# 计算摄影学作业报告
# Lab 3 —— 实现稀疏矩阵以及高斯赛达尔迭代法

Jessie Peng

2019/03/26

# 1 实验内容

## 1.1 稀疏矩阵

本实验要求自行实现一种稀疏矩阵，我在本实验中选用的稀疏矩阵存储方式为 Compressed Row Storage

本实验还要求实现的稀疏矩阵具备以下几种最基本的功能：

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 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。

# 2 实验环境

编程语言：C++
开发环境：CLion 2018.3
操作系统：macOS 10.14.1 (18B75)

# 3 实验原理

## 3.1 行压缩存储

选择采用行压缩的方式来存储稀疏矩阵，主要的数据结构是两个向量：向量 col_val 的每个元素是形如 (col_ind, val) 的数对，表示列号和对应的数值；向量 row_ptr 的每个元素是对应行的数对在 col_val 中的起始下标，用以确定行号与 (col_ind, val) 的对应关系。

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

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 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 |

在具体实现中需要注意几个细节：

1. 向量 row_ptr 的下标是从 0 开始，而行号是从 1 开始计数，为了统一和直观，把 row_ptr[0]忽略掉不使用。
2. 为了方便寻找向量 col_val 的末尾，假设多出一行，即添加一个行指针 row_ptr[row_cnt+1]用于标记。
3. 当一行全为 0 时，行指针仍然需要按顺序放置，因此将每一行在 col_val 中的起始元素作为一个 dummy head 用来占位，不管该行数值如何，行指针都指向这个 dummy head，而该元素的值则无关紧要。

## 3.2 高斯赛德尔迭代法

高斯赛德尔迭代法是一种求解稀疏线性方程组的迭代方法，当求解以下方程组时：

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}, \qquad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \qquad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

把系数矩阵 A 分解为下三角矩阵 $L_*$ 和严格上三角矩阵 U，使得 $A = L_* + U$

$$L_* = \begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ a_{21} & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}, \quad U = \begin{bmatrix} 0 & a_{12} & \cdots & a_{1n} \\ 0 & 0 & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix}.$$

因此，方程组 Ax = b 就可以写成 $L_*x = b - Ux$

改写成可以迭代的形式，就是 $L_*x^{(k+1)} = b - Ux^{(k)}$，也就是

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

# 4 实现过程与代码分析

## 4.1 稀疏矩阵的存储方式与接口

实现的稀疏矩阵模版类声明如下：

```
template <typename T> class MySparseMatrix {
public:
    explicit MySparseMatrix(int m = 10, int n = 10);
    T at(int row, int col) const;
    void insert(T val, int row, int col);
    bool initializeFromVector(const vector<int> &rows, const vector<int> &cols,
const vector<T> &vals);
    void printInfo() const;
    int getRowsCnt() const { return rows_cnt; }
    int getColsCnt() const { return cols_cnt; }
private:
    vector<pair<int, T>> col_val;
    vector<int> row_ptr;
    int rows_cnt;
    int cols_cnt;
    auto getIterator(int ind) const;
};
```

下面的例子展示了具体的数据结构：

$$\begin{pmatrix} 10 & -1 & 2 & 0 \\ -1 & 11 & -1 & 3 \\ 2 & -1 & 10 & -1 \\ 0 & 3 & -1 & 8 \end{pmatrix}$$

```
row_ptr: 0 0 4 9 14 18
col:    0   1   2   3   0   1   2   3   4   0   1   2   3   4   0   2   3   4   0
val:    0  10  -1   2   0  -1  11  -1   3   0   2  -1  10  -1   0   3  -1   8   0
```

其中 row_ptr 的第一个 0 是没有用的，(col, val) = (0, 0) 的数对则是 dummy head。

## 4.2 稀疏矩阵的构造函数

该稀疏矩阵必须在构造的时候设定行数和列数，同时也进行 dummy head 等的一些初始化，具体代码如下：

```
template <typename T> MySparseMatrix<T>::MySparseMatrix(int m, int n): rows_cnt(m),
cols_cnt(n)
{
    row_ptr.push_back(0);
    for (int i = 0; i <= rows_cnt; i++)
    {
        row_ptr.push_back(i);
        col_val.push_back(make_pair(0, 0));
    }
}
```

## 4.3 访问稀疏矩阵的某个元素

at 函数可以读取矩阵中指定行号和列号的元素，实现比较简单，扫描即可：

```
template <typename T> T MySparseMatrix<T>::at(int row, int col) const
{
    int start = row_ptr[row];
    int end = row_ptr[row+1];
    for (int i = start; i < end; i++)
    {
        if (col_val[i].first == col)
        {
            return col_val[i].second;
        }
    }
    return 0;
}
```

## 4.4 向稀疏矩阵中插入/替换数值

因为没有赋值的时候稀疏矩阵的元素默认为 0，因此插入和替换是一个意思，使用 insert 函数来完成。由于使用 CRS 的存储方式，就有三种情况：

1. 非零数变为 0，对应操作为从向量 col_val 中删除一个数对，还要更新行指针
2. 非零数变为另一个非零数，对应操作为直接修改对应的 val 值
3. 0 变为非零数，对应操作为插入一个数对到向量 col_val 中，也要更新行指针

```
template <typename T> void MySparseMatrix<T>::insert(T val, int row, int col)
{
    int start = row_ptr[row];
    int end = row_ptr[row+1];
    int i;
```

```
    for (i = start; i < end; i++)
    {
        if (col_val[i].first == col)
        {
            if (val == 0) // 1. non-zero -> zero
            {
                auto j = getIterator(i);
                col_val.erase(j); // remove pair (col, val)
                for (int k = row+1; k <= rows_cnt+1; k++)
                {
                    row_ptr[k]--; // update row pointers
                }
            }
            else // 2. non-zero -> non-zero
            {
                col_val[i].second = val; // update val
            }
            return;
        }
        if (col_val[i].first > col)
        {
            break;
        }
    }
    if (val != 0) // 3. zero -> non-zero
    {
        auto j = getIterator(i);
        col_val.insert(j, make_pair(col, val)); // add pair (col, val)
        for (int k = row+1; k <= rows_cnt+1; k++)
        {
            row_ptr[k]++; // update row pointers
        }
    }
}
```

## 4.5 通过向量来初始化矩阵

initializeFromVector 函数相当于输入一个 COO (coordinate list) 方式存储的矩阵，用来初始化我的稀疏矩阵。该函数首先判断输入的三个向量长度是否相同，即输入是否合法，然后一个一个元素调用 insert 来赋值。

```
template <typename T> bool MySparseMatrix<T>::initializeFromVector(
        const vector<int> &rows, const vector<int> &cols, const vector<T> &vals)
{
    unsigned long len = rows.size();
    if (cols.size() != len || vals.size() != len)
    {
        return false;
    }
    for (int i = 0; i < len; i++)
    {
        insert(vals[i], rows[i], cols[i]);
    }
    return true;
}
```

## 4.6 高斯赛德尔迭代类

将其写成一个类而不是单独写成一个函数，是为了多次迭代操作便于统计迭代次数和记录结果，模版类声明如下：

```
template <typename T> class GaussSeidel {
public:
    explicit GaussSeidel(int len = 4);
    explicit GaussSeidel(const vector<T> &initval);
    bool solve(const MySparseMatrix<T> &A, const vector<T> &b, int iter = 20, bool
verbose = false);
    vector<T> getResult() const { return x; }
    int getIters() const { return iters; }
private:
    int n;
    vector<T> x;
    int iters;
    bool zeroInDiag(const MySparseMatrix<T> &A) const;
};
```

其中实现了两种构造函数，一个是仅指定要求解的未知数的个数，初始化默认全部为1:

```
template <typename T> GaussSeidel<T>::GaussSeidel(int len): n(len), iters(0)
{
    for (int i = 0; i < n; i++)
    {
        x.push_back(1); // initial values are 1 by default
    }
}
```

另一个则是由用户传入一个向量来初始化 x：

```
template <typename T> GaussSeidel<T>::GaussSeidel(const vector<T> &initval):
n((int)initval.size()), iters(0)
{
    for (int i = 0; i < n; i++)
    {
        x.push_back(initval[i]);
    }
}
```

## 4.7 迭代求解

求解的 solve 函数输入是矩阵 A 和向量 b，迭代次数，以及是否需要打印迭代过程。迭代过程中，如果解已经收敛（相邻两次迭代解相同）则停止。

```
template <typename T> bool GaussSeidel<T>::solve(const MySparseMatrix<T> &A, const
vector<T> &b, int iter, bool verbose)
{
    if (A.getColsCnt() != n || A.getRowsCnt() != n || b.size() != n ||
zeroInDiag(A))
    {
        return false;
    }
    int cnt;
    bool stop = false;
    for (cnt = 0; cnt < iter; cnt++)
    {
        if (verbose)
        {
            for (auto i : x)
            {
                cout << i << " ";
```

```
            }
            cout << endl;
        }
        if (stop)
        {
            break;
        }
        stop = true;
        double sigma;
        double tmp;
        for (int i = 1; i <= n; i++)
        {
            sigma = 0;
            for (int j = 1; j <= n; j++)
            {
                if (j != i)
                {
                    sigma += A.at(i, j) * x[j-1];
                }
            }
            tmp = (double)(b[i-1] - sigma) / A.at(i, i);
            if (tmp != x[i-1])
            {
                stop = false;
                x[i-1] = tmp;
            }
        }
    }
    iters += cnt;
    return true;
}
```

该函数调用了一个辅助函数，用来判断矩阵 A 的主对角线是否有 0，如果是则输入不合法，函数直接返回 false。

```
template <typename T> bool GaussSeidel<T>::zeroInDiag(const MySparseMatrix<T> &A)
const
{
    for (int i = 1; i <= A.getRowsCnt(); i++)
    {
        if (A.at(i, i) == 0) // if there's a zero in A's diagonal
        {
            return true;
        }
    }
    return false;
}
```

# 5　结果分析与实验总结

## 5.1　示例方程组求解

用自己实现的稀疏矩阵和高斯赛德尔迭代法对测试用例进行求解，示例如下：

$$A = \begin{pmatrix} 10 & -1 & 2 & 0 \\ -1 & 11 & -1 & 3 \\ 2 & -1 & 10 & -1 \\ 0 & 3 & -1 & 8 \end{pmatrix}$$

$$b = (6, 25, -11, 15)^T$$

应该得到的结果是 x = [1, 2, -1, 1]

用如下代码进行测试（调用 solve 两次，每次迭代 5 轮，一共迭代了 10 轮）：

```
int main()
{
    MySparseMatrix<double> A(4, 4);
    A.insert(10, 1, 1);
    A.insert(-1, 1, 2);
    A.insert(2, 1, 3);
    A.insert(-1, 2, 1);
    A.insert(11, 2, 2);
    A.insert(-1, 2, 3);
    A.insert(3, 2, 4);
    A.insert(2, 3, 1);
    A.insert(-1, 3, 2);
    A.insert(10, 3, 3);
    A.insert(-1, 3, 4);
    A.insert(3, 4, 2);
    A.insert(-1, 4, 3);
    A.insert(8, 4, 4);
    A.printInfo();

    vector<double> b;
    b.push_back(6);
    b.push_back(25);
    b.push_back(-11);
    b.push_back(15);

    GaussSeidel<double> solver;
    if (solver.solve(A, b, 5, true))
    {
        vector<double> x = solver.getResult();
        for (auto i : x) {
            cout << i << " ";
        }
        cout << endl;
        cout << "iteration number: " << solver.getIters() << endl;
    }
    if (solver.solve(A, b, 5, true))
    {
        vector<double> x = solver.getResult();
        for (auto i : x) {
            cout << i << " ";
        }
        cout << endl;
```

```
        cout << "iteration number: " << solver.getIters() << endl;
    }

    return 0;
}
```

实际运行的结果正确：

```
row_ptr: 0 0 4 9 14 18
col:       0      1      2      3      0      1
val:    0.00  10.00  -1.00   2.00   0.00  -1.00   11.
1 1 1 1
0.5 2.13636 -0.886364 0.963068
0.990909 2.01958 -0.999917 0.992669
1.00194 2.00218 -1.0009 0.999068
1.0004 2.00021 -1.00015 0.999903
1.00005 2.00002 -1.00002 0.999991
iteration number: 5
1.00005 2.00002 -1.00002 0.999991
1.00001 2 -1 0.999999
1 2 -1 1
1 2 -1 1
1 2 -1 1
1 2 -1 1
iteration number: 10

Process finished with exit code 0
```

### 5.2 问题与解决

本次作业没有遇到太大的难题，实现了稀疏矩的一些比较简单接口，其中只遇到了两个小问题：

1. 最开始我忽略了下标从 0 还是从 1 开始的这个细节，导致根据高斯赛德尔算法的伪代码写出来的代码运行不正确，后来经过检查发现是向量 b 的下标本来应该从 1 开始，但在代码实现的时候是从 0 开始的，才导致了数据错位。
2. 最开始没有考虑到检查输入的合法性，自己在测试时输入了一个主对角线上有 0 的矩阵进行求解，结果导致除数为 0，后来修复了这部分代码，增加了对输入矩阵的合法性检验。

虽然本次实验的问题得到了解决，但是代码仍然有诸多不完善的地方，比如稀疏矩阵不能随意扩容的问题，还有一些复杂的矩阵运算接口也还没有实现。

## 6 参考文献

课程网站：http://www.cad.zju.edu.cn/home/gfzhang/course/computational-photography/lab3-gauss-seidel/gauss-seidel.html
Eigen 库的稀疏矩阵：http://eigen.tuxfamily.org/dox/group__TutorialSparse.html