Chapter 1. Basic Linear Algebra

1 基本数据对象

ppx是一组数值计算方法的程序集合，它包括但不限线性代数运算、非线性优化、统计分析、信号处理等等方面。以上功能的绝大多数操作对象都是一般线性群$GL(n)$，因此ppx库最根本的数据容器对象是对$GL(n)$的表示，即矩阵。

本章论述ppx中一般线性群及其子群的表示与程序实现方法。

1.1 一般线性群的表示

矩阵是通常的一般线性群表示，强调群的属性是为了与将要引入的李群这一代数对象做区分。矩阵群或说一般线性群（$GL(n)$）最基本功能包括了矩阵代数运算、矩阵分解、矩阵特征值问题等。

1.1.1 矩阵对象

类型定义

```
#include "matrixs.hpp"
template <std::size_t M, std::size_t N>
class MatrixS : public details::MatrixBase<M, N>
```

1. MatrixS 是储存了一组浮点数的拥有符合STL标准随机访问迭代器的连续容器。

2. MatrixS根据储存元素数量，在栈或堆上分配内存。该界限由常量 *gl_sm_limit* 决定。

MatrixS继承自details::MatrixBase类，在MatrixBase的模板参数中，第一与第二参数决定存储元素数量，而第三参数由第一第二参数决定。MatrixBase会根据第三参数选择特定的偏特化模板：在不同的偏特化模板中，MatrixBase使用的存储容器并不相同。

MatrixS的内存布局由基类实现。它使储存元素连续分布于内存中，这样就能被迭代器或裸指针加上偏移访问。当存储元素数量小于 *gl_sm_limit* 时，这些元素在栈上：它们有平凡构造、赋值、移动构造、移动赋值能力；当储存元素数量大于 *gl_sm_limit* 时，这些元素在堆上：它们有默认非平凡构造、赋值、移动构造、移动赋值能力。

MatrixS类本身实现了群元之间的各类代数操作：群加法、求逆、标量运算等等。此外MatrixS中编码了一些用来进行类型计算的标识符以便进行编译期的惰性求值。

MatrixS中常见操作的复杂度如下:

- 随机访问 *O(1)*
- 末尾安插或移除元素 *O(1)*
- 安插或移除元素 *O(n)*

模板参数

参数	约束	说明
M	size_t	非类型模板参数，矩阵行数
N	size_t	非类型模板参数，矩阵列数

成员方法

构造方法	
constructor	默认构造/移动构造
destructor	默认析构
operator=	默认赋值/移动赋值
元素访问	
operator()	访问处于 i,j 处元素
operator[]	按数组访问处于 idx 处元素
data	返回存储容器裸指针
sub	返回从 i,j 处开始，大小为 M,N 的子矩阵视图
迭代器	
begin/cbegin	返回开始的迭代器/常量迭代器
end/cend	返回结束的迭代器/常量迭代器
容量	
size	返回储存元素数量
row_counts	返回矩阵行数
col_counts	返回矩阵列数
元素修改	
fill	将容器元素全部改写为 val
算术操作	
operator+	二元加法：接受相容矩阵/标量/求值模板
operator-	二元减法：接受相容矩阵/标量/求值模板
operator*	二元乘法：接受相容矩阵/标量
operator/	二元除法：接受标量
operator+=	复合赋值：接受相容矩阵/标量
operator-=	复合赋值：接受相容矩阵/标量
operator*=	复合赋值：接受标量
operator/=	复合赋值：接受标量
operator==	比较运算符，容器中每个元素差值小于 gl_rep_eps 相等
T	转置
I	逆（方阵）/伪逆（非方阵）
det	行列式（方阵）
trace	迹（方阵）
静态方法	
eye	构造单位矩阵
zero	构造零矩阵
diag	按输入构造对角矩阵

非成员方法

矩阵相关非成员函数存在于头文件"linalg.hpp"中：

矩阵操作	
zeros	将矩阵元素全置为零
ones	将矩阵元素全置为1
cofactor	矩阵余子式
adjugate	伴随矩阵
determinant	矩阵行列式
inverse	矩阵逆
slice	返回矩阵处于 i_1, i_2 行与 j_1, j_2 列之间的视图切片
transpose	矩阵转置
norm2	向量范数（2范数）
norminf	向量范数（无穷范数）
inner_product	向量内积
maxloc	返回最大元素索引
trace	矩阵迹
三角分解	
ludcmp	PLU分解，LAPACK式接口
qrdcmp	QR分解，LAPACK式接口
svdcmp	SVD分解
线性求解器	
ludbksb	适用于LU分解的反向提交
qrsolv	适用于QR分解的反向提交
svbksb	适用于SVD分解的反向提交
linsolve	解线性方程组（方阵）/解最小二乘方程组（列满秩）
pinv	（最小二乘）伪逆

1.1.2 矩阵视图

在MatrixS对象上使用sub接口时，将激活代理操作。此时的sub返回结构体MatrixS内部类SubMatrix：它储存了切片操作的范围，并将对MatrixS的引用语义转移到自身，所以SubMatrix是MatrixS的操作代理类型。

该类型仅能由sub创建并返回，并且参与到模板类型计算中，外部无法对其进行移动和代数运算以外的操作。

成员方法

构造方法	
constructor	唯一，接受MatrixS，索引范围（不可访问）
destructor	默认析构（不可访问）
operator=	默认赋值/移动赋值（不可访问）
operator=	从泛矩阵对象对象赋值

1.2 代数运算的表示

ppx的矩阵计算中实现了惰性求值，但相关的工具都遮蔽于details命名空间中，不对外开放。

表达式模板（Expression Templates），以及与其相应的惰性求值（Lazy Evaluation）技术是C++数值计算中关键部分之一。C++常见线性代数库，如uBLAs[1]与Eigen[2]都重度依赖此特性。表达式模板也是C++在高性能领域能接近C与Fortran计算效率的重要工具。然而Eigen中表达式模板泛滥也带来了困扰：大大增加了诸如Openfoam或Moose这些基于Eigen开发的偏微分方程数值库的抽象成本[3]。ppx有限度的使用了表达式模板技术。关于更多表达式模板知识与细节参考*C++ Templates*[4]，此处只简单介绍ppx的实现。

1.2.1 类型表达式expr

定义

```
#include "exprtmpl.hpp"
namespace details
{
    template <typename T>
    class expr
    {
        public:
            using expr_type = expr<T>;
            const T &self() const { return static_cast<const T &>(*this); }
            T &self() { return static_cast<T &>(*this); }

        protected:
            explicit expr(){};
            constexpr size_t size() { return self().size_impl(); }
            auto operator[](size_t idx) const { return self().at_impl(idx); }
            auto operator()() const { return self()(); };
    };
}
```

expr是存在于ppx::details命名空间内的表达式模板基类。因为该类型会大量构造，所以它使用奇异模板递归（CRTP）实现多态，同时避免虚继承开销。它是表达式模板类必须最小特性集合。

成员方法

构造方法	
constructor	默认构造（不可访问）
成员方法	
self	返回自身（子类）
size	返回元素数量（子类）
operator[]	求值
operator()()	隐式类型转换（子类）

1.2.2 终点表达式expr_result

定义

```
#include "exprtmpl.hpp"
namespace details
{
    template <typename T>
    class expr_result : expr<expr_result<T>>;
}
```

expr_result是普通类型接入模板表达式的起点。它继承自expr，这使它能参与模板计算；另一方面它内部有一个类型萃取器：对于值类型保存值得副本，对于引用类型保存其引用。它内部的数据成员是对普通类型得包装，也是模板计算终点。

它唯一数据成员是：

```
typename expr_traits<T>::ExprRef value;
```

expr_traits是简单实现的类型萃取[5]：

```cpp
template <typename T>
struct expr_traits
{
    using ExprRef = T const &;
};

template <typename T>
struct expr_traits<expr_scalar<T>>
{
    using ExprRef = expr_scalar<T>;
};
```

这种萃取使除expr_scalar类型外作为引用储存，对于expr_scalar作为拷贝储存。

成员方法

构造方法	
constructor	默认构造
destructor	默认析构

1.2.3 标量表达式expr_scalar

定义

```cpp
#include "exprtmpl.hpp"
namespace details
{
    template <typename T>
    struct expr_scalar;
}
```

expr_scalar是为了解决表达式模板中标量无法实现operator[]和size函数而实现的包装类，这一技术来自于*C++ Templates*[4]。例如：数组的逐元运算中进行惰性求值，在遍历表达式树调用operator[]求值遇到标量就会出错。因此所有参与模板计算的标量都使用expr_scalar进行包装。

成员方法

构造方法	
constructor	默认构造
destructor	默认析构
成员方法	
operator[]	返回标量值
size	返回1

1.2.4 表达式二元算符biops

定义

```cpp
#include "exprtmpl.hpp"
namespace details
{
    template <typename Ops, typename lExpr, typename rExpr>
    class biops : public expr<biops<Ops, lExpr, rExpr>>;
}
```

biops是bi-operations的缩写，即二元算符。它也继承自expr，因此有参与模板计算能力，同时它也是模板计算中二元运算惰性求值的核心。因此biops也引申出了如下的多个仿函数：

```
struct expr_plus_t
{
    constexpr explicit expr_plus_t() = default;
    template <typename LType, typename RType>
    auto operator()(const LType &lhs, const RType &rhs) const
    {
        return lhs + rhs;
    }
};
```

这种仿函数在模板中代表了用户实际输入的operator+算符；或说是operator+的模板表示类型。

模板参数

参数	约束	说明
Ops	二元算符	类型参数，代表二元运算
lExpr	表达式	类型参数，二元运算左操作数
rExpr	表达式	类型参数，二元运算右操作数

biops类型在构造时会把二元运算类型、左右操作数这些参数储存至自己的模板参数中，在真正调用operator[]算符求值时，通过遍历自身模板参数，实现计算。

成员方法

构造方法	
constructor	默认构造
destructor	默认析构
成员方法	
operator[]	弹出操作数，调用二元算符
size	返回左右操作数size较大者
operator+	储存plus算符，结合右操作数，返回新模板
operator-	储存plus算符，结合右操作数，返回新模板
operator*	储存plus算符，结合右操作数，返回新模板
operator/	储存plus算符，结合右操作数，返回新模板

2 线性代数运算

除一般线性群的基本特征以外，ppx还需要实现三角分解、特征值问题等线性代数相关算法。

2.1 代数余子式

2.1.1 函数原型

```
#include "linalg.hpp"
template <size_t M, size_t N>
Matrix<M - 1u, N - 1u> cofactor(const Matrix<M, N> &mat, size_t p, size_t q)
```

ppx::cofactor 求解 (p,q) 元素的余子式，余子式定义可参考Wiki百科[6]，ppx使用定义求解余子式。

2.2 LU分解

2.2.1 函数原型

```
template <size_t N>
Matrix<N, N> ludcmp(Matrix<N, N> A, std::array<int, N> &indx, bool &even, bool &sing)
```

ppx::lucmp实现了Lapack[7]式选主元LU分解，即PLU分解，详细知识可参考*Numerical analysis*[8]内容。

2.2.2 用户接口

参数	说明
A	M*N输入矩阵
indx	置换数组，记录被交换的行
even	置换次数是否为偶数次
sing	分解是否奇异
Return	M*N矩阵：上三角部分储存L，下三角部分储存U

2.2.3 程序实现

对于满秩方阵A，A可被写作 $A = LU$ ：其中 L 是上三角矩阵 U 是下三角矩阵。为避免数值奇异，程序中更多使用选主元LU分解，即PLU分解：$A = PLU$，其中 P 是置换矩阵，LU 的定义保持不变。

LU 分解本质上是高斯消元法的另一种表示。将A通过初等行变换变成一个上三角矩阵，其变换矩阵就是一个单位下三角矩阵。这也是杜尔里特算法（Doolittle algorithm）：从下至上地对矩阵A做初等行变换，将对角线左下方的元素变成零，然后再证明这些行变换的效果等同于左乘一系列单位下三角矩阵，这一系列单位下三角矩阵的乘积的逆就是L矩阵，它也是一个单位下三角矩阵。LU算法的复杂度是$\frac{2n^3}{3}$左右。

选主元过程：通过std::swap进行列交换，使用数组indx记录置换矩阵。

消去过程：

$$\beta_{ij} = a_{ij} - \sum_{k=1}^{i-1} \alpha_{ik}\beta_{kj}$$

$$\alpha_{ij} = \frac{1}{\beta_{jj}}(\alpha_{ij} - \sum_{k=1}^{j-1} \alpha_{ik}\beta_{kj})$$

代码实现细节参考文献[9]。

2.3 QR分解

2.3.1 函数原型

```
template <size_t M, size_t N>
Matrix<M, N> qrdcmp(Matrix<M, N> a, Matrix<N, 1> &c, Matrix<N, 1> &d, bool &sing)
```

ppx实现了Lapack式基于householder变换的QR分解，

2.3.2 用户接口

参数	说明
a	M*N输入矩阵
c	N维向量，householder变换系数
d	N维向量，R对角线元素
sing	分解是否奇异
Return	M*N矩阵，上三角部分储存R，剩余每列为householder变换因子

2.3.3 程序实现

对于列满秩矩阵A，可以写作 $A = QR$： 这里Q是正交矩阵（$Q^TQ = I$）而R是上三角矩阵。如果A非奇异，且限定R的对角线元素为正，则该因数分解唯一。

QR分解中对列向量消去过程可以使用Givens变换、householder变换，以及Gram-Schmidt正交化等方法[8]。对于稠密小矩阵，householder变换或说镜射变换时最有效。

$$\begin{bmatrix} \mathbf{q}_1 & \cdots & \mathbf{q}_n & \mathbf{q}_{n+1} & \cdots & \mathbf{q}_m \end{bmatrix} \begin{bmatrix} r_{11} & \cdots & r_{1n} \\ & \ddots & \vdots \\ & & r_{nn} \\ 0 & \cdots & 0 \\ \vdots & & \vdots \\ 0 & \cdots & 0 \end{bmatrix} = A(M \times N)$$

QR分解核心在于对每列使用变换将其转换为上Hessenberg矩阵R，变换矩阵乘积为Q，即：

$$Q_1 A = \begin{bmatrix} \alpha_1 & \star & \dots & \star \\ 0 & & & \\ \vdots & & A' & \\ 0 & & & \end{bmatrix}$$

$$R = Q_n \dots Q_1 A$$

$$Q = Q_1^T \dots Q_n^T$$

理论算法参考文献[10]：

$$\begin{aligned} &1: \quad \text{Householder QR}(A) \\ &2: \quad m, n \leftarrow \text{shape}(A) \\ &3: \quad R \leftarrow \text{copy}(A) \\ &4: \quad Q \leftarrow I_m \\ &5: \quad \text{for } k = 0 \dots n-1 \text{ do} \\ &6: \quad \mathbf{u} \leftarrow \text{copy}\left(R_{k:,k}\right) \\ &7: \quad u_0 \leftarrow u_0 + \text{sign}\left(u_0\right)\|\mathbf{u}\| \\ &8: \quad \mathbf{u} \leftarrow \mathbf{u}/\|\mathbf{u}\| \\ &9: \quad R_{k:,k:} \leftarrow R_{k:,k:} - 2\mathbf{u}\left(\mathbf{u}^\top R_{k:,k:}\right) \\ &10: \quad Q_{k:,:} \leftarrow Q_{k:,:} - 2\mathbf{u}\left(\mathbf{u}^\top Q_{k:,:}\right) \\ &11: \quad \text{return } Q^\top, R \end{aligned} \tag{1}$$

这种朴素的实现方式的优点是简单易懂，缺点是生成大量中间变量。事实上，在Lapack中，QR分解将返回：

$$\begin{bmatrix} u_{11} & r_{11} & r_{21} & \dots \\ u_{12} & u_{21} & r_{22} & \dots \\ u_{13} & u_{22} & u_{31} & \dots \\ u_{14} & u_{23} & u_{32} & \dots \\ \dots & \dots & \dots & \dots \end{bmatrix}$$

即上三角矩阵 R 存储于返回矩阵的上三角部分，镜射变换因子 u_i 储存于下三角中，而 R 的对角线单独返回。对于需要 Q 的情形，使用变换因子生成：

$$Q_i = I - c_i u_i u_i^T$$

如不显式需要Q，则QR分解中每次镜射变换因子生成变换矩阵作用在A上过程变为：

$$Q_i x = (I - c_i u_i u_i^T)x = x - c_i u_i u_i^T x = x - c_i u_i (u_i^T x)$$

这种方法也称为隐式QR。

2.4 SVD分解

2.4.1 函数原型

```
template <size_t M, size_t N>
Matrix<M, N> svdcmp(Matrix<M, N> u, Matrix<N, 1> &w, Matrix<N, N> &v)
```

ppx实现了基于QR迭代的SVD分解。

2.4.2 用户接口

参数	说明
u	M*N输入矩阵
w	N维向量，奇异值
v	N*N右奇异矩阵
Return	M*N矩阵，左奇异矩阵

2.4.3 程序实现

假设A是一个矩阵，存在一个分解使得： $A = U\Sigma V^T$ ，其中U是$M\times N$阶列正交矩阵；Σ是$M\times M$阶对角矩阵；而V是$N\times N$阶酉矩阵。这样的分解就称作A的奇异值分解。Σ对角线上的元素$\Sigma_{i,i}$即为A的奇异值。常见做法是将奇异值由大而小排列，如此Σ便能由A唯一确定（但ppx并没有重排）。

SVD分解有多种实现方式，如双边Jacobi，分治法等。ppx实现了经典的Golub–Reinsch方法[11]，该方法先将矩阵A通过householder变换转化双对角矩阵，然后在每个对角块中求解奇异值。

双对角householder变换：

对于矩阵A，有如下变换：

$$A = UBV^T, U \in \mathbb{R}^{n\times n}, V \in \mathbb{R}^{m\times m}$$

其中：

$$B = \begin{bmatrix} \widehat{B} \\ 0 \end{bmatrix} \in \mathbb{R}^{n\times m},$$

$$\widehat{B} = \begin{bmatrix} \psi_1 & \phi_1 & 0 & \cdots & 0 \\ 0 & \psi_2 & \phi_2 & & \\ \vdots & & \ddots & \ddots & \\ & & & \psi_{m-1} & \phi_{m-1} \\ 0 & & & & \psi_m \end{bmatrix} \in \mathbb{R}^{m\times m},$$

则B也写作：

$$B = U_m \cdots U_1 A V_1 \cdots V_{m-2}$$

其中 U_i, V_i都是householder变换。

Golub–Reinsch方法:

反复使用双对角householder变换:

$$\begin{bmatrix} B \\ 0 \end{bmatrix} \leftarrow (U_1 \dots U_n)^T A \, (V_1 \dots V_{n-2})$$

将矩阵变换为下列形状:

$$B = \begin{bmatrix} B_{11} & 0 & 0 \\ 0 & B_{22} & 0 \\ 0 & 0 & B_{33} \end{bmatrix}$$

然后令$T = \hat{B}^T \hat{B}$，选择其右下角 2×2 方阵求解特征值:

$$(\lambda_1, \lambda_2) = eigenvalue(T_i)$$

取 $\mu = min(\lambda_1, \lambda_2)$ ，作为QR迭代求取特征值偏移，最后遍历所有特征值:

$$\begin{aligned} &1: T = \hat{B}^T \hat{B} \\ &2: \text{ for } k = 0, 1, \dots \text{ do} \\ &3: \text{ Determine shift } \mu \\ &4: \quad UR = T - \mu I \text{ (QR)} \\ &5: \quad T = RU + \mu I \\ &6: \text{ end for} \end{aligned}$$

具体细节参考文献[12]。

2.5 求解线性方程组问题

2.5.1 definition

```
template <factorization type, size_t M, size_t N>
Matrix<N, 1> linsolve(const Matrix<M, N> &A, Matrix<M, 1> b)
```

ppx实现了基于矩阵分解的线性方程组求解器，它适用于超定与一般方程组。

2.5.2 用户接口

参数	说明
A	M*N输入矩阵
b	M维输入向量
type	枚举，求解方法（LU/QR/SVD）
Return	N维矩阵，解

2.5.3 程序实现

线性方程求解器面临的主要问题是如何处理超定方程组，ppx中使用最小二乘法来解决过约束问题。

对于一个矩阵 A :

- 方阵且满秩，有三角分解 $A = LU$
- 列满秩，有QR分解 $A = QR$
- 普通矩阵，有SVD分解 $A = U \Sigma V^T$

因此，基于三种分解模式可以处理绝大部分线性方程组求解。

完全约束方程组

LU分解情形：$Ux = L^{-1}b$，即连续求解两个三角方程组。

QR分解情形：$Rx = Q^Tb$，将householder变换因子作用到b上，求解三角方程组。

SVD分解情形：$x = Vdiag(\frac{1}{\sigma_i})U^Tb$，相应矩阵相乘得解。

过约束方程组

过约束方程组没有解，因此转换为求解最小二乘问题 $||Ax - b||$：

对A进行QR分解，由于：

$$A = QR = (Q_1, Q_2)\begin{bmatrix} R_1 \\ 0 \end{bmatrix} = Q_1R_1$$

$$Ax = Q_1R_1x = b \Rightarrow R_1x = Q_1^Tb$$

记Housholder中每次变换因子 H_s （该因子将每一列相应位置变零），有如下过程：

$$H_sH_{s-1}\dots H_1A = R = \begin{bmatrix} R_1 \\ 0 \end{bmatrix}$$

$$H_sH_{s-1}\dots H_1b = Q^Tb = \begin{bmatrix} Q_1^Tb \\ Q_2^Tb \end{bmatrix} = \begin{bmatrix} c_1 \\ c_2 \end{bmatrix}$$

由于 Q^T 正交，有以下问题同解：

$$||Ax - b||_2^2 = ||Q^T(Ax - b)||_2^2 = ||\begin{bmatrix} R_1x - c_1 \\ c_2 \end{bmatrix}||_2^2 = ||R_1x - c_1||_2^2 + ||c_2||_2^2$$

即原方程 $Ax = b$ 的最小二乘问题等价于$R_1x = c_1$ ，即对于列满秩的超定方程组，使用规约QR得到解就是相应问题的最小二乘解。

3 参考文献

1. Boost C++ Libraries ↩

2. https://eigen.tuxfamily.org ↩

3. https://www.osti.gov/servlets/purl/1601096 "Hoemmen, M. F., Badwaik, J., & Brucher, M. (2019). P1417: Historical lessons for C++ linear algebra library standardization (No. SAND2019-1648C). Sandia National Lab.(SNL-NM), Albuquerque, NM (United States)." ↩

4. https://www.speedytemplate.com/forms/general-book-template-2.pdf "Vandevoorde, D., & Josuttis, N. M. (2002). C++ Templates: The Complete Guide, Portable Documents. Addison-Wesley Professional." ↩ ↩

5. https://www.researchgate.net/profile/Martin-Sulzmann/publication/228738465_C_templatestraits_versus_Haskell_type_classes/links/09e415110019081e10000000/C-templates-traits-versus-Haskell-type-classes.pdf "Kothari, S., & Sulzmann, M. (2005). C++ templates/traits versus Haskell type classes. Technical Report TRB2/05, The National Univ. of Singapore, 2005.→ 1 citation on page: 119." ↩

6. Minor (linear algebra) - Wikipedia ↩

7. https://netlib.org/lapack ↩

8. https://dl.acm.org/doi/abs/10.5555/2161609 "Sauer, T. (2011). Numerical analysis. Addison-Wesley Publishing Company." ↩ ↩

9. "Darst, R. B. (1990). Introduction To Linear Programming, Applications and Extensions, Mercel Dekker. Inc., New York." ↩

10. http://ecet.ecs.uni-ruse.bg/cst04/Docs/sIIIA/37.pdf "Stoilov, T., Stoilova, K. (2004, June). Algorithm and software implementation of QR decomposition of rectangular matrices. In *CompSysTech* (pp. 1-6)." ↩

11. https://link.springer.com/chapter/10.1007/978-3-662-39778-7_10 "Golub, G. H., & Reinsch, C. (1971). Singular value decomposition and least squares solutions. In Linear algebra (pp. 134-151). Springer, Berlin, Heidelberg." ↩

12. https://www.mdpi.com/2079-9292/10/1/34 "Alessandrini, M., Biagetti, G., Crippa, P., Falaschetti, L., Manoni, L., & Turchetti, C. (2020). Singular value decomposition in embedded systems based on arm cortex-m architecture. Electronics, 10(1), 34." ↩