实现稀疏矩阵以及 Gauss-Seidel 迭代法

lab 3

姓名： 周炜

学号： 32010103790

稀疏矩阵

实验原理

稀疏矩阵是指大部分元素为零的矩阵。具体来说，如果矩阵的大部分元素都是零，并且非零元素的数量相对于总元素数量来说很小，则该矩阵可以被称为稀疏矩阵。稀疏矩阵在计算机科学和工程领域中经常出现，因为它们可以有效地利用存储空间和计算资源。

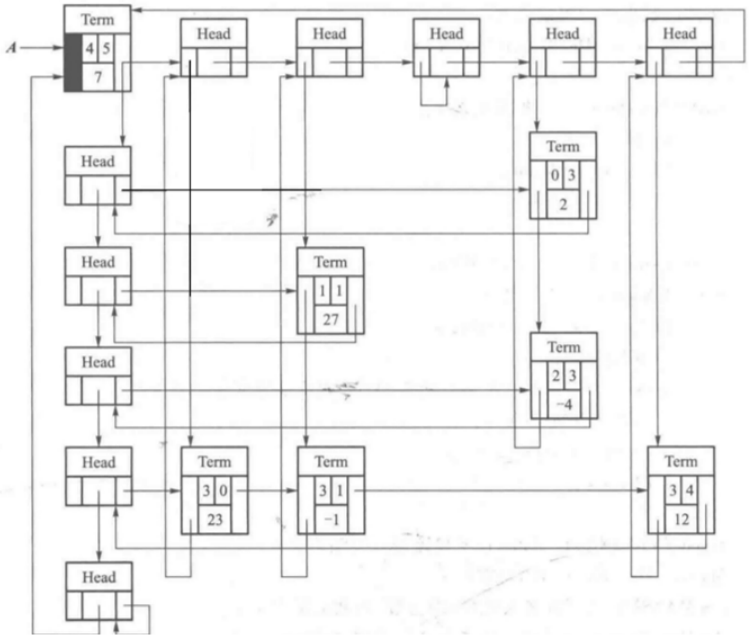
一般来说，矩阵可以表示为一个二维数组，其中每个元素都有一个对应的行索引和列索引。在稀疏矩阵中，只有少量的这些元素具有非零值，而其余的元素都是零。通常，稀疏矩阵可以通过多种方式进行存储和表示，以便在计算中更有效地处理，例如压缩存储格式（如CSR、CSC、COO等）

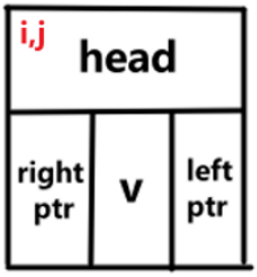
对矩阵做出以下约定：

- 矩阵的非零元素的数量少于 `int`
- 矩阵元素的精度在 `1e-10` 以内，即如果一个矩阵元素的绝对值小于 `1e-10`，则可以认为它就是 0

代码实现

我采用了数据结构基础中的十字链表来进行实现：





我把矩阵中的每个值都定义为了如下格式：

```
class Mnode{
    public:
        int i; // Row id
        int j; // Col id
        double v;   // Element value
        Mnode* right;
        Mnode* down;
        Mnode(int i = 0, int j = 0, double v = 0, Mnode* right = nullptr, Mnode*
down = nullptr)
        : i(i), j(j), v(v), right(right), down(down) {}
};
```

首先介绍成员变量

```
const double epsilon = 1e-10;   //  precision
int RowNum, ColNum, NonzeronNum; // 行数，列数，稀疏矩阵中非零值的数量
Mnode* data;  // 具体的矩阵中的值存在链表中
```

然后介绍成员函数：

构造函数和析构函数：

- `Sparse()`：默认构造函数，初始化一个空的稀疏矩阵，设置行数（RowNum）、列数（ColNum）、非零元素个数（NonzeronNum）为零，并将数据指针（data）设为空。
- `Sparse(int md, int nd)`：构造函数，根据给定的行数（md）和列数（nd）初始化一个稀疏矩阵，分配内存给数据指针（data）。
- `~Sparse()`：析构函数，释放稀疏矩阵占用的内存，调用了 `clear()` 函数来清除所有元素

```
Sparse() : RowNum(0), ColNum(0), NonzeronNum(0), data(nullptr) {}

Sparse(int md, int nd) :RowNum(md), ColNum(nd), NonzeronNum(0){
    data = new Mnode [RowNum];
}

~Sparse() {
    clear();
    delete[] data;
}
```

清除函数:

`clear()`: 清除稀疏矩阵中的所有元素，释放内存。遍历每一行的链表，释放节点，并将行头的右指针重置为空。最后将非零元素计数（NonzeronNum）设为零

```
void clear() {
    for (int i = 0; i < RowNum; ++i) {
        Mnode* currentNode = data[i].right;
        while (currentNode != nullptr) {
            Mnode* toDelete = currentNode;
            currentNode = currentNode->right;
            delete toDelete;
        }
        data[i].right = nullptr; // Reset the row header's next pointer
    }
    NonzeronNum = 0; // Reset the count of non-zero elements
}
```

获取维度信息函数:

- `getRowDimension()`: 返回稀疏矩阵的行数。
- `getColDimension()`: 返回稀疏矩阵的列数。

```
int getRowDimension() {
    return RowNum;
}
int getColDimension() {
    return ColNum;
}
```

读取和修改元素函数:

- `at(int row, int col)`: 返回稀疏矩阵中指定位置（row，col）的元素值。如果指定位置超出了矩阵的范围，则输出错误信息并返回零。
- `insert(double val, int row, int col)`: 插入或修改稀疏矩阵中指定位置（row，col）的元素值为 val。如果超出矩阵的范围或非零元素数量已满，则返回空指针。

```
int at(int row, int col) {
    if (row < 0 || row >= RowNum || col < 0 || col >= ColNum) {
        cout << "out of range" << endl;
        return 0;
    }
```

```
    Mnode* line = &data[row];
    Mnode* value = GetElement(line, col);
    if (value == nullptr) {
        return 0;
    }
    return value->v;
}
Mnode* insert(double val, int row, int col) {
    if (NonzeronNum >= RowNum * ColNum || row > RowNum || col > ColNum){
        return nullptr;
    }
    Mnode* p = &data[row];
    while (p) {
        if (p->j == col) {
            if (p->v == 0 && val != 0) {
                NonzeronNum++;
            }
            p->v = val;
            return p;
        }
        if (p->j < col && p->right && p->right->j > col) {
            Mnode* newele = new Mnode(row, col, val);
            NonzeronNum++;
            newele->right = p->right;
            p->right = newele;
            return newele;
        }
        if (p->right == nullptr) {
            Mnode* newele = new Mnode(row, col, val);
            NonzeronNum++;
            p->right = newele;
            return newele;
        }
        p = p->right;
    }
}
```

从向量初始化函数:

`initializeFromVector(Veci rows, Veci cols, Vecd vals)`：根据三个向量初始化稀疏矩阵。其中 rows 存储行索引，cols 存储列索引，vals 存储对应元素的值。函数首先释放原始的稀疏矩阵数据，然后根据传入的向量重新构建稀疏矩阵。

```
void initializeFromVector(Veci rows, Veci cols, Vecd vals) {
    delete data;
    NonzeronNum = 0;
    auto R = max_element(rows.begin(), rows.end());
    auto C = max_element(cols.begin(), cols.end());
    RowNum = *R + 1;
    ColNum = *C + 1;
    data = new Mnode[RowNum];
    Mnode* p = data;

    for (int i = 0; i < rows.size(); i++) {
        int valuei = rows[i];
```

```
        int valuej = cols[i];
        double value = vals[i];
        insert(value, valuei, valuej);
    }
}
```

其他辅助函数：

- `GetElement(Mnode* line, int j)`：辅助函数，用于获取指定行上的指定列的元素。
- `Print()`：打印当前稀疏矩阵的内容，用于调试和输出。

```
Mnode* GetElement(Mnode* line, int j){
    Mnode* p = line;
    while (p){
        if (p->j == j) {
            return p;
        }
        p = p->right;
    }
    return nullptr;
}

void Print() {
    cout << "Matrix as follow: " << ":" << endl;
    for (int i = 0; i < RowNum; i++) {
        for (int j = 0; j < ColNum; j++) {
            cout << at(i, j) << "  ";
        }
        cout << endl;
    }
}
```

实验结果

测试样例如下：

```
cout << "------------------------- Sparse Test ---------------------------" <<
endl;
Sparse<int> s(2 ,2);
s.insert(1, 0, 0);
cout << "insert 1 in (0,0)" << endl;
cout << "infer (0,0): " << s.at(0, 0) << endl;
cout << "infer (1,1): " << s.at(1, 1) << endl;
s.insert(2, 1, 1);
cout << "insert 2 in (1,1)" << endl;
cout << "infer (0,0): " << s.at(0, 0) << endl;
cout << "infer (1,1): " << s.at(1, 1) << endl;
cout << "------------------------- initializeFromVector test --------------" <<
endl;
Veci a = { 0,1,2 };
Veci b = { 0,1,2 };
Veci c = { 7,8,9 };
s.initializeFromVector(a, b, c);
s.printmatrix();
```

```
-------------------------Sparse Test-------------------------
insert 1 in (0,0)
infer (0,0): 1
infer (1,1): 0
insert 2 in (1,1)
infer (0,0): 1
infer (1,1): 2
-------------------------initializeFromVector test-------------------------
Matrix A :
7  0  0
0  8  0
0  0  9
```

稀疏矩阵的 Gauss-Seidel 迭代法

实验原理

稀疏矩阵的 Gauss-Seidel 迭代法是一种用于解线性方程组的迭代方法，特别适用于处理稀疏矩阵。稀疏矩阵是指其中绝大多数元素都是零的矩阵。

Gauss-Seidel 迭代法是一种迭代求解线性方程组的方法，它基于迭代更新近似解的过程，直至达到满足精度要求的解。与传统的直接解法相比，迭代方法在处理大规模稀疏矩阵时更为高效。

下面是 Gauss-Seidel 迭代法的基本思想：

1. 首先，将线性方程组表示为矩阵形式 $Ax = b$，其中 A 是系数矩阵，x是未知向量，b 是已知向量。
2. 对于 $Ax = b$，我们将 x 分解为$x = x_0 + dx$，其中x_0是初始解向量，dx是修正向量。
3. 将方程组 $Ax = b$ 重写为$x_0 + dx = x_0 + (L + D + U)dx = b$，其中 L、D、U分别是A 的严格下三角部分、对角线部分和严格上三角部分。
4. 利用这种分解，我们可以按照下面的方式迭代更新解向量：
    - 对于第 i 个方程，使用前面已经更新的解向量的最新值来更新x_i，即 $x_i = (b_i - (L_i * x_i + 1 + U_i * x_i - 1))/D_i$。
    - 迭代过程一直进行，直到解向量的变化量满足一定的收敛标准或者达到最大迭代次数。

Gauss-Seidel 迭代法的优点在于它可以有效地利用稀疏矩阵的结构，因为在每一次迭代中，只需要计算一个未知数的值，这大大降低了计算量。但需要注意的是，Gauss-Seidel 迭代法并不总是收敛的，收敛性取决于系数矩阵的性质。

Python的伪代码如下：

```python
def gauss_seidel_iter(A, y, epsilon=1e-6):
    n = len(A)
    B = [[0 for _ in range(n)] for _ in range(n)]
    g = [0 for _ in range(n)]
    for i in range(n):
        for j in range(n):
            if j == i:
                continue
            B[i][j] = -A[i][j] / A[i][i]
        g[i] = y[i] / A[i][i]

    x = [0 for _ in range(n)]
    cnt = 0
    for _ in range(10**6):
        for i in range(n):
```

```
            z = g[i]
            for j in range(n):
                z += B[i][j] * x[j]
            if abs(z-x[i]) < epsilon:
                cnt += 1
            else:
                cnt = 0
            x[i] = z

            if cnt >= n:
                break
        if cnt >= n:
            break
    return x
```

代码实现

接口 `Gauss_Seidel(A, b, error)` 进行代码的编写，其中有

- A: 线性方程组 Ax=b 的系数矩阵
- b: 参照线性方程组 Ax=b
- error: 当前后两次迭代的解向量之差的无穷范数小于等于 error 时，认为已经收敛并停止迭代

$$\|x_{k+1} - x_k\|_\infty \leqslant error$$

考虑到不一定稀疏矩阵的值为int或者double，我使用了模板

并且由于代码中存在除法操作，我们需要注意让分母不等于0：

```
if (std::abs(A.at(i, i)) < A.epsilon) {
    std::cerr << "Diagonal element too close to zero at row " << i << ". Cannot
proceed." << std::endl;
    return Vecd();
}
```

最后代码为：

```
template<class T>
Vecd Gauss_Seidel(Sparse<T>& A, const Vecd& b, double error) {
    // 获取稀疏矩阵 A 的行数
    int rows = A.getRowDimension();
    // 初始化解向量 x，初始值为 0
    Vecd x(rows, 0.0);
    // 初始化旧解向量 x_old，初始值为正无穷
    Vecd x_old(rows, std::numeric_limits<double>::max());
    // 初始化最大误差为正无穷
    double maxDiff = std::numeric_limits<double>::max();

    // 迭代直到误差小于指定的阈值 error
    while (maxDiff > error) {
        // 重置最大误差为 0
        maxDiff = 0.0;
        // 对于每一行 i
        for (int i = 0; i < rows; i++) {
```

```
            // 计算迭代更新的值 sum
            double sum = 0.0;
            // 对于每一列 j
            for (int j = 0; j < rows; j++) {
                // 如果 j 不等于 i，将 A(i, j) * x[j] 加到 sum 上
                if (j != i) {
                    sum += A.at(i, j) * x[j];
                }
            }

            // 如果 A 的对角元素接近于零，输出错误信息并返回空向量
            if (std::abs(A.at(i, i)) < A.epsilon) {
                std::cerr << "Diagonal element too close to zero at row " << i <<
". Cannot proceed." << std::endl;
                return Vecd();
            }

            // 保存旧解向量中的值
            double temp = x[i];
            // 更新解向量中的值，使用高斯-塞德尔迭代公式
            x[i] = (b[i] - sum) / A.at(i, i);
            // 更新最大误差
            maxDiff = std::max(maxDiff, std::abs(x[i] - temp));
        }
    }
    // 返回求解得到的解向量 x
    return x;
}
```

实验结果

顺利通过了课程网站上提供的测试样例:

$$A = \begin{pmatrix} 10 & -1 & 2 & 0 \\ -1 & 11 & -1 & 3 \\ 2 & -1 & 10 & -1 \\ 0 & 3 & -1 & 8 \end{pmatrix}$$
$$b = (6, 25, -11, 15)^T$$

求解得到的结果应该为:

```
x = [1, 2, -1, 1];
```

另外我还设计了一组额外的测试样例:

$$A = \begin{pmatrix} 2 & 2 & 3 \\ 2 & 5 & 2 \\ 3 & 1 & 7 \end{pmatrix}$$
$$b = (15, 19, 27)^T$$

求解得到的结果应该为:

$$x = [1, 2, 3];$$

结果均正确

```
------------------------- Solve Test 1--------------------------
Matrix A :
10  -1  2  0
-1  11  -1  3
2  -1  10  -1
0  3  -1  8
Gauss_Seidel:
1       2       -1      1
Conjugate_Gradient:
1       2       -1      1
------------------------- Solve Test 2--------------------------
Matrix A :
2  2  3
2  5  2
3  1  7
Gauss_Seidel:
1       2       3
Conjugate_Gradient:
0.981889        1.98968 2.97397
```

bonus: 共轭梯度法

实验原理

共轭梯度法是一种用于解决大型稀疏线性方程组的迭代算法。它是一种迭代法，通常用于求解对称正定的线性方程组，因为在这种情况下，共轭梯度法能够收敛到精确解的速度非常快。

该算法基于以下思想：在每次迭代中，共轭梯度法将梯度的负方向作为搜索方向，以确保在每一步中都朝着最速下降的方向前进，并且在搜索方向上相互共轭，从而避免了沿着同一方向多次迭代的情况。

共轭梯度法的迭代过程分为多个步骤：

1. **初始化：** 首先，选择一个初始解向量，并计算残差向量和初始搜索方向。
2. **迭代更新：** 在每次迭代中，算法计算当前搜索方向下的步长，并更新解向量。然后，计算新的残差向量，并利用前一步的搜索方向和新的残差向量来计算下一步的搜索方向。
3. **收敛判断：** 在每次迭代后，算法会检查残差向量的大小，如果小于预先设定的阈值，则停止迭代，得到近似解。

共轭梯度法的优点包括：

- **快速收敛：** 在对称正定矩阵上，共轭梯度法通常能够快速收敛到精确解，迭代次数较少。
- **内存消耗低：** 该算法只需要存储当前解向量、残差向量和搜索方向，因此在处理大型问题时内存消耗较低。
- **适用范围广：** 共轭梯度法适用于大型稀疏线性方程组，并且能够高效地解决这类问题。

然而，共轭梯度法也有一些缺点，例如对矩阵的条件数敏感，如果条件数较大，收敛速度可能会减慢；另外，该算法对初始解向量的选择较为敏感，不同的初始解可能会导致不同的收敛速度。

代码实现

接口 `Conjugate_Gradient(A, b, error, kmax)` 进行代码的编写，其中有

- A: 线性方程组 Ax=b 的系数矩阵
- b: 参照线性方程组 Ax=b
- error: 当残差 r=b-Ax 的 2 范数的平方小于等于 error 时，认为已经收敛而停止迭代
- kmax: 迭代的最大次数

$$||r||_2^2 = ||b - Ax||_2^2 \leq error$$

代码实现如下：

```
template<class T>
Vecd Conjugate_Gradient(Sparse<T>& A, const Vecd& b, double error, int kmax) {
    Vecd x(b.size(), 0.0); // 初始解向量 x 全为零
    Vecd r = subVector(b, multiplyVector(A, x)); // 计算初始残差向量 r = b - Ax
    Vecd p = r; // 设置初始搜索方向为残差向量
    double rsold = dotProduct(r, r); // 计算初始残差向量的平方范数

    for (int k = 0; k < kmax; ++k) {
        Vecd Ap = multiplyVector(A, p); // 计算 A 与搜索方向 p 的乘积
        double alpha = rsold / dotProduct(p, Ap); // 计算步长 alpha
        x = addVector(x, scalarMultiplyVector(alpha, p)); // 更新解向量 x
        r = subVector(r, scalarMultiplyVector(alpha, Ap)); // 更新残差向量 r
        double rsnew = dotProduct(r, r); // 计算更新后的残差向量的平方范数
        // 如果残差的二范数小于指定的误差，则跳出迭代
        if (std::sqrt(rsnew) < error) {
            break;
        }
        // 计算新的搜索方向
        p = addVector(r, scalarMultiplyVector(rsnew / rsold, p));
        rsold = rsnew; // 更新残差向量的平方范数
    }
    return x; // 返回求解得到的解向量 x
}
```

为了使得代码尽量的简洁欸，我定义了许多函数，具体来说，如下所示：

`函数 multiplyVector`：计算稀疏矩阵 A 与向量 x 的乘积

```
template<class T>
Vecd multiplyVector(Sparse<T>& A, Vecd& x) {
    Vecd result(x.size(), 0.0);
    // 逐行逐列计算乘积
    for (int i = 0; i < A.getRowDimension(); ++i) {
        for (int j = 0; j < A.getColDimension(); ++j) {
            result[i] += A.at(i, j) * x[j];
        }
    }
    return result;
}
```

`函数 subVector`：计算向量 x 与向量 y 的差

```
Vecd subVector(const Vecd& x, const Vecd& y) {
    Vecd result(x.size());
    // 逐元素计算差值
    for (size_t i = 0; i < x.size(); ++i) {
        result[i] = x[i] - y[i];
    }
    return result;
}
```

`函数 addVector`：计算向量 x 与向量 y 的和

```
Vecd addVector(const Vecd& x, const Vecd& y) {
    Vecd result(x.size());
    // 逐元素计算和
    for (size_t i = 0; i < x.size(); ++i) {
        result[i] = x[i] + y[i];
    }
    return result;
}
```

`函数 scalarMultiplyVector`：将向量 y 中的每个元素乘以一个标量

```
Vecd scalarMultiplyVector(double scalar, const Vecd& y) {
    Vecd result(y.size());
    // 逐元素计算乘积
    for (size_t i = 0; i < y.size(); ++i) {
        result[i] = scalar * y[i];
    }
    return result;
}
```

`函数 dotProduct`：计算两个向量 x 和 y 的点积

```
double dotProduct(const Vecd& x, const Vecd& y) {
    double result = 0.0;
    // 逐元素计算乘积并累加
    for (size_t i = 0; i < x.size(); ++i) {
        result += x[i] * y[i];
    }
    return result;
}
```

`函数 norm`：计算向量 x 的二范数

```
double norm(const Vecd& x) {
    // 利用 dotProduct 函数计算向量自身的点积，然后取平方根得到二范数
    return std::sqrt(dotProduct(x, x));
}
```

实验结果

测试样例与高斯赛德尔迭代法一致，但是共轭梯度似乎因为步数问题还未收敛

```
------------------------- Solve Test 1-------------------------
Matrix A :
10  -1  2  0
-1  11  -1  3
2  -1  10  -1
0  3  -1  8
Gauss_Seidel:
1       2       -1      1
Conjugate_Gradient:
1       2       -1      1
------------------------- Solve Test 2-------------------------
Matrix A :
2  2  3
2  5  2
3  1  7
Gauss_Seidel:
1       2       3
Conjugate_Gradient:
0.981889        1.98968 2.97397
```

运行环境

我的运行环境为windows10

为了方便助教测试，我最后的代码中把模板部分都去除了，改为了double，并且改进了构造函数与助教给出的代码对齐

我自己的测试代码命名为了`main.cpp`， `main.exe`为其生成的可运行程序

利用课程网站上的`main.cpp`中的测试程序

```
using namespace std;
#include "sparse.h"
#include "solve.h"

void test_solve(Sparse& A, Vecd& b){
    Vecd X = Gauss_Seidel(A, b, 0.000000001);
    cout << "Gauss_Seidel:" << endl;
    for (int i = 0; i < A.ColNum; i++)
        cout << X[i] << '\t';

    Vecd XC = Conjugate_Gradient(A, b, 0.000000001, 100000);
    cout << "\nConjugate_Gradient:" << endl;
    for (int i = 0; i < A.ColNum; i++)
        cout << XC[i] << '\t';
    cout << endl;
}

int main() {
    Sparse S;

    Veci rows{0, 0, 1, 1, 1, 2, 2, 2, 3, 3, 3, 3, 4, 4, 4, 4, 5, 5, 5};
    Veci cols{0, 4, 0, 1, 5, 1, 2, 3, 0, 2, 3, 4, 1, 3, 4, 5, 1, 4, 5};
    Vecd vals{10, -2, 3, 9, 3, 7, 8, 7, 3, 8, 7, 5, 8, 9, 9, 13, 4, 2, -1};
    S.initializeFromVector(rows, cols, vals);

    std::cout << "The matrix:" << std::endl;
    for (int i = 0; i < 6; i++) {
        for (int j = 0; j < 6; j++)
            std::cout << S.at(i, j) << ' ';
        std::cout << std::endl;
```

```
    }

    std::cout << std::endl << "insert(3, 4, 4)" << std::endl;
    S.insert(3, 4, 4);

    std::cout << "The matrix:" << std::endl;
    for (int i = 0; i < 6; i++) {
        for (int j = 0; j < 6; j++)
            std::cout << S.at(i, j) << ' ';
        std::cout << std::endl;
    }

    /* test Gauss-Seidel method */
    vals = {10, -1, 2, 0, -1, 11, -1, 3, 2, -1, 10, -1, 0, 3, -1, 8};
    cols = {0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3};
    rows = {0, 0, 0, 0, 1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3};
    Vecd b = {6, 25, -11, 15};
    S.initializeFromVector(rows, cols, vals);

    Vecd x = Gauss_Seidel(S, b, 1e-12);
    std::cout << std::endl << "Gauss-Seidel result:" << std::endl;
    for (int i = 0; i < x.size(); i++)
        std::cout << x[i] << ' ';
    std::cout << std::endl;

    /* (bonus) test conjugate gradient method */
    x = Conjugate_Gradient(S, b, 1e-12, 100);
    std::cout << std::endl << "Conjugate Gradient result:" << std::endl;
    for (int i = 0; i < x.size(); i++)
        std::cout << x[i] << ' ';
    std::cout << std::endl;

    return 0;
}
```

顺利通过测试