计算摄影学第二次实验报告

123456,张三

1. 实验要求

1.1 稀疏矩阵的实现

实现本实验要求自行实现一种稀疏矩阵。推荐的稀疏矩阵存储方式包括：

1. Compressed Row Storage
2. Compressed Sparse Column
3. 其他形式的稀疏矩阵存储方式

同学们实现的稀疏矩阵必须起码具备以下的几种最基本的功能：

1. at(row, col): 根据row和column的系数来查询矩阵里面的元素的数值
2. insert(val, row, col)： 将val替换/插入到（row, col）这个位置去
3. initializeFromVector(rows, cols, vals): 根据向量来初始化一个稀疏矩阵。其中rows, cols, vals皆为等长度的向量。rows里面存的是行系数，cols里面存的是列系数，vals里面存的是数值。
4. 其余的基本功能可以参考Matlab里面的sparse函数，或者Eigen Library里面的Sparse Matrix的介绍。

1.2 稀疏矩阵的高斯赛达尔迭代法

本实验要求在自己实现的稀疏矩阵的表达的基础上，实现高斯赛达尔迭代法（Gauss-Seidel Method），用于求解大规模的稀疏线性方程组。

1.3 实现共轭梯度法（bonus）

使用共轭梯度法求解线性方程组

> 注： 以上内容来自课程网站lab 3

2. 实现环境

软件环境：Visual Studio 2017

运行环境： OS：Windows 10, 8GB RAM

3. 具体实现

3.1 稀疏矩阵实现

稀疏矩阵，即其中非零元素个数很少的矩阵。在实际操作中，存储大量的0是非常浪费空间的，尤其是当矩阵的尺寸变得很大的时候。于是，我们想到设计一种特殊的数据结构，只存储非0元素，对零元素则不加以保存。这样，面对稀疏矩阵时，就有大量的空间得以保留。稀疏矩阵数据结构的想法由此产生。

稀疏矩阵的数据结构可以有多种，如实验要求中已经提到的，包括压缩行存储、压缩列存储等等。

本次程序中稀疏矩阵的实现参照了课堂上讲过的CRS(compressed Row Storage)，实现起来比较简单，样例如下所示：

$$A = \begin{bmatrix} 10 & 0 & 0 & 0 & -2 & 0 \\ 3 & 9 & 0 & 0 & 0 & 3 \\ 0 & 7 & 8 & 7 & 0 & 0 \\ 3 & 0 & 8 & 7 & 5 & 0 \\ 0 & 8 & 0 & 9 & 9 & 13 \\ 0 & 4 & 0 & 0 & 2 & -1 \end{bmatrix}.$$

val	10	-2	3	9	3	7	8	7	3 ··· 9	13	4	2	-1		
col_ind	1	5	1	2	6	2	3	4	1 ··· 5	6	2	5	6		

row_ptr	1	3	6	9	13	17	20

> 上述资料参考自章国锋老师的PPT

本次实现的稀疏矩阵尤为简单，首先，我们定义矩阵值与列号对应的数据结构`val_col`，其接口定义大致如下所示：

```
class val_col {
public:
    T val;
    int col;
    val_col() {
        col = 0;
    }
    val_col(const T& val, int col) {
        this->val = val;
        this->col = col;
    }

    val_col(T& t) {
        this->val = t.val;
        this->col = t.col;
    }

    T get() {
        return val;
    }
    void set(T val, int col) {
        this->val = val;
        this->col = col;
    }
    void set(T val) {
        this->val = val;
    }
};
```

需要注意的是，上述类`val_col`是一个内部类，其中`T`是模板参数，代表矩阵所存储的数据的类型。

整体上看，这是一个比较简单的数据结构，仅仅实现了很少的接口。但在实际中，这些接口甚至并没有完全用上。

稀疏矩阵的完整定义如下所示：

```
template <class T> class sparseMatrix
{
private:
    //行数
    int Nrow;

    //列数
    int Ncol;
public:
    //行下标列表
    int* row_list;

    //列表与值对照表
    vector<val_col> val_cols_list;

    //构造函数
    sparseMatrix();
    sparseMatrix(int nrow, int ncol);
    sparseMatrix(vector<T> v);

    ~sparseMatrix();

    //返回矩阵中第`i`行，第`j`列的元素
    T at(int row, int col);

    //[get]
    int cols() {
        return Ncol;
    }
    int rows() {
        return Nrow;
    }

    //在第row行第col列插入val值
    bool insert(const T& val, int row, int col);

    //通过向量初始化函数
    bool initializeFromVector(vector<int>& rows,
        vector<int>& cols, vector<T>& vals);

    //未实现完全，将用于共轭梯度法
    bool isSymmetric(void);
    bool isPosDefinite(void);

    //矩阵点乘
    sparseMatrix dot(sparseMatrix<T> mat2);
```

```
    //右乘一个向量(n行1列)
    vector<T> dot_v(vector<T> v);

    //左乘一个向量
    vector<T> v_dot(vector<T> v);

    //高斯赛达尔迭代解线性方程组，其中B为系数
    vector<double> Gauss_Seidel_Iter(vector<double> B);

    //共轭梯度法
    vector<double> ConGrad(vector<double> B);

    //对行数或列数为1的矩阵，可以转化成向量
    vector<T> toVec();

    //运算符重载，包括矩阵之间的加、减；矩阵乘一个常数；矩阵之间的赋值
    sparseMatrix<T> operator-(sparseMatrix<T> m2);
    sparseMatrix<T> operator+(sparseMatrix<T> m2);
    sparseMatrix<T> operator*(double fac);
    sparseMatrix<T> operator=(sparseMatrix<T> m2);

    //矩阵转置，返回一个新矩阵
    sparseMatrix<T> Transpose();

    //辅助函数，用于插入一个新值后更新下标列表
    void updateRow(int row) {
        for (int i = row + 1; i < Nrow; i++) {
            row_list[i]++;
        }
        //pivot
        row_list[Nrow] = 0x7fffffff;
    }

    //辅助函数，用于debug时打印列标
    void printValCols() {
        for (int i = 0; i < val_cols_list.size(); i++) {
            cout << val_cols_list[i].col << "  " << val_cols_list[i].val << endl;
        }
    }

    //清空矩阵的内容
    void clear() {
        Nrow = Ncol = 0;
        memset(row_list, 0, sizeof(row_list));
        for (int i = 0; i < val_cols_list.size(); i++)
            val_cols_list.pop_back();
    }
};
```

其中，各个接口的定义如上述代码中的注释所示。下面，将对作业中特别要求的三个函数进行详细的介绍。

3.1.1 `at` 函数

```
template<class T>
T sparseMatrix<T>::at(int row, int col) {
    try {

        int row_idx = row_list[row];
        int next_row_idx = row_list[row + 1];

        if (row == Nrow - 1)
            next_row_idx = val_cols_list.size();
        if (row_idx < 0)
            return ZERO;

        for (int i = row_idx; i < next_row_idx; i++) {
            if (val_cols_list[i].col == col) {
                T t = (val_cols_list[i].get());
                return t;
            }
        }
        return ZERO;
    }
    catch (exception e) {
        std::cout << "invalid access" << std::endl;
        return ZERO;
    }
}
```

首先，我们根据行号，取得那一行的下标的值和下一行的下标值。如果这一行是最后一行，那么下一行的下标即为`val_cols_list`的长度，这样，我们取元素时就可以取到矩阵中的所有元素。

接下来，我们在`row_idx`和`next_row_idx`中进行遍历，如果发现遍历时某个元素的列号与我们所有访问的列号相同，则代表我们确实正确地访问到了那个元素，于是我们取出那个值并返回。如果遍历完了理论上这一行所有的元素却还没有找到，那么说明这个值在矩阵中不存在，也就意味着在这一行这一列那个元素的值为0，于是我们返回ZERO即可。

3.1.2 插入函数

插入函数的实现相对复杂，具体代码实现如下所示：

```
template<class T>
bool sparseMatrix<T>::insert(const T& val, int row, int col) {
    try {
        int row_idx = row_list[row];
        int next_row_idx = row_list[row + 1];
        int i = 0;
        vector<val_col>::iterator iter = val_cols_list.begin();

        for (i = row_idx; i < next_row_idx && i<val_cols_list.size()&&iter!=val_cols_list.end();
i++, iter++) {
            if (val_cols_list[i].col >= col)
                break;
        }
```

```
        if (i == val_cols_list.size()) { // should push_back
            val_col t(val, col);
            val_cols_list.push_back(t);
            row_list[Nrow]++;
            updateRow(row);
            return true;
        }
        else if (col == val_cols_list[i].col) {//already exist
            val_cols_list[i].val = val;
            return true;
        }
        else {// common insertion
            val_col t(val, col);
            iter = iter + row_idx;
            val_cols_list.insert(++iter, t);
            row_list[Nrow]++;
            updateRow(row);
            return true;
        }
    }
    catch (exception e) {
        cout << e.what() << endl;
        return false;
    }
}
```

在这个实现里，插入主要有三种可能的情况：

- 在那一行那一列已经有元素存在了，这时的插入则应该是元素的替换。即用想要插入的元素替换掉原来存在的元素。
- 遍历到整个矩阵最后一行的值也没有发现它，但是这个元素要被插在最后一行，列号较大而作为最后一个元素。这个时候要插入到`val_cols_list`的队尾。
- 正常的插入。即那个地方原先确实不存在元素，而且插入后又不会作为最后一个元素。

具体的实现正如上述代码所示。首先，我们找到这个元素应该插入的位置，然后判断这个元素应该如何插入，插入方式属于上述三种方式的哪一种。

3.1.3 `initializeFromVector`函数

有了`insert`函数，`initializeFromVector`函数的实现就简单得多了。简单来说就是遍历这些`vector`中的元素，然后多次调用`insert`函数。实现代码如下所示：

```
template<class T>
bool sparseMatrix<T>::initializeFromVector(vector<int>& rows,
    vector<int>& cols, vector<T>& vals) {
    _ASSERT(rows.size() == cols.size() && cols.size() == vals.size());
    int n_error = 0;
    for (unsigned int i = 0; i < rows.size(); i++) {
        if (!insert(vals[i], rows[i], cols[i]))
            n_error++;
    }
    if (n_error == 0)
        return true;
```

```
    else {
        cout << "n_error: " << n_error << endl;
        return false;
    }
}
```

有一点需要注意的是，`insert`函数和`initializeFromVector`函数都是有返回值的，当向量中的全部元素都插入成功后，会返回`true`，否则会返回`false`。

3.2 高斯-赛达尔迭代

高斯-赛达尔迭代（Gauss-Seidel method)是数值线性代数中的一个迭代法，可用来求出线性方程组的近似值。设线性方程组为：

$a_{i1}x_1 + a_{i2}x_2 + \ldots + a_{in}x_n = b_i$

$i = 1, 2, \ldots, n$

高斯-赛达尔迭代法的迭代公式为：

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

$$(i = 1, 2, \cdots, n; k = 0, 1, 2, \cdots,)$$

这里，我们假设$a_{ii} \neq 0$.

与简单迭代法不同的是，它在迭代时利用了刚求出的

$$x_1^{(k+1)}, x_2^{(k+1)}, \cdots, x_{i-1}^{(k+1)}$$

这部分的实现如下所示：

```
template<class T>
vector<double> sparseMatrix<T>::Gauss_Seidel_Iter(vector<double> B) {
    _ASSERT(Nrow == Ncol && Nrow == B.size());
    vector<double> result;
    vector<double> prev_result;

    //Nrow elements init with value of 1.
    result.assign(Nrow, 1.0);

    int iterTimes = 0;
    do {
        iterTimes++;
        //element-wise copy
        prev_result = result;
        if (iterTimes == 1)
            prev_result.assign(Nrow,5.0);
        for (int i = 0; i < Nrow; i++) {
            if (at(i, i)==ZERO)

                continue;
```

```
            double bi = B[i];
            double sigma1 = 0.0;
            double sigma2 = 0.0;
            for (int j = 0; j < i; j++) {
                sigma1 += at(i, j)*result[j];
            }
            for (int j = i + 1; j < Nrow; j++) {
                sigma2 += at(i, j)*prev_result[j];
            }
            result[i] = (bi - sigma1 - sigma2) / at(i, i);
        }
    } while(!converged(result, prev_result) && iterTimes <= MaxIter);

    cout << iterTimes << endl;

    return result;
}
```

函数的实现即如上述原理公式所示。我们先分别求出上述公式中的两个和值，然后求出这一次迭代中对应x的值。循环退出的条件为足够收敛或迭代次数足够多。这里，足够收敛意味着两次算出的结果足够接近，这里采用了L_1距离。这个函数定义如下所示：

```
//The L1 distance is small, then it is converged.
static bool converged(vector<double>& v1, vector<double>& v2) {
    double res = 0.0;
    _ASSERT(v1.size() == v2.size());
    for (int i = 0; i < v1.size(); i++) {
        res += abs(v1[i] - v2[i]);
    }
    if (res <= ConvergeLimit)
        return true;
    else
        return false;
}
```

3.3 共轭梯度法

共轭梯度法适用于解正定、对称矩阵所对应线性方程组的数值解。

共轭梯度法最早由Hestenes和Stiefle提出。在此技术上，Fletcher和Reeves首先提出了解非线性最优化问题的共轭梯度法。共轭梯度法实现的过程如下所示：

$$
\begin{aligned}
&\mathbf{r}_0 := \mathbf{b} - \mathbf{A}\mathbf{x}_0 \\
&\mathbf{p}_0 := \mathbf{r}_0 \\
&k := 0 \\
&\text{repeat} \\
&\quad \alpha_k := \frac{\mathbf{r}_k^{\mathrm{T}} \mathbf{r}_k}{\mathbf{p}_k^{\mathrm{T}} \mathbf{A} \mathbf{p}_k} \\
&\quad \mathbf{x}_{k+1} := \mathbf{x}_k + \alpha_k \mathbf{p}_k \\
&\quad \mathbf{r}_{k+1} := \mathbf{r}_k - \alpha_k \mathbf{A} \mathbf{p}_k \\
&\quad \text{if } r_{k+1} \text{ is sufficiently small then exit loop} \\
&\quad \beta_k := \frac{\mathbf{r}_{k+1}^{\mathrm{T}} \mathbf{r}_{k+1}}{\mathbf{r}_k^{\mathrm{T}} \mathbf{r}_k} \\
&\quad \mathbf{p}_{k+1} := \mathbf{r}_{k+1} + \beta_k \mathbf{p}_k \\
&\quad k := k + 1 \\
&\text{end repeat} \\
&\text{The result is } \mathbf{x}_{k+1}
\end{aligned}
$$

在具体实现中，共轭梯度法的代码如下所示：

```
template<class T>
vector<double> sparseMatrix<T>::ConGrad(vector<double> B) {
    /*if (!this->isSymmetric())
        return vector<double>();*/

    //real calculation
    sparseMatrix b(B);
    sparseMatrix<double> x(Nrow, 1);
    for (int i = 0; i < Nrow; i++) {
        x.insert(0, i, 0);
    }
    sparseMatrix<double> r(1, B.size());
    sparseMatrix<double> p(1, B.size());
    sparseMatrix<double> new_r(1, B.size());
    r = b - dot(x).Transpose();
    p = r;
    int niter = 0;
    do {
        niter++;
        double rkTrk = (r.dot(r.Transpose()).at(0, 0));
        double div = (p.Transpose().dot(*this).dot(p)).at(0, 0);
        double alpha;
        if(div==0)  alpha =  1 ;
        else alpha = rkTrk / div;
        x = x + p.Transpose()*alpha;
        new_r = r - (dot(p.Transpose())).Transpose()*alpha;
        double beta = new_r.dot(new_r.Transpose()).at(0, 0) / rkTrk;
        p = new_r + p*beta;
        r = new_r;
    } while (!converged<T>(r.toVec())&&niter<=MaxIter);
    return x.toVec();
}
```

按照共轭梯度法所应用的条件，其所适用的矩阵应该是正定的、对称的。在开头之前应该做一个检查。在我实际的实现当中，对对称的检查很容易，并且已经实现。但是对矩阵正定的检查却还没有想好该如何实现。在当前版本的实现中，我们没有进行这样的检查。

实现时，我们的步骤基本上都是按照上述的伪码所要求的过程实现的。为了实现这个流程所要求的各种运算，我们首先定义了一些函数，用于做矩阵的运算。包括加、减、乘和赋值等。我们还定义了一些矩阵与向量`vector`之间的运算，虽然这些运算在实际中并没有用上。此外，对于向量`vector`,我们还设计了稀疏矩阵的构造函数，可以将向量转换为1行n列的矩阵。

4. 实现结果

4.1 稀疏矩阵的结果验证

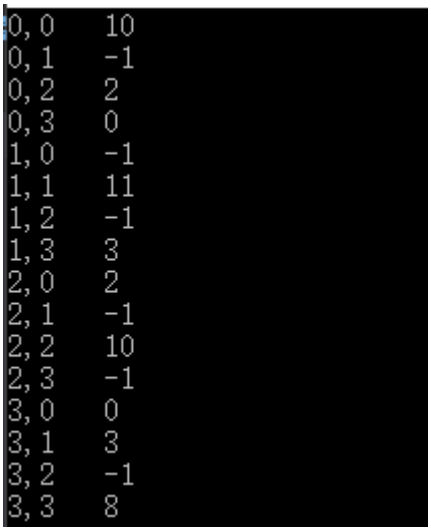
为了验证已经实现的部分的正确性，我们将样例提供的矩阵值逐个插入到矩阵中，然后输出这些值，看看输出的元素是否正确。这验证了`insert`和`at`函数的实现是否正确。

代码如下所示：

```
mat.insert(10, 0, 0);
    mat.insert(-1, 0, 1);
    mat.insert(2, 0, 2);
    mat.insert(-1, 1, 0);
    mat.insert(11, 1, 1);
    mat.insert(-1, 1, 2);
    mat.insert(3,  1, 3);
    mat.insert(2, 2, 0);
    mat.insert(-1, 2, 1);
    mat.insert(10, 2, 2);
    mat.insert(-1, 2, 3);
    mat.insert(3, 3, 1);
    mat.insert(-1, 3, 2);
    mat.insert(8, 3, 3);

    for (int i = 0; i < 4; i++) {
        for (int j = 0; j < 4; j++)
            cout << i << "," << j <<"   "<< mat.at(i, j) << endl;
    }
```

输入结果如下图：



可以看到输出的结果，正好与矩阵的值相同，可知程序的实现基本正确。

4.2 高斯-赛达尔迭代法的结果验证

继续应用上述矩阵，我们构建方程组等号右侧的系数向量`B`：

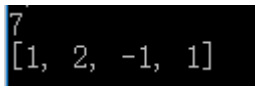
```
vector<double> B(4);

B[0] = 6; B[1] = 25; B[2] = -11; B[3] = 15;
```

然后应用高斯赛达尔迭代法求解，如下所示：

```
    vector<double> res = mat.Gauss_Seidel_Iter(B);
    printVector<double>(res);
```

得到输入结果如下所示：（包括迭代次数和解）



这里的迭代次数上限设定如下所示：

```
#define ConvergeLimit 0.0001
#define MaxIter 1000000
```

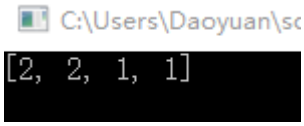
由样例给出的结果可知，这里的实现基本正确。

4.3 共轭梯度法的结果验证

共轭梯度法需要矩阵正定而且对称，这里在验证时给出一个简单的例子：

```
    mat.insert(1, 0, 1);
    mat.insert(1, 0, 3);
    mat.insert(1, 1, 0);
    mat.insert(1, 1, 2);
    mat.insert(1, 2, 1);
    mat.insert(1, 3, 0);
    B[0] = 2; B[1] = 2; B[2] = 1; B[3] = 1;
```

得到输出结果如下所示：



由此可见，这里的输出结果也基本正确。

5. 总结与思考

5.1 总结回顾

在本次实验过程中，我感触最大的一部分就是：实现一个优秀的类是在不是一件容易的事。要考虑的事要包括方方面面。例如：类内接口和成员应该如何定义才能既保证数据安全又能保证接口的灵活？在实现本类时所有的特殊情况都考虑到了吗？这个类应该重载哪些运算符？怎样的运算才是合理的？构造函数应该有哪些？这个类与一些现有类的交互应该如何实现？怎样才能使实现过程的时空效率都很高？

在实现之前，我几乎没有意识到这些问题的严重性。但当我认为自己几乎快要做完的时候，自己其实已经深陷泥潭了，有太多问题困扰着我。而出现这些问题的原因即是没有事先良好的规划。或者说，在实现之前，我也许应该参考一些现有库或者其他人的实现，来为自己填充思路。

另外，知识的灵活运用应该建立在熟练掌握的基础上。在完成这次作业之前，我确是有一段时间没有好好接触过`C++`了，这让我在实现`sparseMatrix`这个模板类时甚至走了一些弯路，浪费了很多时间。

5.2 思考展望

由于时间和精力的限制，我所提交的sparseMatrix类还有许多不能尽善尽美之处。正如我在5.1节中已经提到的，运算的效率、函数接口的定义的思考都不够成熟。类的接口函数仍然稍显不足，亟待丰富。在矩阵运算的实现中，或许还应该考虑一下时间空间的节省。

在这次实验中，我越来越深刻的意识到自己需要学习一些有关程序设计方法论的东西了。学习的方法可以包括现成的开源代码，开源库等等，还可以是一些广为传颂的经典书籍等等。此外，对于一些语法的理解也亟待提高。