High Performance Computing Project Report

Simon Wenchel

August 7, 2023

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:

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

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:

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

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

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_{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 `n_term`$(2 \cdot$ `sizeof`$(\text{int}) +$ `sizeof`$(\text{double}))$

As shown in Table 1 the CSR format also has three lists but needs `n_term` $-$ `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

```
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

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

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 $v[i] > \text{maxVal}$ 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 Ω with triangular finite elements. The input data is shown in Table 3.

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

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));
```

```cpp
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();
}
```

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.

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

4   **for** $i = 1, \ldots, inner$ **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** $k = i - 1 \ldots 0$ **do**

8          $v = v - 2u^{(k)^T} u^{(k)} \cdot v$; // `scalar product and vector update is parallelized`

9   **end**

10       $v = \frac{v}{||v||}$; // `normalization is parallel`

      // `do:` $v = P_i \cdots P_0 A v$

11       $v = Av$; // `sparse Matrix-vector product is parallel`

12   **for** $k = 0 \ldots i$ **do**

13         $v = v - 2u^{(k)^T} u^{(k)} \cdot v$;

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, ..., outer do
    for i = 1, ..., inner do
        // ...
        // Previous code...
        // Continue with Givens rotation...
1       j_0 = v_i; // Store v_i in j_0
2       j_1 = v_{i+1}; // Store v_{i+1} in j_1
3       ρ = √(j_0² + j_1²); // Compute the norm
4       j_0 = j_0/ρ; // Normalize j_0
5       j_1 = j_1/ρ; // Normalize j_1
6       J_{i,0} = j_0; // Store the Givens rotation values
7       J_{i,1} = j_1; // Store the Givens rotation values
8       w_{i+1} = -J_{i,1} · w_i; // Update w_{i+1}
9       w_i = J_{i,0} · w_i; // Update w_i
10      v_i = ρ; // Update v_i
11      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
12      R_{:,i} = v;
13      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 oscilation 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.
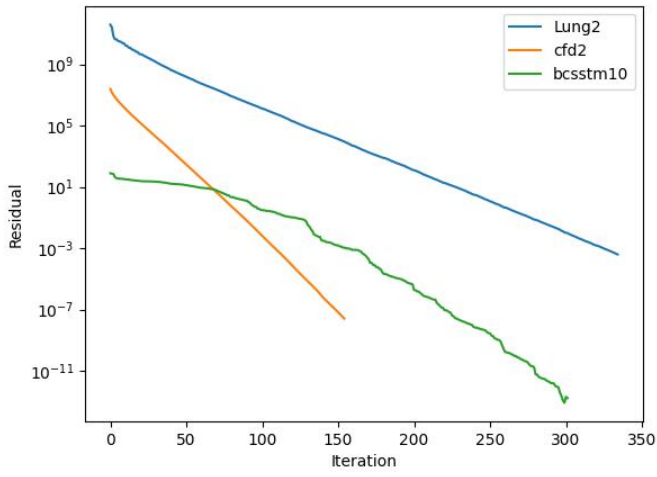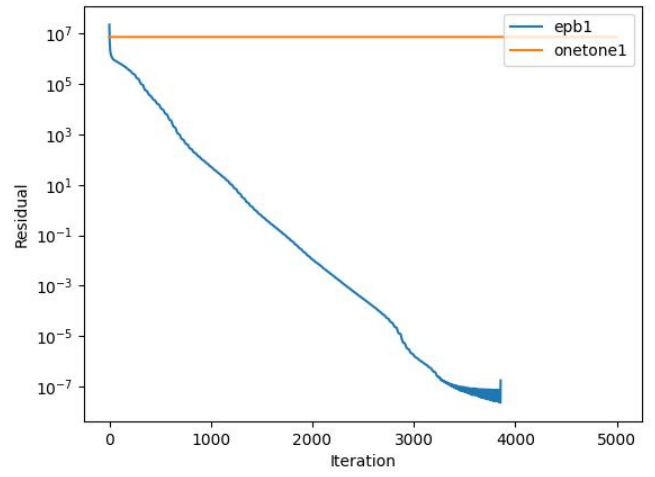
Figure 1: Convergence plots of GMRES for the test matrices. (a) shows fast convergences and (b) slow or no convergence of GMRES
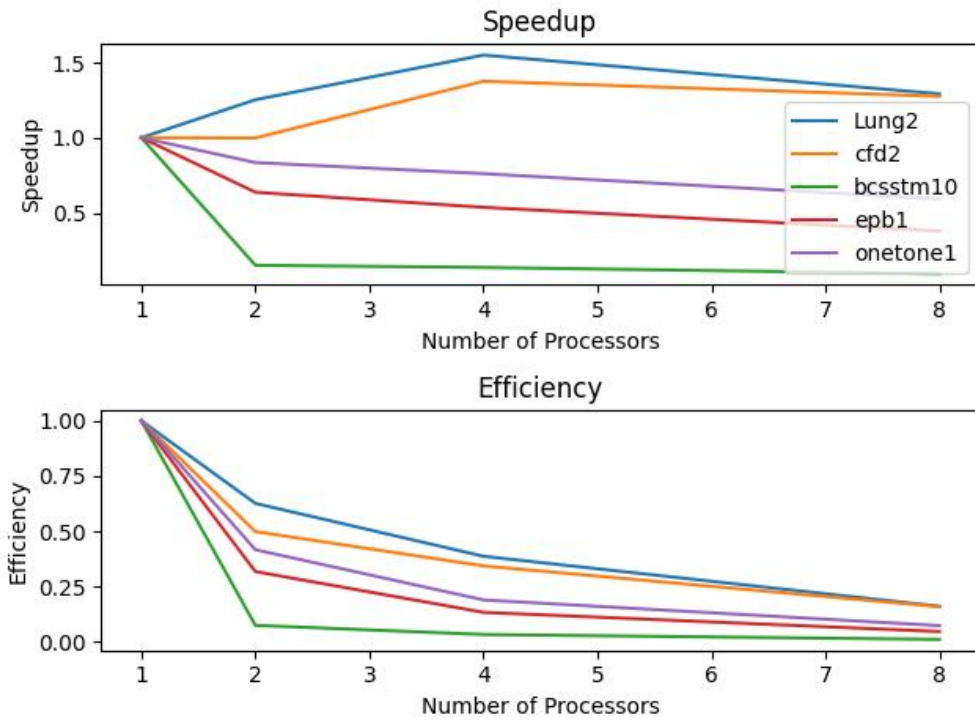

Figure 2: Speedup and Efficiency Metric for Test Matrices

Algorithm 5: Evaluating x_m

---

    for $j = 1, \ldots, outer$ **do**
        for $i = 1, \ldots, inner$ **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** $k = i\text{-}1 \ldots 0$ **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	1.258792×10^5
epb1	14734	95053	5.940657×10^3
onetone1	36057	335552	9.388846×10^6

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

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

where the boundaries Ω_1 and Ω_2 are given by:

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

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.
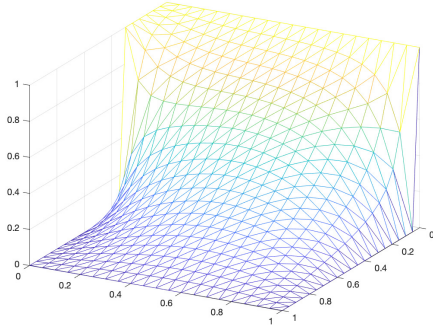
(a) Solution for the small mesh
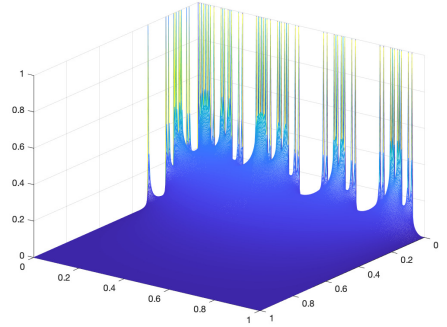
(b) Solution for the big mesh

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;


```

```cpp
    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();
```

```cpp
133        char path[] = "mesh/u.dat";
134        strcpy(sol_path, path);
135    }
136
137
138    //read data
139
140
141
142    // printMat(coord,nb,2);
143
144
145
146    double h = sqrt(2)*(coord[0][1]-coord[1][1]);// why sqrt2
147    double tau = 0.1; // whats tau?
148
149    delta = (double*)malloc(ne*sizeof(double));
150
151    delta_node = (double*)calloc(nn,sizeof(double));
152
153    F = (double*)malloc(nn*sizeof(double));
154    q = (double*)calloc(nn,sizeof(double));
155    b = (double*)malloc(nn*sizeof(double));
156
157
158    FEM = create_FEM_mat(coord, topol, ne, nn, D, h, tau, V, delta);
159
160    int idx;
161    #pragma omp for schedule(dynamic,nn/1000+1)
162    for(int i = 0; i<ne; i++){
163        for(int k=0; k<3; k++){
164            idx = static_cast<int>(floor(topol[i][k])-1);
165            delta_node[idx] += delta[k]/3;
166        }
167    }
168
169    // RHS
170    for(int i = 0; i<nn; i++){
171        F[i] = force(coord[i][0],coord[i][1]*delta_node[i]);
172    }
173
174    // BCs
175    imposeBC(&FEM[0], bound, q, nb);
176    FEM[1].addition_update(&FEM[2]);
177    FEM[0].addition_update(&FEM[1]);
178    // printVec(FEM[0].get_coef(),FEM[0].get_n_term());
179    double* x = (double*)calloc(FEM[0].get_ncol(),sizeof(double));
180
181
182    vector_update(q,F,1,nn);
183    // printVec(q,nn);
184
185    // FEM Solver
186    cout<< norm(q,nn)<<endl;
187    sparse* A = new sparse;
188    // b = getAssembledSystem(A);
189
190    // double* x = (double*)calloc(A->get_ncol(),sizeof(double));
191    // FEM[0].printComponents();
192
193
194    gmres(sol_path,&FEM[0],q, x, true, true, 1000);
195    double end_time = omp_get_wtime();
196    double runtime = (end_time - start_time);
197
198    save_solution(sol_path,x,FEM[0].get_ncol());
199
200
201    free(b);
```

```
202     free(x);
203     delete A;
204
205     return 0;
206
207     }
```

A.2 GMRES.h

```
1   void gmres(char* matname, sparse* A_ptr, double* b, double* x, bool preconditioning, bool
        restarting, int maxit){
2       /*
3       GMRES Algorithm with
4           - Householder projections
5           - Givens rotation
6
7       Input:
8           - A_ptr: pointer to sparse Matrix of the linear system to be solved
9           - b: ptr to rhs
10          - x: ptr to solution vector (and start vector?)
11
12
13
14
15
16      Flag is set to see the status of the Algorithm
17      flag = 0: succesful convergence
18          - inner loop: rormr_act < tolb
19          - outer loop: rormr_act < tolb after the last evaluation
20      flag = 1: maximum iterations reached without convergence
21          - initialization if it stays unchanged: max iterations reached
22      flag = 2: stagnation
23          - no significant change in norm reduction
24
25      */
26      sparse& A = *A_ptr;
27
28      int nc = A.get_ncol();
29      int nr = A.get_nrow();
30
31      if(nc!=nr){
32          cout<<"ERROR need square Matrix n != n"<<endl;
33          return;
34      }
35
36      const int n = nc;
37
38      double make_relres_smaller=1;
39      if (preconditioning){
40          double* M_inv = A.left_Jacobi();
41          elementwise_prod(b,M_inv,n);
42      }else{
43          make_relres_smaller = 1.0e-15;
44      }
45
46      int restart, outer;
47      if(restarting){
48          restart = 20;
49          outer = maxit/restart;
50          maxit = restart; // 10 is small
51      }
52      int inner = maxit;
53
54
55
56
57      double tol = 1e-15;
```

```
58      double n2b = norm(b,n);

59

60

61      int flag = 1;

62

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

66

67      int stag = 0;
68      int moresteps = 0;
69      int maxmsteps = 3;
70      int maxstagsteps = 3;

71

72

73      double* r = (double*)malloc((n)*sizeof(double));
74      double* Ax = (double*)malloc((n)*sizeof(double));
75      double* xm = (double*)malloc((n)*sizeof(double));
76      double* Axm = (double*)malloc((n)*sizeof(double));

77

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

81

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

85

86      // preallocate

87

88      // J is of size (inner x 2)
89      // Values used in one iteration are in the same array
90      // working direction J[row][i] ist ja doch die falsche working direction oder
91      double** J = (double**)malloc(inner * sizeof(double*)); // Matrix for givens rotation J(2,
            inner)
92      for (int i = 0; i < inner; ++i) {
93          J[i] = (double*)malloc(2 * sizeof(double));
94      }

95

96      // U is of size (inner x n)
97      // Values used in one iteration are in the same array
98      // working direction U[row][i] = i;
99      double** U = (double**)malloc((inner+1) * sizeof(double*)); // Matrix for holding
            Householder reflectors u(w) U(n,inner)
100     for (int i = 0; i < (inner+1); ++i) {
101         U[i] = (double*)malloc(n * sizeof(double));
102     }

103

104

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

106

107     double** R = (double**)malloc(inner * sizeof(double*)); // givens rotated Hessenbergmatrix
            R(inner,inner)
108     for (int i = 0; i < inner; ++i) {
109         R[i] = (double*)malloc(inner * sizeof(double));
110     }

111

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

113

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

116

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

118

119

120     double beta;
121     double alpha;

122

123
```

```c
124    double relres; //relative residual
125
126    int initer; // so that they live outside the loop
127    int outiter;
128    int iterations = 0;
129    int idx;
130
131    double* y = (double*)malloc((inner+1)*sizeof(double));
132    double* additive = (double*)malloc(n*sizeof(double));
133
134
135    double start_time = omp_get_wtime();
136
137    normr = norm(r,n);
138    beta = scalarsgn(normr)*normr;
139    copyArray(u, r, n);
140    u[0] += beta;
141    normalize(u, n);
142
143    copyArr2Mat_col(U,u,0,n);
144    //  Apply Householder projection to r.
145    //   w = r - 2*u*u'*r;
146    w[0] = -beta;
147
148    for (outiter = 0; outiter < outer; outiter++){
149        for(initer=0; initer<inner; initer++){
150                iterations = (outiter)*inner+initer+1;
151                // copyArray(v, u, n);        // wofuer das
152                vector_scalarMult(v, u, -2.0*u[initer], n);
153                v[initer] += 1;
154
155                // v = P1*P2*...Pjm1*(Pj*ej)
156                for(int k = initer - 1; k >= 0; k--){
157                    // copyMat_col2Arr(U, u, k, n);
158                    // vector_update(v, u, -2*scalar_prod(u,v,n), n);
159                    vector_update(v, U[k], -2*scalar_prod(U[k],v,n), n);
160
161                }
162
163                normalize(v, n);
164                copyArray(vtemp, v, n);
165
166                A.post_MV(vtemp, v);
167
168                // Form Pj*Pj-1*...*P1*Av
169                for(int k=0; k<=initer; k++){
170                    // copyMat_col2Arr(U, u, k, n);
171                    // vector_update(v, u, -2*scalar_prod(u,v,n), n);
172                    vector_update(v, U[k], -2*scalar_prod(U[k],v,n), n);
173
174                }
175
176
177                // determine Pj+1
178                if(initer!= n-1){ // if not last iteration // not necessary in restarted
                     version
179                    // construct u for Pj+1
180                    copyArray(u, v, n);
181                    vector_zero_out(u, 0, initer, n);
182
183                    alpha = norm(u,n);
184
185                    if (alpha > eps){
186                        alpha *= scalarsgn(v[initer+1]);
187                        //u = v(initer+1:end) + sign(v(initer+1))*||v(initer+1:end)||*e_{initer
                            +1)
188                        u[initer+1] += alpha;
189
190                        normalize(u, n);
```

22

```
                    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;
                }
```

```cpp
259
260                   // copyArray(xm,x,n);
261                   // vector_update(xm,additive,1,n);   // xm = x + additive
262                   vector_add(1,x,1,additive,xm,n);
263
264
265
266
267                   A.post_MV(xm,Axm);
268                   vector_add(1.,b,-1.,Axm,r,n);
269                   if (norm(r,n)<= tolb){
270                       copyArray(x,xm,n);
271                       flag = 0;
272                       break;
273                   }
274                   normr = norm(r,n);
275
276                   resvec[iterations] = normr;
277
278
279
280
281
282                   if (normr <= tolb){
283                       copyArray(x,xm,n);
284                       flag = 0;
285                       jmin = initer;
286                       break;
287                   }else{
288                       if (stag >= maxstagsteps && moresteps ==0){
289                           stag = 0;
290                       }
291                       moresteps += 1;
292                       if (moresteps >= maxmsteps){
293                           cout<<"stagnation"<<endl;
294                           flag = 2;
295                           jmin = initer;
296                           break;
297                       }
298                   }
299               }//endif (normr <= tolb || stag >= maxstagsteps || moresteps)
300
301
302               if (stag>=maxstagsteps){
303                   flag = 2;
304                   break;
305               }
306       }// end for in
307
308       if (flag != 0){ // computes the x with the lowest norm
309           vector_zero_out(additive,n);
310
311           vector_zero_out(y,initer+1);
312           triangularSolver(R,w,y,initer);
313           // copyMat_col2Arr(U, u, initer, n);
314           // vector_scalarMult(additive,u,-2*y[initer]*U[initer][initer],n);
315
316           vector_scalarMult(additive,U[initer],-2*y[initer]*U[initer][initer],n);
317
318
319           additive[initer] += y[initer];
320           for(int k=initer-1;k>=0;k--){
321               // copyMat_col2Arr(U, u, k, n);
322               // additive[k] += y[k];
323               // double tmpscalar = -2*scalar_prod(u,additive,n);
324               // vector_update(additive,u,tmpscalar,n);
325
326               additive[k] += y[k];
327               double tmpscalar = -2*scalar_prod(U[k],additive,n);
```

```cpp
328                    vector_update(additive,U[k],tmpscalar,n);
329                }
330
331            vector_update(x,additive,1,n);
332
333            A.post_MV(x,Ax);
334            vector_add(1.0,b,-1.0,Ax,r,n);
335
336            normr = norm(r,n);
337
338            normr = norm(r,n);
339            beta = scalarsgn(normr)*normr;
340
341            vector_zero_out(u,inner);
342            mat_zero_out(U,n,inner+1);
343            copyArray(u, r, n);
344            u[0] += beta;
345            normalize(u, n);
346
347            copyArr2Mat_col(U,u,0,n);
348            //  Apply Householder projection to r.
349            //  w = r - 2*u*u'*r;
350            w[0] = -beta;
351        }// end if flag~=0
352
353
354        if (flag == 2) { //stagnation
355            break;
356        }
357
358        if (normr <= tolb) { // convergence
359            flag = 0;
360            break;
361        }
362
363
364    } //end for out
365
366
367
368
369    if (normr <= tolb){
370        flag = 0;
371    }
372
373
374    double end_time = omp_get_wtime();
375    double runtime = (end_time - start_time);
376
377    // writeCSV(matname,n, inner, iterations,b, x, w, resvec, U, R);
378    cout<<"GMRES finished, with residual norm: " << normr<< endl << "flag: "<< flag<< "
        iterations: " << iterations << endl;
379    // cout << "Runtime: " << runtime << " seconds" << endl;
380
381    free(r);
382    free(Ax);
383    free(xm);
384    free(Axm);
385
386    for (int i = 0; i < 2; ++i) {
387        free(J[i]);
388    }
389    free(J);
390
391    free(resvec);
392
393    for (int i = 0; i < inner+1; ++i) {
394        free(U[i]);
395    }
```

```
396    free(U);
397
398    free(u);
399
400    for (int i = 0; i < inner; ++i) {
401        free(R[i]);
402    }
403    free(R);
404
405    free(w);
406
407    free(v);
408
409    free(vtemp);
410
411    free(jtemp);
412
413    free(y);
414    free(additive);
415
416 }//this is end of gmres
```

A.3   functions.h

```
1  #ifndef FUNCTIONS_H
2  #define FUNCTIONS_H
3
4  #include <iostream>
5  #include <fstream>
6  #include <string>
7  #include <cstring>
8  #include <omp.h>
9
10 #include "constants.h"
11 #include "sparse.h"
12 #include "read_mat.h"
13
14 #include <fstream>
15 #include <string>
16
17 #include <cmath>
18
19
20 #include <random>
21
22
23 using namespace std;
24
25
26 double scalar_prod(double *v1, double *v2, int nr){
27     double sum = 0.0;
28
29     if(nr>par_threshhold){
30         #pragma omp parallel for reduction(+:sum)
31         for(int i = 0; i < nr; i++){
32             sum += v1[i]*v2[i];
33         }
34         return sum;
35     }else{
36         for(int i = 0; i < nr; i++){
37             sum += v1[i]*v2[i];
38         }
39         return sum;
40     }
41
42 }
43
```

```cpp
44  void elementwise_prod(double *v1, double *v2, int nr){
45      if(nr>par_threshhold){
46          #pragma omp parallel for
47              for(int i = 0; i < nr; i++){
48                  v1[i] *=v2[i];
49              }
50      }else{
51          for(int i = 0; i < nr; i++){
52                  v1[i] *=v2[i];
53              }
54      }
55  }
56
57  double norm(double *v1, int nr){
58      // Euclidean norm
59      return sqrt(scalar_prod(v1,v1,nr));
60  }
61
62  void normalize(double* v1, int nr){
63      double v_norm = norm(v1,nr);
64      if(v_norm<eps){
65          cout<<"Warning: Norm very small:  "<<endl;
66      }else{
67          if(nr>par_threshhold){
68              #pragma omp parallel for schedule(dynamic, nr/1000+1)
69              for(int i=0; i<nr; i++){
70                  v1[i] /= v_norm;
71              }
72          }else{
73              for(int i=0; i<nr; i++){
74                  v1[i] /= v_norm;
75              }
76          }
77      }
78  }
79
80
81
82  void vector_update(double *v1, double *p, double alpha, int nr){
83      if(nr>par_threshhold){
84          #pragma omp parallel for schedule(dynamic, nr/1000+1)
85              for(int i = 0; i < nr; ++i){
86                  v1[i] += alpha * p[i];
87              }
88      }else{
89              for(int i = 0; i < nr; ++i){
90                  v1[i] += alpha * p[i];
91              }
92      }
93  }
94
95
96
97  void vector_scalarMult(double *v1, double *p, double alpha, int nr){
98      if(nr>par_threshhold){
99          #pragma omp parallel for schedule(dynamic, nr/1000+1)
100         for(int i = 0; i < nr; ++i){
101             v1[i] = alpha * p[i];
102         }
103     }else{
104         for(int i = 0; i < nr; ++i){
105             v1[i] = alpha * p[i];
106         }
107     }
108 }
109
110 void vector_add(double alpha,double* v1, double beta, double* v2, double* result, int nr){
111     if(nr>par_threshhold){
112         #pragma omp parallel for schedule(dynamic, nr/1000+1)
```

```cpp
113        for(int i = 0; i < nr; ++i){
114            result[i] = alpha * v1[i] + beta * v2[i];
115        }
116    }else{
117        for(int i = 0; i < nr; ++i){
118            result[i] = alpha * v1[i] + beta * v2[i];
119        }
120    }
121 }
122
123 void vector_zero_out(double* v, int n){
124    if(n>par_threshhold){
125        #pragma omp parallel for schedule(dynamic, n/1000+1)
126        for(int i = 0; i <n; ++i){
127            v[i] = 0;
128        }
129    }else{
130        for(int i = 0; i <n; ++i){
131            v[i] = 0;
132        }
133    }
134 }
135
136 void vector_zero_out(double* v, int from, int to, int n){
137
138 if(n>par_threshhold){
139     #pragma omp parallel for schedule(dynamic, n/1000+1)
140     for(int i = from; i <= to; ++i){
141         v[i] = 0;
142     }
143    }else{
144        for(int i = from; i <= to; ++i){
145            v[i] = 0;
146        }
147    }
148 }
149
150
151 void mat_zero_out(double** A, int nr, int nc){
152    if(nc>par_threshhold){
153        #pragma omp parallel for schedule(dynamic, nr/1000+1)
154        for(int i = 0; i <nc; ++i){
155            for(int j = 0; j<nr; j++){
156                A[i][j] = 0;
157            }
158        }
159    }else{
160        for(int i = 0; i <nc; ++i){
161            for(int j = 0; j<nr; j++){
162                A[i][j] = 0;
163            }
164        }
165    }
166 }
167
168
169 void normalize_copy(double *normalized, double *original, int nr){
170    double v_norm = norm(original,nr);
171    if(v_norm<eps){
172        cout<<"Warning: Norm to small "<<v_norm<<endl;
173    }else{
174        if(nr>par_threshhold){
175            #pragma omp parallel for schedule(dynamic, nr/1000+1)
176            for(int i=0; i<nr; i++){
177                normalized[i] = original[i]/v_norm;
178            }
179        }else{
180            for(int i=0; i<nr; i++){
181                normalized[i] = original[i]/v_norm;
```

```
182                     }
183                 }
184         }
185 }
186
187
188 void copyArray(double* dest, const double* src, int size) {
189     if(size>par_threshhold){
190             #pragma omp parallel for schedule(dynamic, size/1000+1)
191             for(int i=0; i<size; i++){
192                 dest[i] = src[i];
193             }
194         }else{
195             for(int i=0; i<size; i++){
196                 dest[i] = src[i];
197             }
198         }
199 }
200
201 void copyArr2Mat_col(double** matrix, const double* arr, int col, int size){
202     // #pragma omp parallel for schedule(dynamic, nr/1000+1)
203     for(int i=0; i<size; i++){
204         matrix[col][i] = arr[i];
205     }
206 }
207
208 void copyArr2Mat_col(double** matrix, const double* arr, int col, int size, int from, int to){
209     // #pragma omp parallel for schedule(dynamic, nr/1000+1)
210     for(int i=from; i<=to; i++){
211         matrix[col][i] = arr[i];
212     }
213 }
214
215 void copyArr2Mat_row(double** matrix, const double* arr, int row, int size){
216     // #pragma omp parallel for schedule(dynamic, nr/1000+1)
217     for(int i=0; i<size; i++){
218         matrix[i][row] = arr[i];
219     }
220 }
221
222 void copyArr2Mat_row(double** matrix, const double* arr, int row, int size, int from, int to){
223     // #pragma omp parallel for schedule(dynamic, nr/1000+1)
224     for(int i=from; i<=to; i++){
225         matrix[i][row] = arr[i];
226     }
227 }
228
229 void copyMat_col2Arr(double** matrix, double* arr, int col, int size){
230     // #pragma omp parallel for schedule(dynamic, nr/1000+1)
231     for(int i=0; i<size; i++){
232         arr[i] = matrix[col][i];
233     }
234 }
235
236
237 double scalarsgn(double x){
238     if (x >= 0){
239         return 1.0;
240     }
241     else{
242         return -1.0;
243     }
244 }
245
246
247
248
249
250
```

```cpp
251  void printMat(double** matrix, int cols, int rows) {
252      for (int i = 0; i < cols; i++) {
253          for (int j = 0; j < rows; j++){
254              cout << matrix[i][j] << " ";
255          }
256          cout << endl;
257      }
258  }
259
260
261  void triangularSolver(double** A, double* b, double* x, int n){
262      for(int row = n-1; row>=0; row--){
263          double sum = 0.0;
264          //could be parralelized here or reduction operator
265          for(int col = row+1; col<n; col++){
266              sum+=A[row][col]*x[col];
267          }
268          x[row] = (b[row]-sum)/A[row][row];
269      }
270  }
271
272
273
274
275  void writeCSV(char* matname, int n, int inner, int iterations,double* b, double* xstar, double*
        w, double* resvec, double** U, double** R) {
276
277      char filename[100];
278      strcpy(filename, "data_output/");
279      strcat(filename, matname);
280      strcat(filename, "_data.csv");
281      ofstream file(filename, ofstream::trunc); // Open the file in write mode with truncation
282
283
284      // Write b (1D array)
285      file << "b";
286      for (int i = 0; i < n; i++) {
287          file << "," << b[i];
288      }
289      file << std::endl;
290
291      // Write xstar (1D array)
292      file << "xstar";
293      for (int i = 0; i < n; i++) {
294          file << "," << xstar[i];
295      }
296      file << std::endl;
297
298      // Write w (1D array)
299      file << "w";
300      for (int i = 0; i < n + 1; i++) {
301          file << "," << w[i];
302      }
303      file << std::endl;
304
305      // Write resvec (1D array)
306      file << "resvec";
307      for (int i = 0; i < iterations; i++) {
308          file << "," << resvec[i];
309      }
310      file << std::endl;
311
312      // Write U (2D array)
313      for (int i = 0; i < inner; i++) {
314          file << "U_" << i;
315          for (int j = 0; j < n; j++) {
316              file << "," << U[i][j];
317          }
318          file << std::endl;
```

```cpp
319          }
320
321      // Write R (2D array)
322      for (int i = 0; i < inner; i++) {
323          file << "R_" << i;
324          for (int j = 0; j < inner; j++) {
325              file << "," << R[i][j];
326          }
327          file << std::endl;
328      }
329
330      file.close(); // Close the file
331  }
332
333
334
335  void createMat(sparse* mat, char* fname) {
336      std::ifstream file(fname);
337
338      if (file.is_open()) {
339          int nr, nc, nt;
340          file >> nr >> nc >> nt;
341
342          int* iat = (int*)malloc((nr + 1) * sizeof(int));
343          int* ja = (int*)malloc(nt * sizeof(int));
344          double* elem = (double*)malloc(nt * sizeof(double));
345
346          readCSRmat(fname, &nr, &nc, &nt, &iat, &ja, &elem, false);
347
348          mat->set_nrow(nr);
349          mat->set_ncol(nc);
350          mat->set_n_term(nt);
351          mat->set_iat(iat);
352          mat->set_ja(ja);
353          mat->set_coef(elem);
354
355          file.close();
356
357          // wo muss ich hier freen? oder nicht weil in dekonstruktor gemacht wird?
358      } else {
359          cout << "FILE NOT OPENED " <<fname<< endl;
360      }
361  }
362
363  bool getRHS(char* fname, double* b){
364      std::ifstream file(fname);
365
366      if (file.is_open()) {
367          double temp_nr;
368          int nr;
369          file >> temp_nr;
370          nr = static_cast<int>(temp_nr);
371          for(int i=0;i<nr;i++){
372              file >> b[i];
373          }
374          cout<<"Vector file "<<fname<<endl;
375          cout<<"Vector length " << nr<<endl;
376          return true;
377      }else{
378          cout << "FILE NOT OPENED " << fname<< endl;
379          return false;
380      }
381  }
382
383  void make_b(char* fname,int nr){
384      std::ofstream file(fname,std::ofstream::trunc);
385
386      std::random_device rd;
387      std::mt19937 generator(rd());
```

```cpp
388        std::uniform_real_distribution<double> distribution(0.0, 1.0);
389        file<<nr<<endl;
390        #pragma omp parallel for schedule(dynamic, nr/1000+1)
391        for(int i=0;i<nr;i++){
392            file<<distribution(generator)<<endl;
393        }
394
395        cout<<"random Vector of length created " << nr<<endl;
396        file.close();
397    }
398
399
400    void save_solution(char* fname, double* xstar, int n){
401        std::ofstream file(fname,std::ofstream::trunc);
402        file<<n<<endl;
403        for(int i=0;i<n;i++){
404            file<<xstar[i]<<endl;
405        }
406        cout<<"Vector saved in: " << fname << " length: "<< n<<endl;
407        file.close();
408    }
409
410    double* getFEM_sparse(sparse* mat){
411        std::ifstream f_coef("FEM_output/coef.txt");
412        std::ifstream f_iat("FEM_output/iat.txt");
413        std::ifstream f_ja("FEM_output/Ja.txt");
414        std::ifstream f_b("FEM_output/b.txt");
415
416        int n_term, n_row_plus1, n_row;
417        f_coef >> n_term;
418        f_ja >> n_term;
419        f_iat >> n_row_plus1;
420        f_b >> n_row;
421
422        double* elem = (double*) malloc(n_term*sizeof(double));
423        int* ja = (int*) malloc(n_term*sizeof(int));
424        int* iat = (int*) malloc(n_row_plus1*sizeof(int));
425
426
427        double* b = (double*) malloc(n_row*sizeof(double));
428
429        for(int i = 0; i < n_term; i++){
430            f_coef >> elem[i];
431            f_ja >> ja[i];
432        }
433
434        for (int i = 0; i < n_row_plus1; i++){
435            f_iat >> iat[i];
436        }
437        for (int i = 0; i < n_row; i++){
438            f_b >> b[i];
439        }
440
441        mat->set_nrow(n_row);
442        mat->set_ncol(n_row);
443        mat->set_n_term(n_term);
444        mat->set_iat(iat);
445        mat->set_ja(ja);
446        mat->set_coef(elem);
447
448        printf("Matrix rows %d columns %d nterm %d\n",n_row,n_row,n_term);
449        f_coef.close();
450        f_iat.close();
451        f_ja.close();
452        f_b.close();
453        return b;
454    }
455
456    double** mat_allocation(int n_row , int n_col){
```

```
457    double **A = (double**) malloc(n_row*sizeof(double*));
458    for (int i = 0; i < n_row; ++i) {
459        A[i] = (double*)malloc(n_col * sizeof(double));
460    }
461    return A;
462 }
463
464 void mat_free(double** A, int n_col, int n_row){
465    for (int i = 0; i < n_row; ++i) {
466        free(A[i]);
467    }
468    free(A);
469 }
470
471
472 int getNZ(double** A, int n){
473    // number of nonzeros in mtx
474    // A is square
475    int nnz = 0;
476    #pragma omp parallel for schedule(dynamic,n/1000+1)
477    for(int i = 0; i < n; i++){
478        for(int j = 0; j < n; j++){
479            if(abs(A[i][j])>eps){
480                #pragma omp atomic
481                nnz++;
482            }
483        }
484    }
485    return nnz;
486 }
487
488 int getNZ_idx(double** A, double** idx, int n){
489    // number of nonzeros in mtx
490    // A is square
491    int nnz = 0;
492    for(int i = 0; i < n; i++){
493        for(int j = 0; j < n; j++){
494            if(abs(A[i][j])>eps){
495                nnz++;
496                idx[nnz][0] = i;
497                idx[nnz][1] = j;
498            }
499        }
500    }
501    return nnz;
502 }
503
504 double max(double* v, int n){
505    double maxVal = v[0];
506    #pragma omp parallel for schedule(dynamic, n/1000+1)
507    for (int i = 1; i < n; i++) {
508        if (v[i] > maxVal) {
509            #pragma omp critical
510            {
511                if (v[i] > maxVal) {
512                    maxVal = v[i];
513                }
514            }
515        }
516    }
517    return maxVal;
518 }
519
520 #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;
```

```cpp
69      // Copy the rest of the content from the temporary file to the original file
70      std::string line;
71      while (std::getline(inFile, line)) {
72          if (!line.empty()) {
73              outFile << line << std::endl;
74          }
75      }
76
77      // Close both files
78      inFile.close();
79      outFile.close();
80
81      // Delete the temporary file
82      std::remove(tempFile.c_str());
83  }
84
85
86  void stiffness_struct_para(char* fname, int ne, int nn, double** topol) {
87      double** Adj = mat_allocation(ne, nn);
88      double* i_idx = (double*)malloc(static_cast<int>(nn*nn*0.2));
89      double* j_idx = (double*)malloc(static_cast<int>(nn*nn*0.2));
90
91      int idx;
92      #pragma omp parallel for schedule(dynamic, nn/1000+1)
93      for (int i = 0; i < ne; i++) {
94          for (int k = 0; k < 3; k++) {
95              idx = static_cast<int>(floor(topol[i][k]) - 1);
96              Adj[i][idx] = 1;
97          }
98      }
99
100     double sum;
101     int nnz = 0;
102     std::string tempFile = "temp_file.txt";
103     std::ofstream file(tempFile, std::ofstream::trunc); // Open the temporary file in write
            mode with truncation
104
105     #pragma omp for schedule(dynamic,nn/1000+1)
106     for (int i = 0; i < nn; i++) {
107         for (int j = 0; j < nn; j++) {
108             sum = 0;
109             for (int k = 0; k < ne; k++) {
110                 if (Adj[k][i] * Adj[k][j] > 0) {
111                 nnz++;
112                 file << j + 1 << " " << i + 1 << " " << 0 << std::endl;
113                 break;
114             }
115             }
116
117         }
118     }
119     file.close();
120
121     // Reopen the temporary file in read mode
122     std::ifstream inFile(tempFile);
123     if (!inFile) {
124         std::cerr << "Error opening the temporary file for reading." << std::endl;
125         return;
126     }
127
128     // Open the original file in write mode with truncation
129     std::ofstream outFile(fname, std::ofstream::trunc);
130     if (!outFile) {
131         std::cerr << "Error opening the file for writing." << std::endl;
132         inFile.close();
133         return;
134     }
135
136     // Write the new data at the beginning of the original file
```

```
137    outFile << nn << " " << nn << " " << nnz << std::endl;

138

139    // Copy the rest of the content from the temporary file to the original file
140    std::string line;
141    while (std::getline(inFile, line)) {
142        if (!line.empty()) {
143            outFile << line << std::endl;
144        }
145    }

146

147    // Close both files
148    inFile.close();
149    outFile.close();

150

151    // Delete the temporary file
152    std::remove(tempFile.c_str());
153 }

154

155

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

158

159    // Create vectors for i_idx and j_idx
160    std::vector<int> i_idx;
161    std::vector<int> j_idx;
162    int idx;

163

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

169

170    #pragma omp parallel for schedule(dynamic, nn/1000+1)
171    for (int i = 0; i < ne; i++) {
172        for (int k = 0; k < 3; k++) {
173            idx = static_cast<int>(floor(topol[i][k]) - 1);
174            Adj[i][idx] = 1;
175        }
176    }

177

178    double sum;
179    int nnz = 0;
180    #pragma omp parallel for schedule(dynamic,nn/1000+1)
181    for (int i = 0; i < nn; i++) {
182        for (int j = 0; j < nn; j++) {
183            sum = 0;
184            for (int k = 0; k < ne; k++) {
185                if (Adj[k][i] * Adj[k][j] > 0) {
186                    j_idx[nnz] = (i +1);
187                    i_idx[nnz] = (j);
188                    #pragma omp atomic
189                    nnz++;
190                    // Optionally, check the capacity and reserve more space if needed (e.g.,
                        double the current capacity)
191                    if (nnz >= i_idx.capacity()) {
192                        std::cout<<"capacity to small"<<nnz<<endl;
193                        int newCapacity = i_idx.capacity() * 1.5;
194                        i_idx.reserve(newCapacity);
195                        j_idx.reserve(newCapacity);
196                    }

197

198                    break;
199                }
200            }
201        }
202    }
203    int* ja_loc = &i_idx[0];
204    int* irow = &j_idx[0];
```

```
205
206       int* iat_loc = (int*)malloc(nn*sizeof(int));
207       irow2iat(nn,nnz,irow,iat_loc);
208       H->set_n_term(nnz);
209       H->set_ncol(nn);
210       H->copy_ja(ja_loc);
211       H->copy_iat(iat_loc);
212
213       free(iat_loc);
214 }
215
216 void stiffness_struct_small_loops(sparse* H, int ne, int nn, double** topol) {
217         // Create vectors to store the i_idx and j_idx pairs
218       std::vector<int> i_idx;
219       std::vector<int> j_idx;
220
221       // Loop through each element
222       for (int i = 0; i < ne; i++) {
223           for (int k = 0; k < 3; k++) {
224               int idx = static_cast<int>(floor(topol[i][k]) - 1);
225               for (int l = 0; l < 3; l++) {
226                   int jdx = static_cast<int>(floor(topol[i][l]) - 1);
227                   i_idx.push_back(idx);
228                   j_idx.push_back(jdx);
229               }
230           }
231       }
232
233       // Combine i_idx and j_idx into pairs for sorting
234       std::vector<std::pair<int, int>> pairs;
235       for (int i = 0; i < i_idx.size(); i++) {
236           pairs.push_back(std::make_pair(i_idx[i], j_idx[i]));
237       }
238
239       // Sort the pairs to bring duplicates together
240       std::sort(pairs.begin(), pairs.end());
241
242       // Remove duplicates and copy the unique pairs back to i_idx and j_idx
243       int nnz = 0;
244       for (int i = 0; i < pairs.size(); i++) {
245           if (i == 0 || pairs[i] != pairs[i - 1]) {
246               i_idx[nnz] = pairs[i].first + 1;
247               j_idx[nnz] = pairs[i].second;
248               nnz++;
249           }
250       }
251
252       // Allocate memory for the CSR arrays
253       int* ja_loc = new int[nnz];
254       int* iat_loc = new int[nn + 1];
255
256       // Copy the data from i_idx and j_idx to the CSR arrays
257       for (int i = 0; i < nnz; i++) {
258           ja_loc[i] = j_idx[i];
259       }
260
261       // Compute the CSR arrays from i_idx and ja_loc
262       irow2iat(nn, nnz, &i_idx[0], iat_loc);
263
264       H->set_n_term(nnz);
265       H->set_ncol(nn);
266       H->copy_ja(ja_loc);
267       H->copy_iat(iat_loc);
268       // H->printComponents();
269       delete[] ja_loc;
270       delete[] iat_loc;
271 }
272
273 double get_loc(double** coord, double** topol, double *D, double*V, int el, double*** Loc){
```

```cpp
274         double *t = topol[el];
275         double x[3];   //i,j,m
276         double y[3];   //i,j,m
277         double a[3],b[3],c[3], delta, temp, mod_v;
278
279         for(int i = 0; i<3; i++){
280             x[i] = coord[static_cast<int>(t[i]-1)][0];
281             y[i] = coord[static_cast<int>(t[i]-1)][1];
282         }
283
284         a[0] = x[1]*y[2] - x[2]*y[1];
285         a[1] = x[2]*y[0] - x[0]*y[2];
286         a[2] = x[0]*y[1] - x[1]*y[0];
287
288         int idx1, idx2;
289         for(int i = 0; i<3;i++){
290             idx1 = (i+1)%3;
291             idx2 = (i+2)%3;
292             b[i] = y[idx1] - y[idx2];
293             c[i] = x[idx2] - x[idx1];
294         }
295
296         delta = (a[0]+a[1]+a[2])/2;
297         mod_v = norm(V,2);
298
299         if(mod_v>eps){// prevent zero division
300             double temp;
301             for(int i = 0; i<3; i++){
302                 for(int j = 0; j<3; j++){
303                     temp = (V[0]*b[j] + V[1]*c[j]);
304                     Loc[0][i][j] = (D[0]*b[i]*b[j]+ D[1]*c[i]*c[j])/(4*delta);
305                     Loc[1][i][j] = temp/6;
306                     Loc[2][i][j] = (V[0]*b[i] + V[1]*c[i])*temp/(8*delta*mod_v*max(D,2));
307                 }
308             }
309         }else{
310             std::cout<<" Norm v zero, just compute stiffness Matrix H \n";
311             for(int i = 0; i<3; i++){
312                 for(int j = 0; j<3; j++){
313                     Loc[0][i][j] = (D[0]*b[i]*b[j]+ D[1]*c[i]*c[j])/(4*delta);
314                 }
315             }
316         }
317
318
319         return delta;
320 }
321
322 void loc2glob(sparse* A_ptr, double** Loc, double** topol, int e){
323     double* coef = A_ptr->get_coef();
324     int* ja = A_ptr->get_ja();
325     int* iat = A_ptr->get_iat();
326     int row, col;
327     double* top = topol[e];
328     for(int i = 0; i<3; i++){    // choose row
329         row = static_cast<int>(top[i]) - 1;
330         for(int j = 0; j<3; j++){// chose col
331             col = static_cast<int>(top[j]) - 1;
332             int k;
333             for(k = iat[row]; k<iat[row+1];k++){ //go over column indices in ja
334                 if(ja[k]==col){break;}
335             }
336             #pragma omp atomic
337             coef[k] += Loc[i][j];
338         }
339     }
340 }
341
342 sparse* create_FEM_mat(double** coord, double** topol, int ne, int nn, double* D,\
```

```cpp
343                          double h, double tau, double* V, double* delta){
344      sparse* mat_list;
345      mat_list = new sparse[3]; // H,B,S
346      double*** Loc = (double***) malloc(3*sizeof(double**));
347      // double ***Loc;
348      // Loc = new double**[3];
349      int buffer_length = static_cast<int>(nn * nn * 0.1);
350
351      int nterm, nr;
352
353      int* ja = (int*) calloc(buffer_length,sizeof(int));
354      int* iat = (int*) calloc(nn+1,sizeof(int));
355
356      //allocate space for the local H, B, S
357      for(int i = 0; i<3;i++){Loc[i] = mat_allocation(3,3);}
358
359      char fname[] = "fullmat.txt";
360      // stiffness_struct(fname, ne,  nn,  topol);
361      // createMat(&mat_list[0], fname);
362      stiffness_struct_small_loops(&mat_list[0], ne,  nn,  topol);
363
364      mat_list[2] = mat_list[1] = mat_list[0];
365
366      for(int i = 0; i<ne; i++){
367          delta[i] = get_loc(coord,topol,D,V,i,Loc);
368          for(int k = 0; k<3;k++){
369              loc2glob(&mat_list[k],Loc[k],topol,i);
370          }
371      }
372      // printVec(delta,ne);
373      mat_list[2].scalarMult(tau*h);
374
375      return mat_list;
376 }
377
378
379 double force(double x, double y){
380      double f = 0;
381      // do stuff
382      return f;
383 }
384
385
386 void imposeBC(sparse* H, double** bound, double* q, int nb){
387      double R = 1e15;
388      int j, bound_idx, start, end;
389      int* iat = H->get_iat();
390      int* ja = H->get_ja();
391      double* coef = H->get_coef();
392
393      for(int i = 0; i<nb; ++i){
394          bound_idx = static_cast<int>(bound[i][0])-1;
395          start = iat[bound_idx];
396          end = iat[bound_idx+1];
397
398          for(j = start; j<end; j++){
399              if (ja[j] == bound_idx){break;}
400          }
401
402          coef[j] = R;
403          q[bound_idx] = R*bound[i][1];
404      }
405 }
406
407
408
409 double* getAssembledSystem(sparse* A){
410      // FEM assembly
411      char mesh_path[] = "mesh/mesh.dat";
```

```cpp
412        char node_path[] = "mesh/dirnod.dat";
413        char coord_path[] = "mesh/xy.dat";
414
415        double D[2] = {1,1};
416        double V[2] = {1,3};
417        int ne, nn, nb; // number of mesh (800), number of nodes(441), number of ? (80)
418
419        double **coord, **bound, *delta, *delta_node, *F;
420        double **topol, *temp, *q;
421
422        sparse* FEM;
423
424
425        ifstream mesh(mesh_path);
426        ifstream nodes(node_path);
427        ifstream xy(coord_path);
428
429        //read data
430        int not_used;
431
432        xy >> nn >> not_used;
433        nodes >> nb >> not_used;
434        mesh >> ne >> not_used >> not_used >> not_used;
435
436        bound = mat_allocation(nb,2);
437        coord = mat_allocation(nn,2);
438        topol = mat_allocation(ne,3);
439
440
441
442        for (int i = 0; i < nn; ++i){xy >> coord[i][0] >> coord[i][1];}
443        for (int i = 0; i < nb; ++i){nodes >> bound[i][0] >> bound[i][1];}
444        for (int i = 0; i < ne; ++i){mesh >> topol[i][0] >> topol[i][1] >> topol[i][2]>>not_used;}
445
446
447        // printMat(coord,nb,2);
448
449        xy.close();
450        nodes.close();
451        mesh.close();
452
453        double h = sqrt(2)*(coord[0][1]-coord[1][1]);// why sqrt2
454        double tau = 0.1; // whats tau?
455
456        delta = (double*)malloc(ne*sizeof(double));
457
458        delta_node = (double*)calloc(nn,sizeof(double));
459
460        F = (double*)malloc(nn*sizeof(double));
461        q = (double*)calloc(nn,sizeof(double));
462
463
464
465        FEM = create_FEM_mat(coord, topol, ne, nn, D, h, tau, V, delta);
466
467        int idx;
468        #pragma omp parallel for schedule(dynamic,nn/1000+1)
469        for(int i = 0; i<ne; i++){
470            for(int k=0; k<3; k++){
471                idx = static_cast<int>(floor(topol[i][k])-1);
472                delta_node[idx] += delta[k]/3;
473            }
474        }
475
476        // RHS
477        for(int i = 0; i<nn; i++){
478            F[i] = force(coord[i][0],coord[i][1]*delta_node[i]);
479        }
480
```

```
481     // BCs
482     imposeBC(&FEM[0], bound, q, nb);
483     FEM[1].addition_update(&FEM[2]);
484     FEM[0].addition_update(&FEM[1]);
485
486     *A = FEM[0];
487     vector_update(q,F,1,nn);
488
489     mat_free(bound,nb,2);
490     mat_free(coord,nn,2);
491     mat_free(topol,ne,3);
492     free(delta);
493     free(delta_node);
494     free(F);
495     delete[] FEM;
496
497
498     return q;
499 }
```

A.5   sparse.h

```
1  #ifndef SPARSE_H
2  #define SPARSE_H
3
4  #include <omp.h>
5  #include <iostream>
6  #include <cmath>
7
8  #include "constants.h"
9
10 class sparse{
11 private:
12     int nrow, ncol, n_term;
13     double* coef; // non zero matrixelements
14     int *iat, *ja; // first non zero column index iat, what is ja?
15
16
17 public:
18     // constructors
19     sparse();
20     sparse(int nrow, int ncol, int n_term);
21     ~sparse(); // destructor
22
23
24     // get
25     int get_nrow(){return nrow;}
26     int get_ncol(){return ncol;}
27     int get_n_term(){return n_term;}
28
29     double* get_coef(){return coef;}
30     int* get_iat(){return iat;}
31     int* get_ja(){return ja;}
32
33
34     // set
35     void set_nrow(int n){
36         nrow = n;
37         ncol = n;
38         free(iat);
39         iat = (int*) malloc(ncol*sizeof(int));
40     }
41     void set_ncol(int n){
42         nrow = n;
43         ncol = n;
44         free(iat);
45         iat = (int*) malloc(ncol*sizeof(int));
```

41

```cpp
46        }
47        void set_n_term(int n){
48            n_term = n;
49            free(coef);
50            free(ja);
51            ja = (int*) malloc(n_term*sizeof(int));
52            coef = (double*) malloc(n_term*sizeof(double));
53        }
54
55        // void alloc_coef(int n){coef = (double*)malloc((n)*sizeof(double));}
56        // void alloc_iat(int n){iat = (int*)malloc((n+1)*sizeof(int));}
57        // void alloc_ja(int n){ja = (int*)malloc((n)*sizeof(int));}
58
59        void set_coef(double* v){coef = v;}
60        void set_iat(int* v){iat = v;}
61        void set_ja(int* v){ja = v;}
62
63        void copy_coef(double* v){
64            #pragma omp parallel for
65            for(int i = 0; i<n_term;i++){
66                coef[i]=v[i];
67            }
68        }
69
70        void copy_iat(int* v){
71            #pragma omp parallel for
72            for(int i = 0; i<nrow+1;i++){
73                iat[i]=v[i];
74            }
75        }
76
77        void copy_ja(int* v){
78            #pragma omp parallel for
79            for(int i = 0; i<n_term;i++){
80                ja[i]=v[i];
81            }
82        }
83
84
85        //operators
86        sparse& operator=(sparse&);
87        friend bool operator==(const sparse& lhs, const sparse& rhs);
88
89        //Data Operations
90        void full2sparse(double** A, int n_row);
91
92
93        // matrix funcitons
94        void addition_update(sparse* B_ptr);
95        void scalarMult(double alpha);
96        void post_MV(double* v, double* y); // A*b
97        void pre_MV(double* v, double* y); // b'*A
98        void matrixProduct(sparse* A_ptr, sparse* result);
99        void diag(double* v);
100       void getJacobi(double* v);
101       void diag_x_sparse(double* diag);
102       double* left_Jacobi();
103
104
105       // display mtx
106       void printMat();
107       void printComponents();
108   };
109
110
111   using namespace std;
112
113   sparse::sparse(){
114       // cout<<"empty constructor\n";
```

```
115     nrow = 0;
116     ncol = 0;
117     n_term = 0;
118
119     coef = nullptr;
120     ja = nullptr;
121     iat = nullptr;
122 };
123
124 sparse :: sparse(int nr, int nc, int nt){
125     nrow = nr;
126     ncol = nc;
127     n_term = nt;
128
129     coef = (double*)malloc(nt*sizeof(double));
130     ja = (int*)malloc(nt*sizeof(int));
131     iat = (int*)malloc((nr+1)*sizeof(int)); //+1
132 };
133
134 sparse :: ~sparse(){
135     free(coef);
136     free(ja);
137     free(iat);
138 }
139
140
141 void sparse::post_MV(double* v, double* y ){
142     // #pragma omp parallel for schedule(dynamic,nrow/1000+1)
143     for(int i=0; i<nrow; i++){
144         double sum = 0.0;
145         #pragma omp simd reduction(+:sum)
146         for(int j=iat[i]; j<iat[i+1]; j++){
147             sum += coef[j]*v[ja[j]];
148         }
149         y[i] = sum;
150     }
151 }
152
153 void sparse::pre_MV(double* v, double* y ){
154     #pragma omp parallel for schedule(dynamic,nrow/1000+1)
155     for(int j=0; j<ncol; j++){
156         double sum = 0.0;
157         // #pragma omp simd reduction(+:sum)
158         for(int i=0; i<nrow; i++){
159             for(int k=iat[i]; k<iat[i+1]; k++){
160                 if (ja[k] == j){
161                     sum += v[i]*coef[k];
162                 }
163             }
164         }
165         y[j] = sum;
166     }
167 }
168
169 sparse& sparse::operator=(sparse& A){
170     // std::cout<<"cpy"<<endl;
171     int n_row = A.get_nrow();
172     int n_term = A.get_n_term();
173
174     this->set_n_term(n_term);
175     this->set_ncol(n_row);
176     this->copy_coef(A.get_coef());
177     this->copy_iat(A.get_iat());
178     this->copy_ja(A.get_ja());
179
180     return *this;
181 }
182
183 bool operator==(const sparse& lhs, const sparse& rhs) {
```

```cpp
184        if (lhs.nrow != rhs.nrow || lhs.ncol != rhs.ncol || lhs.n_term != rhs.n_term) {
185            cout<<" Shapes are not equal\n";
186            return false;
187
188        }
189
190        // Compare ja arrays
191        for (int i = 0; i < lhs.n_term; i++) {
192            if (lhs.ja[i] != rhs.ja[i]) {
193                cout<<" ja arrays are not equal\n";
194                return false; //
195
196            }
197        }
198
199        // Compare iat arrays
200        for (int i = 0; i <= lhs.nrow; i++) {
201            if (lhs.iat[i] != rhs.iat[i]) {
202                cout<<" iat arrays are not equal\n";
203                return false; //
204
205            }
206        }
207
208        return true; // Shapes, ja, and iat arrays are equal
209 }
210
211
212 void sparse::full2sparse(double** A, int n){
213        // A is square
214        // A is sparse --> suppose only max 10% of elements are nonzero
215        double sparse_ratio = 1;
216        int n_term_max = static_cast<int>(n * n * sparse_ratio);
217
218        double* coef_temp = (double*)malloc(n_term_max*sizeof(double));
219        int* Ja_temp = (int*)malloc(n_term_max*sizeof(int));
220        int* iat_temp = (int*)malloc((n+1)*sizeof(int));
221
222        this->set_ncol(n);
223        this->set_nrow(n);
224
225        int idx_nt = 0;
226        int idx_nr = 0;
227        bool new_row = true;
228
229        #pragma omp parallel for schedule(dynamic,n/1000+1)
230        for(int i = 0; i < n; i++){
231            for(int j = 0; j < n; j++){
232                if(abs(A[i][j])>eps){
233                    coef_temp[idx_nt] = A[i][j];
234                    Ja_temp[idx_nt] = j;
235                    if(new_row){
236                        iat_temp[idx_nr] = idx_nt;
237                        idx_nr++;
238                        new_row = false;
239                    }
240                    idx_nt++;
241                }
242            }
243            new_row = true;
244        }
245        iat_temp[idx_nr] = idx_nt;
246
247        this->set_n_term(idx_nt);
248        this->copy_coef(coef_temp);
249        this->copy_ja(Ja_temp);
250        this->copy_iat(iat_temp);
251
252        free(iat_temp);
```

```cpp
253        free ( coef_temp );
254        free ( Ja_temp );
255 }
256
257 void sparse :: addition_update ( sparse * B_ptr ){
258        if (* this ==* B_ptr ){
259            double * add = B_ptr ->get_coef ();
260            #pragma omp parallel for schedule ( dynamic , n_term /1000 +1)
261            for (int i = 0; i<n_term ; i ++){
262                coef [i] += add [i];
263            }
264        } else {
265            cout <<" Sparsity  structure  is  not  the  same  \n";
266        }
267
268 }
269
270
271 void sparse :: scalarMult ( double alpha ){
272        #pragma omp parallel for schedule ( dynamic , n_term /1000 +1)
273        for (int i = 0; i<n_term ; i ++){
274            coef [i] *= alpha ;
275        }
276 }
277 void sparse :: matrixProduct ( sparse * B_ptr , sparse * result ){
278        // (nxm )*( mxk ) = (nxk ) but usually square anyways
279
280
281        sparse & B = * B_ptr ;
282
283        if ( nrow != B.ncol ){
284            std :: cout <<" Incompatible  Size" << endl ;
285        }
286
287
288        for (int i = 0; i < nrow ; i ++){
289            double sum = 0.0;
290            for (int j = 0; j < B.ncol ; j ++){
291
292                sum += i+j;
293            }
294
295
296        }
297
298
299 }
300
301
302 void sparse :: diag ( double * v ){
303        int start , end ;
304        #pragma omp parallel for schedule ( dynamic , ncol /1000+1)
305        for (int i = 0; i<ncol ; i ++){
306            start = iat [i];
307            end = iat [i+1];
308            for (int j = start ; j<end ; j ++){
309                if (ja [j] == i){
310                    v[i] = coef [j];
311                    break ;
312                }
313            }
314        }
315 }
316
317 void sparse :: getJacobi ( double * v ){
318        this ->diag (v);
319        int zeros = 0;
320        #pragma omp parallel for schedule ( dynamic , ncol /1000 +1)
321        for (int i = 0; i<ncol ; i ++){
```

```cpp
322          if (v[i] == 0){
323              v[i]=1;
324              #pragma omp atomic
325              zeros++;
326              }
327          v[i] = 1/v[i];
328      }
329      if(zeros>0){
330          cout<< "WARNING: Number of zeros in diagonal: "<<zeros<<endl;
331      }
332  }
333
334  void sparse::diag_x_sparse(double* diag){
335      int start, end;
336      #pragma omp parallel for schedule(dynamic, ncol/1000+1)
337      for(int i = 0; i<ncol; i++){
338          start = iat[i];
339          end = iat[i+1];
340          for(int j = start; j<end; j++){
341              coef[j] *= diag[i];
342
343          }
344      }
345  }
346
347
348  double* sparse::left_Jacobi(){
349      double* v = (double*) calloc(ncol,sizeof(double));
350      this->getJacobi(v);
351      this->diag_x_sparse(v);
352      return v;
353  }
354
355  #include <iomanip>
356
357  void sparse::printMat() {
358      // Create a 2D matrix to represent the sparse matrix
359      double** matrix = new double*[nrow];
360      for (int i = 0; i < nrow; i++) {
361          matrix[i] = new double[ncol];
362          // Initialize all elements to 0
363          for (int j = 0; j < ncol; j++) {
364              matrix[i][j] = 0.0;
365          }
366      }
367
368      // Fill the matrix with the non-zero elements from the sparse matrix
369      for (int row = 0; row < nrow; row++) {
370          int start = iat[row];                    // Starting index for the current row
371          int end = iat[row + 1];                  // Ending index for the current row
372
373          // Iterate over the non-zero elements in the current row
374          for (int index = start; index < end; index++) {
375              int column = ja[index];              // Column index of the non-zero element
376              double value = coef[index];          // Value of the non-zero element
377              matrix[row][column] = value;         // Store the value in the matrix
378          }
379      }
380
381      // Print the matrix
382      for (int i = 0; i < nrow; i++) {
383          for (int j = 0; j < ncol; j++) {
384              std::cout << setw(8) << matrix[i][j] << " ";
385          }
386          std::cout << endl;
387      }
388
389      // Free the memory allocated for the matrix
390      for (int i = 0; i < nrow; i++) {
```

```cpp
                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
```