Lab3 实现稀疏矩阵以及高斯赛德尔迭代法

张三 123456

1. 实验内容

本次实验的内容分成三部分。

1.1 稀疏矩阵

需要实现稀疏矩阵的存储、访问指定位置的元素、在指定位置插入和更新元素、使用向量初始化稀疏矩阵。

1.2 稀疏矩阵的高斯赛德尔迭代法

在实现稀疏矩阵的基础上，使用稀疏矩阵存储线性方程组的系数矩阵。使用高斯赛德尔迭代法求解线性方程组。

1.3 共轭梯度法

在实现稀疏矩阵的基础上，使用稀疏矩阵存储线性方程组的系数矩阵。使用共轭梯度法求解线性方程组。

2. 理论分析

2.1 稀疏矩阵

实现这部分内容的核心在于稀疏矩阵的存储。稀疏矩阵的存储有几种方式。

1. List of lists
2. Coordinate list
3. Compressed row storage

具体内容在课件中都有清晰的描述，我就不在此赘述了。

我在实现中使用的是LIL的方式。这种存储方式为矩阵的每一行创建一个列表，这个列表中的每项是一个二元组，存储该行中非零元素的值和列索引。每行的列表可以按列索引进行排序来提高查找速度。

2.2 高斯赛德尔迭代法

高斯赛德尔迭代法的迭代表达式如下：

$$L_* * x^{k+1} = b - Ux^k \tag{2.2.1}$$

其中L_*和U来自系数矩阵A的分解。L_*是A的下三角矩阵，U是其余的部分。

根据公式$(2.2.1)$可以求解出x的表达式：

$$x_i^{k+1} = \frac{1}{a_{ii}}(b_i - \sum_{j<i} a_{ij}x_j^{k+1} - \sum_{j>i} a_{ij}x_j^k) \tag{2.2.2}$$

这样就可以通过迭代计算x的值，直到解收敛。当系数矩阵严格对角占优或对称正定时，该方法一定收敛。

2.3 共轭梯度法

共轭梯度法用于求解系数矩阵为对称正定矩阵的线性方程组。
若$u^T Av = 0$，则向量u,v关于A是共轭的。
假设$P = \{p_k\}$是线性空间R^n的一组基。这样可以通过P获得方程组的一个解：

$$x_* = \sum_i n\alpha_i p_i$$

则有：

$$b = Ax_* = \sum_i n\alpha_i Ap_i$$

因为$\forall i \neq k$，p_i，p_k是共轭的。所以，对任意$p_k \in P$，有：

$$p_k^T = p_k^T = \sum_i n\alpha_i p_k^T Ap_i = \alpha_k p_k^T Ap_k \tag{2.3.1}$$

这样，通过选择基向量p_k，就可以得到近似的解。通过迭代来使结果与真实的解更加接近。需要定义一个度量函数来判断当前解与真实解的距离。度量如下：

$$f(x) = \frac{1}{2}x^T Ax - x^T b, x \in R^n$$

为了优化这个函数，采用梯度下降法。$r_k = b - Ax_k$为$f(x)$的梯度，也就是梯度下降的方向。同时，还要求这个方向p_k彼此共轭。则有：

$$p_k = r_k - \sum_{i<k} \frac{p_i^T Ar_k}{p_i^T Ap_i} p_i$$

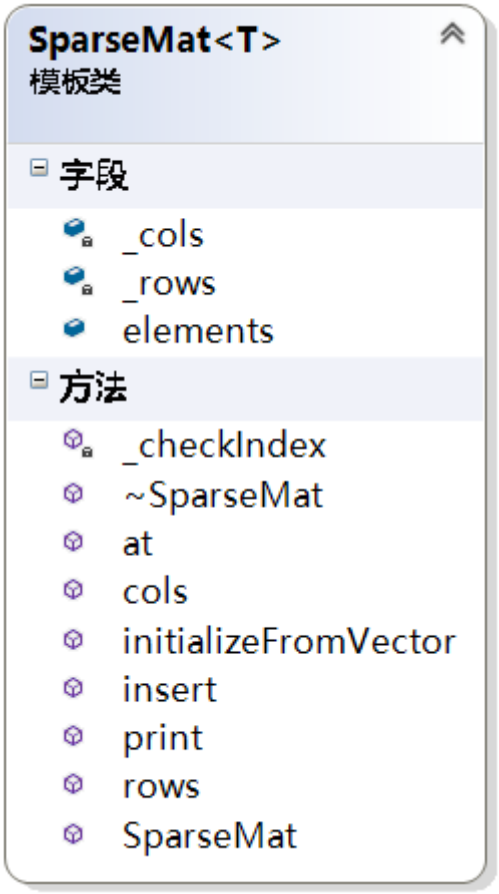
于是，有$x_{k+1}=x_k+\alpha_k p_k$。其中，α_k由公式$(2.3.1)$解出：

$$\alpha_k=\frac{p_k^Tb}{p_k^TAp_k}=\frac{p_k^T(r_{k-1}-Ax_{k-1})}{p_k^TAp_k}=\frac{p_k^Tr_{k-1}}{p_k^TAp_k}$$

3. 实验细节

3.1 稀疏矩阵

为稀疏矩阵类的定义如下。类的私有成员变量为行数和列数，公有成员变量矩阵的元素。



其中命名以下划线开头的为私有成员。这里把矩阵的元素作为了公有成员，主要是为了在求解线性方程组的时候更容易调用，但并不是一个好的设计。
为了使用起来更加灵活，我将这个类定义为了模板类，这样矩阵对象中元素的类型就可以自由定义。

其中 `elements` 的定义如下：

```
map<int, T> * elements;
```

我使用的是list of lists的方式。 `elements` 是所有行列表的列表。每行的列表由一个（列索引，元素）的映射列表构成，在c++中可以方便地使用 `map` 实现。

稀疏矩阵类只有一个构造函数，需要指定矩阵的行列数。并对元素列表进行初始化。

```
this->elements = new map<int, T>[rows]; // 为矩阵的每行创建元素列表
```

有一个私有成员函数 `inline bool _checkIndex(int row, int col)` ，用于检查索引是否越界。方法很简单，判断行列索引是否为小于矩阵行列数的非负整数。

这样，获得矩阵某个位置的元素就非常简单。首先检查索引是否越界。如果合法，通过行索引定位到某行的 `map` ，然后在这个映射中以列索引作为键进行查找，如果炸到，那么返回值，否则该位置为空，返回0。

```
// 在该行中根据列索引查找元素
auto item = elements[row].find(col);

if (item != elements[row].end())
    return item->second;
// 稀疏矩阵中某个位置没有储存元素，则该位置为0
else
    return 0;
```

插入或更新某位置元素的值的方法非常类似，也是先定位到某行的元素列表。但之后的操作更加容易。因为c++中 `map` 在引用某键进行赋值时，如果有值，则直接更新，如果没有，则会自动插入。

```
// 更新行元素列表
elements[row][col] = val;
```

使用向量初始化矩阵，先判断行索引向量、列索引向量、值向量的维度是否匹配。之后只需要对向量中的每个（行，列，值）三元组调用之前的插入函数即可。

在以上的要求之外，我还实现了一个打印矩阵所有元素的方法，这是为了更容易地检查代码编写是否正确。

3.2 高斯塞德尔迭代法

这部分实现起来很容易，只需要根据课件中的伪代码编写代码就可以了。

函数定义如下：

```
// 高斯赛德法
template<class T1, class T2, class T3>
vector<T1> gauss_seidel_solve(SparseMat<T2> A, vector<T3> b,
    int max_iteration_num = 10000, float threshold=0.00001)
```

函数的定义也使用了模板，为了区分结果向量、系数矩阵、常数向量元素的类型。默认设定了最大迭代次数和判断收敛的阈值。
在每次迭代中，有两层循环，外层循环用于更新解的每一维，内层循环根据公式计算该维的值。
这里需要注意的是判断解收敛的方法，应该比较当前迭代和上一次迭代中解向量的每一个维度差的绝

对值是否都小于阈值，如果均满足，则迭代收敛。直观的方式是在每一次迭代计算中，先将本次计算得到的解保存下来，和上一次迭代的解进行比较，再进行更行。但是考虑到每一次迭代中的外层循环就是向量的维度，所以在遍历每一维的时候，都进行对该位的比较，当有任何一次不满足条件时，本次迭代不收敛。
一次迭代的代码大致如下：

```
bool if_converge = true;
// 对向量的每一维使用公式进行计算
for (int i = 0; i < A.rows(); i++)
{
    /*
    使用公式进行计算
    */

    // 记录当前迭代解该维的值
    double tmp = 1.0 * (b[i] - sigma) / A.at(i, i);
    // 比较当前迭代与上一次迭代解的该维的差是否小于阈值
    if (tmp - x[i] > threshold || tmp - x[i] < -threshold)
        if_converge = false;
    // 更新解
    x[i] = tmp;
}
// 解的每一维在当前迭代和上一次迭代中的差都小于阈值，收敛，结束循环
if (if_converge)
    break;
```

3.3 共轭梯度法

在共轭梯度法的计算中，需要用到多种矩阵和向量的运算，具体包括：

- 矩阵乘向量
- 两向量的内积
- 两向量相加
- 两向量相减
- 向量数乘

在实现中使用了C++的 `vector` 类，但是该类并不包含上述的运算，所以我为每种运算编写了单独的函数。函数采用了模板，用于区别计算结果和两运算数元素的变量类型。上述计算的实现方法都很简单，只是稍微麻烦，具体方式不在此赘述。

共轭梯度法的实现也是完全按照课件上的伪代码。判断收敛的方法在算法中已经给出：r_{k+1}是否充分小。在计算中，注意到Ap_k在α_k和r_{k+1}的计算中都被使用到，所以定义一个临时变量来记录这个值。其他的就没有特别需要关注的地方了。

4. 结果展示

4.1 稀疏矩阵

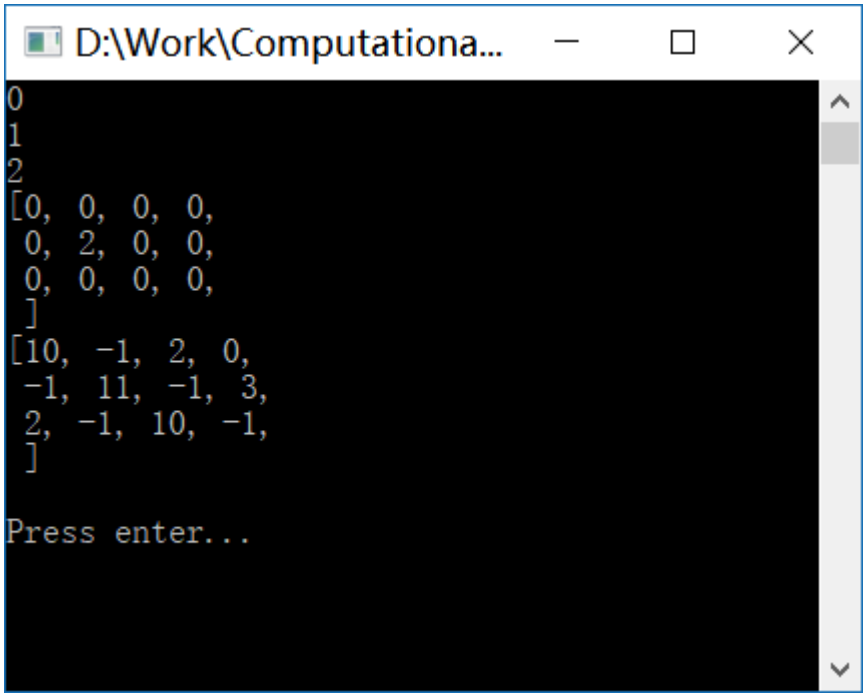
运行如下代码：

```
// 创建4*4矩阵
SparseMat<int> mat(4, 4);
// 访问 (1, 1)
cout << mat.at(1, 1) << endl;
// 将1插入 (1, 1)
mat.insert(1, 1, 1);
// 访问 (1, 1)
cout << mat.at(1, 1) << endl;
// 将2插入 (1, 1)
mat.insert(2, 1, 1);
// 访问 (1, 1)
cout << mat.at(1, 1) << endl;
// 打印整个矩阵
mat.print();

// 使用向量初始化矩阵
int vals_array[] = { 10, -1, 2, -1, 11, -1, 3, 2, -1, 10, -1, 3, -1, 8 };
int rows_index_array[] = { 0, 0, 0, 1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3 };
int cols_index_array[] = { 0, 1, 2, 0, 1, 2, 3, 0, 1, 2, 3, 1, 2, 3 };
vector<int> vals(vals_array, vals_array + 14);
vector<int> rows(rows_index_array, rows_index_array + 14);
vector<int> cols(cols_index_array, cols_index_array + 14);

mat.initializeFromVector(rows, cols, vals);
mat.print();
```

得到的结果如下：



4.2 高斯塞德尔迭代法

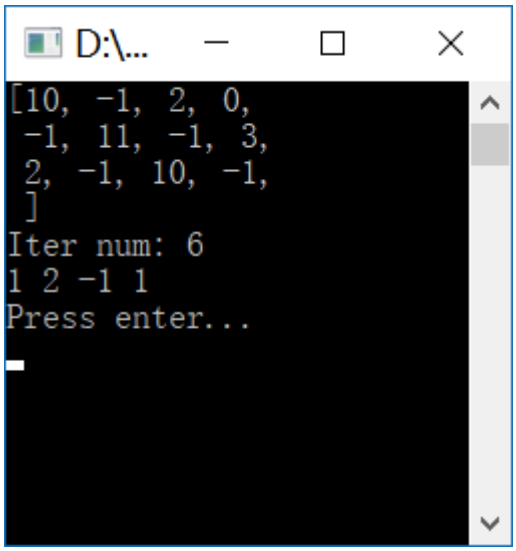
使用上述系数矩阵，为方程组定义常数向量。之后调用函数解方程组。代码如下：

```
int b_array[] = { 6, 25, -11, 15 };
vector<int> b(b_array, b_array + 4);

vector<double> x = gauss_seidel_solve<double, int, int>(mat, b);

for (auto i = x.begin(); i != x.end(); i++)
{
    cout << *i << " ";
}
```

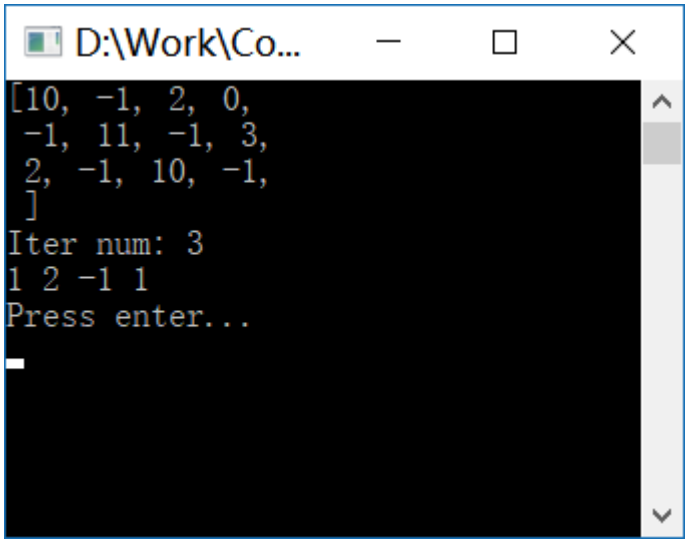
获得的结果为：



可以看到，在6次迭代之后，解已经收敛。

4.3 共轭梯度法

使用相同的方法，将高斯塞德尔函数替换为共轭梯度法的函数。获得的结果如下：



可以看出经过3次迭代，解收敛。