# High Performance Computing Project Report

Simon Wenchel

July 28, 2025

**Abstract**

In this report the work on my HPC-project will be explained and the results will be shown.

## 1 Assignment

Consider the transient convection-diffusion equation in 1D:

$$-D\frac{\partial^2 c}{\partial x^2} - v_x\frac{\partial c}{\partial x} = \frac{\partial c}{\partial t} \quad 0 \leq x \leq L, \quad t \geq 0$$

with boundary conditions:

$$\begin{cases} c(0,x) = \bar{c} \\ D\frac{\partial c}{\partial x}\Big|_{x=L} = \bar{q} \end{cases}$$

Discetization leads to the algebraic system:

$$(H+B)c + P\frac{\mathrm{d}c}{\mathrm{d}t} - q = 0 \ ,$$

where $H$,$B$,$P$ are tridiagonal matrices.

In the 2D case the convection-diffusion equation becomes:

$$\frac{\partial}{\partial x}\left(D_x\frac{\partial c}{\partial x}\right) + \frac{\partial}{\partial y}\left(D_y\frac{\partial c}{\partial y}\right) - v_x\frac{\partial c}{\partial x} - v_y\frac{\partial c}{\partial y} = \frac{\partial c}{\partial t} \quad + f$$

with boundary conditions:

$$\begin{cases} c(x,y,t) = \bar{c},(t) \quad \forall(x,y) \in \partial\Omega_1 \\ D_x\frac{\partial u}{\partial x}n_x + D_y\frac{\partial u}{\partial y}n_y = \bar{q}, \quad \forall(x,y) \in \partial\Omega_2 \end{cases}$$

The local contributions of the element $e$ on the entries $h_{ij}$ of the stiffness matrix $H$ and $p_{ij}$ of the capacity matrix $P$ read:

$$h_{ij}^{(e)} = \int_{\Delta_e}\left(D_x\frac{\partial\xi_i^{(e)}}{\partial x}\frac{\partial\xi_j^{(e)}}{\partial x} + D_y\frac{\partial\xi_i^{(e)}}{\partial y}\frac{\partial\xi_j^{(e)}}{\partial y}\right)\mathrm{d}x\ \mathrm{d}y = \frac{D_x b_i b_j + D_y c_i c_j}{4\Delta_e}$$

$$p_{ij}^{(e)} = \int_{\Delta_e}\xi_i^{(e)}\xi_j^{(e)}\mathrm{d}x\ \mathrm{d}y = \begin{cases}\Delta_e/6 & i=j \\ \Delta_e/12 & i\neq j\end{cases}$$

where $b_i, b_j, c_i$ e $c_j$ are computed using the triangle $e$ coordinates. If $i$ is a node belonging to the Neumman boundary $\partial\Omega_2$, the contribution to the $i$-th right-hand side component arises from (4):

$$q_i^{(e)} = \int_{\partial\Omega_2^{(e)}}\bar{q}\xi_i^{(e)}\mathrm{d}S = \bar{q}\frac{l^{(e)}}{2}$$

with $l^{(e)}$ the length of the side a triangle belonging to $\partial\Omega_2$ and the node $i$ as vertex.
The $b_{ij}$ entry of the convective matrix $B$ is given by:

$$b_{ij} = \int_\Omega\left(v_x\frac{\partial\xi_j}{\partial x} + v_y\frac{\partial\xi_j}{\partial y}\right)\xi_i\ \mathrm{d}x\ \mathrm{d}y$$

To solve this problem two main tasks need to be handled:

1. The assembly of the linear system given a predefined mesh structure.
2. The efficient solution of the large linear system obtained

The primary objective of this report is to highlight the implementation of parallel programming paradigms as the preferred solution for these tasks.

## 2 Implementation Framework

### 2.1 Used OpenMP Paradigms

OpenMP is a programming model that helps in parallelizing tasks on shared memory architectures by dividing the work into smaller tasks that can be executed concurrently by multiple threads.
The number of threads can be set in the terminal with command:

```
export OMP_NUM_THREADS=n
```

#### 2.1.1 `for`

The `for` paradigm is used to parallelize loops. It divides the iteration between the available threads.

```
#pragma omp parallel for
for (int i = 0; i < n; i++) {
    // do something
    // ...
}
```

#### 2.1.2 `schedule`

The `schedule` command is used with `for` loops to decide how threads are scheduled. In this project, we use the `dynamic` scheduling approach as shown in the following example. Using dynamic scheduling helps reduce the amount of time threads spend waiting. The formula `n/1000+1` determines the size of each chunk. After a thread finishes processing one chunk, it receives a smaller chunk to prevent a situation where all threads have to wait for one thread to finish.

```
#pragma omp parallel for schedule(dynamic, n/1000+1)
for (int i = 0; i < n; i++) {
    // do something
    // ...
}
```

#### 2.1.3 `reduction`

`reduction` is used for reducing a simple operation, such as summation or multiplication, on a variable over multiple threads. Pairwise operations will be done such that the number of total operations is reduced by half each step.

```
int sum = 0;
#pragma omp parallel for reduction(+:sum)
for (int i = 0; i < n; i++) {
    sum += array[i];
}
```

#### 2.1.4 `atomic`

`atomic` ensures that a special location in the memory is only accessed by one thread at a time to prevent data race. This leads to potential performance reduction as the threads may need to wait to access the desired memory location.

```
#pragma omp parallel for
for (int i = 0; i < n; i++) {
    #pragma omp atomic
    counter++;
}
```

### 2.1.5 critical

The clause `critical` makes sure that only one thread enters a special section in the code. This also reduces performance because threads may need to wait for executing the `critical` code.

```
#pragma omp parallel for
for (int i = 0; i < 10; i++) {
        // do something
        // ..
        #pragma omp critical
        {
            // do something only one thread is allowed once at a time
        }
    }
```

## 2.2 Data Structures and Basic Functions

### 2.2.1 The Compressed Sparse Row Format (CSR)

The CSR format is widely used representation to store large sparse matrices in an efficient way and minimize the needed memory.
In the project, we only deal with square matrices. That's why, for the explanations, we only consider square matrices and look at an $n \times n$ matrix with $n_{\text{term}}$ entries.
The standard approach in storing a sparse matrix would be storing three lists containing the column and row index and the value of the non-zero elements. This leads to a memory occupation of $\texttt{n\_term}(2 \cdot \texttt{sizeof}(\texttt{int}) + \texttt{sizeof}(\texttt{double}))$
As shown in Table 1 the CSR format also has three lists but needs $\texttt{n\_term} - \texttt{n}$ fewer elements. The larger the matrix and the more non-zero elements the matrix get this format gets more efficient.

| **Name** | **Data Type** | **Length** | **Explanation** |
|---|---|---|---|
| values | double∗ | n_term | Array storing the non-zero values of the matrix |
| column_indices | int∗ | n_term | Array storing the column indices of the corresponding non-zero values |
| row_pointers | int∗ | nrow + 1 | Array storing the indices where each row starts in the values and column_indices arrays last index for these arrays |

Table 1: Arrays for CSR Format

### 2.2.2 The Sparse Class

The CSR format was utilized for creating an own data structure to store and manage large sparse matrices in the code all attributes and methods of this class are shown in Table 2. In the following special functions and their parallelization will be explained.

**Matrix-vector Product postMV** computes the matrix-vector product $Av = y$ in a parallel and efficient way.

Explanation:

- outer loop iterates over each row as each row gets multiplied with the vector
- inner loop iterates from the first element in the row i iat[i] to the last element in the row iat[i]-1
- This is embarrassingly parallel as the rows have no relation as well as the saved component for each y

Code:

```
void post_MV(double* v, double* y ){
    #pragma omp parallel for \\
            schedule(dynamic,nrow/1000+1)
    for(int i=0; i<nrow; i++){
        double sum = 0.0;
        // #pragma omp simd reduction(+:sum)
        for(int j=iat[i]; j<iat[i+1]; j++){
            sum += coef[j]*v[ja[j]];
        }
        y[i] = sum;
    }
}
```

| **Class Name** | sparse | |
|---|---|---|
| **Private Members** | | |
| nrow | int | Number of rows |
| ncol | int | Number of columns |
| n_term | int | Number of non-zero matrix elements |
| coef | double∗ | Pointer to non-zero matrix elements |
| iat | int∗ | Pointer to the first non-zero column index |
| ja | int∗ | Pointer to the column indices |
| **Public Members** | | |
| sparse() | sparse | Default constructor |
| sparse(int nrow, int ncol, int n_term) | sparse | Parameterized constructor |
| ˜sparse() | void | Destructor |
| *Get and Set Methods* | | |
| full2sparse (double∗∗ A, int n_row) | void | Convert full matrix to sparse |
| addition_update(sparse∗ B_ptr) | void | Addition and update operation |
| scalarMult(double alpha) | void | Scalar multiplication |
| post_MV(double∗ v, double∗ y) | void | Matrix-vector multiplication |
| pre_MV(double∗ v, double∗ y) | void | Vector-matrix multiplication |
| matrixProduct(sparse∗ A_ptr, sparse∗ result) | void | Matrix product |
| diag(double∗ v) | void | diagonal submatrix $D$ |
| getJacobi(double∗ v) | void | Inverse of the Diagonal matrix $D^{-1}$ |
| diag_x_sparse(double∗ diag) | void | Diagonal matrix times sparse |
| left_Jacobi () | double* | Left Jacobi preconditioning $D^{-1}A$ |
| printMat() | void | Print matrix |
| printComponents() | void | Print components |

Table 2: UML Diagram of the `sparse` Class

**Jacobi Preconditioning** Computes the Matrix multiplied by the Jacobi preconditioner from the left side. The left side multiplication utilizes the CSR format because it is easier to iterate over each row. This is done by four methods.

Explanation:

- the diagonal matrix is represented as an 1-D array, of which all values were set to zero before function call
- loops are the same as in matrix-vector product
- stop iterating in the row when diagonal index is met
- also embarrassingly parallel as the rows have no relation as well as the saved component for each v

Code:

```
void diag(double* v) {
    int start, end;
    #pragma omp parallel for \\
            schedule(dynamic, ncol/1000 + 1)
    for(int i = 0; i < ncol; i++) {
        start = iat[i];
        end = iat[i + 1];
        for(int j = start; j < end; j++) {
            if (ja[j] == i) {
                v[i] = coef[j];
                break;
            }
        }
    }
}
```

- compute inverse of diagonal matrix $D$
- prevent division by zero and set diagonal element to one s.t. no change in that row will take place
- count the zeros as the effectiveness of Jacobi reduces with increasing number of zeroes

```
void getJacobi(double* v) {
    this->diag(v);
    int zeros = 0;
    #pragma omp parallel for \\
            schedule(dynamic, ncol/1000 + 1)
    for(int i = 0; i < ncol; i++) {
        if (v[i] == 0) {
            v[i] = 1;
            #pragma omp atomic
            zeros++;
        }
        v[i] = 1 / v[i];
    }
    if (zeros > 0) {
        cout << "WARNING: Number of zeros\\
        in diagonal: " << zeros << endl;
    }
}
```

- loops are the same as in Matrix-vector product
- simplified matrix product because of diagonal structure
- every element in row i is multiplied by v[i]

```
void diag_x_sparse(double* diag) {
    int start, end;
    #pragma omp parallel for \\
            schedule(dynamic, ncol/1000 + 1)
    for(int i = 0; i < ncol; i++) {
        start = iat[i];
        end = iat[i + 1];
        for(int j = start; j < end; j++) {
            coef[j] *= diag[i];
        }
    }
}
```

- allocate array with guaranteed zeroes in every entry
- compute the preconditioned matrix

```
double* left_Jacobi() {
    double* v =(double*)calloc(ncol,sizeof(double));
    this->getJacobi(v);
    this->diag_x_sparse(v);
    return v;
}
```

### 2.2.3 Basic Linear Algebra Operations

In the following basic functions for standard operations are described. Some others are equivalent and can be found in the complete code.

**Scalar Product** calculate the standard scalar product of two vectors.

Explanation:

- perfect use for the reduction clause
- reduction does not need an atomic statement
- embarrassingly parallel

Code:

```
double scalar_prod(double *v1, double *v2, int nr){
    double sum = 0.0;
    #pragma omp parallel for reduction(+:sum)
    for(int i = 0; i < nr; i++){
        sum += v1[i] * v2[i];
    }

    return sum;
}
```

**Vector Update** Add a vector onto another and scale it with a factor.

Explanation:

- standard parallel for loop
- embarrassingly parallel

Code:

```
void vector_update(double *v1, double *p,\\
                        double alpha, int nr) {
    #pragma omp parallel for \\
            schedule(dynamic, nr/1000+1)
    for (int i = 0; i < nr; ++i) {
        v1[i] += alpha * p[i];
    }
}
```

**Upper Triangular System Solver** Solving an upper triangular system using backward substitution is straightforward. However, when working with large systems, efficient parallelization row-by-row is difficult due to the dependency on solving the rows below first. The basic operation is

$$\mathtt{x[row]} = \big(\mathtt{b[row]} - \sum_{\mathtt{col=row+1}}^{n} \mathtt{A[row][col]} \cdot \mathtt{x[col]}\big)/\mathtt{A[row][row]}$$

To parallelize this process, the reduction operator can be used as the summation grows with each iteration of row. However, it needs to be evaluated if the size of the system being solved is large enough to actually improve the performance using parallelization.

Code:

```
void triangularSolver(double** A, double* b,\\
                        double* x, int n) {
    for (int row = n - 1; row >= 0; row--) {
        double sum = 0.0;
        #pragma omp parallel for reduction(+:sum)
        for (int col = row + 1; col < n; col++) {
            sum += A[row][col] * x[col];
        }
        x[row] = (b[row] - sum) / A[row][row];
    }
}
```

**Maximum** Finding the maximum value in an array utilizing parallelization can be achieved by using the critical directive to prevent data races. This is necessary because other threads may have modified the maxVal variable in the meantime. The code snippet provided demonstrates the usage of critical to ensure that the comparison v[i] > maxVal is still valid before updating maxVal if needed.

```
double max(double* v, int n) {
    double maxVal = v[0];
    #pragma omp parallel for \\
            schedule(dynamic, n/1000+1)
    for (int i = 1; i < n; i++) {
        if (v[i] > maxVal) {
            #pragma omp critical
            {
                if (v[i] > maxVal) {
                    maxVal = v[i];
                }
            }
        }
    }
    return maxVal;
}
```

## 3 Assembly

### 3.1 Mesh Data

For the solution we used a non overlapping triangulation of $\Omega$ with triangular finite elements. The input data is shown in Table 3.

| File Name | File containing | Input Representation |
|---|---|---|
| `mesh` | number of elements `ne`<br>the triangle topology | 800<br>1 23 22<br>22 44 43<br>43 65 64<br>...... |
| `coord` | number of nodes `nn`<br>the node coordinates | 441<br>0.0 1.00<br>0.0 0.95<br>0.0 0.90<br>...... |
| `bound` | number of dirichlet nodes `nb`<br>the set of Dirichlet nodes and their values | 80<br>1 1.0<br>2 1.0<br>3 1.0<br>... |

Table 3: Description of the input files describing the mesh.

### 3.2 Nonzero Structure of the System Matrix

Determining the nonzero structure of a matrix is essential before further computations can proceed, especially in the CSR format. Changing a zero to a nonzero value is not feasible without storing a new matrix. A matrix entry, denoted as $H_{i,j}$, is considered nonzero if and only if two finite elements share nodes $i$ and $j$.
The mesh topology input allows for an easy derivation of this pattern. We construct the adjacency matrix $A$ with `nn` rows and `ne` columns, where each element will have a 1 at every node that is part of it.
The final pattern of the system matrix is then received by the product: $H = A^T A$.This is shown in the code below.

```
void stiffness_struct(char* fname, int ne, int nn, double** topol) {
    double** Adj = mat_allocation(ne, nn);
    double* i_idx = (double*)malloc(static_cast<int>(nn * nn * 0.2));
    double* j_idx = (double*)malloc(static_cast<int>(nn * nn * 0.2));

    int idx;
    #pragma omp parallel for schedule(dynamic, nn / 1000 + 1)
    for (int i = 0; i < ne; i++) {
        for (int k = 0; k < 3; k++) {
            idx = static_cast<int>(floor(topol[i][k]) - 1);
            Adj[i][idx] = 1;
        }
    }

    double sum;
    int nnz = 0;
    std::string tempFile = "temp_file.txt";
    std::ofstream file(tempFile, std::ofstream::trunc); // Open the temporary file in write mode with truncat
    for (int i = 0; i < nn; i++) {
        for (int j = 0; j < nn; j++) {
            sum = 0;
            for (int k = 0; k < ne; k++) {
                sum += Adj[k][i] * Adj[k][j];
            }
            if (sum > 0) {
                nnz++;
                file << j + 1 << " " << i + 1 << " " << 0 << std::endl;
            }
        }
    }
    file.close();
}
```

This technique works perfectly fine for small meshes, but with growing meshsize the performance and memory consumption are the reason why this is not working.
Therefore another function with way less memory consumption and operations was implemented as shown below.
In this version the creation of the adjacency matrix is omitted and the indices are saved directly to be ordered afterwards and than added to the CSR matrix structure directly. This could be parallelized with only adjusting vector containers by using the `atomic` operator or creating subvectors and merging them after the loop, it would be solved.
This approach lead to unsolvable errors and was not continued because of time consumption.

```
 void stiffness_struct_small_loops(sparse* H, int ne, int nn, double** topol) {
    // Create vectors to store the i_idx and j_idx pairs
    std::vector<int> i_idx;
    std::vector<int> j_idx;

    // Loop through each element
    for (int i = 0; i < ne; i++) {
        for (int k = 0; k < 3; k++) {
            int idx = static_cast<int>(floor(topol[i][k]) - 1);
            for (int l = 0; l < 3; l++) {
                int jdx = static_cast<int>(floor(topol[i][l]) - 1);
                i_idx.push_back(idx);
                j_idx.push_back(jdx);
            }
        }
    }

    // Combine i_idx and j_idx into pairs for sorting
    std::vector<std::pair<int, int>> pairs;
    for (int i = 0; i < i_idx.size(); i++) {
        pairs.push_back(std::make_pair(i_idx[i], j_idx[i]));
    }

    // Sort the pairs to bring duplicates together
    std::sort(pairs.begin(), pairs.end());

    // Remove duplicates and copy the unique pairs back to i_idx and j_idx
    int nnz = 0;
    for (int i = 0; i < pairs.size(); i++) {
        if (i == 0 || pairs[i] != pairs[i - 1]) {
            i_idx[nnz] = pairs[i].first + 1;
            j_idx[nnz] = pairs[i].second;
            nnz++;
        }
    }

    // Allocate memory for the CSR arrays
    int* ja_loc = new int[nnz];
    int* iat_loc = new int[nn + 1];

    // Copy the data from i_idx and j_idx to the CSR arrays
    for (int i = 0; i < nnz; i++) {
        ja_loc[i] = j_idx[i];
    }

    // Compute the CSR arrays from i_idx and ja_loc
    irow2iat(nn, nnz, &i_idx[0], iat_loc);

    H->set_n_term(nnz);
    H->set_ncol(nn);
    H->copy_ja(ja_loc);
    H->copy_iat(iat_loc);

    delete[] ja_loc;
    delete[] iat_loc;
}
```

## 3.3 Computation of the Right Hand Side

The right hand side is composed of the forcing function $f$ and the boundary condition that is also given as input file. In this report the forcing function is zero always.

## 3.4 Boundary Condition Enforcement

The boundary condition is enforced with the penalty method with the value $10^{15}$

```
void imposeBC(sparse* H, double** bound, double* q, int nb){
 double R = 1e15;
 int j, bound_idx, start, end;
 int* iat = H->get_iat();
 int* ja = H->get_ja();
 double* coef = H->get_coef();

 for(int i = 0; i<nb; ++i){
  bound_idx = static_cast<int>(bound[i][0])-1;
  start = iat[bound_idx];
  end = iat[bound_idx+1];

  for(j = start; j<end; j++){
   if (ja[j] == bound_idx){break;}
  }

  coef[j] = R;
  q[bound_idx] = R*bound[i][1];
 }
}
```

# 4 Generalized Minimal Residual Algorithm

GMRES is an iterative method to solve linear systems of equations. The solution is approximated by the Krylov subspace with minimal residual.

## 4.1 The Householder Version

The orthogonalization in the Arnoldi process is commonly performed using the Gram-Schmidt procedure, which can suffer from numerical instability. However, an alternative technique known as Householder projections provides improved numerical stability for the orthogonalization step. The Householder projection matrix is defined as $P = I - 2u^T u$, where $u$ is a unit vector. This definition can be utilized in the GMRES algorithm as only the vector $v$ needs to be stored to define a $P_i$ as well as $P_i v = v - u^T uv$ where the matrix-vector product reduces to simple vector operations.
In Algorithm 1 GMRES with Householder projections is presented taken from [Saad, 2003], who developed this method in 1986.

**Algorithm 1:** GMRES Algorithm with Householder projections

```
   Input  : $A$, $b$, $m$, $x_0$
   Output: $x_m$
 1 Compute $r_0 = b - Ax_0$, $z = r_0$;
 2 for $j = 1, \dots, m, m+1$ do
 3     Compute the Householder unit vector $w_j$ such that
 4     $w_j(i) = 0, i = 1, \dots, j-1$ and $(P_j z)(i) = 0, i = j+1, \dots, n$
 5     where $P_j = I - 2w_j w_j^T$;
 6     $h_{j-1} = P_j z$;
 7     if $j = 1$ then
 8         Let $\beta = e_1^T h_0$;
 9     end
10     $v = P_1 P_2 \dots P_j w_j$;
11     if $j \leq m$ then
12         Compute $z = P_j P_{j-1} \dots P_1 Av$;
13     end
14 end
15 Define $\bar{H}_m$ as the $(m+1) \times m$ upper part of the matrix $[h_1, \dots, h_m]$;
16 Compute $y_m = \text{Argmin}_y \|\beta e_1 - \bar{H}_m y\|_2$;
17 Let $y_m = (\eta_1, \eta_2, \dots, \eta_m)^T$;
18 $z = 0$;
19 for $j = m, m-1, \dots, 1$ do
20     $z = P_j(\eta_j e_j + z)$;
21 end
22 Compute $x_m = x_0 + z$;
```

**Givens Rotation** Saad also suggests in his book to use Givens rotations to transform the linear system, that needs to be solved in line 16, from Householder structure to an upper triangular matrix making it easy to solve.
The rotation matrix $\Omega_i$ is defined as

$$\Omega_i = \begin{pmatrix} 1 & & & & & & \\ & \ddots & & & & & \\ & & 1 & & & & \\ & & & c_i & s_i & & \\ & & & -s_i & c_i & & \\ & & & & & 1 & \\ & & & & & & \ddots \\ & & & & & & & 1 \end{pmatrix}$$

with $c_i^2 + s_i^2 = 1$ and $c_i$ and $s_i$ of the $i^{\text{th}}$ rotation

$$s_i = \frac{h_{i+1,i}}{\sqrt{\left(h_{ii}^{(i-1)}\right)^2 + h_{i+1,i}^2}}, \quad c_i = \frac{h_{ii}^{(i-1)}}{\sqrt{\left(h_{ii}^{(i-1)}\right)^2 + h_{i+1,i}^2}}.$$

Applying the rotation matrices on $\bar{H}_4$ the transformed system is shown below.

$$\bar{H}_4 = \begin{pmatrix} h_{11} & h_{12} & h_{13} & h_{14} \\ h_{21} & h_{22} & h_{23} & h_{24} \\ & h_{32} & h_{33} & h_{34} \\ & & h_{43} & h_{44} \\ & & & h_{54} \end{pmatrix}, \quad \bar{g}_0 = \begin{pmatrix} \beta \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}.$$

$$\bar{H}_4^{(4)} = \begin{pmatrix} h_{11}^{(4)} & h_{12}^{(4)} & h_{13}^{(4)} & h_{14}^{(4)} \\ & h_{22}^{(4)} & h_{23}^{(4)} & h_{24}^{(4)} \\ & & h_{33}^{(4)} & h_{34}^{(4)} \\ & & & h_{44}^{(4)} \\ & & & 0 \end{pmatrix}, \quad \bar{g}_4 = \begin{pmatrix} \gamma_1 \\ \gamma_2 \\ \gamma_3 \\ \gamma_4 \\ \gamma_5 \end{pmatrix}$$

**Restarting** Another suggestion from Saad is to apply restarting after some iterations to the GMRES algorithm. This is beneficial since the start vector improves by every restart and so does the generated Krylov subspace. Another improvement is that the triangular system to solve is relatively small as well as the matrices that store projections or the transformed Hessenberg matrix. For the standard GMRES the convergence is guaranteed after $n$ iterations. This property gets lost with the restarted version, but since $n$ is large this property is impractical.
The algorithm 2 is also taken from [Saad, 2003].

**Algorithm 2:** Restarted GMRES

```
  Input  : $A$, $b$, $m$, $x_0$
  Output: $x_m$
1 Compute $r_0 = b - Ax_0$, $\beta = \|r_0\|_2$, and $v_1 = r_0/\beta$;
2 Generate the Arnoldi basis and the matrix $\bar{H}_m$ using the Arnoldi algorithm starting with $v_1$;
3 Compute $y_m$ which minimizes $\|\beta e_1 - \bar{H}_m y\|_2$;
4 If satisfied, then Stop. Otherwise, set $x_0 := x_m$ and go to Step 1;
```

## 4.2 Implementation

In the following, parts of the implemented GMRES will be written down in pseudo-code and details of the implementation will be explained. When an operation is parallelized, it is mentioned once. Subsequent occurrences of the same operation are parallelized as well, without explicitly mentioning.
The restarted part will be omitted in this explanation since it is straightforward.
The full GMRES implementation can be viewed in the appendix Code A.2.
The input values can be viewed in table 4. The function does not have a return value the solution vector is stored in start vector from the input.

| Name | Data Type | Explanation |
|---|---|---|
| z_ptr | sparse* | Pointer to the sparse matrix of the linear system to be solved |
| b | double* | Pointer to the right-hand side array |
| x | double* | Pointer to the start and solution array |
| preconditioning | bool | Boolean indicating whether preconditioning is used |
| restarting | bool | Boolean indicating whether restarting is used |
| maxit | int | Maximum number of iterations |

Table 4: Input variables for the GMRES function.

**Arnoldi process** In Algorithm 3 the Arnoldi process with Householder projections of my implementation is presented and parts of the parallelization are mentioned. The parallelization can just be applied on single operations as there is data dependence on the outer loops or even the application of the projection matrices.

**Givens Rotation** In Algorithm 4 the givens rotation to obtain the triangular system is explained. In this part no real parralelization can be done because the operations are mostly on scalars and not Vectors that can be divided in parallel calculations.

**Evaluation of $x_i$** This part solves the triangular system and computes the approximation $x_i$ of the solution vector. This part will be entered right after the Givens rotation if the process is stagnating or the norm smaller than the relative tolerance

**Algorithm 3:** Householder part

```
   Input  : $A$, $x$, $b$, $U \in \mathbb{R}^{n \times inner}$
   Output: $x_m$
 1 Compute $r = b - Ax$, $u = r$, $normr = ||r||$, $\beta = \text{sig}(normr) \cdot normr$, $u = r$
 2           $u_0 = u_0 + \beta$, $u = \frac{u}{||u||}$, $U_{0,:} = u$, $w_0 = -\beta$ ;
 3 for $j = 1, \dots, outer$ do
 4     for $i = 1, \dots, inner$ do
 5         $v = -2u_i \cdot u^{(i)}$ ; // scalar multiplication done in parallel
 6         $v_i + = 1$; // just update one value
           // do:  $v = P_1 \cdots P_i e_i$, but why ei?
 7         for $k = i - 1 \dots 0$ do
 8             $v = v - 2{u^{(k)}}^T u^{(k)} \cdot v$; // scalar product and vector update is parallelized
 9         end
10         $v = \frac{v}{||v||}$; // normalization is parallel
           // do:  $v = P_i \cdots P_0 Av$
11         $v = Av$; // sparse Matrix-vector product is parallel
12         for $k = 0 \dots i$ do
13             $v = v - 2{u^{(k)}}^T u^{(k)} \cdot v$;
14         end
           // compute $u^{i+1}$ for $P_{j+1}$
15         $u^{i+1} = v$;
16         $u_{0:initer} = 0$; // zero_out is parallel
17         $\alpha = ||u||$; // norm is parallel
18         if $alpha > 0$ then
19             $\alpha = v_{i+1}\alpha$;
20             $u^{(i+1)}_{i+1} = u^{(i+1)}_{i+1} + \alpha$;
               // normalize $u^{(i+1)}$;
               // do $v = P_{i+1} v$
21             $v_{i+1:n} = 0$;
22             $v_{i+1} = -\alpha$;
23         end
           // Continue with Givens rotation...
           // ...
24     end
25 end
```

*normr* < *tolb* or after the iteration of the inner loop.

With that the main outline of the GMRES is concluded. As mentioned the parallelization could just be applied on the single operations.

## 5 Numerical Experiments

For the experiments, a MacBook Pro 2019 with an Intel(R) Core(TM) i5-8279U CPU running at 2.40GHz is used. The system has 4 physical CPU cores and 8 logical CPU cores due to hyper-threading. Additionally, the system has 8 GB of RAM.

### 5.1 Performance Metrics

The following metrics are used to evaluate the performance of the parallelization in the implemented program.

#### 5.1.1 Speedup

The speedup measures the improvement of the performance compared to the used threads. It compares the runtime of the program without parallelization divided by the runtime with parallelization.

$$S_p := \frac{T_{\text{one thread}}}{T_{p \text{ threads}}}$$

An optimal speedup is equal to the number of threads used, i.e. if 4 threads are used and the speed up is 4, the program runs 4 times faster which means the utilization of parallelization is optimal.

**Algorithm 4:** Givens rotation part

```
for $j = 1, \dots, outer$ do
    for $i = 1, \dots, inner$ do
        // ...
        // Previous code...
        // Continue with Givens rotation...
        $j_0 = v_i$; // Store $v_i$ in $j_0$
        $j_1 = v_{i+1}$; // Store $v_{i+1}$ in $j_1$
        $\rho = \sqrt{j_0^2 + j_1^2}$; // Compute the norm
        $j_0 = \frac{j_0}{\rho}$; // Normalize $j_0$
        $j_1 = \frac{j_1}{\rho}$; // Normalize $j_1$
        $J_{i,0} = j_0$; // Store the Givens rotation values
        $J_{i,1} = j_1$; // Store the Givens rotation values
        $w_{i+1} = -J_{i,1} \cdot w_i$; // Update $w_{i+1}$
        $w_i = J_{i,0} \cdot w_i$; // Update $w_i$
        $v_i = \rho$; // Update $v_i$
        $v_{i+1} = 0$; // Set $v_{i+1}$ to 0
        // $v$ is a row vector in $R$ the matrix of the triangular system
        // $w$ is the RHS of this system
        $R_{:,i} = v$;
        $normr = |w_{i+1}|$;
        // Following Code ...
        // ...
    end
end
```

### 5.1.2 Efficiency

The efficiency describes the utilization of parallelism in the program, i.e. the mean time a processor is really working.

$$E_p := \frac{S_p}{p} = \frac{T_1}{pT_p}$$

Following the definition an efficiency of 1 is optimal and means that the program is using the parallel resources 100% all the time.

### 5.1.3 Scalability

Scalability describes how well the program handles increasing workloads, in our case increased Matrix size or number of non zero elements. $r = \frac{n_{\text{increased}}}{n_{\text{reference}}}$ is the factor by which the data size increased.

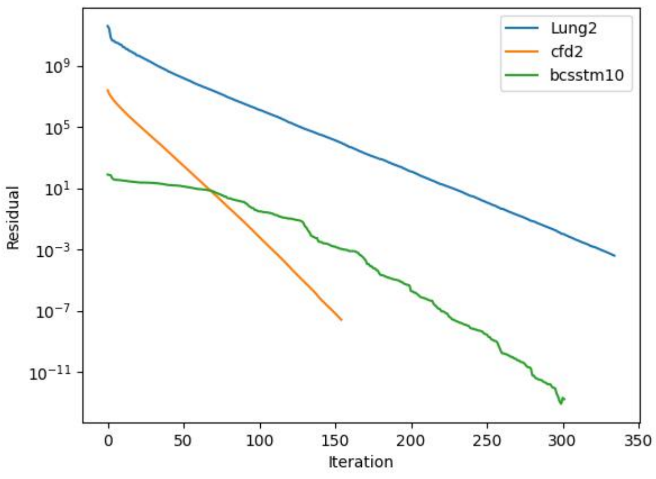
$$SB_r = r \cdot \frac{T_{\text{reference}}}{T_{\text{increased}}}$$

If $SB_r$ is greater than 1 it means the Performance decreases with increasing problem size and vice versa.

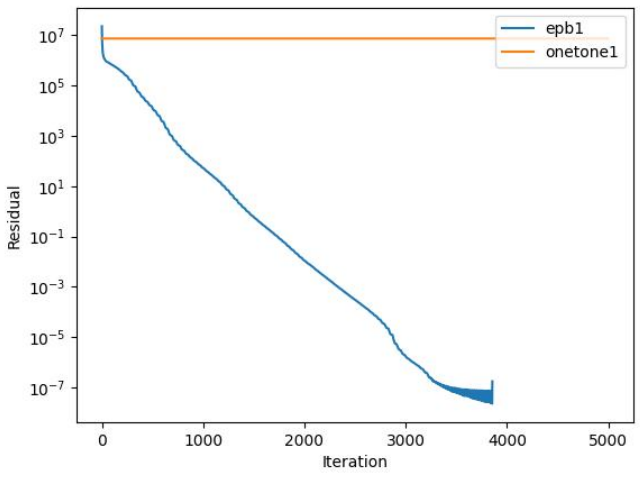
## 5.2 Performance of the GMRES

### 5.2.1 Convergence

For evaluating the quality of the implemented GMRES algorithm large matrices from SuiteSparse Matrix Collection were taken and there properties are shown in Table 5. For the right hand side of the linear systems a random vector with values between 0 and 1 with the suiting length is created.
The convergence for the different test matrices can be seen in Figure 1. The positive symmetric matrices converge very fast and do not need many iterations with respect to their size as well as the relatively small *bcsstm10* matrix. Another interesting thing we can observe is that the convergence for the two SPD matrices follow a constant exponential rate and the convergence of *bcsstm10* is very irregular.
For *epb1* it takes many iterations to converge but it does. The osciliation at the end come from the described restarted version of GMRES. Comparing the sizes of *epb1* and *cfd2* we see that the limiting factor in fast convergence is the non SPD property because the number of nonzeros is just $\frac{95053}{1605669} \approx 5.9\%$ compared to *cfd2* that converges very fast.
GMRES is not capable of solving a linear system with *onetone1* as system matrix, an improved preconditioner would be needed to solve this system.



(a)



(b)

Figure 1: Convergence plots of GMRES for the test matrices. (a) shows fast convergences and (b) slow or no convergence of GMRES

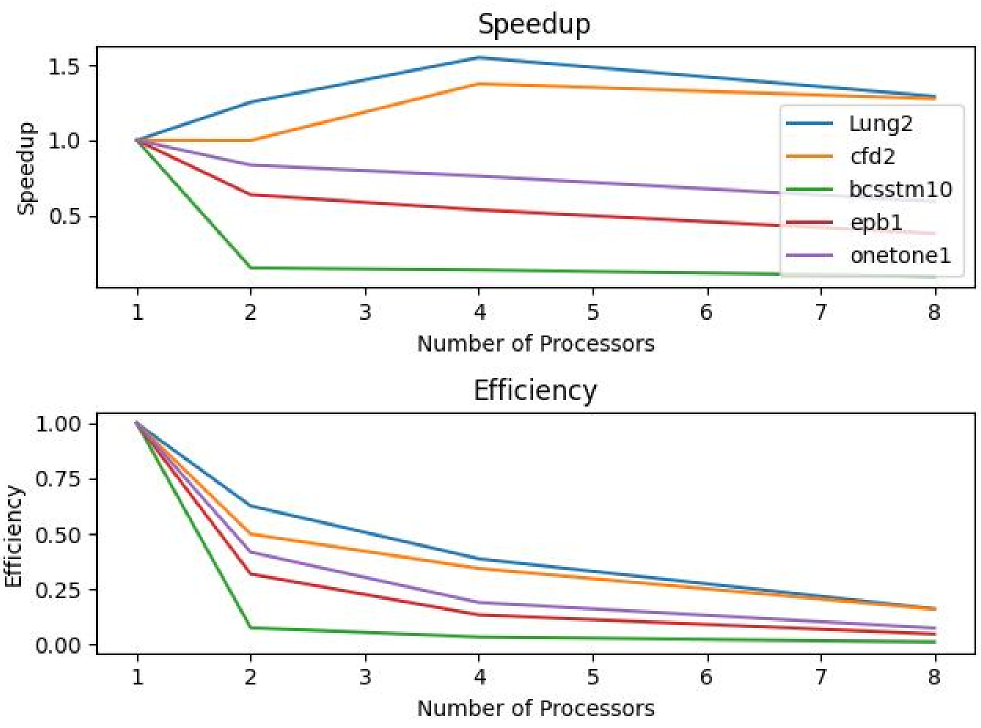


Figure 2: Speedup and Efficiency Metric for Test Matrices

**Algorithm 5:** Evaluating $x_m$

```
for $j = 1, \dots, outer$ do
    for $i = 1, \dots, inner$ do
        // ...
        // Previous code...
        // Continue with evaluation of $x_m$...
1       $y = R^{-1}w$; // solve triangular system
        // Compute additive vector from $x_m = x_0 + z$, with $z := P_j(\eta_j e_j + z)$
2       $z = -2u_i^{(i)} y_i u$;
3       $z_i = z_i + y_i$;
4       for $k$ = i-1...0 do
5           $z_k = z_k + y_k$;
6           $z = z - 2uu^T z$;
7       end
8       $x_i = x_0 + z$;
9       $r = b - Ax_i$;
        // Following Code ...
        // ...
    end
    // evaluation will be done here if not happened before
    // ...
end
```

| Name | Dimension | Nonzeros | Condition Number |
|---|---|---|---|
| **Symmetric Positive Definite (Spd) Matrices** | | | |
| Lung2 | 109460 | 492564 | - |
| cfd2 | 123440 | 1605669 | - |
| **Non-Symmetric Positive Definite Matrices** | | | |
| bcsstm10 | 1086 | 11589 | $1.258792 \times 10^5$ |
| epb1 | 14734 | 95053 | $5.940657 \times 10^3$ |
| onetone1 | 36057 | 335552 | $9.388846 \times 10^6$ |

Table 5: Test Matrices

### 5.2.2 Parallelization Performance

The runtimes for solving the linear system with the GMRES algorithm are shown in Table 6 and some performance metrics are displayed in 2.
For the SPD matrices *lung* and *cfd2* the runtime actually increases. This is the expected behaviour as the program has few steps where no parallelization can be applied and the datasize is very large (see Table 5). The speedup is still bad as the optimal speedup is equal to the number of used processors. Additionally, the runtime increases when 8 threads are used.
The non SPD matrices on the other side increase their runtime with increasing number of threads. This behaviour is inexplicable as the data size is also big and the parallelization overhead neccessary to distribute the parallel processes on different threads should increase the runtime with such a large ratio. Additionally for comparable size the speedup is greater than 1 for the SPD matrices, but the influence on runtime should not depend on the property if SPD or not.

| number of threads | 1 | 2 | 4 | 8 |
|---|---|---|---|---|
| Lung2 | 2.52803 | 2.01759 | 1.63059 | 1.95627 |
| cfd2 | 1.50406 | 1.50657 | 1.09299 | 1.17903 |
| bcsstm10 | 0.098761 | 0.65573 | 0.725824 | 1.04566 |
| epb1 | 5.65513 | 8.87665 | 10.5441 | 14.9338 |
| onetone1 | 15.0989 | 18.0937 | 19.8476 | 25.4774 |

Table 6: Execution times for different processes.

## 5.3 Solution of the Convection-Diffusion Equation

Consider the problem

$$\frac{\partial}{\partial x}\left(D_x \frac{\partial c}{\partial x}\right) + \frac{\partial}{\partial y}\left(D_y \frac{\partial c}{\partial y}\right) - v_x \frac{\partial c}{\partial x} - v_y \frac{\partial c}{\partial y} = \frac{\partial c}{\partial t} \quad + f$$

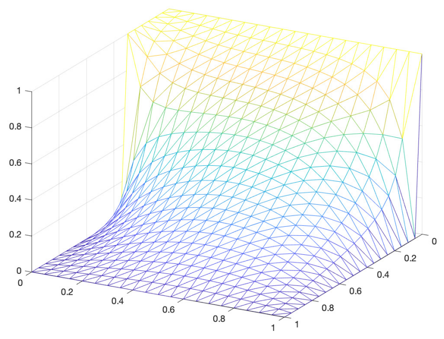
with the boundary conditions

$$\begin{aligned} c(x,t) &= [x \in \Omega_1] \qquad && \text{for } x \in \Omega_1, t > 0 \\ c(x,t) &= [x \in \Omega_2] \qquad && \text{for } x \in \Omega_2, t > 0 \end{aligned}$$

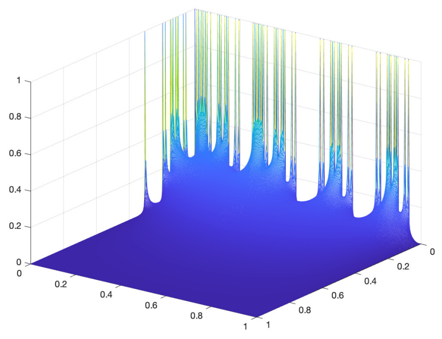
where the boundaries $\Omega_1$ and $\Omega_2$ are given by:

$$\begin{aligned} \Omega_1 &= \{(x,y) : [x = 0, 0 \le y \le 1] \cup [0 \le x \le 0.3, y = 0]\} \\ \Omega_2 &= \{(x,y) : [x = 1, 0 \le y \le 1] \cup [0 \le x \le 1, y = 1] \cup [0.3 \le x \le 1, y = 0]\} \end{aligned}$$

The correct solution of this problem on the two different meshes is shown in 3. The solutions for small and big mesh should look the same but there occurred an error in the mesh creation for the big mesh and the boundary conditions are not created correctly. Because of the small size of the problem, it is not necessary to look at the performance for parallelization. Unfortunately the evaluation for the big mesh could not be done because as mentioned the parallelization for the assembly which takes the most computational effort could not be resolved.



(a) Solution for the small mesh



(b) Solution for the big mesh

Figure 3: Solution of the Convection-Diffusion Problem.

## References

[Saad, 2003] Saad, Y. (2003). *Iterative Methods for Sparse Linear Systems Second Edition.*

# A CPP Files

## A.1 main.cpp

```cpp
#include <iostream>
#include <cmath>
#include <fstream>
#include <omp.h>
#include <cstring>

#include "sparse.h"
#include "read_mat.h"
#include "functions.h"
#include "GMRES.h"
#include "constants.h"
#include "fem_functions.h"


const int n_threads = 8;
using namespace std;

int main(int argc, char* argv[]) {
    //allocate memory for the solution of the Matrix Vector prod
    // char filename[] = "mesh/u_mattia.dat";

    // sparse mat_mattia = sparse();
    // double* b_mattia = getFEM_sparse(&mat_mattia);

    // double *xstar_mattia = (double*)malloc(mat_mattia.get_nrow()*sizeof(double));

    // // sparse* newmat = &mat_mattia;
    // gmres(filename, &mat_mattia,b_mattia, xstar_mattia, true, true, 1000);

    // save_solution(filename, xstar_mattia, mat_mattia.get_ncol());

    // free(b_mattia);
    // free(xstar_mattia);




    // FEM assembly




    //     int num_threads = 4;
    // omp_set_num_threads(num_threads);

    // Enable dynamic threads to support task-based parallelism
    // omp_set_dynamic(1);

    // Get the actual number of threads being used by OpenMP
    int actual_num_threads = omp_get_max_threads();
    std::cout << "Actual number of threads: " << actual_num_threads << std::endl;


    double start_time = omp_get_wtime();

    double D[2] = {1,1};
    double V[2] = {1,3};
    int ne, nn, nb; // number of mesh (800), number of nodes(441), number of ? (80)

    double **coord, **bound, *delta, *delta_node, *F;
    double **topol, *temp, *q, *b;



    sparse* FEM;
    bool bigmesh = true;

    char sol_path[100];


    if (bigmesh){
        string mesh_path = "bigmesh/mesh.dat";
        string nodes_path = "bigmesh/dirtot.dat";
        string coord_path = "bigmesh/xy.dat";

        ifstream mesh(mesh_path);
        ifstream nodes(nodes_path);
        ifstream xy(coord_path);


        xy >> nn;
        nodes >> nb;
        mesh >> ne;

        bound = mat_allocation(nb,2);
        coord = mat_allocation(nn,2);
        topol = mat_allocation(ne,3);

        for (int i = 0; i < nn; ++i){xy >> coord[i][0] >> coord[i][1];}
        for (int i = 0; i < nb; ++i){nodes >> bound[i][0] >> bound[i][1];}
        for (int i = 0; i < ne; ++i){mesh >> topol[i][0] >> topol[i][1] >> topol[i][2];}

        xy.close();
        nodes.close();
        mesh.close();
        char path[] = "bigmesh/u.dat";
        strcpy(sol_path, path);


    }else{
        string mesh_path = "mesh/mesh.dat";
        string nodes_path = "mesh/dirnod.dat";
        string coord_path = "mesh/xy.dat";


        ifstream mesh(mesh_path);
        ifstream nodes(nodes_path);
        ifstream xy(coord_path);

        int not_used;

        xy >> nn;
        nodes >> nb;
        mesh >> ne;


        bound = mat_allocation(nb,2);
        coord = mat_allocation(nn,2);
        topol = mat_allocation(ne,3);

        for (int i = 0; i < nn; ++i){xy >> coord[i][0] >> coord[i][1];}
        for (int i = 0; i < nb; ++i){nodes >> bound[i][0] >> bound[i][1];}
        for (int i = 0; i < ne; ++i){mesh >> topol[i][0] >> topol[i][1] >> topol[i][2]>>
            not_used;}


        for (int i = 0; i < nn; ++i){xy >> coord[i][0] >> coord[i][1];}
        for (int i = 0; i < nb; ++i){nodes >> bound[i][0] >> bound[i][1];}
        for (int i = 0; i < ne; ++i){mesh >> topol[i][0] >> topol[i][1] >> topol[i][2];}

        xy.close();
        nodes.close();
        mesh.close();
        char path[] = "mesh/u.dat";
        strcpy(sol_path, path);
    }


    //read data



    // printMat(coord,nb,2);



    double h = sqrt(2)*(coord[0][1]-coord[1][1]);// why sqrt2
    double tau = 0.1; // whats tau?

    delta = (double*)malloc(ne*sizeof(double));

    delta_node = (double*)calloc(nn,sizeof(double));

    F = (double*)malloc(nn*sizeof(double));
    q = (double*)calloc(nn,sizeof(double));
    b = (double*)malloc(nn*sizeof(double));


    FEM = create_FEM_mat(coord, topol, ne, nn, D, h, tau, V, delta);

    int idx;
    #pragma omp for schedule(dynamic,nn/1000+1)
    for(int i = 0; i<ne; i++){
        for(int k=0; k<3; k++){
            idx = static_cast<int>(floor(topol[i][k])-1);
            delta_node[idx] += delta[k]/3;
        }
    }

    // RHS
    for(int i = 0; i<nn; i++){
        F[i] = force(coord[i][0],coord[i][1]*delta_node[i]);
    }

    // BCs
    imposeBC(&FEM[0], bound, q, nb);
    FEM[1].addition_update(&FEM[2]);
    FEM[0].addition_update(&FEM[1]);
    // printVec(FEM[0].get_coef(),FEM[0].get_n_term());
    double* x = (double*)calloc(FEM[0].get_ncol(),sizeof(double));


    vector_update(q,F,1,nn);
    // printVec(q,nn);

    // FEM Solver
    cout<< norm(q,nn)<<endl;
    sparse* A = new sparse;
    // b = getAssembledSystem(A);

    // double* x = (double*)calloc(A->get_ncol(),sizeof(double));
    // FEM[0].printComponents();


    gmres(sol_path,&FEM[0],q, x, true, true, 1000);
    double end_time = omp_get_wtime();
    double runtime = (end_time - start_time);

    save_solution(sol_path,x,FEM[0].get_ncol());


    free(b);
    free(x);
    delete A;

    return 0;

    }
```

## A.2 GMRES.h

```
void gmres(char* matname, sparse* A_ptr, double* b, double* x, bool preconditioning, bool
    restarting, int maxit){
    /*
    GMRES Algorithm with
        - Householder projections
        - Givens rotation

    Input:
        - A_ptr: pointer to sparse Matrix of the linear system to be solved
        - b: ptr to rhs
        - x: ptr to solution vector (and start vector?)





    Flag is set to see the status of the Algorithm
    flag = 0: succesful convergence
        - inner loop: rormr_act < tolb
        - outer loop: rormr_act < tolb after the last evaluation
    flag = 1: maximum iterations reached without convergence
        - initialization if it stays unchanged: max iterations reached
    flag = 2: stagnation
        - no significant change in norm reduction

    */
    sparse& A = *A_ptr;

    int nc = A.get_ncol();
    int nr = A.get_nrow();

    if(nc!=nr){
        cout<<"ERROR need square Matrix n != n"<<endl;
        return;
    }

    const int n = nc;

    double make_relres_smaller=1;
    if (preconditioning){
        double* M_inv = A.left_Jacobi();
        elementwise_prod(b,M_inv,n);
    }else{
        make_relres_smaller = 1.0e-15;
    }

    int restart, outer;
    if(restarting){
        restart = 20;
        outer = maxit/restart;
        maxit = restart; // 10 is small
    }
    int inner = maxit;




    double tol = 1e-15;
    double n2b = norm(b,n);


    int flag = 1;

    int imin = 0;
    int jmin = 0;
    double tolb = tol*n2b*make_relres_smaller;

    int stag = 0;
    int moresteps = 0;
    int maxmsteps = 3;
    int maxstagsteps = 3;


    double* r = (double*)malloc((n)*sizeof(double));
    double* Ax = (double*)malloc((n)*sizeof(double));
    double* xm = (double*)malloc((n)*sizeof(double));
    double* Axm = (double*)malloc((n)*sizeof(double));

    A.post_MV(x,Ax);
    vector_add(1.,b,-1.,Ax,r,n);
    double normr = norm(r, n);

    double* resvec = (double*)malloc((inner*outer+1)*sizeof(double));
    resvec[0] = normr;
    double normrmin = normr;

    // preallocate

    // J is of size (inner x 2)
    // Values used in one iteration are in the same array
    // working direction J[row][i] ist ja doch die falsche working direction oder
    double** J = (double**)malloc(inner * sizeof(double*)); // Matrix for givens rotation J(2,
        inner)
    for (int i = 0; i < inner; ++i) {
        J[i] = (double*)malloc(2 * sizeof(double));
    }

    // U is of size (inner x n)
    // Values used in one iteration are in the same array
    // working direction U[row][i] = i;
    double** U = (double**)malloc((inner+1) * sizeof(double*)); // Matrix for holding
        Householder reflectors u(w) U(n,inner)
    for (int i = 0; i < (inner+1); ++i) {
        U[i] = (double*)malloc(n * sizeof(double));
    }


    double* u = (double*)malloc(n * sizeof(double)); // row vector of u as helper;

    double** R = (double**)malloc(inner * sizeof(double*)); // givens rotated Hessenbergmatrix
        R(inner,inner)
    for (int i = 0; i < inner; ++i) {
        R[i] = (double*)malloc(inner * sizeof(double));
    }

    double* w = (double*)malloc((n+1)*sizeof(double));

    double* v = (double*)malloc((n+1)*sizeof(double));
    double* vtemp = (double*)malloc((n+1)*sizeof(double));

    double* jtemp = (double*)malloc(2*sizeof(double)); // there to save v(initer:initer+1)


    double beta;
    double alpha;


    double relres; //relative residual

    int initer; // so that they live outside the loop
    int outiter;
    int iterations = 0;
    int idx;

    double* y = (double*)malloc((inner+1)*sizeof(double));
    double* additive = (double*)malloc(n*sizeof(double));


    double start_time = omp_get_wtime();

    normr = norm(r,n);
    beta = scalarsgn(normr)*normr;
    copyArray(u, r, n);
    u[0] += beta;
    normalize(u, n);

    copyArr2Mat_col(U,u,0,n);
    //  Apply Householder projection to r.
    //  w = r - 2*u*u'*r;
    w[0] = -beta;

    for (outiter = 0; outiter < outer; outiter++){
        for(initer=0; initer<inner; initer++){
                iterations = (outiter)*inner+initer+1;
                // copyArray(v, u, n);      // wofuer das
                vector_scalarMult(v, u, -2.0*u[initer], n);
                v[initer] += 1;

                // v = P1*P2*...Pjm1*(Pj*ej)
                for(int k = initer - 1; k >= 0; k--){
                    // copyMat_col2Arr(U, u, k, n);
                    // vector_update(v, u, -2*scalar_prod(u,v,n), n);
                    vector_update(v, U[k], -2*scalar_prod(U[k],v,n), n);

                }

                normalize(v, n);
                copyArray(vtemp, v, n);

                A.post_MV(vtemp, v);

                // Form Pj*Pj-1*...*P1*Av
                for(int k=0; k<=initer; k++){
                    // copyMat_col2Arr(U, u, k, n);
                    // vector_update(v, u, -2*scalar_prod(u,v,n), n);
                    vector_update(v, U[k], -2*scalar_prod(U[k],v,n), n);

                }


                // determine Pj+1
                if(initer!= n-1){ // if not last iteration // not necessary in restarted
                    version
                    // construct u for Pj+1
                    copyArray(u, v, n);
                    vector_zero_out(u, 0, initer, n);

                    alpha = norm(u,n);

                    if (alpha > eps){
                        alpha *= scalarsgn(v[initer+1]);
                        //u = v(initer+1:end) + sign(v(initer+1))*||v(initer+1:end)||*e_{initer
                            +1)
                        u[initer+1] += alpha;

                        normalize(u, n);
                        copyArr2Mat_col(U, u, initer+1, n);

                        // Apply Pj+1 to v
                        vector_zero_out(v, initer+1, n-1, n);
                        v[initer+1] -= alpha;
                    }
                }


                for(int colJ = 0; colJ<initer; colJ++){
                    double tmpv = v[colJ];
                    v[colJ] = J[colJ][0]*v[colJ] + J[colJ][1]*v[colJ+1];
                    v[colJ+1] = -J[colJ][1]*tmpv + J[colJ][0]*v[colJ+1];
                }


                // compute Given's rotation Jm
                if (initer!= n-1){
                    jtemp[0] = v[initer];
                    jtemp[1] = v[initer+1];

                    double rho = norm(jtemp,2);
                    normalize(jtemp,2);
                    copyArr2Mat_col(J,jtemp,initer,2);

                    w[initer+1] = -J[initer][1]*w[initer];
                    w[initer] *= J[initer][0];

                    v[initer] = rho;
                    v[initer+1] = 0;
                }


                copyArr2Mat_row(R,v,initer,inner);

                normr = scalarsgn(w[initer+1])*w[initer+1];     // abs(w[initer+1])
                resvec[iterations] = normr;


                if (normr <= tolb || stag >= maxstagsteps || moresteps){
                // normr smaller than relative tolerance
                // stagnation larger than allowed
                // if moresteps set to 1 to allow further improvement
                // one of these is true: allow to compute the solution vector x

                    vector_zero_out(additive,n);
                    vector_zero_out(y,initer+1);

                    triangularSolver(R,w,y,initer+1);
                    copyMat_col2Arr(U, u, initer, n);
                    vector_scalarMult(additive,u,-2*y[initer]*U[initer][initer],n);
                    additive[initer] += y[initer];
                    for(int k=initer-1;k>=0;k--){
                        // copyMat_col2Arr(U, u, k, n);
                        // additive[k] += y[k];
                        // double tmpscalar = -2*scalar_prod(u,additive,n);
                        // vector_update(additive,u,tmpscalar,n);

                        additive[k] += y[k];
                        double tmpscalar = -2*scalar_prod(U[k],additive,n);
                        vector_update(additive,U[k],tmpscalar,n);
                    }
                    if(norm(additive,n)<eps*normr){ //check if the additive vector is big
                       enough to have an influence
                        stag += 1;
                    }else{
                        stag = 0;
                    }
```

```
                // copyArray(xm,x,n);
                // vector_update(xm,additive,1,n);  // xm = x + additive
                vector_add(1,x,1,additive,xm,n);




                A.post_MV(xm,Axm);
                vector_add(1.,b,-1.,Axm,r,n);
                if (norm(r,n)<= tolb){
                    copyArray(x,xm,n);
                    flag = 0;
                    break;
                }
                normr = norm(r,n);

                resvec[iterations] = normr;





                if (normr <= tolb){
                    copyArray(x,xm,n);
                    flag = 0;
                    jmin = initer;
                    break;
                }else{
                    if (stag >= maxstagsteps && moresteps ==0){
                        stag = 0;
                    }
                    moresteps += 1;
                    if (moresteps >= maxmsteps){
                        cout<<"stagnation"<<endl;
                        flag = 2;
                        jmin = initer;
                        break;
                    }
                }
            }//endif (normr <= tolb || stag >= maxstagsteps || moresteps)


            if (stag>=maxstagsteps){
                flag = 2;
                break;
            }
        }// end for in

        if (flag != 0){ // computes the x with the lowest norm
            vector_zero_out(additive,n);

            vector_zero_out(y,initer+1);
            triangularSolver(R,w,y,initer);
            // copyMat_col2Arr(U, u, initer, n);
            // vector_scalarMult(additive,u,-2*y[initer]*U[initer][initer],n);

            vector_scalarMult(additive,U[initer],-2*y[initer]*U[initer][initer],n);


            additive[initer] += y[initer];
            for(int k=initer-1;k>=0;k--){
                // copyMat_col2Arr(U, u, k, n);
                // additive[k] += y[k];
                // double tmpscalar = -2*scalar_prod(u,additive,n);
                // vector_update(additive,u,tmpscalar,n);

                additive[k] += y[k];
                double tmpscalar = -2*scalar_prod(U[k],additive,n);
                vector_update(additive,U[k],tmpscalar,n);
            }

            vector_update(x,additive,1,n);

            A.post_MV(x,Ax);
            vector_add(1.0,b,-1.0,Ax,r,n);

            normr = norm(r,n);

            normr = norm(r,n);
            beta = scalarsgn(normr)*normr;

            vector_zero_out(u,inner);
            mat_zero_out(U,n,inner+1);
            copyArray(u, r, n);
            u[0] += beta;
            normalize(u, n);

            copyArr2Mat_col(U,u,0,n);
            //  Apply Householder projection to r.
            //  w = r - 2*u*u'*r;
            w[0] = -beta;
        }// end if flag~=0


        if (flag == 2) { //stagnation
            break;
        }

        if (normr <= tolb) { // convergence
            flag = 0;
            break;
        }


    } //end for out




    if (normr <= tolb){
        flag = 0;
    }


    double end_time = omp_get_wtime();
    double runtime = (end_time - start_time);

    // writeCSV(matname,n, inner, iterations,b, x, w, resvec, U, R);
    cout<<"GMRES finished, with residual norm: " << normr<< endl << "flag: "<< flag<< "
        iterations: " << iterations << endl;
    // cout << "Runtime: " << runtime << " seconds" << endl;

    free(r);
    free(Ax);
    free(xm);
    free(Axm);

    for (int i = 0; i < 2; ++i) {
        free(J[i]);
    }
    free(J);

    free(resvec);

    for (int i = 0; i < inner+1; ++i) {
        free(U[i]);
    }
    free(U);

    free(u);

    for (int i = 0; i < inner; ++i) {
        free(R[i]);
    }
    free(R);

    free(w);

    free(v);

    free(vtemp);

    free(jtemp);

    free(y);
    free(additive);

}//this is end of gmres
```

## A.3 functions.h

```
#ifndef FUNCTIONS_H
#define FUNCTIONS_H

#include <iostream>
#include <fstream>
#include <string>
#include <cstring>
#include <omp.h>

#include "constants.h"
#include "sparse.h"
#include "read_mat.h"

#include <fstream>
#include <string>

#include <cmath>


#include <random>


using namespace std;


double scalar_prod(double *v1, double *v2, int nr){
    double sum = 0.0;

    if(nr>par_threshhold){
        #pragma omp parallel for reduction(+:sum)
        for(int i = 0; i < nr; i++){
            sum += v1[i]*v2[i];
        }
        return sum;
    }else{
        for(int i = 0; i < nr; i++){
            sum += v1[i]*v2[i];
        }
        return sum;
    }

}

```

```
void elementwise_prod(double *v1, double *v2, int nr){
    if(nr>par_threshhold){
        #pragma omp parallel for
            for(int i = 0; i < nr; i++){
                v1[i] *=v2[i];
            }
    }else{
        for(int i = 0; i < nr; i++){
                v1[i] *=v2[i];
            }
    }
}

double norm(double *v1, int nr){
    // Euclidean norm
    return sqrt(scalar_prod(v1,v1,nr));
}

void normalize(double* v1, int nr){
    double v_norm = norm(v1,nr);
    if(v_norm<eps){
        cout<<"Warning: Norm very small:  "<<endl;
    }else{
        if(nr>par_threshhold){
            #pragma omp parallel for schedule(dynamic, nr/1000+1)
            for(int i=0; i<nr; i++){
                v1[i] /= v_norm;
            }
        }else{
            for(int i=0; i<nr; i++){
                v1[i] /= v_norm;
            }
        }
    }
}



void vector_update(double *v1, double *p, double alpha, int nr){
    if(nr>par_threshhold){
        #pragma omp parallel for schedule(dynamic, nr/1000+1)
            for(int i = 0; i < nr; ++i){
                v1[i] += alpha * p[i];
            }
        }else{
            for(int i = 0; i < nr; ++i){
                v1[i] += alpha * p[i];
            }
        }
}



void vector_scalarMult(double *v1, double *p, double alpha, int nr){
    if(nr>par_threshhold){
        #pragma omp parallel for schedule(dynamic, nr/1000+1)
        for(int i = 0; i < nr; ++i){
            v1[i] = alpha * p[i];
        }
    }else{
        for(int i = 0; i < nr; ++i){
            v1[i] = alpha * p[i];
        }
    }
}

void vector_add(double alpha,double* v1, double beta, double* v2, double* result, int nr){
    if(nr>par_threshhold){
        #pragma omp parallel for schedule(dynamic, nr/1000+1)
        for(int i = 0; i < nr; ++i){
            result[i] = alpha * v1[i] + beta * v2[i];
        }
    }else{
        for(int i = 0; i < nr; ++i){
            result[i] = alpha * v1[i] + beta * v2[i];
        }
    }
}

void vector_zero_out(double* v, int n){
    if(n>par_threshhold){
        #pragma omp parallel for schedule(dynamic, n/1000+1)
        for(int i = 0; i <n; ++i){
            v[i] = 0;
        }
    }else{
        for(int i = 0; i <n; ++i){
            v[i] = 0;
        }
    }
}

void vector_zero_out(double* v, int from, int to, int n){

if(n>par_threshhold){
    #pragma omp parallel for schedule(dynamic, n/1000+1)
    for(int i = from; i <= to; ++i){
        v[i] = 0;
    }
    }else{
        for(int i = from; i <= to; ++i){
            v[i] = 0;
        }
    }
}


void mat_zero_out(double** A, int nr, int nc){
    if(nc>par_threshhold){
        #pragma omp parallel for schedule(dynamic, nr/1000+1)
        for(int i = 0; i <nc; ++i){
            for(int j = 0; j<nr; j++){
                A[i][j] = 0;
            }
        }
    }else{
        for(int i = 0; i <nc; ++i){
            for(int j = 0; j<nr; j++){
                A[i][j] = 0;
            }
        }
    }
}


void normalize_copy(double *normalized, double *original, int nr){
    double v_norm = norm(original,nr);
    if(v_norm<eps){
        cout<<"Warning: Norm to small "<<v_norm<<endl;
    }else{
        if(nr>par_threshhold){
            #pragma omp parallel for schedule(dynamic, nr/1000+1)
            for(int i=0; i<nr; i++){
                normalized[i] = original[i]/v_norm;
            }
        }else{
            for(int i=0; i<nr; i++){
                normalized[i] = original[i]/v_norm;
            }
        }
    }
}


void copyArray(double* dest, const double* src, int size) {
    if(size>par_threshhold){
            #pragma omp parallel for schedule(dynamic, size/1000+1)
            for(int i=0; i<size; i++){
                dest[i] = src[i];
            }
        }else{
            for(int i=0; i<size; i++){
                dest[i] = src[i];
            }
        }
}

void copyArr2Mat_col(double** matrix, const double* arr, int col, int size){
    // #pragma omp parallel for schedule(dynamic, nr/1000+1)
    for(int i=0; i<size; i++){
        matrix[col][i] = arr[i];
    }
}

void copyArr2Mat_col(double** matrix, const double* arr, int col, int size, int from, int to){
    // #pragma omp parallel for schedule(dynamic, nr/1000+1)
    for(int i=from; i<=to; i++){
        matrix[col][i] = arr[i];
    }
}

void copyArr2Mat_row(double** matrix, const double* arr, int row, int size){
    // #pragma omp parallel for schedule(dynamic, nr/1000+1)
    for(int i=0; i<size; i++){
        matrix[i][row] = arr[i];
    }
}

void copyArr2Mat_row(double** matrix, const double* arr, int row, int size, int from, int to){
    // #pragma omp parallel for schedule(dynamic, nr/1000+1)
    for(int i=from; i<=to; i++){
        matrix[i][row] = arr[i];
    }
}

void copyMat_col2Arr(double** matrix, double* arr, int col, int size){
    // #pragma omp parallel for schedule(dynamic, nr/1000+1)
    for(int i=0; i<size; i++){
        arr[i] = matrix[col][i];
    }
}


double scalarsgn(double x){
    if (x >= 0){
        return 1.0;
    }
    else{
        return -1.0;
    }
}
```

```
void printMat(double** matrix, int cols, int rows) {
    for (int i = 0; i < cols; i++) {
        for (int j = 0; j < rows; j++){
            cout << matrix[i][j] << " ";
        }
        cout << endl;
    }
}


void triangularSolver(double** A, double* b, double* x, int n){
    for(int row = n-1; row>=0; row--){
        double sum = 0.0;
        //could be parralelized here or reduction operator
        for(int col = row+1; col<n; col++){
            sum+=A[row][col]*x[col];
        }
        x[row] = (b[row]-sum)/A[row][row];
    }
}




void writeCSV(char* matname, int n, int inner, int iterations,double* b, double* xstar, double*
    w, double* resvec, double** U, double** R) {

    char filename[100];
    strcpy(filename, "data_output/");
    strcat(filename, matname);
    strcat(filename, "_data.csv");
    ofstream file(filename, ofstream::trunc); // Open the file in write mode with truncation


    // Write b (1D array)
    file << "b";
    for (int i = 0; i < n; i++) {
        file << "," << b[i];
    }
    file << std::endl;

    // Write xstar (1D array)
    file << "xstar";
    for (int i = 0; i < n; i++) {
        file << "," << xstar[i];
    }
    file << std::endl;

    // Write w (1D array)
    file << "w";
    for (int i = 0; i < n + 1; i++) {
        file << "," << w[i];
    }
    file << std::endl;

    // Write resvec (1D array)
    file << "resvec";
    for (int i = 0; i < iterations; i++) {
        file << "," << resvec[i];
    }
    file << std::endl;

    // Write U (2D array)
    for (int i = 0; i < inner; i++) {
        file << "U_" << i;
        for (int j = 0; j < n; j++) {
            file << "," << U[i][j];
        }
        file << std::endl;
    }

    // Write R (2D array)
    for (int i = 0; i < inner; i++) {
        file << "R_" << i;
        for (int j = 0; j < inner; j++) {
            file << "," << R[i][j];
        }
        file << std::endl;
    }

    file.close(); // Close the file
}



void createMat(sparse* mat, char* fname) {
    std::ifstream file(fname);

    if (file.is_open()) {
        int nr, nc, nt;
        file >> nr >> nc >> nt;

        int* iat = (int*)malloc((nr + 1) * sizeof(int));
        int* ja = (int*)malloc(nt * sizeof(int));
        double* elem = (double*)malloc(nt * sizeof(double));

        readCSRmat(fname, &nr, &nc, &nt, &iat, &ja, &elem, false);

        mat->set_nrow(nr);
        mat->set_ncol(nc);
        mat->set_n_term(nt);
        mat->set_iat(iat);
        mat->set_ja(ja);
        mat->set_coef(elem);

        file.close();

        // wo muss ich hier freen? oder nicht weil in dekonstruktor gemacht wird?
    } else {
        cout << "FILE NOT OPENED " <<fname<< endl;
    }
}

bool getRHS(char* fname, double* b){
    std::ifstream file(fname);

    if (file.is_open()) {
        double temp_nr;
        int nr;
        file >> temp_nr;
        nr = static_cast<int>(temp_nr);
        for(int i=0;i<nr;i++){
            file >> b[i];
        }
        cout<<"Vector file "<<fname<<endl;
        cout<<"Vector length " << nr<<endl;
        return true;
    }else{
        cout << "FILE NOT OPENED " << fname<< endl;
        return false;
    }
}

void make_b(char* fname,int nr){
    std::ofstream file(fname,std::ofstream::trunc);

    std::random_device rd;
    std::mt19937 generator(rd());
    std::uniform_real_distribution<double> distribution(0.0, 1.0);
    file<<nr<<endl;
    #pragma omp parallel for schedule(dynamic, nr/1000+1)
    for(int i=0;i<nr;i++){
         file<<distribution(generator)<<endl;
    }

    cout<<"random Vector of length created " << nr<<endl;
    file.close();
}


void save_solution(char* fname, double* xstar, int n){
    std::ofstream file(fname,std::ofstream::trunc);
    file<<n<<endl;
    for(int i=0;i<n;i++){
         file<<xstar[i]<<endl;
    }
    cout<<"Vector saved in: " << fname << " length: "<< n<<endl;
    file.close();
}

double* getFEM_sparse(sparse* mat){
    std::ifstream f_coef("FEM_output/coef.txt");
    std::ifstream f_iat("FEM_output/iat.txt");
    std::ifstream f_ja("FEM_output/Ja.txt");
    std::ifstream f_b("FEM_output/b.txt");

    int n_term, n_row_plus1, n_row;
    f_coef >> n_term;
    f_ja >> n_term;
    f_iat >> n_row_plus1;
    f_b >> n_row;

    double* elem = (double*) malloc(n_term*sizeof(double));
    int* ja = (int*) malloc(n_term*sizeof(int));
    int* iat = (int*) malloc(n_row_plus1*sizeof(int));


    double* b = (double*) malloc(n_row*sizeof(double));

    for(int i = 0; i < n_term; i++){
        f_coef >> elem[i];
        f_ja >> ja[i];
    }

    for (int i = 0; i < n_row_plus1; i++){
        f_iat >> iat[i];
    }
    for (int i = 0; i < n_row; i++){
        f_b >> b[i];
    }

    mat->set_nrow(n_row);
    mat->set_ncol(n_row);
    mat->set_n_term(n_term);
    mat->set_iat(iat);
    mat->set_ja(ja);
    mat->set_coef(elem);

    printf("Matrix rows %d columns %d nterm %d\n",n_row,n_row,n_term);
    f_coef.close();
    f_iat.close();
    f_ja.close();
    f_b.close();
    return b;
}

double** mat_allocation(int n_row , int n_col){
```

```c
    double **A = (double**) malloc(n_row*sizeof(double*));
    for (int i = 0; i < n_row; ++i) {
        A[i] = (double*)malloc(n_col * sizeof(double));
    }
    return A;
}

void mat_free(double** A, int n_col, int n_row){
    for (int i = 0; i < n_row; ++i) {
        free(A[i]);
    }
    free(A);
}


int getNZ(double** A, int n){
    // number of nonzeros in mtx
    // A is square
    int nnz = 0;
    #pragma omp parallel for schedule(dynamic,n/1000+1)
    for(int i = 0; i < n; i++){
        for(int j = 0; j < n; j++){
            if(abs(A[i][j])>eps){
                #pragma omp atomic
                nnz++;
            }
        }
    }
    return nnz;
}

int getNZ_idx(double** A, double** idx, int n){
    // number of nonzeros in mtx
    // A is square
    int nnz = 0;
    for(int i = 0; i < n; i++){
        for(int j = 0; j < n; j++){
            if(abs(A[i][j])>eps){
                nnz++;
                idx[nnz][0] = i;
                idx[nnz][1] = j;
            }
        }
    }
    return nnz;
}

double max(double* v, int n){
    double maxVal = v[0];
    #pragma omp parallel for schedule(dynamic, n/1000+1)
    for (int i = 1; i < n; i++) {
        if (v[i] > maxVal) {
            #pragma omp critical
            {
                if (v[i] > maxVal) {
                    maxVal = v[i];
                }
            }
        }
    }
    return maxVal;
}

#endif
```

## A.4 fem_functions.h

```cpp
#include <iostream>
#include <fstream>
#include <string>
#include <omp.h>

#include "constants.h"
#include "sparse.h"
#include "functions.h"

#include <fstream>
#include <string>
#include <cmath>

#include <algorithm>
#include <vector>

using namespace std;

void stiffness_struct(char* fname, int ne, int nn, double** topol) {
    double** Adj = mat_allocation(ne, nn);
    double* i_idx = (double*)malloc(static_cast<int>(nn*nn*0.2));
    double* j_idx = (double*)malloc(static_cast<int>(nn*nn*0.2));

    int idx;
    #pragma omp parallel for schedule(dynamic, nn/1000+1)
    for (int i = 0; i < ne; i++) {
        for (int k = 0; k < 3; k++) {
            idx = static_cast<int>(floor(topol[i][k]) - 1);
            Adj[i][idx] = 1;
        }
    }

    double sum;
    int nnz = 0;
    std::string tempFile = "temp_file.txt";
    std::ofstream file(tempFile, std::ofstream::trunc); // Open the temporary file in write
        mode with truncation
    for (int i = 0; i < nn; i++) {
        for (int j = 0; j < nn; j++) {
            sum = 0;
            for (int k = 0; k < ne; k++) {
                sum += Adj[k][i] * Adj[k][j];
            }
            if (sum > 0) {
                nnz++;
                file << j + 1 << " " << i + 1 << " " << 0 << std::endl;
            }
        }
    }
    file.close();

    // Reopen the temporary file in read mode
    std::ifstream inFile(tempFile);
    if (!inFile) {
        std::cerr << "Error opening the temporary file for reading." << std::endl;
        return;
    }

    // Open the original file in write mode with truncation
    std::ofstream outFile(fname, std::ofstream::trunc);
    if (!outFile) {
        std::cerr << "Error opening the file for writing." << std::endl;
        inFile.close();
        return;
    }

    // Write the new data at the beginning of the original file
    outFile << nn << " " << nn << " " << nnz << std::endl;

    // Copy the rest of the content from the temporary file to the original file
    std::string line;
    while (std::getline(inFile, line)) {
        if (!line.empty()) {
            outFile << line << std::endl;
        }
    }

    // Close both files
    inFile.close();
    outFile.close();

    // Delete the temporary file
    std::remove(tempFile.c_str());
}


void stiffness_struct_para(char* fname, int ne, int nn, double** topol) {
    double** Adj = mat_allocation(ne, nn);
    double* i_idx = (double*)malloc(static_cast<int>(nn*nn*0.2));
    double* j_idx = (double*)malloc(static_cast<int>(nn*nn*0.2));

    int idx;
    #pragma omp parallel for schedule(dynamic, nn/1000+1)
    for (int i = 0; i < ne; i++) {
        for (int k = 0; k < 3; k++) {
            idx = static_cast<int>(floor(topol[i][k]) - 1);
            Adj[i][idx] = 1;
        }
    }

    double sum;
    int nnz = 0;
    std::string tempFile = "temp_file.txt";
    std::ofstream file(tempFile, std::ofstream::trunc); // Open the temporary file in write
        mode with truncation

    #pragma omp for schedule(dynamic,nn/1000+1)
    for (int i = 0; i < nn; i++) {
        for (int j = 0; j < nn; j++) {
            sum = 0;
            for (int k = 0; k < ne; k++) {
                if (Adj[k][i] * Adj[k][j] > 0) {
                nnz++;
                file << j + 1 << " " << i + 1 << " " << 0 << std::endl;
                break;
            }
            }

        }
    }
    file.close();

    // Reopen the temporary file in read mode
    std::ifstream inFile(tempFile);
    if (!inFile) {
        std::cerr << "Error opening the temporary file for reading." << std::endl;
        return;
    }

    // Open the original file in write mode with truncation
    std::ofstream outFile(fname, std::ofstream::trunc);
    if (!outFile) {
        std::cerr << "Error opening the file for writing." << std::endl;
        inFile.close();
        return;
    }

    // Write the new data at the beginning of the original file
    outFile << nn << " " << nn << " " << nnz << std::endl;

    // Copy the rest of the content from the temporary file to the original file
    std::string line;
    while (std::getline(inFile, line)) {
        if (!line.empty()) {
            outFile << line << std::endl;
        }
    }

    // Close both files
    inFile.close();
    outFile.close();

    // Delete the temporary file
    std::remove(tempFile.c_str());
}


void stiffness_struct_perf(sparse* H, int ne, int nn, double** topol){
    double** Adj = mat_allocation(ne+1, nn);

    // Create vectors for i_idx and j_idx
    std::vector<int> i_idx;
    std::vector<int> j_idx;
    int idx;

    // Reserve an initial capacity for the vectors
    double n_temp = static_cast<double>(nn);
    int initialCapacity = static_cast<int>(n_temp * n_temp * 0.05 );
    i_idx.reserve(initialCapacity);
    j_idx.reserve(initialCapacity);

    #pragma omp parallel for schedule(dynamic, nn/1000+1)
    for (int i = 0; i < ne; i++) {
        for (int k = 0; k < 3; k++) {
            idx = static_cast<int>(floor(topol[i][k]) - 1);
            Adj[i][idx] = 1;
        }
    }

    double sum;
    int nnz = 0;
    #pragma omp parallel for schedule(dynamic,nn/1000+1)
    for (int i = 0; i < nn; i++) {
        for (int j = 0; j < nn; j++) {
            sum = 0;
            for (int k = 0; k < ne; k++) {
                if (Adj[k][i] * Adj[k][j] > 0) {
                    j_idx[nnz] = (i +1);
                    i_idx[nnz] = (j);
                    #pragma omp atomic
                    nnz++;
                    // Optionally, check the capacity and reserve more space if needed (e.g.,
                       double the current capacity)
                    if (nnz >= i_idx.capacity()) {
                        std::cout<<"capacity to small"<<nnz<<endl;
                        int newCapacity = i_idx.capacity() * 1.5;
                        i_idx.reserve(newCapacity);
                        j_idx.reserve(newCapacity);
                    }

                    break;
                }
            }
        }
    }
    int* ja_loc = &i_idx[0];
    int* irow = &j_idx[0];

    int* iat_loc = (int*)malloc(nn*sizeof(int));
    irow2iat(nn,nnz,irow,iat_loc);
    H->set_n_term(nnz);
    H->set_ncol(nn);
    H->copy_ja(ja_loc);
    H->copy_iat(iat_loc);

    free(iat_loc);
}

void stiffness_struct_small_loops(sparse* H, int ne, int nn, double** topol) {
        // Create vectors to store the i_idx and j_idx pairs
    std::vector<int> i_idx;
    std::vector<int> j_idx;

    // Loop through each element
    for (int i = 0; i < ne; i++) {
        for (int k = 0; k < 3; k++) {
            int idx = static_cast<int>(floor(topol[i][k]) - 1);
            for (int l = 0; l < 3; l++) {
                int jdx = static_cast<int>(floor(topol[i][l]) - 1);
                i_idx.push_back(idx);
                j_idx.push_back(jdx);
            }
        }
    }

    // Combine i_idx and j_idx into pairs for sorting
    std::vector<std::pair<int, int>> pairs;
    for (int i = 0; i < i_idx.size(); i++) {
        pairs.push_back(std::make_pair(i_idx[i], j_idx[i]));
    }

    // Sort the pairs to bring duplicates together
    std::sort(pairs.begin(), pairs.end());

    // Remove duplicates and copy the unique pairs back to i_idx and j_idx
    int nnz = 0;
    for (int i = 0; i < pairs.size(); i++) {
        if (i == 0 || pairs[i] != pairs[i - 1]) {
            i_idx[nnz] = pairs[i].first + 1;
            j_idx[nnz] = pairs[i].second;
            nnz++;
        }
    }

    // Allocate memory for the CSR arrays
    int* ja_loc = new int[nnz];
    int* iat_loc = new int[nn + 1];

    // Copy the data from i_idx and j_idx to the CSR arrays
    for (int i = 0; i < nnz; i++) {
        ja_loc[i] = j_idx[i];
    }

    // Compute the CSR arrays from i_idx and ja_loc
    irow2iat(nn, nnz, &i_idx[0], iat_loc);

    H->set_n_term(nnz);
    H->set_ncol(nn);
    H->copy_ja(ja_loc);
    H->copy_iat(iat_loc);
    // H->printComponents();
    delete[] ja_loc;
    delete[] iat_loc;
}

double get_loc(double** coord, double** topol, double *D, double*V, int el, double*** Loc){
    double *t = topol[el];
    double x[3];  //i,j,m
    double y[3];  //i,j,m
    double a[3],b[3],c[3], delta, temp, mod_v;

    for(int i = 0; i<3; i++){
        x[i] = coord[static_cast<int>(t[i]-1)][0];
        y[i] = coord[static_cast<int>(t[i]-1)][1];
    }

    a[0] = x[1]*y[2] - x[2]*y[1];
    a[1] = x[2]*y[0] - x[0]*y[2];
    a[2] = x[0]*y[1] - x[1]*y[0];

    int idx1, idx2;
    for(int i = 0; i<3;i++){
        idx1 = (i+1)%3;
        idx2 = (i+2)%3;
        b[i] = y[idx1] - y[idx2];
        c[i] = x[idx2] - x[idx1];
    }

    delta = (a[0]+a[1]+a[2])/2;
    mod_v = norm(V,2);

    if(mod_v>eps){// prevent zero division
        double temp;
        for(int i = 0; i<3; i++){
            for(int j = 0; j<3; j++){
                temp = (V[0]*b[j] + V[1]*c[j]);
                Loc[0][i][j] = (D[0]*b[i]*b[j]+ D[1]*c[i]*c[j])/(4*delta);
                Loc[1][i][j] = temp/6;
                Loc[2][i][j] = (V[0]*b[i] + V[1]*c[i])*temp/(8*delta*mod_v*max(D,2));
            }
        }
    }else{
        std::cout<<" Norm v zero, just compute stiffness Matrix H \n";
        for(int i = 0; i<3; i++){
            for(int j = 0; j<3; j++){
                Loc[0][i][j] = (D[0]*b[i]*b[j]+ D[1]*c[i]*c[j])/(4*delta);
            }
        }
    }


    return delta;
}

void loc2glob(sparse* A_ptr, double** Loc, double** topol, int e){
    double* coef = A_ptr->get_coef();
    int* ja = A_ptr->get_ja();
    int* iat = A_ptr->get_iat();
    int row, col;
    double* top = topol[e];
    for(int i = 0; i<3; i++){   // choose row
        row = static_cast<int>(top[i]) - 1;
        for(int j = 0; j<3; j++){// chose col
            col = static_cast<int>(top[j]) - 1;
            int k;
            for(k = iat[row]; k<iat[row+1];k++){ //go over column indices in ja
                if(ja[k]==col){break;}
            }
            #pragma omp atomic
            coef[k] += Loc[i][j];
        }
    }
}

sparse* create_FEM_mat(double** coord, double** topol, int ne, int nn, double* D,\
                          double h, double tau, double* V, double* delta){
    sparse* mat_list;
    mat_list = new sparse[3]; // H,B,S
    double*** Loc = (double***) malloc(3*sizeof(double**));
    // double ***Loc;
    // Loc = new double**[3];
    int buffer_length = static_cast<int>(nn * nn * 0.1);

    int nterm, nr;

    int* ja = (int*) calloc(buffer_length,sizeof(int));
    int* iat = (int*) calloc(nn+1,sizeof(int));

    //allocate space for the local H, B, S
    for(int i = 0; i<3;i++){Loc[i] = mat_allocation(3,3);}

    char fname[] = "fullmat.txt";
    // stiffness_struct(fname, ne,  nn,  topol);
    // createMat(&mat_list[0], fname);
    stiffness_struct_small_loops(&mat_list[0], ne,  nn,  topol);

    mat_list[2] = mat_list[1] = mat_list[0];

    for(int i = 0; i<ne; i++){
        delta[i] = get_loc(coord,topol,D,V,i,Loc);
        for(int k = 0; k<3;k++){
            loc2glob(&mat_list[k],Loc[k],topol,i);
        }
    }
    // printVec(delta,ne);
    mat_list[2].scalarMult(tau*h);

    return mat_list;
}


double force(double x, double y){
    double f = 0;
    // do stuff
    return f;
}


void imposeBC(sparse* H, double** bound, double* q, int nb){
    double R = 1e15;
    int j, bound_idx, start, end;
    int* iat = H->get_iat();
    int* ja = H->get_ja();
    double* coef = H->get_coef();

    for(int i = 0; i<nb; ++i){
        bound_idx = static_cast<int>(bound[i][0])-1;
        start = iat[bound_idx];
        end = iat[bound_idx+1];

        for(j = start; j<end; j++){
            if (ja[j] == bound_idx){break;}
        }

        coef[j] = R;
        q[bound_idx] = R*bound[i][1];
    }
}



double* getAssembledSystem(sparse* A){
    // FEM assembly
    char mesh_path[] = "mesh/mesh.dat";
    char node_path[] = "mesh/dirnod.dat";
    char coord_path[] = "mesh/xy.dat";

    double D[2] = {1,1};
    double V[2] = {1,3};
    int ne, nn, nb; // number of mesh (800), number of nodes(441), number of ? (80)

    double **coord, **bound, *delta, *delta_node, *F;
    double **topol, *temp, *q;

    sparse* FEM;


    ifstream mesh(mesh_path);
    ifstream nodes(node_path);
    ifstream xy(coord_path);

    //read data
    int not_used;

    xy >> nn >> not_used;
    nodes >> nb >> not_used;
    mesh >> ne >> not_used >> not_used >> not_used;

    bound = mat_allocation(nb,2);
    coord = mat_allocation(nn,2);
    topol = mat_allocation(ne,3);



    for (int i = 0; i < nn; ++i){xy >> coord[i][0] >> coord[i][1];}
    for (int i = 0; i < nb; ++i){nodes >> bound[i][0] >> bound[i][1];}
    for (int i = 0; i < ne; ++i){mesh >> topol[i][0] >> topol[i][1] >> topol[i][2]>>not_used;}


    // printMat(coord,nb,2);

    xy.close();
    nodes.close();
    mesh.close();

    double h = sqrt(2)*(coord[0][1]-coord[1][1]);// why sqrt2
    double tau = 0.1; // whats tau?

    delta = (double*)malloc(ne*sizeof(double));

    delta_node = (double*)calloc(nn,sizeof(double));

    F = (double*)malloc(nn*sizeof(double));
    q = (double*)calloc(nn,sizeof(double));



    FEM = create_FEM_mat(coord, topol, ne, nn, D, h, tau, V, delta);

    int idx;
    #pragma omp parallel for schedule(dynamic,nn/1000+1)
    for(int i = 0; i<ne; i++){
        for(int k=0; k<3; k++){
            idx = static_cast<int>(floor(topol[i][k])-1);
            delta_node[idx] += delta[k]/3;
        }
    }

    // RHS
    for(int i = 0; i<nn; i++){
        F[i] = force(coord[i][0],coord[i][1]*delta_node[i]);
    }

    // BCs
    imposeBC(&FEM[0], bound, q, nb);
    FEM[1].addition_update(&FEM[2]);
    FEM[0].addition_update(&FEM[1]);

    *A = FEM[0];
    vector_update(q,F,1,nn);

    mat_free(bound,nb,2);
    mat_free(coord,nn,2);
    mat_free(topol,ne,3);
    free(delta);
    free(delta_node);
    free(F);
    delete[] FEM;


    return q;
}
```

## A.5 sparse.h

```
#ifndef SPARSE_H
#define SPARSE_H

#include <omp.h>
#include <iostream>
#include <cmath>

#include "constants.h"

class sparse{
private:
    int nrow, ncol, n_term;
    double* coef; // non zero matrixelements
    int *iat, *ja; // first non zero column index iat, what is ja?


public:
    // constructors
    sparse();
    sparse(int nrow, int ncol, int n_term);
    ~sparse(); // destructor


    // get
    int get_nrow(){return nrow;}
    int get_ncol(){return ncol;}
    int get_n_term(){return n_term;}

    double* get_coef(){return coef;}
    int* get_iat(){return iat;}
    int* get_ja(){return ja;}


    // set
    void set_nrow(int n){
        nrow = n;
        ncol = n;
        free(iat);
        iat = (int*) malloc(ncol*sizeof(int));
    }
    void set_ncol(int n){
        nrow = n;
        ncol = n;
        free(iat);
        iat = (int*) malloc(ncol*sizeof(int));
    }
    void set_n_term(int n){
        n_term = n;
        free(coef);
        free(ja);
        ja = (int*) malloc(n_term*sizeof(int));
        coef = (double*) malloc(n_term*sizeof(double));
    }

    // void alloc_coef(int n){coef = (double*)malloc((n)*sizeof(double));}
    // void alloc_iat(int n){iat = (int*)malloc((n+1)*sizeof(int));}
    // void alloc_ja(int n){ja = (int*)malloc((n)*sizeof(int));}

    void set_coef(double* v){coef = v;}
    void set_iat(int* v){iat = v;}
    void set_ja(int* v){ja = v;}

    void copy_coef(double* v){
        #pragma omp parallel for
        for(int i = 0; i<n_term;i++){
            coef[i]=v[i];
        }
    }

    void copy_iat(int* v){
        #pragma omp parallel for
        for(int i = 0; i<nrow+1;i++){
            iat[i]=v[i];
        }
    }

    void copy_ja(int* v){
        #pragma omp parallel for
        for(int i = 0; i<n_term;i++){
            ja[i]=v[i];
        }
    }


    //operators
    sparse& operator=(sparse&);
    friend bool operator==(const sparse& lhs, const sparse& rhs);

    //Data Operations
    void full2sparse(double** A, int n_row);


    // matrix funcitons
    void addition_update(sparse* B_ptr);
    void scalarMult(double alpha);
    void post_MV(double* v, double* y); // A*b
    void pre_MV(double* v, double* y); // b'*A
    void matrixProduct(sparse* A_ptr, sparse* result);
    void diag(double* v);
    void getJacobi(double* v);
    void diag_x_sparse(double* diag);
    double* left_Jacobi();


    // display mtx
    void printMat();
    void printComponents();
};


using namespace std;

sparse::sparse(){
    // cout<<"empty constructor\n";
    nrow = 0;
    ncol = 0;
    n_term = 0;

    coef = nullptr;
    ja = nullptr;
    iat = nullptr;
};

sparse :: sparse(int nr, int nc, int nt){
    nrow = nr;
    ncol = nc;
    n_term = nt;

    coef = (double*)malloc(nt*sizeof(double));
    ja = (int*)malloc(nt*sizeof(int));
    iat = (int*)malloc((nr+1)*sizeof(int)); //+1
};

sparse :: ~sparse(){
    free(coef);
    free(ja);
    free(iat);
}


void sparse::post_MV(double* v, double* y ){
    // #pragma omp parallel for schedule(dynamic,nrow/1000+1)
    for(int i=0; i<nrow; i++){
        double sum = 0.0;
        #pragma omp simd reduction(+:sum)
        for(int j=iat[i]; j<iat[i+1]; j++){
            sum += coef[j]*v[ja[j]];
        }
        y[i] = sum;
    }
}

void sparse::pre_MV(double* v, double* y ){
    #pragma omp parallel for schedule(dynamic,nrow/1000+1)
    for(int j=0; j<ncol; j++){
        double sum = 0.0;
        // #pragma omp simd reduction(+:sum)
        for(int i=0; i<nrow; i++){
            for(int k=iat[i]; k<iat[i+1]; k++){
                if (ja[k] == j){
                    sum += v[i]*coef[k];
                }
            }
        }
        y[j] = sum;
    }
}

sparse& sparse::operator=(sparse& A){
    // std::cout<<"cpy"<<endl;
    int n_row = A.get_nrow();
    int n_term = A.get_n_term();

    this->set_n_term(n_term);
    this->set_ncol(n_row);
    this->copy_coef(A.get_coef());
    this->copy_iat(A.get_iat());
    this->copy_ja(A.get_ja());

    return *this;
}

bool operator==(const sparse& lhs, const sparse& rhs) {
    if (lhs.nrow != rhs.nrow || lhs.ncol != rhs.ncol || lhs.n_term != rhs.n_term) {
        cout<<" Shapes are not equal\n";
        return false;

    }

    // Compare ja arrays
    for (int i = 0; i < lhs.n_term; i++) {
        if (lhs.ja[i] != rhs.ja[i]) {
            cout<<" ja arrays are not equal\n";
            return false; //

        }
    }

    // Compare iat arrays
    for (int i = 0; i <= lhs.nrow; i++) {
        if (lhs.iat[i] != rhs.iat[i]) {
            cout<<" iat arrays are not equal\n";
            return false; //

        }
    }

    return true; // Shapes, ja, and iat arrays are equal
}


void sparse::full2sparse(double** A, int n){
    // A is square
    // A is sparse --> suppose only max 10% of elements are nonzero
    double sparse_ratio = 1;
    int n_term_max = static_cast<int>(n * n * sparse_ratio);

    double* coef_temp = (double*)malloc(n_term_max*sizeof(double));
    int* Ja_temp = (int*)malloc(n_term_max*sizeof(int));
    int* iat_temp = (int*)malloc((n+1)*sizeof(int));

    this->set_ncol(n);
    this->set_nrow(n);

    int idx_nt = 0;
    int idx_nr = 0;
    bool new_row = true;

    #pragma omp parallel for schedule(dynamic,n/1000+1)
    for(int i = 0; i < n; i++){
        for(int j = 0; j < n; j++){
            if(abs(A[i][j])>eps){
                coef_temp[idx_nt] = A[i][j];
                Ja_temp[idx_nt] = j;
                if(new_row){
                    iat_temp[idx_nr] = idx_nt;
                    idx_nr++;
                    new_row = false;
                }
                idx_nt++;
            }
        }
        new_row = true;
    }
    iat_temp[idx_nr] = idx_nt;

    this->set_n_term(idx_nt);
    this->copy_coef(coef_temp);
    this->copy_ja(Ja_temp);
    this->copy_iat(iat_temp);

    free(iat_temp);
    free(coef_temp);
    free(Ja_temp);
}

void sparse::addition_update(sparse* B_ptr){
    if(*this==*B_ptr){
        double* add = B_ptr->get_coef();
        #pragma omp parallel for schedule(dynamic,n_term/1000 +1)
        for(int i = 0; i<n_term; i++){
            coef[i] += add[i];
        }
    }else{
        cout<<"Sparsity structure is not the same \n";
    }

}


void sparse::scalarMult(double alpha){
    #pragma omp parallel for schedule(dynamic,n_term/1000 +1)
    for(int i = 0; i<n_term; i++){
        coef[i] *= alpha;
    }
}
void sparse::matrixProduct(sparse* B_ptr, sparse* result){
    // (nxm)*(mxk) = (nxk) but usually square anyways


    sparse& B = *B_ptr;

    if (nrow != B.ncol){
        std::cout<<"Incompatible Size"<<endl;
    }


    for(int i = 0; i < nrow; i++){
        double sum = 0.0;
        for(int j = 0; j < B.ncol; j++){

            sum += i+j;
        }


    }


}


void sparse::diag(double* v){
    int start, end;
    #pragma omp parallel for schedule(dynamic, ncol/1000+1)
    for(int i = 0; i<ncol; i++){
        start = iat[i];
        end = iat[i+1];
        for(int j = start; j<end; j++){
            if(ja[j] == i){
                v[i] = coef[j];
                break;
            }
        }
    }
}

void sparse::getJacobi(double* v){
    this->diag(v);
    int zeros = 0;
    #pragma omp parallel for schedule(dynamic, ncol/1000 +1)
    for(int i = 0; i<ncol; i++){
```

```cpp
        if (v[i] == 0){
            v[i]=1;
            #pragma omp atomic
            zeros++;
            }
        v[i] = 1/v[i];
    }
    if(zeros>0){
        cout<< "WARNING: Number of zeros in diagonal: "<<zeros<<endl;
    }
}

void sparse::diag_x_sparse(double* diag){
    int start, end;
    #pragma omp parallel for schedule(dynamic, ncol/1000+1)
    for(int i = 0; i<ncol; i++){
        start = iat[i];
        end = iat[i+1];
        for(int j = start; j<end; j++){
            coef[j] *= diag[i];

        }
    }
}


double* sparse::left_Jacobi(){
    double* v = (double*) calloc(ncol,sizeof(double));
    this->getJacobi(v);
    this->diag_x_sparse(v);
    return v;
}

#include <iomanip>

void sparse::printMat() {
    // Create a 2D matrix to represent the sparse matrix
    double** matrix = new double*[nrow];
    for (int i = 0; i < nrow; i++) {
        matrix[i] = new double[ncol];
        // Initialize all elements to 0
        for (int j = 0; j < ncol; j++) {
            matrix[i][j] = 0.0;
        }
    }

    // Fill the matrix with the non-zero elements from the sparse matrix
    for (int row = 0; row < nrow; row++) {
        int start = iat[row];                    // Starting index for the current row
        int end = iat[row + 1];                  // Ending index for the current row

        // Iterate over the non-zero elements in the current row
        for (int index = start; index < end; index++) {
            int column = ja[index];              // Column index of the non-zero element
            double value = coef[index];          // Value of the non-zero element
            matrix[row][column] = value;         // Store the value in the matrix
        }
    }

    // Print the matrix
    for (int i = 0; i < nrow; i++) {
        for (int j = 0; j < ncol; j++) {
            std::cout << setw(8) << matrix[i][j] << " ";
        }
        std::cout << endl;
    }

    // Free the memory allocated for the matrix
    for (int i = 0; i < nrow; i++) {
        delete[] matrix[i];
    }
    delete[] matrix;
}

void sparse::printComponents(){
    std::cout<<"Coef: ";
    for (int i=0; i<n_term; i++){std::cout<<coef[i]<<" ";}
    std::cout<<std::endl<<"ja: ";
    for (int i=0; i<n_term; i++){std::cout<<ja[i]<<" ";}
    std::cout<<std::endl<<"iat: ";
    for (int i=0; i<nrow+1; i++){std::cout<<iat[i]<<" ";}
    std::cout<<std::endl;
}

#endif
```