于表示右值引用。
常见用法为将一个对象转换为右值引用，从而可以被移动构造或赋值。
语法示例：`int x = 1;`，其中 `x` 是一个右值引用。

知识点：std::accumulate

std::accumulate

std::accumulate 是 C++ 标准库中的泛型算法，用于对区间内元素进行累加、累乘或自定义聚合操作。
常见用法包括：求和、求积、字符串拼接、结构体成员累加等。
基本语法：std::accumulate(first, last, init);，其中 first 和 last 为迭代器，init 为初始值。
支持自定义二元操作：std::accumulate(first, last, init, binary_op);，可实现如乘积、最大值、字符串拼接等多种聚合。
结合 Lambda 表达式可灵活实现复杂聚合逻辑。例如：std::accumulate(v.begin(), v.end(), 0.0, [](double a, double b){ return a + b * b; });
相比手写循环，std::accumulate 代码更简洁、可读性更高，易于编译器优化。

知识点：std::for_each + Lambda 表达式

std::for_each 优化用法

`std::for_each` 是 C++ 标准库中的泛型算法，用于对区间内每个元素执行指定操作，常用于遍历和批量处理。

推荐结合 Lambda 表达式使用，可实现就地处理、捕获外部变量、简化代码。例如：

```
std::for_each(v.begin(), v.end(), [](int &x){ x *= 2; });
```

语法：`std::for_each(first, last, func);`，其中 `func` 可为函数指针、函数对象或 Lambda 表达式。

支持对任意容器（如 `std::vector`、`std::list` 等）进行高效遍历和操作，代码更简洁、可读性更高。

与传统 for 循环相比，`std::for_each` 更易于并行化和编译器优化，适合大规模数据处理场景。

Matrix 类

矩阵类设计

基于 `std::vector<std::vector<double>>` 实现的二维数值矩阵类
支持 Lambda 表达式优化矩阵运算
提供完整的矩阵代数操作接口

核心特性

- **存储结构**：二维向量，行优先存储
- **构造函数**：默认、数据、移动构造
- **元素访问**：带边界检查的 `at()` 方法
- **矩阵运算**：`multiply()` 矩阵-向量乘法
- **奇异性检测**：基于行列式的精确判断
- **输出**：格式化的控制台打印

Lambda 优化

- 矩阵-向量乘：`std::inner_product`
- 打印输出：嵌套 `std::for_each`
- 行列式计算：递归算法优化
- 对角元素检查：简化的奇异性判断
- 性能提升：编译时内联和常量传播

知识点：std::inner_product

std::inner_product 优化用法

`std::inner_product` 是 C++ 标准库中的泛型算法，常用于高效计算两个区间（如向量）元素的内积，广泛应用于矩阵运算、信号处理等领域。

基本用法：`std::inner_product(first1, last1, first2, init);`，其中 `first1` 和 `last1` 为第一个区间的迭代器，`first2` 为第二个区间的起始迭代器，`init` 为初始值。

支持自定义二元操作和累加操作：`std::inner_product(first1, last1, first2, init, binary_op1, binary_op2);`，可实现如加权和、逻辑运算等复杂聚合。

推荐结合 Lambda 表达式灵活定制内积逻辑。例如：`std::inner_product(a.begin(), a.end(), b.begin(), 0.0, std::plus<>(), [](double x, double y){ return x*y; });`

相比手写循环，`std::inner_product` 代码更简洁、可读性更高，易于编译器优化和并行化。

SolverBase 求解器基类

抽象基类设计

采用面向对象多态设计，实现算法抽象
定义统一的求解接口，便于扩展新算法
使用智能指针管理内存，异常安全

设计模式

- **策略模式**：不同算法的封装和替换
- **工厂模式**：动态创建具体求解器实例
- **模板方法**：定义求解流程的骨架
- **RAII 模式**：智能指针自动资源管理

核心接口

- **纯虚函数**：`solve()` 统一求解接口
- **智能指针**：`std::unique_ptr<Vector>` 返回值
- **异常处理**：返回 nullptr 表示求解失败
- **const 正确性**：不修改输入参数

GaussianSolver 高斯消元法

具体实现

继承 SolverBase 基类，实现高斯消元算法
支持部分选主元策略，提高数值稳定性
使用 Lambda 表达式优化行交换操作

算法流程

- **前向消元**：逐列消元，将矩阵化为上三角
- **部分选主元**：选择绝对值最大的主元
- **行交换**：使用 `std::swap_ranges` 优化
- **回代求解**：从最后一列向上递推求解
- **奇异性检测**：检查主元是否接近零

Lambda 优化

- 行交换：`std::swap_ranges` 替代循环
- 边界检查：编译时类型安全
- 内存布局：连续存储保证缓存友好
- 异常处理：返回 nullptr 表示失败

知识点：std::unique$_{p}tr + std :: make_{u}nique$

std::unique$_{p}tr$

std::unique_ptr 是 C++ 标准库中的一个智能指针，用于管理动态分配的内存。
常见用法为将一个对象转换为右值引用，从而可以被移动构造或赋值。
语法示例：`std::unique_ptr<int> x = std::make_unique<int>(1);`，其中 x 是一个智能指针。

std::make$_{u}nique$

std::make_unique 是 C++ 标准库中的一个函数，用于创建一个智能指针。
常见用法为创建一个智能指针。
语法示例：`std::unique_ptr<int> x = std::make_unique<int>(1);`，其中 x 是一个智能指针。

知识点：std::swap_ranges

std::swap 简介

`std::swap` 是 C++ 标准库中用于高效交换两个对象内容的函数，简化代码并提升性能。
常见场景：变量交换、容器元素交换等，适用于所有支持交换操作的类型。
示例：`std::swap(a, b);` 可直接交换 a 和 b 的值。
注意：参与交换的对象需支持交换操作，否则结果未定义。

std::swap_ranges 优化与应用

`std::swap_ranges` 可高效地一次性交换两个区间的全部元素，避免手动循环，提升代码可读性与执行效率。
典型应用：矩阵行交换、批量数据重排等，适用于如 `std::vector`、原生数组等支持随机访问迭代器的容器。
示例：`std::swap_ranges(row1.begin(), row1.end(), row2.begin());` 可高效交换两行数据。
注意：两个区间长度必须一致且不能重叠，否则行为未定义。
优化建议：结合 Lambda 表达式与类型安全检查，可进一步提升代码健壮性和灵活性。

LUSolver LU 分解法

具体实现

继承 SolverBase 基类，实现 Doolittle LU 分解算法
支持多次求解不同右端项，提高效率
使用 Lambda 表达式优化前向/后向替代

算法流程

- **LU 分解**：将 A 分解为 L（下三角）和 U (上三角)
- **Doolittle 方法**：L 的对角线元素为 1
- **前向替代**：解 $Ly = b$ 得到中间向量 y
- **后向替代**：解 $Ux = y$ 得到最终解 x
- **效率优势**：适合多次求解相同系数矩阵

Lambda 优化

- 矩阵分解：`std::accumulate` 优化求和
- 前向替代：Lambda 表达式简化循环
- 后向替代：Lambda 表达式优化计算
- 内存效率：避免重复矩阵拷贝

JacobiSolver 雅可比迭代法

具体实现

继承 SolverBase 基类，实现雅可比迭代算法
支持自定义收敛容差和最大迭代次数
使用 Lambda 表达式优化收敛性检查

算法流程

- **矩阵分解**：$A = D + R$（D 为对角矩阵）
- **迭代公式**：$x^{(k+1)} = D^{-1}(b - Rx^{(k)})$
- **分量计算**：每个分量独立计算
- **收敛检查**：检查前后两次迭代的差值
- **适用条件**：严格对角占优矩阵

Lambda 优化

- 收敛检查：`std::accumulate` 优化误差计算
- 迭代过程：Lambda 表达式简化循环
- 边界检查：编译时类型安全
- 内存效率：避免临时变量创建

SORSolver SOR 迭代法

具体实现

继承 SolverBase 基类，实现 SOR 加速迭代算法
支持自定义松弛因子、收敛容差和最大迭代次数
使用 Lambda 表达式优化收敛性检查和迭代过程

算法流程

- **矩阵分解**：$A = D + L + U$（D 对角，L 严格下三角，U 严格上三角）
- **SOR 公式**：$x^{(k+1)} = (1-\omega)x^{(k)} + \omega(D+L)^{-1}(b - Ux^{(k)})$
- **松弛因子**：ω 控制收敛速度 $(0 < \omega < 2)$
- **收敛加速**：比雅可比方法收敛更快
- **适用条件**：对角占优矩阵

Lambda 优化

- 收敛检查：`std::accumulate` 优化误差计算
- 迭代计算：Lambda 表达式简化求和
- 超松弛：Lambda 表达式优化参数应用
- 内存效率：避免临时向量创建

LinearSystem 系统管理类

门面模式设计

封装复杂的求解系统，提供统一接口
门面模式隐藏子系统的复杂性
提供完整的求解流程和验证功能

核心功能

- **系统封装**：封装矩阵 A 和向量 b
- **统一求解**：提供统一的 solve 接口
- **解管理**：存储精确解和数值解
- **精度验证**：残差计算和误差分析
- **准确性验证**：解的准确性检查
- **报告生成**：完整的精度分析报告

Lambda 优化

- 残差计算：`std::transform` 优化
- 误差计算：`std::transform` 优化
- 数值验证：Lambda 表达式简化计算
- 报告生成：Lambda 表达式格式化输出

主程序 main.cpp

系统集成与演示

完整的系统集成和算法测试演示
展示所有组件的协同工作
验证系统的正确性和性能

程序流程

- **编码设置**：UTF-8 中文显示支持
- **测试矩阵**：创建 5×5 对角占优系统
- **求解器创建**：实例化四种求解器
- **算法测试**：逐个测试求解器
- **精度验证**：残差计算和误差分析
- **报告生成**：完整的精度分析报告

测试策略

- 5×5 对角占优矩阵测试
- 四种算法全面验证
- 数值精度对比分析
- 收敛性检查验证
- 中文界面友好展示