

浙江大学

本科生毕业论文（设计）



题目 稀疏矩阵存储的 C++ 实现

姓名与学号 唐周若愚 3110104942

指导教师 王何宇 副教授

年级与专业 信息与计算科学 2011 级

所在学院 数学系

浙江大学本科生毕业论文（设计）诚信承诺书

1. 本人郑重地承诺所呈交的毕业论文（设计），是在指导教师的指导下严格按照学校和学院有关规定完成的。
2. 本人在毕业论文（设计）中引用他人的观点和参考资料均加以注释和说明。
3. 本人承诺在毕业论文（设计）选题和研究内容过程中没有抄袭他人研究成果和伪造相关数据等行为。
4. 在毕业论文（设计）中对侵犯任何方面知识产权的行为，由本人承担相应的法律责任。

班级 信息与计算科学 1102 学号 3110104942

毕业论文（设计）作者签名 _____

2015 年 05 月 15 日

摘 要

在现代的科学计算与工程中，由于经常会对大规模甚至是超大规模的矩阵进行运算，传统的全矩阵运算已经无法满足实际的需求。为了能对这些矩阵进行数值运算，利用矩阵的稀疏性的稀疏矩阵便被创造了出来，大大地缩减了计算的复杂度。而本文则是旨在建立一个底层的稀疏矩阵存储与计算的 C++ 库，供计算使用者使用，方便快捷地实现稀疏矩阵的运算而无需顾及稀疏矩阵的底层存储。

关键词： 稀疏矩阵、C++、存储

Abstract

In modern scientific computation and industrial projects, the traditional dense matrix cannot satisfy the increasing requirement of calculating large scale matrixes. In order to be able to calculate such matrixes, sparse matrix is created by taking advantage of the sparsity. The complexity of calculating is reduced in a large amount by using sparse matrix. The purpose of this paper is to create a C++ library to storage the sparse matrix for programmers who want to use sparse matrix. With this library, programmers can calculate sparse matrix easily and quickly without thinkink about the storage of the sparse matrix.

Keywords: Sparse Matrix;C++;Storage

目 录

摘要	i
Abstract	ii
目录	iii
第一章 引言	1
第二章 研究背景综述	2
2.1 稀疏矩阵的定义	2
2.2 稀疏矩阵已有存储方式	2
2.2.1 三元组 (Triplet) 存储	2
2.2.2 十字链表 (Orthogonal List) 存储	3
2.2.3 列压缩 CSS(Compressed-column storage) 存储	4
2.2.4 行压缩存储方式 (Compressed Row Storage)	5
2.3 已有的处理稀疏矩阵存储与运算的开源库	7
2.4 稀疏矩阵的数乘	8
2.5 稀疏矩阵的矩阵向量乘法	9
2.6 稀疏矩阵的矩阵矩阵乘法	9
第三章 研究的目标、内容及研究方法	11
3.1 研究的目标	11
3.2 研究内容	11
3.3 研究方法	11
第四章 实验结果	13
第五章 结论	16
附录 A sparseMatrix 库	17
A.1 sparseMatrix.h	17
A.2 tmpSparse.h	31

A.3	<code>sparseVector.h</code>	34
A.4	<code>complexNumber.h</code>	35
A.5	<code>complexSparseMatrix.h</code>	37
A.6	<code>tmpComplexSparse.h</code>	53
A.7	<code>complexSparseVector.h</code>	55
参考文献		59
致谢		60

第一章 引言

在实际工程计算中，尤其是在微分方程数值解、线性规划等的有限元分析中，经常出现高阶的矩阵运算，经常出现百万阶、千万阶的矩阵运算，而如果使用全矩阵运算，假设是百万阶的 float 类型的矩阵，则需要 $4 * 10^{12}B$ 来存储，也就是 4TB 的空间，这显然是无法被微型计算机甚至是工作站所接受的。但是通常这些矩阵具有稀疏的性质，拥有着大量的零元，而且通常阶数越高稀疏度越高。这就可以在相当大的程度上减少了存储对于硬件的压力。同时，在计算中，大量对于 0 的运算是无谓的消耗资源的操作，也可以利用稀疏矩阵来避免这些无谓的操作。而在本研究中编写的稀疏矩阵库可以用来方便快捷的实现稀疏矩阵的存储以及一些基本运算，避免了使用者大量编写存储底层的代码，提高开发效率。

第二章 研究背景综述

近年来，在工程应用中，求解高阶矩阵的需求日益增长，全矩阵运算脱离了实际的硬件限制，为了满足这一日益增长的需求，同时这些矩阵通常都有着一个特征——非零元远少于零元，稀疏矩阵这门学科便应运而生。在 20 世纪 60 年代研发电子网络的电子工程师们是最早的去利用稀疏性来应用稀疏矩阵进行工程上的计算的。[\[1\]](#)

而在微分方程数值解、线性规划等的有限元分析中，经常出现求解高阶稀疏线性方程组，如利用全矩阵进行存储，则需要 n^2 的空间复杂度和 n^3 的乘法运算时间复杂度，显然，这种程度的运算量是无法被微型计算机，甚至是工作站所接受的。而利用矩阵的稀疏性，可以有效地减小消耗很多无谓的存储空间以及无谓的计算，在很大的程度上降低了时间和空间复杂度，降低了计算对硬件的需求，使计算成为可能。

2.1 稀疏矩阵的定义

稀疏矩阵是非零元远小于矩阵元素总数，且非零元分布没有规律的矩阵。[\[1\]](#)

2.2 稀疏矩阵已有存储方式

在这些年的发展，出现了很多的存储方法，比如：对角线存储法、对称矩阵的变带宽存储法、坐标存储法、Elipack-Itpack 存储法、CSR 存储法、Shermans 存储法、超矩阵存储法、动态存储方案等。[\[2\]](#)

2.2.1 三元组（Triplet）存储

顾名思义，三元组存储方法就是分别存储矩阵的非零元所在的行列索引值，以及与之对应的非零元的值的存储方案。[\[3\]](#)

三元组存储方案有着直观、易于实现、顺序无关的特性，但是与传统的存储方法一样，三元组存储方案不便于求解算法的使用，不利于

利用矩阵的稀疏性。典型的 C 语言实现为

```
struct triplet_matrix{
    int *Ti; /*row pointer*/
    int *Tj; /*column pointer*/
    double *Tx; /*value pointer*/
    int Tnz; /*number of entries*/
    int Tnrow; /*number of rows*/
    int Tncol; /*number of columns*/
};
```

下面对应于矩阵的三元组存储:

COLP	0	2	5	9	10	12						
ROWIND	0	1	0	2	4	1	2	3	4	2	1	4
VAL	a	c	b	f	j	d	g	i	k	h	e	l

2.2.2 十字链表 (Orthogonal List) 存储

稀疏矩阵可以用上述的三元组存储方式存储,但是当稀疏矩阵的非零元的位置或个数经常变化时,使用三元组存储方式就效率低下,而十字链表法能有效的提高存取效率。十字链表法不仅为稀疏矩阵的每一行设置一个单独的行循环链表,而且同样为每一列设置一个单独的列循环链表。因此,每一个非零元同时存在于两个链表中,即包含它们的行循环链表和列循环链表,亦即这两个链表的交汇处。稀疏矩阵的链表节点需要同时存储所在的行号 (row)、列号 (col)、相应的元素值 (value)、向下指针 (down)、向右指针 (right)。典型的 C 语言结构实现为:

```
typedef struct node {
    int row, col;
```

```

union {
    int val;
    struct node *ptr;
};           //value 域包括两种类型
struct node *right, *down;
}CNode, *CrossLink;

```

2.2.3 列压缩 CSS(Compressed-column storage) 存储

任何矩阵的运算都涉及到矩阵的存储与读取操作，因此，存取效率对于矩阵运算的效率有着极大的影响。

许多开源库采用了该存储结构，如 UMFPACK、TAUCS 等。列压缩存储方案较前面的三元组存储方案较不好理解，但是在矩阵计算中能更为高效地利用矩阵的稀疏性，而这也就是各大求解器采用这一存储方案的原因。[\[3\]](#)

列压缩存储维护了三组数据——(各列非零元的累加值，按递增的顺序存储的每列的非零元的行索引值，对应第二组数据行索引值位置对应的非零元的值)。典型的 C 语言结构实现为

```

struct cc_matrix{
    int *Ai; /*row index*/
    int *Ap; /*length ncol+1*/
    double *Ax;

    int Ancol;
    int Anrow;
};

```

下面对应于矩阵的列压缩存储：

COLP	0	2	5	9	10	12						
ROWIND	0	1	0	2	4	1	2	3	4	2	1	4
VAL	a	c	b	f	j	d	g	i	k	h	e	l

2.2.4 行压缩存储方式 (Compressed Row Storage)

CRS 存储可以高效地存取任意一行非零元素，但存取任意一列非零元则需要遍历整个 CRS 存储结构。相应地，与 CRS 存储的稀疏矩阵相关的算法要高效的编程实现，算法的计算顺序必须按行来进行。[\[3\]](#) 下面对应于 CRS 存储：

ROWP	0	2	5	8	9	12						
COLIND	0	1	0	2	4	1	2	3	2	1	2	4
VAL	a	b	c	d	e	f	g	h	i	j	k	l

我们可以发现，ROWP 数组存储的时行非零元的增量。COLIND 则存储的是列索引值，典型的 C 语言结构实现为

```
struct cr_matrix{
    int *Ri; /* col index */
    int *Rp; /* length nrow+1 */
    double *Rx;
    int Rncol;
    int Rnrow;
};
```

显然，越是简单的数据结构，存取操作的效率越高，亦即寻址操作的效率越高。例如，传统的稠密矩阵的存储方案只需根据行列索引值即可计算得到数据所在的地址，轻松高效的实现存取操作。在列压缩存储的稀疏矩阵中，显然能高效地存取矩阵的一列，但是，存

取行的效率是极为低下的。因此，在矩阵的运算中，我们希望能尽可能地按列进行，而不是按行进行。CRS 存储可以高效地存取任意一行非零元素，但存取任意一列非零元则需要遍历整个 CRS 存储结构。相应地，与 CRS 存储的稀疏矩阵相关的算法要高效的编程实现，算法的计算顺序必须按行来进行。[3] 下面对应于 CRS 存储：

ROWP	0	2	5	8	9	12						
COLIND	0	1	0	2	4	1	2	3	2	1	2	4
VAL	a	b	c	d	e	f	g	h	i	j	k	l

我们可以发现，ROWP 数组存储的时行非零元的增量。COLIND 则存储的是列索引值，典型的 C 语言结构实现为

```
struct cr_matrix{
    int *Ri; /* col index */
    int *Rp; /* length nrow+1 */
    double *Rx;
    int Rncol;
    int Rnrow;
};
```

显然，越是简单的数据结构，存取操作的效率越高，亦即寻址操作的效率越高。例如，传统的稠密矩阵的存储方案只需根据行列索引值即可计算得到数据所在的地址，轻松高效的实现存取操作。在列压缩存储的稀疏矩阵中，显然能高效地存取矩阵的一列，但是，存取行的效率是极为低下的。因此，在矩阵的运算中，我们希望能尽可能地按列进行，而不是按行进行。

2.3 已有的处理稀疏矩阵存储与运算的开源库

在做稀疏矩阵的计算时，通常都是做一系列的基本运算，如：矩阵转置、矩阵向量乘法、矩阵矩阵乘法、数乘等。为了能得到更好的效率，许多研究者致力于寻找对于这些计算最优的存储结构及计算算法，同时提供了许多类库供科学计算使用，如：Portable, Extensible Toolkit for Scientific Computation (PETSc)、Boost、GNU Scientific Library (GSL) 等。

例如，在 Boost-uBLAS 中有着稀疏矩阵的模板 `mapped_matrix<T, F, A>`（元素映射矩阵存储形式）、`compressed_matrix<T, F, IB, IA, TA>`（压缩存储格式）、`coordinate_matrix<T, F, IB, IA, TA>`（坐标存储格式）。分别有着如下示例：[\[4\]](#)

mapped_matrix:

```
#include <boost/numeric/ublas/matrix_sparse.hpp>
#include <boost/numeric/ublas/io.hpp>

int main () {
    using namespace boost::numeric::ublas;
    mapped_matrix<double> m (3, 3, 3 * 3);
    for (unsigned i = 0; i < m.size1 (); ++ i)
        for (unsigned j = 0; j < m.size2 (); ++ j)
            m (i, j) = 3 * i + j;
    std::cout << m << std::endl;
}
```

compressed_matrix:

```
#include <boost/numeric/ublas/matrix_sparse.hpp>
#include <boost/numeric/ublas/io.hpp>

int main () {
```

```

using namespace boost::numeric::ublas;
compressed_matrix<double> m (3, 3, 3 * 3);
for (unsigned i = 0; i < m.size1 (); ++ i)
    for (unsigned j = 0; j < m.size2 (); ++ j)
        m (i, j) = 3 * i + j;
std::cout << m << std::endl;
}

```

coordinate_matrix:

```

#include <boost/numeric/ublas/matrix_sparse.hpp>
#include <boost/numeric/ublas/io.hpp>

int main () {
    using namespace boost::numeric::ublas;
    coordinate_matrix<double> m (3, 3, 3 * 3);
    for (unsigned i = 0; i < m.size1 (); ++ i)
        for (unsigned j = 0; j < m.size2 (); ++ j)
            m (i, j) = 3 * i + j;
    std::cout << m << std::endl;
}

```

2.4 稀疏矩阵的数乘

根据数乘的定义，若矩阵 A 的 $=\{a_{ij}\}$ ，则 $kA=\{ka_{ij}\}$ ，则稀疏矩阵的数乘可以简单的定义为所有非零元乘上这个数。例如：对 (1) 实现的三元组存储的稀疏矩阵的数乘可以用如下函数进行计算：

```

void multi(triplet_matrix matrix, double k){
    for(int i=0;i<Tnz;i++){
        *Tx[i]=*Tx[i]*k;
    }
}

```

}
}

2.5 稀疏矩阵的矩阵向量乘法

根据矩阵向量乘法的定义：

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix} = \begin{pmatrix} a_{11}b_1 + a_{12}b_2 + \cdots + a_{1n}b_n \\ a_{21}b_1 + a_{22}b_2 + \cdots + a_{2n}b_n \\ \vdots \\ a_{n1}b_1 + a_{n2}b_2 + \cdots + a_{nn}b_n \end{pmatrix}$$

由于零元参与运算对计算结果没有影响，所以在计算中可以不对零元计算，亦即在稀疏矩阵的矩阵向量运算中，可以只对非零元进行计算。亦即对 A 的每一列 j 的非零元 a_{ij} 执行

$$z_i = y_i + a_{ij}x_j$$

那么，我们可以很方便的利用矩阵的稀疏性对列压缩存储的稀疏矩阵进行上述操作，示例代码如下

```
for (int j=0;j<Ancol;j++){
    for (int p=Ap[j];p<Ap[j+1];p++){
        z[Ai[p]]=Ax[p]*x[j]+y[Ai[p]];
    }
}
```

2.6 稀疏矩阵的矩阵矩阵乘法

根据矩阵向量乘法的定义：

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

其中，对于任意的 i, j ，有

$$c_{ij} = \sum_{k=1}^n a_{ik} b_{kj}$$

由于零元参与运算无意义，在计算中可以将零元不进行运算。亦即

$$c_{ij} = \sum_{k=1, a_{ik} \neq 0, b_{kj} \neq 0}^n a_{ik} b_{kj}$$

第三章 研究的目标、内容及研究方法

3.1 研究的目标

利用 c++ 语言在 linux 平台下的 g++ 编译器上编写一个库，实现稀疏矩阵的行压缩存储方式 (Compressed Row Storage) 存储，并可以实现并行计算稀疏矩阵的矩阵乘法、数乘等运算，可以在实际的科学计算中进行使用。

3.2 研究内容

在 linux 平台下，基于 g++ 编译器、C++ 标准库，编写稀疏矩阵存储的 C++ 库，实现稀疏矩阵的行压缩存储方式 (Compressed Row Storage)，同时实现一些稀疏矩阵的基本运算，如矩阵乘法、数乘等。而为了更好地发挥现代多核心 CPU 的性能，需要实现多线程并行计算，为以后可能进行的集群运算留下可以使用的接口。

3.3 研究方法

实现稀疏矩阵的行压缩存储方式 (Compressed Row Storage)，通过数组存储非零元的增长量、列索引值和非零元值来实现稀疏矩阵的存储。为了实现可变长数组的存储，利用 C++ 的 STL 中的 vector 来存储数据。

具体实现数据结构如下：

```
class sparseMatrix
{
private:
    int n_col;           /**< number of the
                        columns. */
    int n_row;           /**< number of the rows.
                        */
    int n_couple;
```

```

public:
    std::vector<double> ele; /**< the array to
        story the values of the elements. */
    std::vector<int> col; /**< the column
        index of the non-zero elements. */
    std::vector<int> row; /**< the row
        index of the non-zero elements. */
}

```

而为了更好地利用现代多核心 CPU 的性能，利用 thread 库实现多线程并行计算，更为有效地利用 CPU 的并行性能，为未来用于集群计算留下可以使用的接口。而在并行计算中，由于存在多个线程同时操作某一块内存，出现“脏数据”的情况，为了防止多线程并行计算中“脏数据”的出现，利用 mutex 互斥锁保障进程安全性，将正在运算的内存进行加锁，防止被另一线程同时操作，在操作完成后，进行解锁，供另一线程操作。而在不同的机型上，由于硬件配置、软件配置不一，最优线程数也不一，为了实现更好的利用计算机硬件性能，利用 C++ 的预定义来由程序编写者自行设定线程数。

由于在实际工程中，经常遇到的矩阵操作有矩阵数乘运算、矩阵矩阵乘法、矩阵向量乘法等等，需要实现稀疏矩阵的一些基本运算（矩阵数乘运算、矩阵矩阵乘法、矩阵向量乘法等）。而为了实现稀疏矩阵的矩阵矩阵乘法和矩阵向量乘法，可以参考全矩阵矩阵乘法和矩阵向量乘法算法，利用稀疏矩阵的零元不参与计算的特性，按照稀疏矩阵存储的结构，较全矩阵计算省略大量计算，实现稀疏矩阵计算的优势——更为高效的计算。

第四章 实验结果

在 iMac 5k 平台上进行测试 (3.5 GHz Intel Core i5, 8 GB 1600 MHz DDR3), 分别测试 sparseMatrix 和 boost-ublas, 测试代码如下:

使用 sparseMatrix.h

```
#include <iostream>
#include "sparseMatrix.h"
#include <stdlib.h>
#include "complexSparseMatrix.h"

#define ROW 100000 //行数
#define COL 100000 //列数
#define COUPLE 1500 //每行最多非零数
#define NONE_ZERO_PERCENT 0.01

int main(int argc, const char * argv[]) {
    //初始化稀疏矩阵
    sparseMatrix* sparse=new sparseMatrix(ROW,COL
        ,COUPLE);
    sparse->init(NONE_ZERO_PERCENT);
    sparse->compress();

    clock_t start, finish; double duration;
    start=clock();
    multi(sparse, sparse);
    finish=clock();
    duration=1000*(double)(finish-start)/
        CLOCKS_PER_SEC;
    std::cout<<duration<<std::endl;
```

```

    multi(sparse , sparse)->show( 'B' );

    return 0;
}

```

使用 boost_ublas

```

#include <iostream>
#include<fstream>
#include "boost/numeric/ublas/matrix_sparse.hpp"
#include "boost/numeric/ublas/io.hpp"
using namespace std;
int main ( ) {
    using namespace boost::numeric::ublas ;
    compressed_matrix<double> m ( 10000 , 1000)
        ;

    ifstream infile ;
    infile.open("/Users/tzry/Desktop/graduation-
        project/sparse/output.txt");

    while ( infile.good() && !infile.eof() ) {
        int i,j;
        double v;
        infile>>i>>j>>v;
        m(i-1,j-1)=v;

    }

    clock_t start , finish ; double duration ;
    start=clock();

```

```
m=boost::numeric::ublas::prod(m,m);

finish=clock();
duration=1000*(double)(finish-start)/
    CLOCKS_PER_SEC;
std::cout<<duration<<std::endl;

}
```

测试结果如下：

10000 阶稀疏度为 0.01 的稀疏矩阵的矩阵矩阵乘法			
单行最大非零元个数	200	500	1000
sparseMatrix	85404.1ms	85129.4ms	87244.9ms
boost_ublas	43823.6ms	330923ms	1.470550e+06ms

第五章 结论

根据本文第四部分内容，boost_ublas 库的效率在很大程度上取决于输入参数的好坏，而 sparseMatrix 则效率与输入参数的好坏相关系数很小。虽然在好的参数条件下，boost_ublas 的效率更高，但是，sparseMatrix 的鲁棒性更好，不容易受到差的输入参数的影响。sparseMatrix 在较差的参数条件下与较好地参数条件下，效率基本没有区别。

在实际计算中，遇到不容易知道参数如何选择，或者每行的非零元数量经常变化，且变化较大时，使用 sparseMatrix 库能有效地提高使用效率。

附录 A sparseMatrix 库

A.1 sparseMatrix.h

```
#ifndef sparseMatrix_h
#define sparseMatrix_h
#include <vector>
#include <iostream>
#include <fstream>
#include "tmpSparse.h"
#include<thread>
#include"sparseVector.h"

#define THREADCOUNT 1//线程数

class sparseMatrix
{
private:
    int n_col;           /**< number of the
                        columns. */
    int n_row;           /**< number of the rows.
                        */
    int n_couple;

    std::mutex m; //互斥量

public:
```

```

int missionRow=0;

std::vector<double> ele; /**< the array to
    story the values of the elements. */
std::vector<int> col; /**< the column
    index of the non-zero elements. */
std::vector<int> row; /**< the row
    index of the non-zero elements. */

int getCol(){
    return this->n_col;
}
int getRow(){
    return this->n_row;
}
int getCouple(){
    return this->n_couple;
}

//构造函数
sparseMatrix(int __n_row, int __n_col){
    n_col = __n_col;
    n_row = __n_row;
}

/**
 * Construction function.
 *
 * @param __n_row the init. value of the
    nubmer of the rows.

```



```

* @param __n_col the init. value of the
  nubmer of the columns.
* @param __n_couple the max number of the non
  -zero elements in each rows.
*/
sparseMatrix(int __n_row, int __n_col, int
  __n_couple)
{
    n_col = __n_col;
    n_row = __n_row;
    n_couple=__n_couple;
    ele.resize(__n_row*__n_couple);
    col.resize(__n_row*__n_couple, -1);
    row.resize(__n_row + 1);
    for (int i = 1; i < __n_row + 1; ++i)
        row[i] = __n_couple * i;
    for (int i = 0; i < __n_row; ++i)
        col[row[i]] = i;
};

/**
* Set a non-zero element.
*
* @param __i the row index of the element to
  put in.
* @param __j the column index of the element
  to put in.
* @param __ele the value of the element to
  put in.
*
* @return the finish status.
*/

```

```
//rewritten by tzry
//fix some bugs on special issues
int set_ele(int __i, int __j, double __ele)
{
    int i; bool flag=false;
    /**
     * If the input element is at the
     * diagonal, just put
     * it to the first place of its row.
     *
     */
    int start=row[__i];
    int maxI=start+n_couple;
    for(i=start; i<maxI; i++){
        int nowcol=col[i];

        if(nowcol==__j){
            col[i]=__j;
            ele[i]=__ele;
            flag=true;
            break;
        }
        else if(nowcol==-1){//尚未使用过
            col[i]=__j;
            ele[i]=__ele;
            flag=true;
            break;
        }
        else if(ele[i]==0){//初始的对角数据
            col[i]=__j;
            ele[i]=__ele;
            flag=true;
        }
    }
}
```

```

        break;
    }
}
if (flag)
    return 0;
else
    return -1;
};

/**
 * Get an element form the matrix with its
 * row and column
 * index.
 *
 * @param _i the row index.
 * @param _j the column index.
 *
 * @return the element value.
 */
double get_ele(int _i, int _j)
{
    for (int j = row[_i]; j < row[_i+1]; ++j)
        if (col[j] == _j)
            return ele[j];
    return 0;
};

/**
 * To list all the non-zero elements with the
 * format: (row

```

```

    * index, column index) = value of the
      element.
    *
    */
void show()
{
    for (int i = 0; i < n_row; ++i)
        for (int j = row[i]; j < row[i + 1];
            ++j)
            std::cout << "(" << i << ", " <<
                col[j] << ")=" << ele[j] << std
                ::endl;

    return;
};

int gs_step(std::vector<double> &_b, std::
vector<double> &_u)
{
    for (int i = 0; i < n_row; ++i)
    {
        double S = _b[i];
        for (int j = row[i] + 1; j < row[i +
            1]; ++j)
            S -= ele[j] * _u[col[j]];
        _u[i] = S / ele[row[i]];
    }
    return 0;
};

/**

```

```

* Compress the empty non-zero elements. Just
  renumber the
* index of the non-zero elements and push
  them to the front
* places of the element array and the column
  array. And
* release the memory.
*
*/
//rewritten by tzry
//fix some bugs on special issues
int compress()
{

    int p1 = 0;
    int i;
    int old_row = row[0];
    for(i = 0; i < n_row; i++)
    {

        for(int k = old_row; k < old_row+
            n_couple; k++)
        {
            if( col[k]==-1){//未分配
                break;
            }
            //已分配，记录
            col[p1]=col[k];
            ele[p1]=ele[k];

            p1++;
        }
    }
}

```

```

        }
        old_row = row[i + 1];
        row[i + 1] = p1;
    }
    ele.resize(p1);
    col.resize(p1);
    return 0;
};

```

```

//Added by tzry
//noZeroPercent是稀疏度
int init(double noZeroPercent){
    //初始化稀疏矩阵

    unsigned int a = 0;

    int MAX = (~a) / 2;

    srand((unsigned)time(NULL));
    for(int i=0;i<n_row;i++){
        for(int j=0;j<n_col;j++){
            if(rand()*1.0/MAX<noZeroPercent){
                this->set_ele(i, j, (float)(
                    rand()*1.0/MAX));
            }
        }
    }
    return 0;
}

```

```
//数乘
void multi(double times){
    std::vector<double>::iterator it;
    for(it=this->ele.begin();it!=this->ele.
        end();it++){
        (*it)*=times;
    }
}

//多线程获取任务行数
int getMissionRow(){
    this->m.lock();
    try
    {
        int mR=this->missionRow;
        this->missionRow++;
        this->m.unlock();
        return mR;
    }
    catch(...)
    {
        this->m.unlock();
        throw;
    }
}

//测试文件输出
void log(char* filename,char key,int row,int
    col,double value){
    std::ofstream fout(filename,std::ios::app
        );
```

```

        //fout<<key<<"("<<row<<","<<col<<")="<<
        value<<";"<<std::endl;
        fout<<row<<'_'<<col<<'_'<<value<<std::
        endl;
    }
    void show(char key)
    {
        for (int i = 0; i < n_row; ++i)
            for (int j = row[i]; j < row[i + 1];
                ++j)
                log("/Users/tzry/Desktop/
                graduation-project/sparse/
                output.txt",key,i+1,col[j]+1,
                ele[j]);
        return;
    }
};

```

//线程工作函数

```

void workFun(sparseMatrix* a,sparseMatrix* b,
tmpSparse* result,int choice){
    while(choice<a->getRow()){
        //choice为当前A行数

        double orin[a->getCol()];
        double r[b->getCol()];
        for(int i=0;i<a->getCol();i++)
            orin[i]=0;
        for(int i=0;i<b->getCol();i++)
            r[i]=0;
    }
}

```



```

        for (int i=a->row.at(choice); i<a->row.at(
            choice+1)&&a->col.at(i)!=-1&&a->ele.at
            (i)!=0; i++){
            orin[a->col.at(i)]=a->ele.at(i);
        }

        for (int row=0; row<b->getRow(); row++){
            for (int colIndex=b->row[row]; b->col[
                colIndex]!=-1&&colIndex<b->row[row
                +1]; colIndex++){
                r[b->col[colIndex]]=r[b->col[
                    colIndex]+orin[row]*b->ele[
                    colIndex];
            }
        }

        for (int i=0; i<b->getCol(); i++){
            if (r[i]!=0)
                result->setEle(choice, i, r[i]);
        }

        choice+=THREADCOUNT;
    }
}

```

//格式转换

```

sparseMatrix* sparse(tmpSparse* tps){
    int index=0;
    int couple=0;
    sparseMatrix* result=new sparseMatrix(tps->
        getRow(),tps->getCol());
    for(int i=0;i<tps->getRow();i++){
        //每一行
        if(tps->col[i].size()==0){
            //无 ele
            result->ele.push_back(0);
            if(i<=tps->getCol())
                result->col.push_back(i);
            else
                result->col.push_back(tps->getCol
                    ());
            result->row.push_back(index);
            index++;
        }
        else{
            result->row.push_back(index);
            for(int tpsi=0;tpsi<tps->col[i].size
                ();tpsi++){
                //每一个符合要求的
                result->col.push_back(tps->col[i]
                    ][tpsi]);
                result->ele.push_back(tps->ele[i]
                    ][tpsi]);
                index++;
            }
        }
    }
    result->row.push_back(index);
}

```

```

        return result;
    }

//稀疏矩阵乘法
//认为阶数是正常的
sparseMatrix* multi(sparseMatrix* a, sparseMatrix*
    b){
    a->missionRow=0;
    if (a->getCol() != b->getRow()) {
        throw "cannot be multied";
    }

    tmpSparse* result=new tmpSparse(a->getRow(), b
        ->getCol(), a->getCouple()>b->getCouple()?a
        ->getCouple():b->getCouple());

    std::thread threads[THREADCOUNT];
    for (int i=0; i<THREADCOUNT; i++){
        threads[i]=std::thread(workFun, a, b, result
            , i);
    }
    for (int i=0; i<THREADCOUNT; i++)
    {
        (threads[i]).join();
    }
    return sparse(result);
}

```

//向量乘法线程工作函数

```
void sparseVectorWorkFun(sparseMatrix* sM,
    sparseVector* sV, sparseVector* result, int row){
    while(row<sV->getLength()){

        for(int i=sM->row.at(row); i<sM->row.at(
            row+1)&& sM->col.at(i)!=-1&& sM->ele.at(i)
            )!=0; i++){
            result->addEle(row, sM->ele.at(i)*sV
                ->getValue(sM->col.at(i)));
        }

        row+=THREADCOUNT;
    }
}
```

//向量乘法

```
sparseVector* multi(sparseMatrix* sM, sparseVector
    * sV){
    sM->missionRow=0;
    if(sM->getCol()!=sV->getLength()){
        throw "cannot be multied";
    }
    sparseVector* result=new sparseVector(sM->
        getCol());
    std::thread threads[THREADCOUNT];
    for(int i=0; i<THREADCOUNT; i++){
        threads[i]=std::thread(
            sparseVectorWorkFun, sM, sV, result, i);
    }
}
```

```

    for (int i=0; i<THREADCOUNT; i++)
    {
        (threads[i]).join();
    }
    return result;
}

```

```

#endif

```

A.2 tmpSparse.h

```

#ifndef tmpSparse_h
#define tmpSparse_h
#include <vector>
#include <iostream>
#include <thread>
#include <mutex>

class tmpSparse
{
private:
    int n_col;           /**< number of the
                        columns. */
    int n_row;           /**< number of the rows.
                        */

    std::mutex m; //互斥量

public:
    int getRow() {
        return this->n_row;
    }

```

```
}  
int getCol() {  
    return this->n_col;  
}  
  
std::vector<double> *ele; //存储的数值  
std::vector<int> *col;    //存储的列号  
  
tmpSparse(int __row,int __col,int initLength){  
    n_col=__col;  
    n_row=__row;  
    ele=new std::vector<double>[__row];  
    col=new std::vector<int>[__row];  
}  
  
//设置单个位置的值（不考虑重复）  
void setEle(int __row,int __col,double __ele){  
    this->m.lock();  
    try  
    {  
        std::vector<double> *ev=&ele[__row];  
        std::vector<int> *cv=&col[__row];  
        ev->push_back(__ele);  
        cv->push_back(__col);  
  
        this->m.unlock();  
    }  
    catch (...)  
    {  
        this->m.unlock();  
        throw;  
    }  
}
```

```

    }

}

//在指定位置增加指定数
void addEle(int __row,int __col,double __ele){
    this->m.lock();
    try
    {
        std::vector<double> *ev=&ele[__row];
        std::vector<int> *cv=&col[__row];
        for(int i=0;i<cv->size();i++){
            if((*cv)[i]==__col){//已存在
                (*cv)[i]+=__ele;
                this->m.unlock();
                return;
            }
        }
        //不存在
        ev->push_back(__ele);
        cv->push_back(__col);

        this->m.unlock();
    }
    catch(...)
    {
        this->m.unlock();
        throw;
    }
}

```

```
};
```

```
#endif
```

A.3 sparseVector.h

```
#include<stdlib.h>
```

```
#include<iostream>
```

```
#ifndef sparse_sparseVector_h
```

```
#define sparse_sparseVector_h
```

```
class sparseVector{
```

```
private:
```

```
    int length;
```

```
    double* ele; public:
```

```
    int getLength(){
```

```
        return length;
```

```
    }
```

```
    double getValue(int pos){
```

```
        return *(ele+pos);
```

```
    }
```

```
    //构造函数
```

```
    sparseVector(int l){
```

```
        length=l;
```

```
        ele=(double*) malloc(sizeof(double)*l);
```

```
        double* now=ele;
```

```
        for(int i=0;i<l;i++){
```



```

        *now=0;
        now++;
    }
}
//随机初始化
void init() {
    unsigned int a = 0;
    int MAX = (~a) / 2;
    srand((unsigned)time(NULL));

    for(int i=0;i<length;i++){
        *(ele+i)=(float)(rand()*1.0/MAX);
    }
}

void addEle(int pos,double value){
    (*(ele+pos))+=value;
}
//呈现
void show() {
    for(int i=0;i<length;i++)
        std::cout<<*(ele+i)<<' ','';
    std::cout<<std::endl;
}
};

#endif

```

A.4 complexNumber.h

```

#ifndef sparse_complexNumber_h
#define sparse_complexNumber_h

```

```

#include <stdlib.h>
struct complexNumber{
    double realPart;
    double imaginaryPart;
    complexNumber(double a, double b){
        realPart=a;
        imaginaryPart=b;
    }
    complexNumber() {
        realPart=0;
        imaginaryPart=0;
    }
    //格式化输出
    char* toString() {
        char* ret=(char*) malloc(sizeof(char)*30);
        sprintf(ret, "%lf+%lfi", realPart,
            imaginaryPart);
        return ret;
    }
    //乘法重载
    friend complexNumber operator *(complexNumber
        &a, complexNumber &b) {
        complexNumber cN(a.realPart*b.realPart-a.
            imaginaryPart*b.imaginaryPart,
            a.realPart*b.
                imaginaryPart+a.
                    imaginaryPart*b.
                        imaginaryPart);
        return cN;
    }
    friend complexNumber operator *(complexNumber
        &a, double b) {

```

```

        complexNumber cN(a.realPart*b,
                          a.imaginaryPart*b);

    return cN;
}

friend complexNumber operator *(double b,
complexNumber &a){
    complexNumber cN(a.realPart*b,
                      a.imaginaryPart*b);

    return cN;
}

//加法重载
friend complexNumber operator +(complexNumber
&a,complexNumber &b){
    complexNumber cN(a.realPart+b.realPart ,a.
        imaginaryPart+b.imaginaryPart);
    return cN;
}
};

#endif

```

A.5 complexSparseMatrix.h

```

#ifndef sparse_complexSparseMatrix_h
#define sparse_complexSparseMatrix_h

#include <vector>
#include "complexNumber.h"
#include <iostream>
#include <fstream>

```

```

#include<thread>
#include"complexSparseVector.h"
#include "tmpComplexSparse.h"

#define THREADCOUNT 4//线程数

class complexSparseMatrix
{
private:
    int n_col;           /**< number of the
        columns. */
    int n_row;           /**< number of the rows.
        */
    int n_couple;

    std::mutex m; //互斥量

public:

    int missionRow=0;

    std::vector<complexNumber> ele; /**< the
        array to store the values of the elements.
        */
    std::vector<int> col;           /**< the column
        index of the non-zero elements. */
    std::vector<int> row;           /**< the row
        index of the non-zero elements. */

```

```
int getCol() {  
    return this->n_col;  
}  
int getRow() {  
    return this->n_row;  
}  
int getCouple() {  
    return this->n_couple;  
}
```

//构造函数

```
complexSparseMatrix(int __n_row, int __n_col){  
    n_col = __n_col;  
    n_row = __n_row;  
}  
  
/**  
 * Construction function.  
 *  
 * @param __n_row the init. value of the  
 *       nubmer of the rows.  
 * @param __n_col the init. value of the  
 *       nubmer of the columns.  
 * @param __n_couple the max number of the non  
 *       -zero elements in each rows.  
 */  
complexSparseMatrix(int __n_row, int __n_col,  
    int __n_couple)  
{
```

```

        n_col = __n_col;
        n_row = __n_row;
        n_couple=__n_couple;
        ele.resize(__n_row*__n_couple);
        col.resize(__n_row*__n_couple, -1);
        row.resize(__n_row + 1);
        for (int i = 1; i < __n_row + 1; ++i)
            row[i] = __n_couple * i;
        for (int i = 0; i < __n_row; ++i)
            col[row[i]] = i;
    };

    /**
     * Set a non-zero element.
     *
     * @param __i the row index of the element to
     * put in.
     * @param __j the column index of the element
     * to put in.
     * @param __ele the value of the element to
     * put in.
     *
     * @return the finish status.
     */
    //rewritten by tzry
    //fix some bugs on special issues
    int set_ele(int __i, int __j, complexNumber
        __ele)
    {
        int i; bool flag=false;
        /**

```

```

    * If the input element is at the
      diagonal, just put
    * it to the first place of its row.
    *
    */
int start=row[_i];
int maxI=start+n_couple;
for( i=start; i<maxI; i++){
    int nowcol=col[i];

    if(nowcol==_j){
        col[i]=_j;
        ele[i]=_ele;
        flag=true;
        break;
    }
    else if(nowcol==-1){//尚未使用过
        col[i]=_j;
        ele[i]=_ele;
        flag=true;
        break;
    }
    else if(ele[i].realPart==0&&ele[i].
        imaginaryPart==0){//初始的对角数据
        col[i]=_j;
        ele[i]=_ele;
        flag=true;
        break;
    }
}
if(flag)
    return 0;

```

```

        else
            return -1;
    };

    /**
     * Get an element form the matrix with its
     * row and column
     * index.
     *
     * @param _i the row index.
     * @param _j the column index.
     *
     * @return the element value.
     */
    complexNumber get_ele(int _i, int _j)
    {
        for (int j = row[_i]; j < row[_i+1]; ++j)
            if (col[j] == _j)
                return ele[j];
        complexNumber cN(0,0);
        return cN;
    };

    /**
     * To list all the non-zero elements with the
     * format: (row
     * index, column index) = value of the
     * element.
     *
     */
    void show()

```



```

{
    for (int i = 0; i < n_row; ++i)
        for (int j = row[i]; j < row[i + 1];
              ++j)
            std::cout << "(" << i << ", " <<
                col[j] << ")=" << ele[j].
                toString() << std::endl;
    return;
};

/*
int gs_step(std::vector<double> &b, std::
vector<double> &u)
{
    for (int i = 0; i < n_row; ++i)
    {
        double S = b[i];
        for (int j = row[i] + 1; j < row[i +
            1]; ++j)
            S -= ele[j] * u[col[j]];
        u[i] = S / ele[row[i]];
    }
    return 0;
};
*/

/**
 * Compress the empty non-zero elements. Just
   renumber the
 * index of the non-zero elements and push
   them to the front

```

```

    * places of the element array and the column
      array. And
    * release the memory.
    *
    */
//rewritten by tzry
//fix some bugs on special issues
int compress()
{

    int p1 = 0;
    int i;
    int old_row = row[0];
    for(i = 0; i < n_row; i++)
    {

        for(int k = old_row; k < old_row+
            n_couple; k++)
        {
            if( col[k]==-1){//未分配
                break;
            }
            //已分配，记录
            col[p1]=col[k];
            ele[p1]=ele[k];

            p1++;

        }
        old_row = row[i + 1];
        row[i + 1] = p1;
    }
}

```

```

        ele.resize(p1);
        col.resize(p1);
        return 0;
    };

```

//Added by tzry

//noZeroPercent是稀疏度

```

int init(double noZeroPercent){
    //初始化稀疏矩阵

```

```

    unsigned int a = 0;

```

```

    int MAX = (~a)/2;

```

```

    srand((unsigned)time(NULL));

```

```

    for(int i=0;i<n_row;i++){

```

```

        for(int j=0;j<n_col;j++){

```

```

            if(rand()*1.0/MAX<noZeroPercent){
                complexNumber cN((float)(rand()
                    *1.0/MAX),(float)(rand()
                    *1.0/MAX));

```

```

                this->set_ele(i, j, cN);

```

```

            }

```

```

        }

```

```

    }

```

```

    return 0;

```

```

}

```

//数乘

```

void multi(double times){

```

```

        std::vector<complexNumber>::iterator it;
        for(it=this->ele.begin(); it!=this->ele.
            end(); it++){
            (*it).realPart*=times;
            (*it).imaginaryPart*=times;
        }
    }

```

//多线程获取任务行数

```

int getMissionRow() {
    this->m.lock();
    try
    {
        int mR=this->missionRow;
        this->missionRow++;
        this->m.unlock();
        return mR;
    }
    catch(...)
    {
        this->m.unlock();
        throw;
    }
}

```

//测试文件输出

```

void log(char* filename, char key, int row, int
    col, char* value) {
    std::ofstream fout(filename, std::ios::app
        );
}

```

```

        fout << key << " ( " << row << " , " << col << " ) = " <<
            value << " ; " << std::endl;
    }
    void show(char key)
    {
        for (int i = 0; i < n_row; ++i)
            for (int j = row[i]; j < row[i + 1];
                ++j)
                log("/Users/tzry/Documents/
                    graduation-project/sparse/
                    output.txt", key, i + 1, col[j] + 1,
                    ele[j].toString());
        return;
    }
};

```

//线程工作函数

```

void complexWorkFun(complexSparseMatrix* a,
    complexSparseMatrix* b, tmpComplexSparse* result
    , int choice) {
    while (choice < a->getRow()) {
        //choice 为当前A行数

        complexNumber *orin;
        complexNumber *r;

        orin = (complexNumber*) malloc(sizeof(
            complexNumber)*a->getCol());
        r = (complexNumber*) malloc(sizeof(
            complexNumber)*b->getCol());
    }
}

```

```

for (int i=0;i<a->getCol();i++){
    orin[i].realPart=0;
    orin[i].imaginaryPart=0;
}
for (int i=0;i<b->getCol();i++){
    r[i].realPart=0;
    r[i].imaginaryPart=0;
}

for (int i=a->row.at(choice);i<a->row.at(
choice+1)&&a->col.at(i)!=-1&&a->ele.at(
i).realPart!=0&&a->ele.at(i).
imaginaryPart!=0;i++){
    orin[a->col.at(i)].realPart=a->ele.at
    (i).realPart;
    orin[a->col.at(i)].imaginaryPart=a->
    ele.at(i).imaginaryPart;
}

for (int row=0;row<b->getRow();row++){
    for (int colIndex=b->row[row];b->col[
colIndex]!=-1&&colIndex<b->row[row
+1];colIndex++){
        //r[b->col[colIndex]]=r[b->col[
        colIndex]+orin[row]*b->ele[
        colIndex];
        complexNumber t=*(orin+row)*b->
        ele[colIndex];
        (*(r+b->col[colIndex])).realPart
        +=t.realPart;
    }
}

```

```

        (* (r+b->col [ colIndex ] ) ) .
        imaginaryPart+=t . imaginaryPart ;
    }
}

for ( int i=0; i<b->getCol () ; i++) {
    if ( r [ i ] . realPart !=0 || r [ i ] .
        imaginaryPart !=0 )
        result -> setEle ( choice , i , *(r+i) )
        ;
}

choice+=THREADCOUNT;
}

}

```

//格式转换

```

complexSparseMatrix* sparse (tmpComplexSparse* tps
) {
    int index=0;
    int couple=0;
    complexSparseMatrix* result=new
        complexSparseMatrix ( tps->getRow () , tps->
        getCol () );
    for ( int i=0; i<tps->getRow () ; i++) {
        //每一行
        if ( tps->col [ i ] . size () ==0 ) {

```

```

        //无 ele
        complexNumber cN(0,0);
        result->ele.push_back(cN);
        if(i<=tps->getCol())
            result->col.push_back(i);
        else
            result->col.push_back(tps->getCol
                ());
        result->row.push_back(index);
        index++;
    }
    else{
        result->row.push_back(index);
        for(int tpsi=0; tpsi<tps->col[i].size
            (); tpsi++){
            //每一个符合要求的
            result->col.push_back(tps->col[i
                ][ tpsi]);
            result->ele.push_back(tps->ele[i
                ][ tpsi]);
            index++;
        }
    }
}
result->row.push_back(index);
return result;
}

```

//稀疏矩阵乘法

//认为阶数是正常的

```

complexSparseMatrix* multi(complexSparseMatrix* a
    ,complexSparseMatrix* b){

```



```

a->missionRow=0;
if (a->getCol()!=b->getRow()) {
    throw "cannot be multied";
}

tmpComplexSparse* result=new tmpComplexSparse
(a->getRow(),b->getCol(),a->getCouple()>b->
getCouple()?a->getCouple():b->getCouple());

std::thread threads[THREADCOUNT];
for (int i=0;i<THREADCOUNT;i++){
    threads[i]=std::thread(complexWorkFun,a,b
        ,result,i);
}

for (int i=0;i<THREADCOUNT;i++)
{
    (threads[i]).join();
}
return sparse(result);
}

```

//向量乘法线程工作函数

```

void complexSparseVectorWorkFun(
    complexSparseMatrix* sM,complexSparseVector* sV
    ,complexSparseVector* result,int row){
    while (row<sV->getLength()){

```

```

        for (int i=sM->row.at(row); i<sM->row.at(
            row+1)&& sM->col.at(i)!=-1&& sM->ele.at(i)
            ).realPart!=0&& sM->ele.at(i).
            imaginaryPart!=0; i++){
            complexNumber t=sM->ele.at(i);
            complexNumber tt=sV->getValue(sM->col
                .at(i));
            result->addEle(row, t*tt);
        }

        row+=THREADCOUNT;
    }
}

```

//向量乘法

```

complexSparseVector* multi(complexSparseMatrix*
    sM, complexSparseVector* sV){
    sM->missionRow=0;
    if (sM->getCol()!=sV->getLength()){
        throw "cannot be multied";
    }
    complexSparseVector* result=new
        complexSparseVector(sM->getCol());
    std::thread threads[THREADCOUNT];
    for (int i=0; i<THREADCOUNT; i++){
        threads[i]=std::thread(
            complexSparseVectorWorkFun, sM, sV, result
            , i);
    }
    for (int i=0; i<THREADCOUNT; i++)

```

```

    {
        (threads[i]).join();
    }
    return result;
}

```

```
#endif
```

A.6 tmpComplexSparse.h

```

#ifndef sparse_tmpComplexSparse_h
#define sparse_tmpComplexSparse_h
#include <vector>
#include <iostream>
#include <thread>
#include <mutex>
#include "complexNumber.h"
class tmpComplexSparse
{
private:
    int n_col;           /**< number of the
                        columns. */
    int n_row;           /**< number of the rows.
                        */

    std::mutex m; //互斥量

public:
    int getRow() {
        return this->n_row;
    }
}

```

```

int getCol() {
    return this->n_col;
}

std::vector<complexNumber> *ele; //存储的数值
std::vector<int> *col;           //存储的列号

tmpComplexSparse(int __row,int __col,int
    initLength){
    n_col=__col;
    n_row=__row;
    ele=new std::vector<complexNumber>[__row];
    col=new std::vector<int>[__row];
}

//设置单个位置的值（不考虑重复）
void setEle(int __row,int __col,complexNumber
    __ele){
    this->m.lock();
    try
    {
        std::vector<complexNumber> *ev=&ele[
            __row];
        std::vector<int> *cv=&col[__row];
        ev->push_back(__ele);
        cv->push_back(__col);

        this->m.unlock();
    }
    catch (...)
    {

```

```

        this -> m. unlock ( ) ;
        throw ;
    }

}

```

```
};
```

```
#endif
```

A.7 complexSparseVector.h

```

#include <stdlib.h>
#include <iostream>
#include "complexNumber.h"
#ifndef sparse_complexSparseVector_h
#define sparse_complexSparseVector_h

class complexSparseVector{
private:
    int length;
    complexNumber* ele;
public:
    int getLength() {
        return length;
    }
    complexNumber getValue(int pos){

```

```

        return *(ele+pos);
    }
    //构造函数
    complexSparseVector(int l){
        length=l;
        ele=(complexNumber*) malloc(sizeof(
            complexNumber)*l);
        complexNumber* now=ele;

        for(int i=0;i<l;i++){
            (*now).realPart=0;
            (*now).imaginaryPart=0;
            now++;
        }
    }
    //随机初始化
    void init(){
        unsigned int a = 0;
        int MAX = (~a)/2;
        srand((unsigned)time(NULL));

        for(int i=0;i<length;i++){
            (*(ele+i)).realPart=(float)(rand()
                *1.0/MAX);
            (*(ele+i)).imaginaryPart=(float)(rand()
                *1.0/MAX);
        }
    }

    void addEle(int pos,complexNumber value){
        (*(ele+pos)).realPart+=value.realPart;

```

```
        (*(ele+pos)).imaginaryPart+=value.
        imaginaryPart;
    }
    //呈现
    void show() {
        for(int i=0;i<length;i++)
            std::cout<<(ele+i)->toString()<<', ';
        std::cout<<std::endl;
    }
};

#endif
```


参考文献

- [1] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. Springer, second edition, 2003.
- [2] 张永杰、孙秦. 稀疏矩阵存储技术. 长春理工大学学报, 03:38–41, 2006.
- [3] 冯广祥. 大型稀疏矩阵直接求解算法的研究及实现. Master's thesis, 东北大学, 2010.
- [4] Joerg Walter and Mathias Koch. Boost 1.57.0 library documentation-ublas.

致 谢

在完成终稿的今天，在敲完最后一个句号的时刻，我的思想同周围凝固的热气一样停驻了，思绪万千，心情久久不能平静。光阴似箭，白驹过隙，转眼间四年大学本科生活就即将结束，四年浙江大学学习生活注定将成为我人生中的一段重要旅程。四年来，我的师长、同学给予我的关心和帮助，使我终身收益，我真心地感谢他们。

感谢浙江大学数学系副教授王何宇老师，一位平易近人的良师，一位和蔼可亲的朋友。伟人、名人为我所崇拜，可是我更急切地要把我的敬意和赞美献给一位平凡的人——我的导师。我不是您最出色的学生，而您却是最尊敬的老师。您治学严谨，学识渊博，视野宽广，为我营造了一种良好的学术氛围。置身其间，耳濡目染，潜移默化，使我不仅深入了解了全新的奥尔夫教学思想体系的先进理念，树立了明确的学术目标，领会了基本的思考方式，掌握了通用的研究方法，而且还明白了许多待人接物与为人处世的道理。您严以律己、宽以待人的崇高风范，朴实无华、平易近人的人格魅力，与无微不至、感人至深的人文关怀，令人如沐春风，倍感温馨。感谢可爱可亲的王老师，感谢您在百忙之中对我毕业论文从选题到写作再到最后定稿所付出的辛劳！感谢您在这个我即将离开浙江大学最后的炎热夏天对我人生方向的指引！

在论文即将完成之际，我的心情无法平静，从开始进入课题到论文的顺利完成，有多少可敬的师长、同学、朋友给了我无言的帮助，在这里请接受我诚挚谢意！同时也感谢学院为我提供良好的做毕业设计的环境，以及在设计中被我引用或参考的论著的作者。

随着这篇本科毕业论文的最后落笔，我四年的大学生活也即将划上一个圆满的句号。回忆这四年生活的点点滴滴，从入学时对大学生活的无限憧憬到课堂上对各位老师学术学识的深沉沉湎，从奔波于教室图书馆的来去匆匆到业余生活的五彩缤纷，一切中的一切都是历历在目，让人倍感留恋，倍感珍惜。