C 程序与算法分析
项目实践课部分

授课教师
陆佳亮、龚禾林、邱思琦

助教
曹家齐

上海交通大学
巴黎卓越工程师学院

目 录

图 目 录

表 目 录

1 项目实践课教学大纲

The course project focuses on the development of an industrial-grade matrix computation library, guiding students to implement a functional matrix library in C from scratch. It emphasizes modular design, memory management, and performance optimization, incorporating industrial practices such as OpenBLAS et al. Students will develop practical programming and algorithm implementation skills in the field of high-performance computing.

Grading: Assiduity + Lab: 30 %; Quiz 30%, **Project: 40%.**

表 1 单元安排

单元	主题	基础任务	提高任务
1 第 9 周	工业背景与环境搭建	• 矩阵库应用背景 • 内存管理 • 矩阵结构体定义	• 内存泄漏检测
2 第 10 周	模块化编程	• 项目框架设计 • 头文件设计 • 防御性检查	• 错误编码系统 • Quiz（结构体）
3 第 11 周	矩阵基本运算	• 矩阵加法 • 矩阵减法 • 矩阵转置 • 标量乘法	• 矩阵运算的 SIMD 向量化优化
4 第 12 周	矩阵乘法	• 三重循环实现 • 理解时间复杂度	• 内存访问模式优化
	课程第三次作业		
5 第 13 周	矩阵求逆（1）	• 矩阵代数余子式 • 矩阵行列式	• 递归算法的时间复杂度分析与优化策略
6 第 14 周	矩阵求逆（2）	• 伴随矩阵 • 矩阵求逆	• 矩阵求逆替代算法（如 LU 分解）
7 第 15 周	综合测试	• 功能验证 • 性能测试	• 算法优化
8 第 16 周	项目答辩	• 项目报告文档 • 现场答辩	

2 矩阵计算的时代背景与重要性

在人工智能、工业控制以及科学计算等众多领域中，矩阵及其运算（矩阵加减、乘法、求逆、行列式、特征分解等）是不可或缺的数学工具。随着算法复杂度与数据规模的持续增长，高性能矩阵计算已成为现代科学与工程计算的核心支撑。

（1）人工智能中的核心基础

在机器学习与深度学习系统中，数据通常以向量或矩阵形式表示。模型中的线性变换（例如全连接层、卷积核、特征降维、主成分分析等）均可视为矩阵运算。矩阵乘法是神经网络中最频繁的计算操作之一，支撑着参数更新、特征提取与预测等关键环节。

例如，Transformer 等大型语言模型的推理过程，本质上是多组矩阵 - 矩阵乘法与非线性激活函数的组合。可以说，每一次智能决策背后都在进行大量矩阵计算。

（2）工业与科学计算中的关键支撑

在工程仿真、结构力学、流体力学、电子设计及数字孪生系统中，矩阵运算是数值求解的核心环节。有限元分析（FEM）或有限差分法（FDM）在求解过程中都会生成大型稀疏矩阵，通过矩阵方程组的求解获得系统状态的精确描述。在能源、航天、核能及制造等工业场景中，矩阵求解的性能直接影响系统建模和优化设计的效率。

（3）性能挑战与硬件优化

随着矩阵规模的扩大,矩阵乘法(MatMul)成为计算性能的瓶颈。针对不同硬件(CPU、GPU、TPU）及不同精度（FP32、BF16、Int8）进行优化已成为工业界的重要课题。现代高性能计算平台通过缓存优化、指令并行和分布式计算显著提升了矩阵运算速度。

矩阵计算因此被称为“人工智能与科学计算的引擎”，其效率直接决定系统运行的可扩展性与可靠性。

2.1 主流矩阵计算库

为满足不同应用场景的需求,矩阵计算库形成了从底层基础到高层框架的多层体系，主要包括三大类：基础线性代数库、大规模科学计算库和现代 C++ 矩阵库等。

2.1.1 基础线性代数库（BLAS / LAPACK）

- BLAS（Basic Linear Algebra Subprograms）

https://www.netlib.org/blas/

BLAS 定义了基础的向量与矩阵运算接口，如向量加法、标量乘法、矩阵乘法等。其分为三级：

Level 1：向量运算

Level 2：矩阵 - 向量运算

Level 3：矩阵 - 矩阵运算

由于 BLAS 具备高效、可移植和广泛可用的特点，BLAS 为各类高性能实现提供统一标准，常见实现包括 OpenBLAS、Intel MKL 等。

BLAS is now included in the LAPACK library.See LAPACK website for Download http://www.netlib.org/lapack

- LAPACK（Linear Algebra PACKage）

LAPACK 基于 BLAS 构建，提供更高层次的线性代数功能，如：

线性方程组求解

矩阵分解（LU、QR、Cholesky）

特征值与奇异值分解（SVD）

LAPACK 是 MATLAB、NumPy、SciPy 等主流科学计算软件的底层核心库。

- **主要特点**

高度优化，支持多种硬件平台。

接口标准化，适用于科学与工程计算。

在高性能计算中广泛应用。

2.1.2 大规模科学计算库

为应对分布式计算和超大规模矩阵处理的需求，出现了面向并行和加速器架构的科学计算库。

ScaLAPACK：在 LAPACK 基础上扩展，用于分布式内存并行环境。采用块循环分布方式实现高效矩阵分解与求解。

cuBLAS：由 NVIDIA 开发的 GPU 加速 BLAS 库，广泛应用于深度学习训练与推理。

MAGMA、PETSc 等库进一步支持 GPU 与 CPU 协同计算，能够处理超大规模稀疏矩阵与多物理场耦合问题。

这些库构成了现代高性能科学计算的核心支柱，使得从桌面计算到超算平台的线性代数计算得以高效扩展。

2.1.3 现代 C++ 矩阵库

C++ 语言的模板与泛型特性推动了高层次矩阵库的发展，这些库在保持高性能的同时，提供了简洁灵活的接口。

Eigen：

现代 C++ 矩阵库的代表，采用表达式模板（Expression Templates）技术，支持矩阵、向量、稀疏矩阵及各种分解算法。

特点：轻量、跨平台、接口友好、自动优化。

Armadillo、Blaze、TNT 等库同样提供高效的线性代数功能，并与标准 C++ 容器和数值类型兼容。

现代 C++ 矩阵库在科研、工业仿真与机器学习等场景中广泛使用，为用户提供了比传统 Fortran 式库更高的易用性与可扩展性。

2.2 示例程序

2.2.1 LAPACK

- **例行程序级别**

LAPACK 中的子例行程序分类如下：

（1）驱动例行程序（driver routines），每个程序解决一个完整的问题，例如求解线性方程组，或计算实对称矩阵的特征值。建议用户如果存在满足其需求的驱动例行程序，应优先使用。它们列在第 2.3 节。

（2）计算例行程序（computational routines），每个程序执行一个独立的计算任务，例如 LU 分解，或将实对称矩阵约化为三对角形式。每个驱动例行程序会调用一系列计算例行程序。用户（特别是软件开发者）可能需要直接调用计算例行程序，以执行驱动例行程序不便完成的单个任务或任务序列。它们列在第 2.4 节。

（3）辅助例行程序（auxiliary routines），又可进一步分类为：

执行分块算法子任务的例行程序——特别是实现算法非分块版本的例行程序；

执行一些常用低级计算的例行程序，例如矩阵缩放、计算矩阵范数，或生成初等 Householder 矩阵；其中一些可能对数值分析人员或软件开发者有用，并可能被考虑纳入未来的 BLAS；

少量对 BLAS 的扩展，例如应用复平面旋转的例行程序，或涉及复对称矩阵的矩阵-向量运算（严格来说，BLAS 本身并不属于 LAPACK）。

驱动例行程序和计算例行程序在本用户指南中均有完整描述，但辅助例行程序则没有。其用户手册附录 B 中给出了辅助例行程序的列表，并附有简要功能描述。

- **数据类型与精度**

LAPACK 为实数和复数数据提供了同样范围的功能支持。

对于大多数计算，LAPACK 都提供了配对的例程，一个用于实数数据，另一个用于复数数据，但也存在少数例外。例如，对应于实数对称不定线性方程组的求解例程，存在针对复数厄米特（Hermitian）方程组和复数对称方程组的例程，因为这两种复数系统类型在实际应用中都会出现。然而，并没有与"求解实对称三对角矩阵的选定特征值"例程直接对应的复数版本，这是因为一个复数厄米特矩阵总是可以被约化为一个实对称三对角矩阵。

实数与复数数据的配对例程在编码实现上，尽可能地保持了二者之间的高度一致性。然而，在某些领域（尤其是不对称特征值问题），这种一致性必然较弱。

LAPACK 中的所有例程都同时提供了单精度和双精度版本。这些双精度版本是通过使用 Toolpack/1 [88] 工具自动生成的。

用于复数矩阵的双精度例程需要非标准的 Fortran 数据类型 COMPLEX*16。在通常进行双精度计算的大多数机器上，该数据类型都是可用的。

- **命名规则**

每个 LAPACK 例程的名称都是对其功能的编码说明（受标准 Fortran 77 的 6 字符长度严格限制）。

所有驱动例程和计算例程的名称形式为 XYYZZZ，其中某些驱动例程的第 6 个字符为空。

第一个字母 X 表示数据类型，含义如下：

S REAL（单精度实数）

D DOUBLE PRECISION（双精度实数）

C COMPLEX（单精度复数）

Z COMPLEX*16 或 DOUBLE COMPLEX（双精度复数）

当我们希望泛指某个 LAPACK 例程（不考虑数据类型）时，将第一个字母替换为“x”。例如，xGESV 泛指 SGESV、CGESV、DGESV 和 ZGESV 这四个例程。

接下来的两个字母 YY 表示矩阵（或最主要矩阵）的类型。大多数这两位代码同时适用于实矩阵和复矩阵；少数仅适用于其中一种，如表 2.1 所示。

Table 2.1: Matrix types in the LAPACK naming scheme

BD bidiagonal

DI diagonal

GB general band

GE general (i.e., unsymmetric, in some cases rectangular)

GG general matrices, generalized problem (i.e., a pair of general matrices)

GT general tridiagonal

HB (complex) Hermitian band

HE (complex) Hermitian

HG upper Hessenberg matrix, generalized problem (i.e a Hessenberg and a triangular matrix)

HP (complex) Hermitian, packed storage

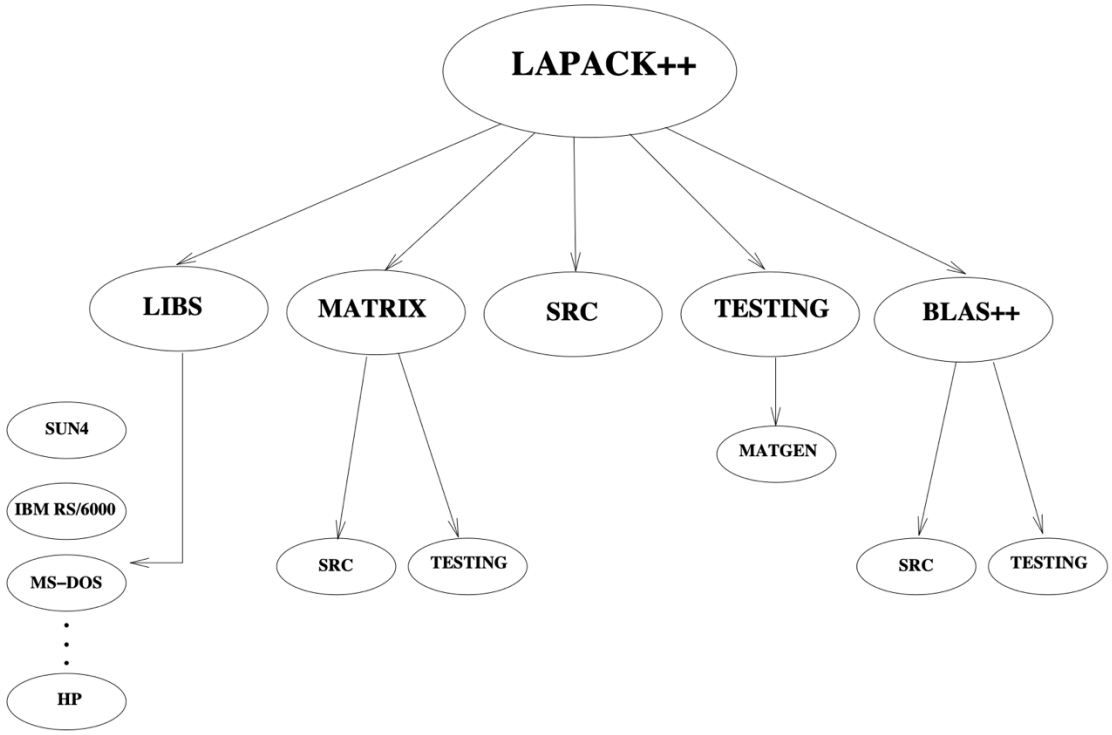
HS upper Hessenberg

OP (real) orthogonal, packed storage

OR (real) orthogonal

PB symmetric or Hermitian positive definite band

PO symmetric or Hermitian positive definite
PP symmetric or Hermitian positive definite, packed storage
PT symmetric or Hermitian positive definite tridiagonal
SB (real) symmetric band
SP symmetric, packed storage
ST (real) symmetric tridiagonal
SY symmetric
TB triangular band
TG triangular matrices, generalized problem (i.e., a pair of triangular matrices)
TP triangular, packed storage
TR triangular (or in some cases quasi-triangular)
TZ trapezoidal
UN (complex) unitary
UP (complex) unitary, packed storage



- main `LAPACK++` directory: C++ matrix libraries `liblamatrix++.a` and LAPACK++ library `liblapack++.a`.
- `MATRIX` subdirectory: Source and testing for LAPACK++ matrices and vectors, including General, Symmetric, SPD, Banded, Triangular cases. This includes only access, assignment, construction, and other non-arithmetic operations.
- `BLAS++` subdirectory: object-oriented extensions to the Basic Linear Algebra Subprograms (BLAS). These call either Fortran or assembly computational kernels of Level 3 BLAS for operations such as matrix multiply, triangular solve, and *rank-k* updates.
- `SRC` subdirectory: main LAPACK++ routines for solving linear systems and linear least-squares. Includes support for symmetric, SPD, banded, tridiagonal, and triangular matrices.
- `TESTING` subdirectory: main testing module call drivers for solving linear systems and linear least squares, together with various matrix decomposition modules for LU, Cholesky (LL^T) and QR factorizations. (See section 4 for details.)
- `LIBS` subdirectory: this is where the LAPACK++ libraries (`lapack.a, blas++.a, matrix++.a`) are for each architecture (e.g. `SUN4`, `RS6K`, etc.).

图 1 Directory structure of LAPACK++ source code for LAPACK++ v. 1.1LAPACK++ v. 1.1

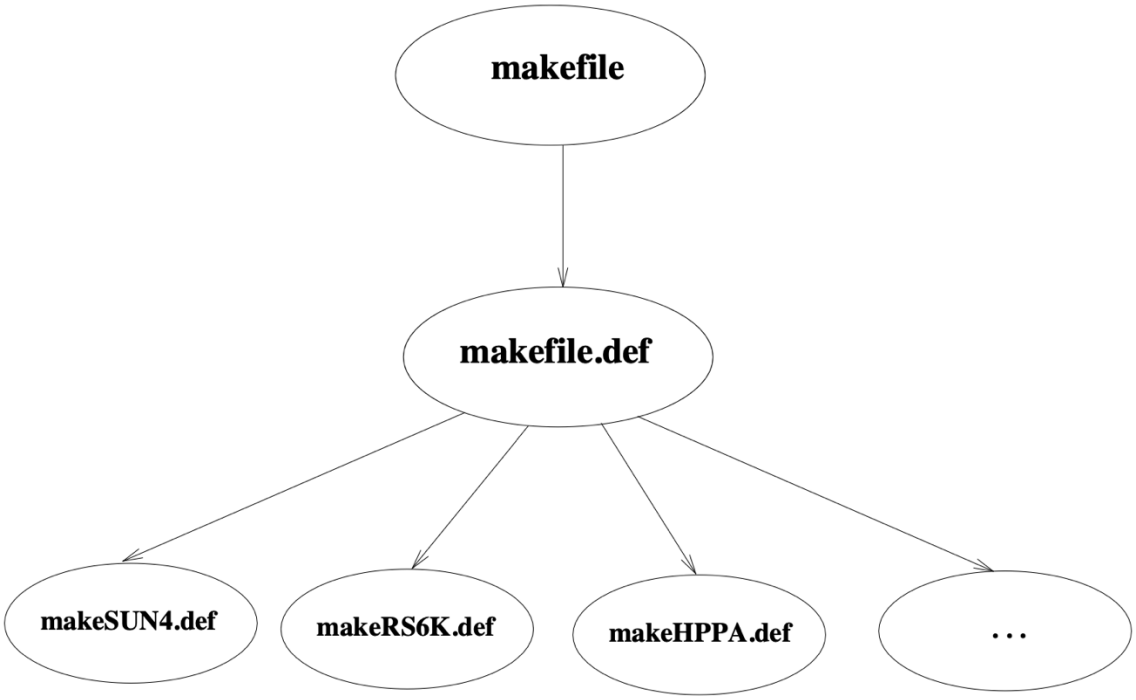


图 2 Makefile structure of LAPACK++ source codes for LAPACK++ v. 1.1LAPACK++ v. 1.1

2.2.2 PETSc

PETSc（Portable, Extensible Toolkit for Scientific Computation，可移植、可扩展的科学计算工具包）是专门为求解科学和工程应用中出现的偏微分方程（PDEs）的数值解而设计的，特别是涉及超大规模稀疏矩阵的求解，属于大规模科学计算库的定位。

https://petsc.org/release/

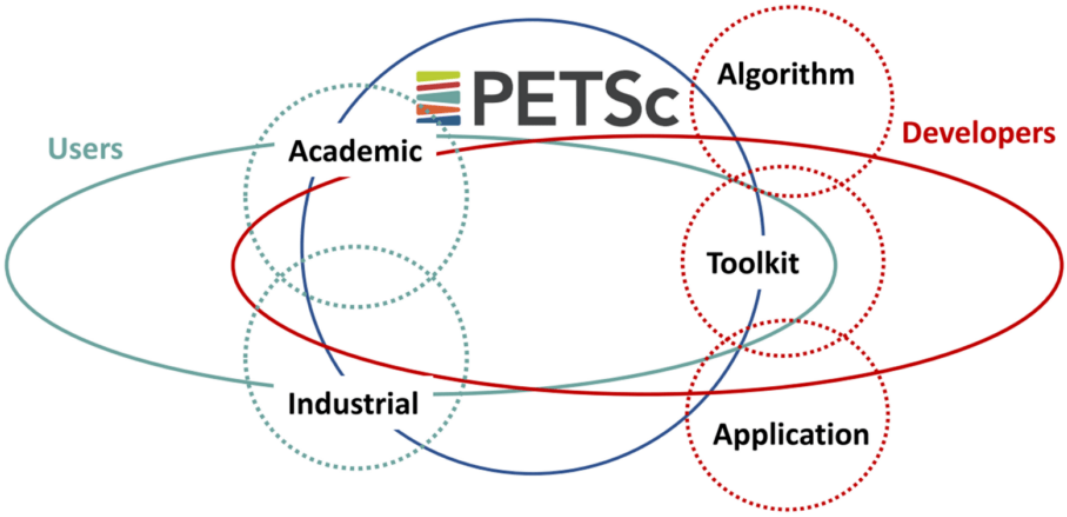


图 3 PETSc connects users and developers.

From:
https://www.exascaleproject.org/highlight/petsc-tao-how-to-create-maintain-and-modernize-a-numerical-toolkit-throughout-decades-of-supercomputer-innovations/

Toolkits/libraries that use PETSc

- ADflow An open-source computational fluid dynamics solver for aerodynamic and multidisciplinary optimization
- BOUT++ Plasma simulation in curvilinear coordinate systems
- Chaste Cancer, Heart and Soft Tissue Environment
- code_aster open-source general purpose finite element code for solid and structural mechanics
- code_saturne open-source general purpose code for fluid dynamics
- COOLFluiD CFD, plasma and multi-physics simulation package
- DAFoam Discrete adjoint solvers with OpenFOAM for aerodynamic optimization
- DAMASK Unified multi-physics crystal plasticity simulation package
- DEAL.II C++-based finite element simulation package
- DUNE-FEM Python and C++-based finite element simulation package
- FEniCS Python-based finite element simulation package
- Firedrake Python-based finite element simulation package
- Fluidity a finite element/volume fluids code
- FreeFEM finite element and boundary element PDE solver with embedded domain specific language
- GetDP a General Environment for the Treatment of Discrete Problems
- Gridap a Julia-based finite element simulation package through the GridapPETSc plugin

- hIPPYlib FEniCS-based toolkit for solving deterministic and Bayesian inverse problems governed by PDEs
- libMesh adaptive finite element library
- MFEM lightweight, scalable C++ library for finite element methods
- MLSVM, Multilevel Support Vector Machines with PETSc.
- MoFEM, An open source, parallel finite element library
- MOOSE - Multiphysics Object-Oriented Simulation Environment finite element framework, built on libMesh.
- OOFEM object-oriented finite element library
- OpenCarp Cardiac electrophysiology simulator
- OpenFOAM Available as an extension for linear solvers for OpenFOAM
- OpenFPM framework for particles and mesh simulation
- OpenFVM finite volume based CFD solver
- PermonSVM support vector machines and PermonQP quadratic programming
- PetIGA A framework for high performance Isogeometric Analysis
- PFLOTRAN An open source, state-of-the-art code for massively parallel simulation of subsurface flow, reactive transport, geomechanics, and electrical resistivity tomography
- PHAML The Parallel Hierarchical Adaptive MultiLevel Project
- preCICE - A fully parallel coupling library for partitioned multi-physics simulations
- PyClaw A massively parallel, high order accurate, hyperbolic PDE solver
- SLEPc Scalable Library for Eigenvalue Problems
- tmm4py Transport Matrix Method for simulating ocean biogeochemical processes

3 自主构建矩阵计算库

在理解矩阵计算库的重要性与分类之后，自主实现一个简易矩阵库具有重要的学习意义。通过从零设计与编码，希望大家能够深入掌握矩阵算法的原理与实现细节，并培养工程化思维。

3.1 教学目标

掌握矩阵加减、乘法、行列式、逆矩阵、线性方程组求解等基本算法。
理解矩阵存储结构（二维数组、一维线性存储等）。
理解算法复杂度与内存访问优化。
具备分析和改进数值计算性能的能力。

3.2 项目功能模块

矩阵类设计：定义矩阵数据结构、构造函数、内存管理及元素访问。

矩阵加减法：实现维度检查与逐元素运算。

矩阵乘法：实现 $O(n^3)$ 算法并分析性能瓶颈。

行列式计算：采用递归展开（本课程要求）或 LU 分解方法（自主提高）。

矩阵求逆：利用伴随矩阵法（本课程要求）、 Gauss-Jordan 消元或 LU 分解实现（自主提高）。

线性方程组求解：基于矩阵分解方法求 Ax = b（自主提高）。

3.3 进一步探索

模板化设计以支持 float、double、complex 等多种数值类型。

探索矩阵乘法的优化与并行化方法。

对比自编矩阵库与 Eigen、BLAS 等成熟库的性能与易用性。

3.4 小结

矩阵计算是人工智能与工业计算的核心基础。从底层 BLAS/LAPACK 到分布式科学计算库，再到现代 C++ 矩阵库，这一体系支撑了当今大部分智能算法与仿真系统。

通过在 C 语言实践课程中自主构建一个矩阵求解器，不仅能帮助学生巩固线性代数与编程技能，更能理解高性能计算软件设计的思想，为未来从事科学计算、人工智能与工程仿真奠定坚实基础。

4 开发思想

在学习的初级阶段,打造一个优雅易用的 C 语言矩阵运算库,涉及预定义数据类型、对象编程、内存管理、输入验证和编程细节。

我们参考 Marc Pony 的博客进行讲解。

https://blog.csdn.net/maple_2014/article/details/119720296

4.1 基本原则

程序开发过程中，以下的一些基本原则：

(1)工具性(toolability):应该使用可用的工具来建立新的应用程序；

(2)可移植性(portability):应用程序应该容易被移植到不同的软件和硬件平台；

(3)可重用性(reusability):程序的编写应该便于重复使用代码段；

(4)可检验性(testability):程序应该简单一致，以便于测试和调试；

(5)可靠性(reliability):对程序运行过程中可能出现的各种错误应该进行合理、一致的处理，以使系统稳定、可靠；

(6)可扩展性(enhanceability):代码必须易于理解，以便可以容易地增加新的功能；

(7)可维护性(fixability):易于找出程序错误的位置；

(8)一致性(consistency):在整个库中，编程习惯应保持一致；

(9)可读性(communicability):程序应该易于阅读和理解;

(10)编程风格(style of programming):代码看起来像书上的数学公式那样以便于读者理解，同时遵循用户友好的编程风格;

(11)易用性(usability):应该使一些非专业的用户也能够方便地使用所开发的库来开发各种更高层次的应用程序;

(12)数值高效性(numerical efficiency):所有函数必须仔细推敲，保证其数值高效性;

(13)基于对象编程(object based programming):避免在函数间传递大量数据，并且使代码易于理解。

4.2 设计思想

(1)采用预定义的数据类型，避免直接使用编译器定义的数据类型

```
typedef  unsigned int ERROR_ID;
typedef int INDEX;
typedef short FLAG;
typedef int INTEGER;
typedef double REAL;
typedef char* STRING;
typedef void VOID;
```

使用预定义的数据类型，有利于程序移植，且可以提高可读性。例如，如果一个系统只支持单精度浮点数，那么只需修改数据类型 REAL 为 float，如果需要更高精度，则只需要改为 long double，达到一劳永逸的效果。定义 INDEX 与 INTEGER 数据类型是为了在编程时区分索引变量与普通整形变量，同样提高可读性。

(2)基于对象编程，定义矩阵对象

```
typedef  struct matrix
{
   INTEGER rows;
   INTEGER columns;
   REAL* p;
   // MatrixType type;   // 矩阵类型（可能后期添加）
   // int isSparseB;
   // ... 未来可以继续扩展属性
}MATRIX;
```

这里，用一级指针而非二级指针指向矩阵的数据内存地址，有诸多原因：

1、对于 n 行 m 列的二维数组 a， a[i][j]与 a[i*m+j]这两种数组元素的访问方式的效率区别;

2、二维数组作为函数参数时，其列数必须是已经确定的常数，这使得函数不通用;

3、如果使用二级指针创建动态二维数组，对堆内存分配是否成功的判断显得特别繁琐，使用一级指针则简洁得多。

基于对象编程使函数接口更加稳定，不需要经常改动。例如，实现一个函数相乘的函数，你的函数接口可能是这样的:

```
void matrix_multiplication(double* A, int rowA, int colA, double* B, int rowB, int colB, double* C)
```

虽然输入参数较多，但一般情况下接口也无需改动。直到有一天，你需要计算维度很大的稀疏矩阵乘法，大量的零元素相乘消耗了大量时间，为了提升计算效率，你需要传入一个参数来表征矩阵是否为大型稀疏矩阵，对稀疏矩阵乘法做一些优化，这时，你的函数接口可能变成这样：

```
void matrix_multiplication(double* A, int rowA, int colA, int isSparseA, double* B, int rowB, int colB, int isSparseB, double* C)
```

函数接口的改变，必然迫使你修改所有的函数调用。使用基于对象编程很好地解决了这个问题，接口可以简洁地定义为：

```
ERROR_ID matrix_multiplication(_IN MATRIX* A, _IN MATRIX* B, _OUT MATRIX* C)
```

无论未来的业务需求需要增加什么功能，只需要对矩阵对象增加属性，修改函数的实现，接口可以永远不变！这对于矩阵运算库，太重要了，因为矩阵运算太常用了，为基础的函数库，接口一旦改变，调用者苦不堪言。

```
/*
// common_defines.h 或 matrix_common.h
// 参数方向标记宏定义
#define _IN       const
#define _OUT
#define _IN_OUT
*/
// 最终效果：创建自解释的 API
ERROR_ID matrix_multiply(_IN MATRIX* left, _IN MATRIX* right, _OUT MATRIX* result);
// 任何人看到这个声明都能立即理解：
// - left 和 right 是输入，不会被修改
// - result 是输出，用于接收计算结果
```

这种设计体现了优秀的软件工程实践：通过编译时检查和人机友好的标记，同时提升代码的可靠性和可读性。

(3) 除了特别编写的内存处理函数(使用栈链表保存、释放动态分配的内存地址)，不允许任何函数直接分配和释放内存

不恰当的分配、使用与释放内存很可能导致内存泄漏、系统崩溃等致命的错误。如果一个函数需动态申请多个内存，那么可能会写出这样啰嗦的程序:

```
double* x = NULL, * y = NULL, * z = NULL;

x = (double*)malloc(n1 * sizeof(double));
if (x == NULL) return -1;

y = (double*)malloc(n2 * sizeof(double));
if (y == NULL)
{
free(x);
x = NULL;
```

```
return -1;
}

z = (double*)malloc(n3 * sizeof(double));
if (z == NULL)
{
free(x);
x = NULL;
free(y);
y = NULL;
return -1;
}
```

为实现动态内存分配与释放，可分 3 步来处理内存申请与释放：

a)在进入一个新的程序时，一个内存堆栈被初始化为空；

b)当需要申请内存时，调用特定的函数来分配所需的内存，并将指向内存的指针存入堆栈中的正确位置；

c)在离开程序时，遍历内存堆栈，释放其中的指针所指向的内存。

程序结构大致如下：

```
STACKS S;
MATRIX* m = NULL;
INTEGER rows = 3, columns = 4;
ERROR_ID errorID = _ERROR_NO_ERROR;

init_stack(&S);

m = creat_matrix(rows, columns, &errorID, &S);
if (m == NULL) goto EXIT;
//do something
// ...

EXIT:
free_stack(&S);
return errorID;
```

(4)防御性编程，对输入参数做有效性检查，并返回错误号

例如输入的矩阵行数、列数应该是正整数，指针参数必须非空等等。这一点在工业级程序中特别重要。

(5)注意编程细节的打磨

这个一般都有不同单位或者项目的相应程序开发规范来指导。比如我硕士课题参与的一个工程项目，即是如此。

a)操作符(逗号，等号等)两边必须空一格；
b)逻辑功能相同的程序间不加空行，逻辑功能独立的程序间加空行；
c)条件判断关键字(for if while 等)后必须加一空格，起到强调作用，也更清晰；
d)函数内部定义局部变量后，必须空一行后再编写函数主体。

5 从零开始

5.1 总体设计

5.2 创建一个矩阵结构体

5.2.1 第一种结构体：double** data (二级指针)

```
C
typedef struct {
    size_t rows;
    size_t cols;
    double** data;   // 二级指针：指针的指针
} Matrix;
```

优点：

1. 直观的索引方式： 可以像使用标准二维数组一样，直接通过 mat.data[i][j] 来访问元素，非常符合人类和数学习惯。

2. 灵活的行大小： 理论上每行可以独立分配内存，允许创建不规则矩阵（虽然矩阵运算中很少见）。

缺点（致命缺陷）：

1. 内存不连续：每一行的数据都是单独 malloc 分配的,它们在内存中是分散的。

2. 性能低下： 这种分散导致 CPU 缓存效率极低。当程序访问元素时，CPU 必须多次跳转内存地址，无法利用现代 CPU 的缓存预取机制，导致矩阵遍历和运算速度变慢。

3. 两次解引用： 访问 mat.data[i][j] 需要两次指针解引用（一次找到行指针，一次找到数据），增加了开销。

5.2.2 第二种结构体：REAL *data (一维指针)

```
C
typedef struct
{
    INTEGER row,column;
    REAL *data;  // 一维指针：使用一维数组存储矩阵元素数据
}Matrix;
```

优点（主流库的选择）：

1. 内存高度连续（核心优势）： 整个矩阵的数据只需要调用一次 malloc 或 calloc，所有元素在内存中是连续存放的。

2. 极高的效率：连续内存极大地提升了 CPU 缓存命中率，对大规模矩阵的遍历、线性代数运算（如矩阵乘法）的速度有决定性的优化作用。

3. 易于传递和操作： 连续内存块便于与外部库（如 BLAS, CUDA）进行数据交换。

4. 一次解引用： 访问元素时只需要一次指针解引用。

缺点：

1. 不直观的索引： 必须使用转换公式 index = (i - 1) * cols + (j - 1) 来访问元素，不符合直觉。

5.2.3 总结与建议

结构体	存储方式	内存连续性	访问效率	适用场景
第一种	二级指针 (**)	**不连续**	低	小型、对性能要求不高的教学示例
第二种	一维指针 (*)	**高度连续**	高	**实际的数值计算、高性能计算、矩阵库**

5.3 接下来的任务

表 2 单元安排

单元	主题	基础任务	提高任务
1 第 9 周	工业背景与环境搭建	• 矩阵库应用背景 • 内存管理 • 矩阵结构体定义	• 内存泄漏检测
2 第 10 周	模块化编程	• 项目框架设计 • 头文件设计 • 防御性检查	• 错误编码系统 • Quiz（结构体）
3 第 11 周	矩阵基本运算	• 矩阵加法 • 矩阵减法 • 矩阵转置 • 标量乘法	• 矩阵运算的 SIMD 向量化优化
4 第 12 周	矩阵乘法	• 三重循环实现 • 理解时间复杂度	• 内存访问模式优化
	课程第三次作业		
5 第 13 周	矩阵求逆（1）	• 矩阵代数余子式 • 矩阵行列式	• 递归算法的时间复杂度分析与优化策略
6 第 14 周	矩阵求逆（2）	• 伴随矩阵 • 矩阵求逆	• 矩阵求逆替代算法（如 LU 分解）
7 第 15 周	综合测试	• 功能验证 • 性能测试	• 算法优化
8 第 16 周	项目答辩	• 项目报告文档 • 现场答辩	