Lab 1 - OpenCV 的安装和使用

学号：3210106034
姓名：王伟杰

实验环境：

```
计算机名: LHMD-LAPTOP
操作系统: Windows 10 专业版 64 位 (10.0，内部版本 19045)
语言: 中文(简体) (区域设置: 中文(简体))
系统制造商: ASUSTeK COMPUTER INC.
系统型号: ROG Zephyrus G14 GA401QM_GA401QM
BIOS: GA401QM.415
处理器: AMD Ryzen 9 5900HS with Radeon Graphics         (16 CPUs), ~3.3GHz
内存: 32768MB RAM
```

```
Microsoft Visual Studio Professional 2017
版本 15.9.59
```

实验内容

- 自行实现一种稀疏矩阵
- 在自己实现的稀疏矩阵的表达的基础上，实现 Gauss-Seidel 方法，用于求解大规模的稀疏线性方程组
- [Bonus]实现共轭梯度法求解线性方程组

理论分析

稀疏矩阵

稀疏矩阵(sparse matrix)，在数值分析中，是其元素大部分为零的矩阵。反之，如果大部分元素都非零，则这个矩阵是**稠密(dense)**的。在科学与工程领域中求解线性模型时经常出现大型的稀疏矩阵。

在使用计算机存储和操作稀疏矩阵时，经常需要修改标准算法以利用矩阵的稀疏结构。由于其自身的稀疏特性，通过压缩可以大大节省稀疏矩阵的内存代价。更为重要的是，由于过大的尺寸，标准的算法经常无法操作这些稀疏矩阵。

在存储时，我们有以下较常用的方法：

COO，Coordinate List

由三个数组组成，分别存储非零元素的行号、列号和值。这种方法简单直接，便于构造稀疏矩阵，但对于矩阵的运算和访问并不高效，尤其是在需要频繁查找特定元素的情况下。

CSR，Compressed Sparse Row

压缩行存储是一种更为高效的稀疏矩阵存储方法，特别适合行访问。它使用三个数组：`values` 存储非零元素的值，`col_indices` 存储对应的列索引，`row_pointers` 存储每行第一个非零元素在 `values` 数组中的位置。CSR 格式便于快速地按行访问稀疏矩阵，特别适合矩阵乘法等操作。

CSC，Compressed Sparse Column

压缩列存储与压缩行存储类似，但它是按列来组织数据的。它也使用三个数组：`values` 存储非零元素的值，`row_indices` 存储对应的行索引，`col_pointers` 存储每列第一个非零元素在 `values` 数组中的位置。CSC 格式特别适用于列访问和基于列的计算。

BSR，Block Compressed Row Storage

块状压缩存储是对 CSR 的一种扩展，适用于那些有大量相同大小的密集子矩阵的稀疏矩阵。在 BSR 中，矩阵被划分为较大的“块”，每个块内部是密集的，但在块之间是稀疏的。这种方法可以减少存储空间并提高矩阵运算的效率。

DIA，Diagonal Storage

对角线存储专门用于存储那些非零元素主要分布在对角线附近的稀疏矩阵。在这种方法中，矩阵是按对角线来存储的，每个对角线上的元素存放在数组的一行中，非对角线位置上的元素如果为零则不存储，这样可以有效减少存储空间。

本次实验我们使用CSR的存储方法，CSR格式主要由三个部分组成：

1. values（或data）数组：这个数组按照行的顺序存储矩阵中的所有非零元素值。
2. col_indices数组：这个数组存储与 `values` 数组中每个元素相对应的列索引，表示该非零元素在其所在行中的位置。
3. row_pointers数组：这个数组存储每一行第一个非零元素在 `values` 数组中的索引。 `row_pointers` 的长度比行数多1，最后一个元素是非零元素总数。

考虑下面这个5x5的稀疏矩阵：

```
0 0 0 0 0
5 8 0 0 0
0 0 3 0 0
0 6 0 0 0
0 0 0 0 1
```

使用CSR格式存储上述矩阵：

1. values数组：[5, 8, 3, 6, 1]。这是矩阵中所有非零元素的值，按照行的顺序排列。
2. col_indices数组：[0, 1, 2, 1, 4]。这表示每个非零元素在其所在行的列索引。例如，第一个非零元素5位于第二行第一列，因此对应的列索引是0（基于0的索引）。
3. row_pointers数组：[0, 0, 2, 3, 4, 5]。这表明：
   - 第一行的第一个非零元素在 `values` 数组中的索引是0（实际上，第一行没有非零元素，所以这里指向的是第二行的第一个非零元素）。
   - 第二行的第一个非零元素在 `values` 数组中的索引是0。
   - 第三行的第一个非零元素在 `values` 数组中的索引是2。
   - 第四行的第一个非零元素在 `values` 数组中的索引是3。
   - 第五行的第一个非零元素在 `values` 数组中的索引是4。
   - 最后一个元素5表示非零元素的总数。

Gauss-Seidel 迭代法

Gauss-Seidel 迭代法是一种用于求解线性方程组的迭代方法，特别适用于大型稀疏矩阵。

首先，我们有线性方程组 $Ax = b$，其中：

- A 是一个 $n \times n$ 的系数矩阵，

- x 是一个未知向量，
- b 是结果向量。

当 A 是稀疏矩阵时，大多数 A 的元素都是0。

Gauss-Seidel 方法迭代求解 x 的值。与简单的迭代方法如雅可比迭代法相比，Gauss-Seidel方法在每一步迭代中都会使用最新更新的值，这通常会加快收敛速度。

Gauss-Seidel迭代公式如下：

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

其中，k 是迭代次数，$x_i^{(k+1)}$ 是第 i 个未知数在第 $k+1$ 次迭代的值。

用矩阵来说，Gauss-Seidel方法的定义可以表示为

$$\mathbf{x}^{(k)} = (\mathbf{D} - \mathbf{L})^{-1} \left(\mathbf{U}\mathbf{x}^{(k-1)} + \mathbf{b} \right)$$

其中D、$-L$和$-U$分别表示A的对角线部分、下三角部分和上三角部分。

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：

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

共轭梯度法

共轭梯度法的核心思想基于共轭方向的概念。在数学上，如果两个非零向量p和q对于一个正定对称矩阵A满足$p^T Aq = 0$，则称p和q关于A共轭。在共轭梯度法中，通过寻找一系列共轭方向，并沿这些方向进行搜索，从而逐步逼近方程的解或优化问题的最小值。

下面是共轭梯度法的迭代伪代码：

$\mathbf{r}_0 := \mathbf{b} - \mathbf{A}\mathbf{x}_0$
if $\mathbf{r}_0$ is sufficiently small, then return $\mathbf{x}_0$ as the result
$\mathbf{p}_0 := \mathbf{r}_0$
$k := 0$
repeat
$$\alpha_k := \frac{\mathbf{r}_k^\mathsf{T}\mathbf{r}_k}{\mathbf{p}_k^\mathsf{T}\mathbf{A}\mathbf{p}_k}$$
$\mathbf{x}_{k+1} := \mathbf{x}_k + \alpha_k\mathbf{p}_k$
$\mathbf{r}_{k+1} := \mathbf{r}_k - \alpha_k\mathbf{A}\mathbf{p}_k$
if $\mathbf{r}_{k+1}$ is sufficiently small, then exit loop
$$\beta_k := \frac{\mathbf{r}_{k+1}^\mathsf{T}\mathbf{r}_{k+1}}{\mathbf{r}_k^\mathsf{T}\mathbf{r}_k}$$
$\mathbf{p}_{k+1} := \mathbf{r}_{k+1} + \beta_k\mathbf{p}_k$
$k := k + 1$
end repeat
return $\mathbf{x}_{k+1}$ as the result

实验细节

稀疏矩阵

我定义了一些变量用来以CSR方法存储稀疏矩阵：

```
int rowNum, colNum;
int nzeroNum;
Vecd val;
Veci colIndex, rowOffset;
```

其中 `rowNum` 和 `colNum` 是矩阵的列数和行数， `nzeroNum` 是非零数的数量， `val` 数组存储所有非零的值， `colIndex` 存储每个非零值所在列， `rowOffset` 存储每一行第一个非零元素在 `val` 数组中的索引。

除此之外，还实现了几种方法：

`Sparse::at(int row, int col) const`

这个函数的目的是获取位于特定行 `row` 和列 `col` 的元素值。它首先使用 `rowOffset` 数组找到对应行的起始和结束索引（ `start` 和 `end` ），这个数组中存储的是每一行非零元素在 `val` 数组中的起始位置。接着，函数遍历这一行的所有非零元素，如果找到一个列索引与所查询的列 `col` 相匹配，就返回该位置的值。如果遍历完毕后都没有找到匹配的列索引，意味着这个位置的值为零，函数返回0。

`Sparse::insert(double val, int row, int col)`

此函数用于在特定位置插入或更新一个元素的值。首先，它找到要操作的行的起始和结束索引，然后遍历该行的非零元素，查找列索引。如果找到了对应的列索引，就更新该位置的值并立即返回。如果没有找到对应的列，意味着这是一个新的非零元素，需要将其插入到合适的位置。此时，它会在 `val` 和 `colIndex` 数组的适当位置插入新元素和其列索引，并更新 `rowOffset` 数组来反映每一行非零元素数量的变化。最后，非零元素数量 `nzeroNum` 增加1。

`Sparse::initializeFromVector(const Veci& rows, const Veci& cols, const Vecd& vals)`

这个函数是用来初始化一个稀疏矩阵的，它接收三个向量： `rows` （每个非零元素的行索引），`cols` （每个非零元素的列索引）和 `vals` （每个非零元素的值）。首先，它计算矩阵的行数和列数，即 `rows` 和 `cols` 中的最大值加1。然后，它遍历 `vals` 向量来统计非零元素的数量（忽略小于某个阈值 `epsilon` 的元素，以处理精度问题）。接着，为 `val` 、 `colIndex` 和 `rowOffset` 分配足够的空间，并初始化 `rowOffset` 为零。之后，它遍历所有的非零元素，填充 `val` 和 `colIndex` ，同时更新 `rowOffset` 来记录每行非零元素的累积总数。最后，通过累加 `rowOffset` 数组来确定每行非零元素在 `val` 数组中的确切起始位置。

Gauss-Seidel 迭代法

```
Vecd Gauss_Seidel(const Sparse& A, const Vecd& b, double error) {
    int n = b.size();
    Vecd x(n, 0);
    Vecd x_new(n, 0);
    double sum1, sum2;
    double diff;
    do {
        for (int i = 0; i < n; i++) {
            sum1 = 0;
            sum2 = 0;
            for (int j = 0; j < i; j++) {
                sum1 += A.at(i, j) * x_new[j];
            }
```

```
            for (int j = i + 1; j < n; j++) {
                sum2 += A.at(i, j) * x[j];
            }
            x_new[i] = (b[i] - sum1 - sum2) / A.at(i, i);
        }
        diff = 0;
        for (int i = 0; i < n; i++) {
            diff += fabs(x_new[i] - x[i]);
        }
        x = x_new;
    } while (diff > error);
    return x;
}
```

初始化

1. **初始化变量**：首先，需要一个初始解向量 `x`，这里将其初始化为0。同时，声明一个新的解向量 `x_new`，用于在每次迭代中存储更新后的解。
2. **设定收敛标准**：`error` 是给定的容差限，用来判断算法是否已经收敛。当连续两次迭代解之间的差异小于这个值时，认为算法已经收敛。

迭代过程

1. **外层循环**：外层循环控制迭代过程，只有当解的更新量 `diff` 大于容差限 `error` 时，才继续迭代。
2. 内层循环（计算新的解）
   - 对于方程组中的每一个方程 `i`，计算该方程的解。
   - 计算解的过程分为两部分：
     - `sum1` 表示所有 `j < i` 的已经计算过的新解 `x_new[j]` 与系数 `A.at(i, j)` 的乘积之和。这一部分体现了Gauss-Seidel 方法的特点，即使用已经更新过的最新解进行计算。
     - `sum2` 表示所有 `j > i` 的当前解 `x[j]` 与系数 `A.at(i, j)` 的乘积之和，这里使用的还是上一次迭代的解。
   - 利用 `sum1` 和 `sum2` 以及当前方程的常数项 `b[i]`，按照公式更新解 `x_new[i]`。
3. **计算更新量 `diff`**：通过计算新旧两个解向量之间的差的绝对值之和，来判断解的更新是否足够小，即是否接近收敛。
4. **更新解向量**：将 `x_new` 的值复制给 `x`，为下一次迭代做准备。

收敛判断

- 循环继续进行，直到新旧解之间的差异 `diff` 小于等于给定的容差限 `error`，此时认为算法已经收敛，返回当前的解向量 `x`。

共轭梯度法

在实现共轭梯度法的时候，我参考了Wikipedia上的伪代码思路，具体伪代码在上述理论分析环节已附上。

```
Vecd Conjugate_Gradient(const Sparse& A, const Vecd& b, double error, int kmax) {
    int n = b.size();
    Vecd x(n, 0);
    Vecd x_new(n, 0);
    Vecd r = b;
    Vecd r_new = r;
    Vecd p = r;
    Vecd Ap(n, 0);
    double r_norm;
    double alpha, beta;

    for (int k = 0; k < kmax; ++k) {
        // calculate Ap
        for (int i = 0; i < n; ++i) {
            Ap[i] = 0;
            for (int j = 0; j < n; ++j) {
                Ap[i] += A.at(i, j) * p[j];
            }
        }

        // calculate alpha
        double deno = 0, nume = 0;
        for (int i = 0; i < n; ++i) {
            deno += p[i] * Ap[i];
        }
        for (int i = 0; i < n; ++i) {
            nume += r[i] * r[i];
        }
        alpha = nume / deno;

        // update x
        for (int i = 0; i < n; ++i) {
```

```
            x_new[i] = x[i] + alpha * p[i];
        }

        // update r
        for (int i = 0; i < n; ++i) {
            r_new[i] = r[i] - alpha * Ap[i];
        }

        // calculate r_norm
        r_norm = 0;
        for (int i = 0; i < n; ++i) {
            r_norm += r_new[i] * r_new[i];
        }
        if (r_norm < error) {
            return x_new;
        }

        // calculate beta
        beta = 0;
        for (int i = 0; i < n; ++i) {
            beta += r_new[i] * r_new[i];
        }
        beta /= nume;

        // update p
        for (int i = 0; i < n; ++i) {
            p[i] = r_new[i] + beta * p[i];
        }

        x = x_new;
        r = r_new;
    }
    return x;
}
```

初始化

1. **初始化解向量** `x` 为0。
2. **计算初始残差** `r`，由于初始 `x` 为0，初始残差等于 `b`。
3. **设置初始搜索方向** `p` 等于初始残差 `r`。
4. **其他变量**：初始化 `Ap`（`A` 乘以 `p` 的结果）和残差范数 `r_norm`。

迭代过程

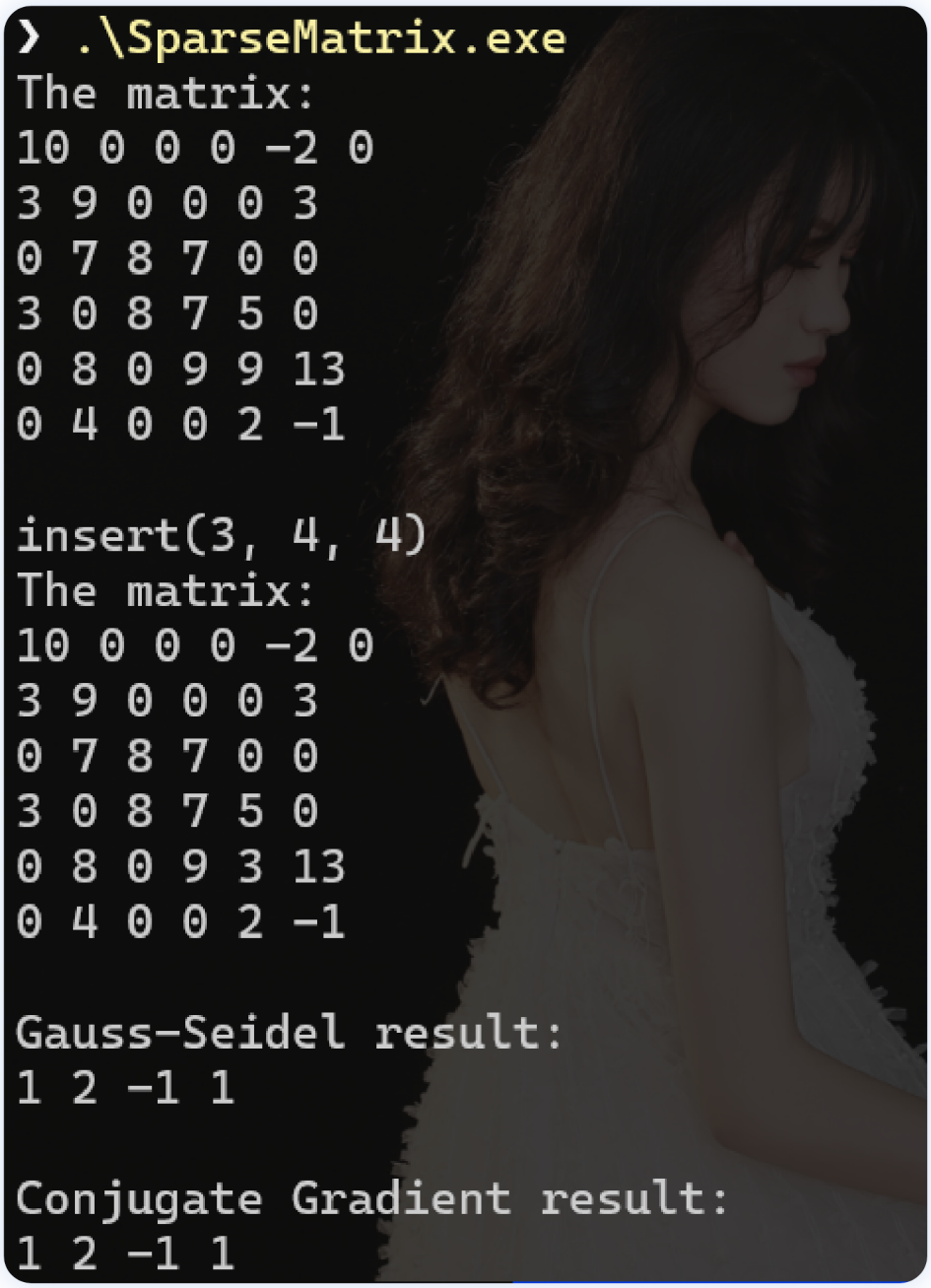
共轭梯度法的每一步迭代都尝试在当前搜索方向 `p` 上找到解的最优更新量。这个过程包括以下步骤：

1. **计算Ap**：对于当前的搜索方向 `p`，计算 `Ap`，即矩阵 `A` 与 `p` 的乘积。
2. **计算步长α**（alpha）：这是沿当前搜索方向 `p` 更新解 `x` 时使用的步长，它基于当前残差 `r` 和 `A p` 的比率计算得出。
3. **更新解x**：根据计算出的步长α和搜索方向 `p`，更新解 `x`。
4. **更新残差r**：根据步长α和 `Ap`，更新残差 `r`。
5. **计算新残差范数**：如果新残差范数小于给定的容差限 `error`，则算法收敛，返回当前解 `x`。
6. **计算β**（beta）：这个值决定了如何调整搜索方向 `p` 以确保新的搜索方向与之前的搜索方向共轭。
7. **更新搜索方向p**：根据新残差和β更新搜索方向 `p`。

收敛性判断

在每次迭代后，算法都会检查残差范数 `r_norm` 是否小于预设的容差限 `error`。如果是，算法认为已经收敛，并返回当前的解 `x_new`。否则，算法继续迭代，直到达到最大迭代次数 `kmax` 或满足收敛条件。

结果展示

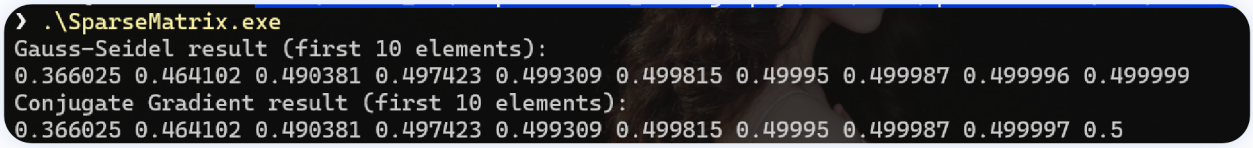


自己写的测试大矩阵的代码：

```cpp
#include "sparse.h"
#include "hw3_solve.h"
#include <iostream>
#include <vector>
#include <cmath>

int main() {
    const int n = 10000; // 矩阵和向量的大小
    Sparse S;
```

```
    std::vector<int> rows;
    std::vector<int> cols;
    std::vector<double> vals;

    for (int i = 0; i < n; ++i) {
        rows.push_back(i);
        cols.push_back(i);
        vals.push_back(4.0); // 对角线上的值较大以确保对角占优

        if (i > 0) {
            rows.push_back(i);
            cols.push_back(i - 1);
            vals.push_back(-1.0);
        }
        if (i < n - 1) {
            rows.push_back(i);
            cols.push_back(i + 1);
            vals.push_back(-1.0);
        }
    }

    S.initializeFromVector(rows, cols, vals);

    std::vector<double> b(n, 1.0);
    std::vector<double> x_gs = Gauss_Seidel(S, b, 1e-10);
    std::vector<double> x_cg = Conjugate_Gradient(S, b, 1e-10, 1000);

    // 仅展示前10个元素
    std::cout << "Gauss-Seidel result (first 10 elements):" << std::endl;
    for (int i = 0; i < 10; ++i) {
        std::cout << x_gs[i] << ' ';
    }
    std::cout << std::endl << "Conjugate Gradient result (first 10
elements):" << std::endl;
    for (int i = 0; i < 10; ++i) {
        std::cout << x_cg[i] << ' ';
    }
    std::cout << std::endl;

    return 0;
}
```

结果：



可以看到有很小的误差存在，但是大体结果是一样的。

思考

之前在数值分析课程上其实已经学过了这两种计算方法，并且使用C语言实现过求解过程，所以写起来还是比较轻松的，复习了一下这两种方法。

主要的收获是知道了稀疏矩阵的各种存储方式，除了十字链表外还有CSR等存储方法，相比链表还是要简单不少的，也知道了各种存储的优缺点和适用范围。