Scientific Computing Computer Exercise
A Comparison of Sparse Direct and Preconditioned Krylov Subspace Solvers in Two and Three Dimensions

Nerine Usman (4349997) Marianne Schaaphok (4355822), Group 19

October 2018

1 Problem statement

The goal of this report is to compare the performance of three sparse direct and iterative solution methods applied to the discrete Laplacian in two and three dimensions. We consider the Poisson problem on the unit square $D = [0,1]^2$ and the unit cube $D = [0,1]^3$. The Poisson equation relates the unknown field $u(\mathbf{x}) = u(x,y)$ in 2D ($u(\mathbf{x}) = u(x,y,z)$ in 3D) to the known excitation $f(x)$:

$$-\Delta\mathbf{u} = -\text{div} \cdot \text{grad}\mathbf{u} = f \text{ on } D.$$

The problem is supplied with non-homogeneous Dirichlet boundary conditions $\mathbf{u} = \mathbf{u}_0$ on ∂D. We discretize the domain D by a uniform grid G^h with mesh width $h = 1/n$ consisting of N nodes in total including those on the boundary. For the two different problems we have

$$N = \begin{cases} (n+1)^2 & \text{in 2D-case} \\ (n+1)^3 & \text{in 3D-case} \end{cases}.$$

For example in the 2D case

$$G^h = \{(x_i, y_j) | x_i = ih, y_i = jh; h = 1/n, 0 \leq i,j \leq n, n \in \mathbb{N}\}.$$

Further we apply a central second order accurate finite difference scheme to the partial differential equations without elimination of the boundaries. The discretization leads to a linear system for the discrete solution $\mathbf{u}^h$: $A\mathbf{u}^h = \mathbf{f}^h$.

The report first gives a theoretical analysis of the solution methods and their complexities. In section 3 the implementations and the analaysis of the performance of the solvers is given.

2 Theoretical analysis

2.1 Source function and boundary data

First we determine the source function $\mathbf{f}(x)$ and the boundary data $\mathbf{u}_0(x)$, such that $\mathbf{u}_{ex}(\mathbf{x}) = \sin(xy)$ in 2D. We know that $\mathbf{f}(x) = -\Delta\mathbf{u}_{ex} = -\left(\frac{\partial^2 \mathbf{u}_{ex}}{\partial x^2} + \frac{\partial^2 \mathbf{u}_{ex}}{\partial y^2}\right)$. Now we want that $\mathbf{u}_{ex} = \sin(xy)$, substituting this in the equation gives:

$$\mathbf{f}(x) = -\left(-\sin(xy) \cdot y^2 - x^2 \sin(xy)\right) = \left(x^2 + y^2\right)\sin(xy).$$

Further on the boundary it should hold $\mathbf{u}_{ex}(\mathbf{x}) = \mathbf{u}_0(\mathbf{x})$ for $\mathbf{x} \in \partial D$. The boundary consists of four parts: $\partial D_1 = [0:1,0], \partial D_2 = [0,0:1], \partial D_3 = [0:1,1]$ and $\partial D_4 = [1,0:1]$. On boundary $\partial D_1, \partial D_3$ it is evident that $\mathbf{u}_0(\mathbf{x}) = 0$, since either x or y is zero and $\sin(0) = 0$. At boundary ∂D_3 $y = 1$ constantly, therefore the

boundary values should be $\mathbf{u}_0(\mathbf{x}) = \sin(x)$, for $\mathbf{x} \in \partial D_3$. For boundary ∂D_4 it holds that x is 1 constantly and therefore the boundary should satisfy $\mathbf{u}_0(\mathbf{x}) = \sin(y)$, for $\mathbf{x} \in \partial D_4$. To conclude we find:

$$\mathbf{u}_0(\mathbf{x}) = \sin(xy) = \begin{cases} 0 & \mathbf{x} \in \partial D_1, \partial D_2 \\ \sin(x) & \mathbf{x} \in \partial D_3 \\ \sin(y) & \mathbf{x} \in \partial D_4 \end{cases}. \tag{1}$$

The approach for the three-dimensional case is analogous for the exact solution $\mathbf{u}_{ex}(\mathbf{x}) = \sin(xyz)$ and results in:

$$\mathbf{f}(x) = \left((xy)^2 + (yz)^2 + (xz)^2\right)\sin(xyz) \tag{2}$$

$$u_0(\mathbf{x}) = \sin(xyz) \text{ for } \mathbf{x} \in \partial D. \tag{3}$$

2.2 Discretization scheme and bandwidth

Discretization is done without eliminating the boundaries. This means that there are $(n+1)^2$ unknowns in the 2D problem. Therefore A^h is of size $(n+1)^2 \times (n+1)^2$. Since a central second order accurate finite difference scheme is used the discretization in stencil notation is given by

$$\begin{bmatrix} & 1 & \\ 1 & 4 & 1 \\ & 1 & \end{bmatrix}.$$

Therefore every row contains at most five nonzero elements. And if we order the unknowns in a lexicographical order the contributing neighbour nodes of node i are the left neighbour $i-1$, right neighbour $i+1$ the lower neighbour $i-(n+1)$ and the upper one $i+(n+1)$. So therefore the upper and lower bandwidth of A are determined by the upper and lower neighbour respectively.

For 3D there are $(n+1)^3$ unknowns, so therefore the matrix A^h is $(n+1)^3 \times (n+1)^3$. With the second order scheme only the direct neighbours influence the solution at one node. Every node has six neighbours so together with the node itself, we find that every row of A^h has at most seven nonzero elements. In a lexicographical order the neighbours of node i are $i-1$, $i+1$, $i-(n+1)$, $i+(n+1)$, $i-(n+1)^2$, $i+(n+1)^2$. So by the latter two neighbours both the upper and lower bandwidth are of size $(n+1)^2$.

2.3 Cholesky decomposition

Next we will motivate that the Cholesky decomposition can be used to solve this problem. First the system matrix A^h must satisfy the SPD condition for which the Cholesky decomposition exists. We know that A^h is positive definite by definition and can be made symmetric.[1] As the factorization exists it can be used to solve the system using forward/backward solving.

The Cholesky decomposition results in savings in both computational cost and in memory storage and the numerical stability is guaranteed by the SPD property of the matrix. Another advantage of the Cholesky decomposition is that it preserves the sparsity structure of matrix A^h outside the bandwidth and is therefore efficient.

2.4 Computational complexity

Since the following complexities highly depend on the bandwidth b of A we will express everything in this bandwidth and will compute the final solutions at the end for the different cases. In 2.2 it was explained that

$$b = \begin{cases} n+1 = \sqrt{N} & \text{in 2D-case} \\ (n+1)^2 = \left((n+1)^3\right)^{2/3} = N^{2/3} & \text{in 3D-case} \end{cases}.$$

[1] Paragraph 3.7 from *Scientific Computing Lecture Notes, C.Vuik and D.J.P. Lahaye*

Cholesky Factorization

First we consider the 2D case again. It is known that because both the upper and lower bandwidth of A are b the lower bandwidth of C is b as well [1, Thm 4.9.1]. This reduces the complexity of the Cholesky factorization significantly, since C contains many zero-elements. We can adjust the algorithm from [1,Page 54] to Algorithm 1. The sums are reduced in size and the second for loop only needs to loop only b times.

Algorithm 1 Cholesky Factorization Step

for $k = 1 \to N$ **do**
  $A(k,k) \leftarrow C(k,k) = \sqrt{A(k,k) - \sum_{j=\max\{k-b,1\}}^{k-1} C(k,j)^2}$
  for $i = k+1 \to k+b$ **do**
    $A(i,k) \leftarrow C(i,k) = \frac{1}{C(k,k)} \left(A(i,k) - \sum_{j=\max\{i-b,1\}}^{k-1} C(i,j)C(k,j) \right)$
  end for
end for

The complexity of calculating $A(k,k)$ for some k is at most b flops for the squares, $b-1$ for additions and 2 extra flops for the subtraction and square root. Adding these results in a total of $2b+1$ flops. The complexity of $A(i,k)$ is at most $b-(i-k-1)$ flops for the multiplications, $b-(i-k-1)-1$ for additions and 2 extra flops for the subtraction and dividing. So the complexity of calculating $A(i,k)$ is $2(b-(i-k-1))+1$. Analyzing the second for-loop first, we see that the computational cost for this loop is

$$\begin{aligned}\sum_{i=k+1}^{k+b} 2(b-(i-k-1))+1 &= \sum_{i=1}^{b} 2(b-(i-1))+1 \\ &= \sum_{i=1}^{b} 2i+1 \\ &= b(b+2) \text{ flops.}\end{aligned}$$

It can be seen that the complexity in the main loop does not depend on k, therefore the total computational cost is

$$N\left(2b+1+b(b+2)\right) = N(b^2+4b+1) = \begin{cases} N(N+4\sqrt{N}+1) = \mathcal{O}(N^2) \text{ flops} & \text{in 2D-case} \\ N(N^{4/3}+4N^{2/3}+1) = \mathcal{O}(N^{7/3}) \text{ flops} & \text{in 3D-case} \end{cases}.$$

Forward substitution

For the forward substitution we adjust Algorithm 2 in [1, p. 49] on the known sparsity of A. Since the Cholesky decomposition is used, $L = C$ in our case. Because of the relative small lower bandwith b of C, the elements $C(i, i-1-b)$ are zero. Therefore the algorithm can be written as in Algorithm 2.

Algorithm 2 Forward Substitution

1: **for** $k = 1 \to N$ **do**
2: $\quad y(i) \leftarrow [f(i) - C(i, i-b : i-1) \cdot f(i-b : i-1)]$
3: **end for**

The computation on line 2 requires b multiplications, $b-1$ summations and 1 subtraction, addition gives a total cost of $2b$ flops. Therefore the total computational cost of this algorithm is

$$N(2b) \text{ flops} = \begin{cases} 2N\sqrt{N} \text{ flops} & \text{in 2D-case} \\ 2N^{5/3} \text{ flops} & \text{in 3D-case} \end{cases}.$$

Backward substitution

The algorithm for backward substitution is very similar to the forward substitution. We adjust algorithm 3 in [1, p. 49]. Since for the Cholesky decomposition $A = CC^T$, we substitute C^T for U. So $U(i, i+1:n) = C(i+1:n, i))$. Furthermore because the lower bandwidth of C is still b; $C(i+1+b:n, i) = 0$. The pseudo code for the adjusted algorithm can be found in Algorithm 3.

Algorithm 3 Backward Substitution

1: **for** $k = 1 \rightarrow N$ **do**
2: $\quad u(i) \leftarrow [y(i) - C(i+1:i+b, i)^T \cdot f(i+1:i+b)]/C(i,i)$
3: **end for**

The transpose in row 2 has been added such that the multiplication is still well defined, otherwise we would multiply two row vectors. The amount of computations needed for one iteration is equal to the ones needed for the computation of line 2 in Algorithm 2 for the forward substitution plus one extra flop for the division by $C(i, i)$. So the total amount of flops for the backward substitution is

$$N(2b+1) \text{ flops} = \begin{cases} N\left(2\sqrt{N}+1\right) \text{ flops} & \text{in 2D-case} \\ N\left(2N^{2/3}+1\right) \text{ flops} & \text{in 3D-case} \end{cases}.$$

2.5 Symmetric Successive Overrelaxation (SSOR)

The Symmetric Successive Overrelaxation (SSOR) can be described as a Basic Iterative Method. In order to do this we define the splitting $A = M - N$, with

$$M = \frac{1}{\omega(2-\omega)}(D - \omega E)D^{-1}(D - \omega F)$$

and

$$N = M - A = \frac{1}{\omega(2-\omega)}(D - \omega E)D^{-1}(D - \omega F) - (D - E - F).$$

The iteration matrix is then given by: $B = I - M^{-1}A$.

The requirement for using SSOR for solving system $A\mathbf{u} = \mathbf{f}$ is that A is SPD. As mentioned before the matrix A from our system is SPD and therefore the SSOR method can be used. The SSOR method will be used with relaxation parameter $\omega = 1.5$.

2.6 Computational Complexity SSOR

Just as in the complexity for forward and backward substitution we can use the sparsity of A to obtain a more efficient algorithm for $SSOR$. Say A has at most s nonzero elements per row ($s = 5$ for 2D, $s = 7$ for 3D as we found in section 2.2). In Algorithm 8 from [1, p. 69], which describes one single step of an SSOR(ω) iteration, the third and eighth line need only $2(s-1)+1$ flops, since there are only $s-1$ nonzero elements on the i-th row of A, which are not on the diagonal. These are all multiplied by an corresponding element from $\mathbf{u}$, subtracted from $\mathbf{f}(i)$ and then divided by a number. Both the for-loops need the same amount of computations, so the computational complexity of one iteration of SSOR needs

$$2N\left(2(s-1)+4\right) = 4N\left(s+1\right) = \begin{cases} 24N \text{ flops} & \text{in 2D-case} \\ 32N \text{ flops} & \text{in 3D-case} \end{cases}.$$

Suppose we need N_{iter} iterations to reach convergence the total computation complexity of SSOR is

$$\begin{cases} 24N \cdot N_{iter} \text{ flops} & \text{in 2D-case} \\ 32N \cdot N_{iter} \text{ flops} & \text{in 3D-case} \end{cases}.$$

2.7 SSOR as preconditioner

Remark: *The question asked to repeat question (5) describing preconditioned CG as a BIM. Prof Vuik told us that the question should have been to show that CG method with $\tilde{A}, \tilde{\mathbf{u}}, \tilde{\mathbf{f}}$ can be rewritten to the preconditioned CG with $A, \mathbf{u}, \mathbf{f}$. And that preconditioned CG with SSOR as a preconditioner can be used on the system. This will be shown below.*

To show that this is the case, we start with the CG algorithm on the system $\tilde{A}, \tilde{\mathbf{f}}, \tilde{\mathbf{u}}$ where $\tilde{A} = P^{-1}AP^{-T}, \tilde{\mathbf{u}} = P^T\mathbf{u}, \tilde{\mathbf{f}} = P^{-1}\mathbf{f}$. Next we replace $\tilde{A}$ by $P^{-1}AP^{-T}$, $\tilde{\mathbf{f}}$ by $P^{-1}\mathbf{f}$ and $\tilde{\mathbf{u}}$ by $P^T\mathbf{u}$. This leads to the following algorithm:

Algorithm 4 Conjugate Gradient Method on $\tilde{A}\tilde{u} = \tilde{f}$

1: $P^T\mathbf{u}^0 = 0, \mathbf{r}^0 = P^{-1}\mathbf{f}, k = 1$
2: **for** k=1,2,... **do**
3: **if** k=1 **then**
4: $\mathbf{p}^1 = \mathbf{r}^0$
5: **else**
6: $\beta_k = \frac{\left(\mathbf{r}^{k-1}\right)^T\mathbf{r}^{k-1}}{\left(\mathbf{r}^{k-2}\right)^T\mathbf{r}^{k-2}}$
7: $\mathbf{p}_k = \mathbf{r}^{k-1} + \beta_k\mathbf{p}^{k-1}$
8: **end if**
9: $\alpha_k = \frac{\left(\mathbf{r}^{k-1}\right)^T\mathbf{r}^{k-1}}{(\mathbf{p}^k)^TP^{-1}A\mathbf{A}P^{-T}\mathbf{p}^{k-2}}$
10: $P^T\mathbf{u}^k = P^T\mathbf{u}^{k-1} + \alpha_k\mathbf{p}^k$
11: $\mathbf{r}^k = \mathbf{r}^{k-1} - \alpha_kP^{-1}AP^{-T}\mathbf{p}^k$
12: $k = k + 1$
13: **end for**

In this algorithm we set $\mathbf{s}^k = P\mathbf{r}^k$ and $\mathbf{t}^k = P^{-T}\mathbf{p}^k$ and rewrite the algorithm. We take $M = PP^T$, and therefore $M^{-1} = P^{-T}P^{-1}$. Below it is shown per line of the algorithm, how it is rewritten:
Line 1: $P^T\mathbf{u}^0 = 0 \Rightarrow \mathbf{u}^0 = 0$. $P^{-1}\mathbf{s}^0 = P^{-1}\mathbf{f} \Rightarrow \mathbf{s}^0 = \mathbf{f}$.
Line 4: $\mathbf{p}^1 = \mathbf{r}^0 \Rightarrow P^{-1}\mathbf{s}^1 = P^{-1}\mathbf{f} \Rightarrow \mathbf{s}^k = \mathbf{f}$.
Line 6: $\beta_k = \frac{\left(P^{-1}\mathbf{s}^{k-1}\right)^TP^{-1}\mathbf{s}^{k-1}}{\left(P^{-1}\mathbf{s}^{k-2}\right)^TP^{-1}\mathbf{s}^{k-2}} = \frac{\left(\mathbf{s}^{k-1}\right)^TP^{-T}P^{-1}\mathbf{s}^{k-1}}{\left(\mathbf{s}^{k-2}\right)^TP^{-T}P^{-1}\mathbf{s}^{k-2}} = \frac{\left(\mathbf{s}^{k-1}\right)^TM^{-1}\mathbf{s}^{k-1}}{\left(\mathbf{s}^{k-2}\right)^TM^{-1}\mathbf{s}^{k-2}}$.
Line 7: $P^Tt^k = P^{-1}s^{k-1} + \beta_kP^Tt^{k-1} \Rightarrow t^k = P^{-T}P^{-1}s^{k-1} + \beta_kP^{-T}P^Tt^{k-1} \Rightarrow t^k = M^{-1}s^{k-1} + \beta_kt^{k-1}$.
Line 9: $\alpha_k = \frac{\left(P^{-1}\mathbf{s}^{k-1}\right)^TP^{-1}\mathbf{s}^{k-1}}{(P^T\mathbf{t}^k)^TP^{-1}A\mathbf{A}P^{-T}P^T\mathbf{t}^{k-2}} = \frac{\left(\mathbf{s}^{k-1}\right)^TP^{-T}P^{-1}\mathbf{s}^{k-1}}{(\mathbf{t}^k)^TPP^{-1}A\mathbf{A}P^{-T}P^T\mathbf{t}^{k-2}} = \frac{\left(\mathbf{s}^{k-1}\right)^TM^{-1}\mathbf{s}^{k-1}}{(\mathbf{t}^k)^TA\mathbf{A}\mathbf{t}^{k-2}}$
Line 10: $P^T\mathbf{u}^k = P^T\mathbf{u}^{k-1} + \alpha_kP^T\mathbf{t}^k \Rightarrow \mathbf{u}^k = \mathbf{u}^{k-1} + \alpha_kP^{-T}P^T\mathbf{t}^k \Rightarrow \mathbf{u}^k = \mathbf{u}^{k-1} + \alpha_k\mathbf{t}^k$.
Line 11: $P^{-1}\mathbf{s}^k = P^{-1}\mathbf{s}^{k-1} - \alpha_kP^{-1}AP^{-T}P^T\mathbf{t}^k \Rightarrow \mathbf{s}^k = \mathbf{s}^{k-1} - \alpha_kPP^{-1}A\mathbf{t}^k \Rightarrow \mathbf{s}^k = \mathbf{s}^{k-1} - \alpha_kA\mathbf{t}^k$

Placing the new equations in the algorithm and setting $\mathbf{z}^k = M^{-1}\mathbf{s}^k$, results in the following algorithm, which is the Preconditioned CG algorithm:

The SSOR preconditioning matrix M is defined by:

$$M_{SSOR} = \frac{1}{\omega(2-\omega)}(D - \omega E)D^{-1}(D - \omega F) \tag{4}$$

Due to the fact that D is symmetric and $E = F^T$ (since A is symmetric), it is easy to verify that $M = PP^T$, with $P = (\frac{1}{\omega(2-\omega)})^{1/2}(D-\omega E)D^{-1/2}$. With $D^{-1/2}$ we mean the matrix with the square root of the diagonal element. To verify that the SSOR matrix can function as a preconditioner, we check the following three conditions:

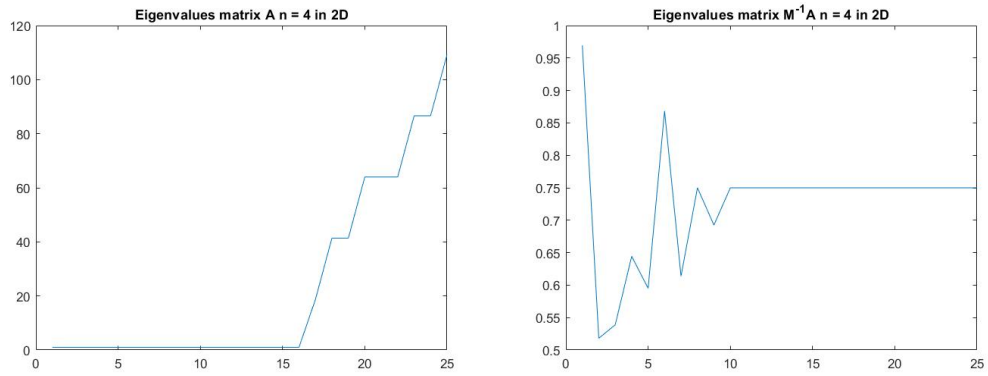
1. Matrix M is SPD $\rightarrow$ A simple check in Matlab shows that indeed matrix M is SPD.

2. The eigenvalues of $M^{-1}A$ should be clustered around 1. In figure 1 it can be seen that the eigenvalues of $M^{-1}A$ are more clustered around 1 than the eigenvalues of A and so this condition is satisfied.

Algorithm 5 Preconditioned Conjugate Gradient Method

1: $\mathbf{u}^0 = 0, \mathbf{s}^0 = \mathbf{f}, k = 1$
2: **while** $||\mathbf{s}^k||/||\mathbf{f}|| > C$ **do**
3: $\quad \mathbf{z}^{k-1} = M^{-1}\mathbf{s}^{k-1}$
4: $\quad$ **if** k=1 **then**
5: $\quad\quad \mathbf{t}^1 = \mathbf{z}^0$
6: $\quad$ **else**
7: $\quad\quad \beta_k = \frac{\left(\mathbf{s}^{k-1}\right)^T \mathbf{z}^{k-1}}{(\mathbf{s}^{k-2})^T \mathbf{z}^{k-2}}$
8: $\quad\quad \mathbf{t}_k = \mathbf{z}^{k-1} + \beta_k \mathbf{t}^{k-1}$
9: $\quad$ **end if**
10: $\quad \alpha_k = \frac{\left(\mathbf{s}^{k-1}\right)^T \mathbf{z}^{k-1}}{(\mathbf{t}^k)^T A \mathbf{A} \mathbf{t}^{k-2}}$
11: $\quad \mathbf{u}^k = \mathbf{u}^{k-1} + \alpha_k \mathbf{t}^k$
12: $\quad \mathbf{s}^k = \mathbf{s}^{k-1} - \alpha_k A \mathbf{t}^k$
13: $\quad k = k + 1$
14: **end while**

3. It should be possible to obtain $M^{-1}\mathbf{y}$ at low cost. This is the case, since obtaining the LU decomposition of M is easy to obtain and M is sparse as will be explained in the next section.



(a) A (b) $M^{-1}A$

Figure 1: Eigenvalues matrix A and $M^{-1}A$.

The standard CG method has the requirement that A is SPD. It is easy to verify that the matrix $\tilde{A} = P^{-1}AP^{-T}$ is indeed SPD. It is known that A is SPD. Further we can say that $\tilde{A}$ is symmetric because A is symmetric. Positive definite can be shown as follows:

$$\mathbf{x}P^{-1}AP^{-T}\mathbf{x}^T = \mathbf{x}P^{-1}AP^{-T}\mathbf{x}^T = (\mathbf{x}P^{-1})A(\mathbf{x}P^{-1})^T = \mathbf{y}A\mathbf{y}^T > 0.$$

.

So it follows that we can indeed use Preconditioned CG on our system with SSOR as a preconditioner.

2.8 Computational Complexity SSOR as preconditioner

A pseudo code for the preconditioned conjugate gradient method (PCG) can be found in Algorithm 6.

The computational complexity for one iteration is determined by lines 3 up to 12. For every line we will determine the amount of flops needed. For line 3 we can determine $M^{-1}\mathbf{r}$ in a smart way. We know the structure of M as in Eq. (4).

Algorithm 6 Preconditioned Conjugate Gradient Method

1: $\mathbf{u}^0 = 0, \mathbf{r}^0 = \mathbf{f}, k = 1$
2: **while** $||\mathbf{r}^k||/||\mathbf{f}|| > C$ **do**
3: $\quad \mathbf{z}^{k-1} = M^{-1}\mathbf{r}^{k-1}$
4: $\quad$ **if** k=1 **then**
5: $\quad\quad \mathbf{p}^1 = \mathbf{z}^0$
6: $\quad$ **else**
7: $\quad\quad \beta_k = \frac{\left(\mathbf{r}^{k-1}\right)^T \mathbf{z}^{k-1}}{(\mathbf{r}^{k-2})^T \mathbf{z}^{k-2}}$
8: $\quad\quad \mathbf{p}_k = \mathbf{z}^{k-1} + \beta_k \mathbf{p}^{k-1}$
9: $\quad$ **end if**
10: $\quad \alpha_k = \frac{\left(\mathbf{r}^{k-1}\right)^T \mathbf{z}^{k-1}}{(\mathbf{p}^k)^T \mathbf{A}\mathbf{p}^{k-2}}$
11: $\quad \mathbf{u}^k = \mathbf{u}^{k-1} + \alpha_k \mathbf{p}^k$
12: $\quad \mathbf{r}^k = \mathbf{r}^{k-1} - \alpha_k A\mathbf{p}^k$
13: $\quad k = k + 1$
14: **end while**

Remark that it is very easy to determine the LU-decomposition of M by this expression since $D - \omega E$ is a lower triangular matrix and $D - \omega F$ a upper triangular one. Therefor, $M = LU$ for

$$L = (D - \omega E)D^{-1},$$
$$U = \frac{1}{\omega(2-\omega)}(D - \omega F),$$

with L a lower triangular matrix with ones on the diagonal and U an upper triangular matrix. Since we have this decomposition we can use forward and backward substitution to determine z^{k-1}. Since the bandwidth of M is the same as A's bandwidth we can use the calculations in section 2.4 for the computational complexity of forward/backward substituting. This results in

$$N(4b+1)\text{flops},$$

for line 3 of Algorithm 6. Next we will look of the second part of the **if**-statement, since the first option will only be calculated in the first iteration. For the calculation in line 7, we need to determine one inner product and one division, since the denominator has already been calculated in the previous iteration. So this results in $2N$ flops. In line 8 there are N multiplication and N additions, since both the vectors contain N elements. So also for this line we need $2N$ flops. For line 10 and 12 the matrix vector product $\mathbf{A}\mathbf{p}^k$ has to be be calculated. Because A has at most s elements per row, this will cost $N(2s-1)$ flops. Since we can use the computations in line 7 for the denominator, the computational complexity left for line 10 is only one inner product in the denominator and one division, so $2N$ flops. Lines 11 and 12 both need $2N$ flops for the scalar multiplication of a vector and addition of two vectors. So in total this means we need

$$\underbrace{N(4b+1)}_{\text{line 3}} + \underbrace{2N}_{\text{line 7}} + \underbrace{2N}_{\text{line 8}} + \underbrace{N(2s-1)}_{\text{calculation } \mathbf{A}\mathbf{p}^k} + \underbrace{2N}_{\text{line 10}} + \underbrace{2N}_{\text{line 11}} + \underbrace{2N}_{\text{line 12}} = N(4b+2s+10) \text{ flops} \tag{5}$$

$$= \begin{cases} 4N\sqrt{N} + 20N \text{ flops} & \text{in 2D-case} \\ 4N^{5/3} + 34N \text{ flops} & \text{in 3D-case} \end{cases}. \tag{6}$$

3 Implementation Analysis

3.1 Implementation of A and f

We know the structure of the discretization matrix for the 1D problem A_1^h is [1, p. 25, Eq. 3.42]

$$A^h = \frac{1}{h^2}\begin{pmatrix} h^2 & 0 & \dots & & \dots & 0 \\ 0 & 2 & -1 & & & \vdots \\ \vdots & -1 & 2 & -1 & & \\ & & \ddots & \ddots & \ddots & \\ \vdots & & & -1 & 2 & 0 \\ 0 & \dots & & \dots & 0 & h^2 \end{pmatrix} \in \mathbb{R}^{(n+1)\times(n+1)}.$$

To construct the matrices for the higher dimensional problems we defined three help matrices

$$D_1 = \begin{pmatrix} 0 & \dots & & \dots & 0 \\ \vdots & 1 & & & \vdots \\ & & \ddots & & \\ \vdots & & & 1 & \vdots \\ 0 & \dots & & \dots & 0 \end{pmatrix} \in \mathbb{R}^{(n+1)\times(n+1)},$$

$$H_1 = \begin{pmatrix} h^2 & \dots & & \dots & 0 \\ \vdots & 0 & & & \vdots \\ & & \ddots & & \\ \vdots & & & 0 & \vdots \\ 0 & \dots & & \dots & h^2 \end{pmatrix} \in \mathbb{R}^{(n+1)\times(n+1)},$$

$$T_1^h = \begin{pmatrix} 0 & 0 & \dots & & \dots & 0 \\ 0 & 0 & -1 & & & \vdots \\ \vdots & -1 & \ddots & \ddots & & \\ & & \ddots & & -1 & \vdots \\ \vdots & & & -1 & 0 & 0 \\ 0 & \dots & & \dots & 0 & 0 \end{pmatrix} \in \mathbb{R}^{(n+1)\times(n+1)}.$$

The help matrices correspond with the one dimensional problem. D contains ones on the diagonal of the rows which correspond with interior nodes, H contains h^2 on the boundary nodes (such that when the whole matrix will be divided by h^2, ones remains, and T contains a -1 for the neighbouring nodes of each node i, which are also interior points. With these matrices we can also construct A_1^h by

$$A_1^h = \frac{1}{h^2}\left(H_1 + 2 \cdot D_1 + T_1\right).$$

To construct A_2^h we can consider the first and last row of the domain as part of the boundary and all the other rows are similar to a one dimensional problem, but they also have neighbours on the previous and next rows of the domain, so

$$A_2^h = \frac{1}{h^2}\left(H_1 \otimes I_1 + D_1 \otimes (A_1 \cdot h^2 + 2D_1) + T_1 \otimes D_1\right) \in \mathbb{R}^{(n+1)^2\times(n+1)^2}, \tag{7}$$

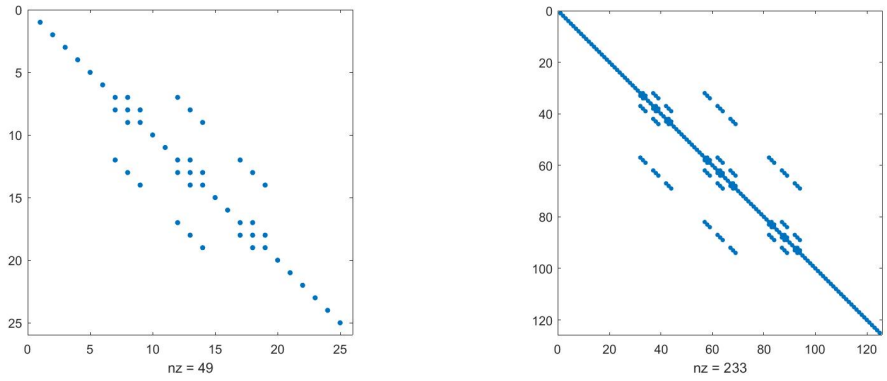
where I_1 is the identity matrix of size $(n+1)\times(n+1)$ and $\otimes$ denotes the tensor product. The first part of this expression makes sure there are h^2 on the rows which correspond with the nodes on the first and

last row of the domain (so for $y \in 0, 1$), since all these nodes belong to the boundary. The second part adds 2 to elements on the diagonal of A_1, which correspond to the interior points of the one-dimensional problem. Taking the the tensor product of D_1 with this matrix makes sure there is a 4 on every element on the diagonal for an interior point of the two-dimensional case and an h^2 for the elements corresponding to the left en right hand side boundary. Because of the structure of D_1 and H_1 we now made sure there is a nonzero value at all diagonal elements of A_2 and the relationships between right or left neighbours are also taken into account by using A_1. The third part of the above equation contains the relationships between different rows of nodes in the domain. Every adjacent row contains neighbours at the same value for x. So node i for example has $i + n + 1$ as a neighbour in the 2D case. This can be added by the tensor product of T_1 and D_1. In Figure 2a the structure of A_2 is shown.

For the 3D case we obtained the following equation

$$A_3 = \frac{1}{h^2}\left(H_1 \otimes I_2 + D_1 \otimes (A_2 \cdot h^2 + 2 * D_2) + T_1 \otimes D_2\right) \in \mathbb{R}^{(n+1)^3 \times (n+1)^3}, \tag{8}$$

where I_2 is the identity matrix of size $(n+1)^2 \times (n+1)^2$ and $D_2 = D_1 \otimes D_1$. This equation is very similar to the previous one, but now we consider the three-dimensional space as $n + 1$ layers of two-dimensional spaces. So the first and last layer are part of the boundary, and so are the boundaries of every layer itself. The structure of both matrices can be viewed in Figure 2b.



(a) 2D (b) 3D

Figure 2: Structure of A_2(a) and A_3(b) for $n = 4$. Every blue dot represents a nonzero value

We will construct $\mathbf{f}$ in three steps. Firstly, we need a vector $\mathbf{f}_{int}$ which contains the the values of $f(\mathbf{x})$ for $\mathbf{x}$ interior points. For that purpose construct a vector $\mathbf{g} = (g_1, \ldots, g_N)$ with $g_i = f(\mathbf{x}_i)$ for $\mathbf{x}_i$ a node with i being lexicographic ordered,

$$\mathbf{f}_{int} = \begin{cases} D_2\mathbf{g} & \text{in 2D-case} \\ D_3\mathbf{g} & \text{in 3D-case} \end{cases},$$

where $D_3 = D_1 \otimes D_2$, which contains ones on the diagonal for rows that correspond with interior nodes.

Secondly, construct a vector $\mathbf{f}_b$ which contains all the values of the Dirichlet boundary condition as in Eq. (1) and (3). Define $(\mathbf{u}_0)_i = u_0(\mathbf{x}_i)$. Then

$$\mathbf{f}_b = \begin{cases} (I_2 - D_2)\mathbf{u}_0 & \text{in 2D-case} \\ (I_3 - D_3)\mathbf{u}_0 & \text{in 3D-case} \end{cases},$$

where I_3 is the identity matrix of size $(n+1)^3 \times (n+1)^3$. Since D_2 contained a one on all the rows corresponding with interior points, So then $I_2 - D_2$ will contain a one on the diagonal for all other rows, which thus correspond with all the boundary points.

Thirdly, we constructed the matrices A_1, A_2 and A_3 such that they are symmetric. Normally one would get in the one dimensional case [1, p. 27]

$$A^h = \frac{1}{h^2}\begin{pmatrix} h^2 & 0 & \dots & & \dots & 0 \\ -1 & 2 & -1 & & & \vdots \\ \vdots & -1 & 2 & -1 & & \\ & & \ddots & \ddots & \ddots & \\ \vdots & & & -1 & 2 & -1 \\ 0 & \dots & & \dots & 0 & h^2 \end{pmatrix}.$$

To eliminate the red minus ones, we have to add u_1/h^2 and u_{n+1}/h^2 to the second and n-th element of f^h. These are exactly the nodes which have a neighbour lying on the boundary. The relation we are searching for between nodes with adjacent boundary points in 1D can be described by

$$B_1 = \begin{pmatrix} 0 & 0 & \dots & \dots & 0 \\ 1 & 0 & & & \vdots \\ 0 & & \ddots & & 0 \\ \vdots & & & 0 & 1 \\ 0 & \dots & \dots & 0 & 0 \end{pmatrix} \in \mathbb{R}^{(n+1)\times(n+1)}.$$

To construct all the relations between interior nodes and adjacent boundary nodes in 2D we get

$$B_2 = D_1 \otimes B_1 + B_1 \otimes D_1.$$

Here, the first part constructs the relations for the interior points adjacent to the boundary left and right and the second part gives the relations between the first and last interior row of the domain with the boundary at the lower and upper side of the domain. This gives a vector

$$\mathbf{f}_{bnb} = \begin{cases} B_2\mathbf{u}_0/h^2 & \text{in 2D-case} \\ B_3\mathbf{u}_0/h^2 & \text{in 3D-case} \end{cases},$$

where $B_3 = D_1 \otimes B_2 + B_1 \otimes D_2$.

Now we can construct our final vector $\mathbf{f}^h$ being

$$\mathbf{f}^h = \mathbf{f}_{int} + \mathbf{f}_b + \mathbf{f}_{bnb}.$$

3.2 Solver and discretization scheme

We solve the linear problem by using a direct solver with the Cholesky decomposition. We define $h = \frac{1}{2^p}$ for $p = 2, ..., 9$ in 2D and $p = 2, ..., 5$ in 3D. (Remark: for $p > 9$ in 2D and for $p > 5$ in 3D the computations were too heavy for our laptops). We consider the max-norm of the discretization error $||\mathbf{u}^h - \mathbf{u}^h_{ex}||_\infty$.

For the Cholesky decomposition the MATLAB function $chol(A)$ is used. Note here that the implementation of $chol(A)$ calls a optimized, very fast, routine in C++. It is therefore likely that the total time taken for the factorization is much faster than would be expected. The problem is solved using forward/backward solving. Algorithm 7 shows the pseudocode for the solver.

In figure 3 the solution to the problem is shown.

Algorithm 7 Forward/Backward solving

1: **procedure** FORWARD/BACKWARD(A,**f**) ▷ solving Au = f
2: $C = chol(A,'lower')$
3: $\mathbf{y} \leftarrow C \backslash \mathbf{f}$
4: $\mathbf{u} \leftarrow C^T \backslash \mathbf{y}$
5: **end procedure**

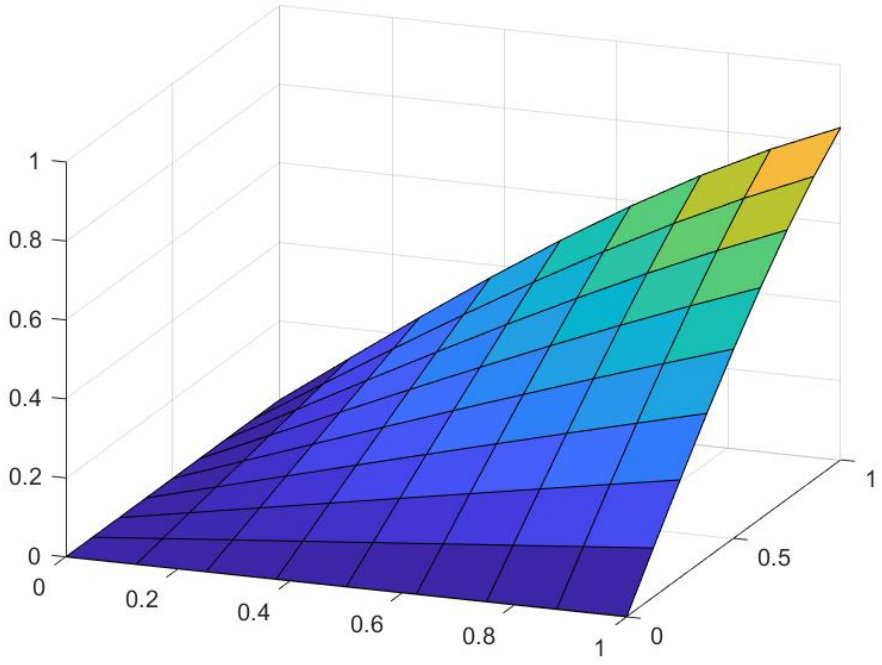
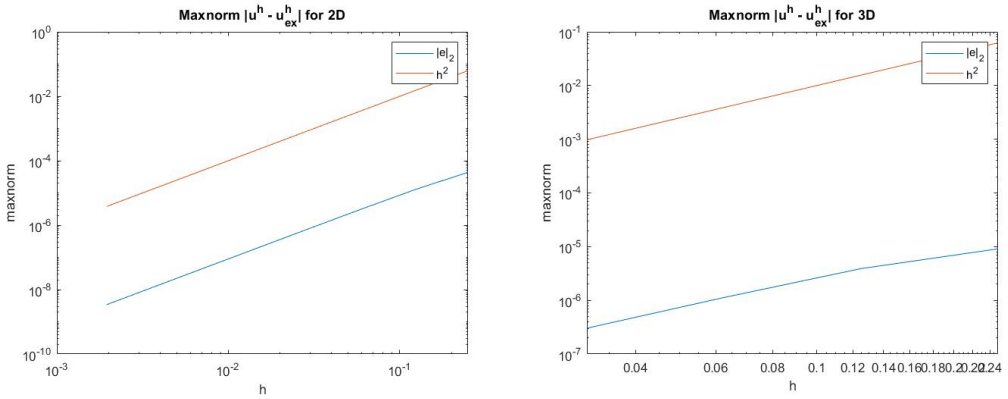


Figure 3: Solution to the problem with $n = 8$, using Cholesky to solve the problem

In order to show that the solver is second order accurate, we look at the max-norm of the error. In figure 4a the max-norm is plotted against the stepsize both on logarithmic scales. In this figure the function h^2 is shown as well. Since the slopes of both figures are the same in these logarithmic plots it is evident that the discretization scheme is second order accurate. In 3D it can even be seen that the method is slightly better than second order accurate.



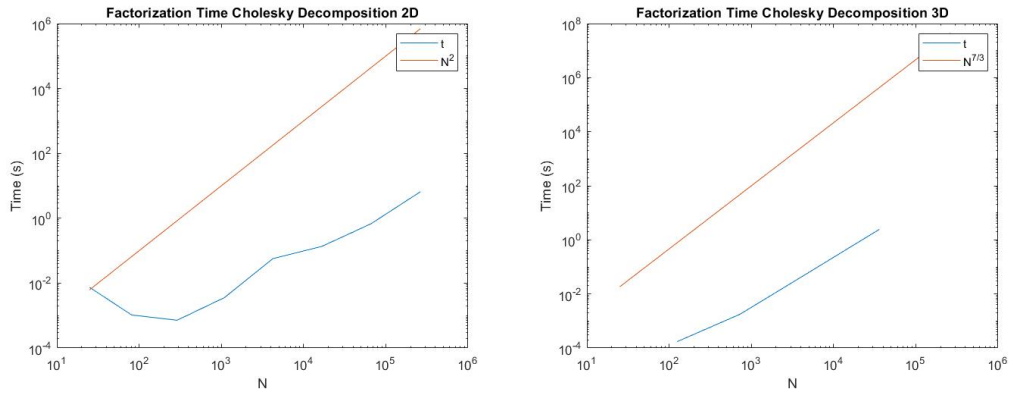
(a) 2D (b) 3D

Figure 4: Maxnorm with Forward/Backward solving

For the 3D case the same approach is used and the results are shown in figure 4b.

3.3 CPU Time factorization and solving

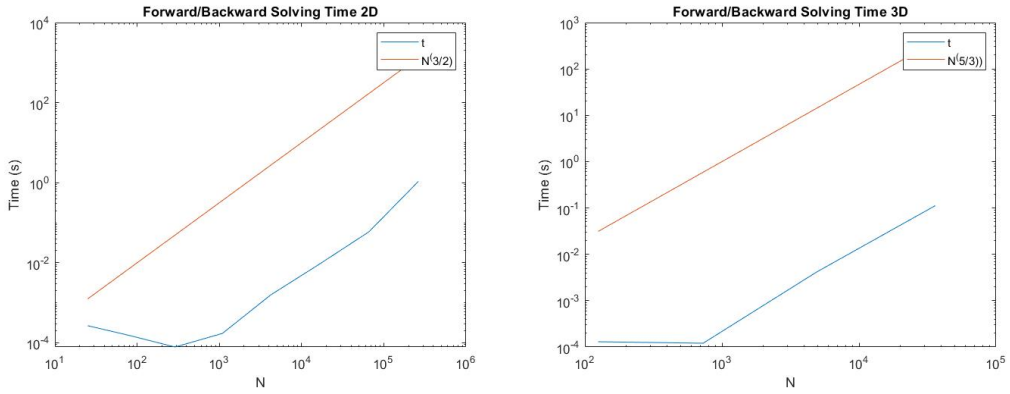
In figure 5a the factorization time of the Cholesky Decomposition is plotted against problem size N. As mentioned before the MATLAB function *chol(A)* is used. Because of the efficiency of this method, the times recorded for the factorization can be faster than the theoretical computed complexity. In both figures the red line gives the theoretical estimate of the time complexity, the blue line shows the computed time needed by MATLAB. It can be seen that for larger N the increase in time is close to the estimated complexity. However for the small N it can be seen that the needed time decreases for 2D and is almost equal for the 3D case. This probably occurs due to the efficiency of the function. It is evident from the figures that the time for factorization increases rapidly for large N.



(a) 2D (b) 3D

Figure 5: Factorization time Cholesky decomposition

In figure 6 the time for forward/backward solving is shown against the problem size N. In question 4 of the theoretical analysis the computational complexity of the forward/backward solving was derived. Both the computed complexities are shown in the figures. From the figures it can be concluded that the solving time confirms the calculated complexity for larger N. For small N the implementation is faster than the computed estimate, which is probably caused by the efficiency of the backslash operator of MATLAB.



(a) 2D (b) 3D

Figure 6: Forward backward solving time

3.4 Fill-in ratio

Next an analysis of the number of nonzeros is given. We denote the number of nonzeros in matrix A by $nnz(A)$. In order to compute the number of nonzeros the function $nnz(A)$ from MATLAB is used. We are interested in the fill-in ratio defined as $\frac{nnz(C)}{nnz(A)}$. In figure 7 the fill-in ratio's are shown against the problem size N. In 2D the fill in ratio is $\mathcal{O}(\sqrt{N})$ and in 3D it is about $\mathcal{O}(N^2/3)$. Notice that this is comparable to the bandwidths. This is because C will fill in the elements in the bandwidth of A.

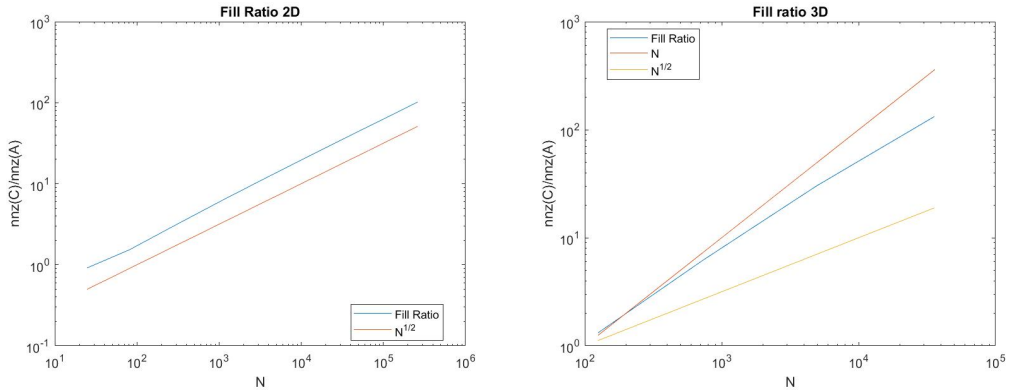
In 2D we have

$$\frac{nnz(C)}{nnz(A)} = c \cdot N^{1/2},$$

for some constant c. So

$$nnz(C) = c \cdot N^{1/2} \cdot nnz(A).$$

The number of non-zeros in C influences the performance of the forward and backward substitution phase. In section 2.4 we derived the performance assuming the whole bandwidth was filled in. But if we would introduce some matrix reordering scheme with a lower fill in we could substitute something smaller for the variable b in these computations. Since this influences the order of computations needed, it is interesting to look for a better reordering scheme.



(a) 2D (b) 3D

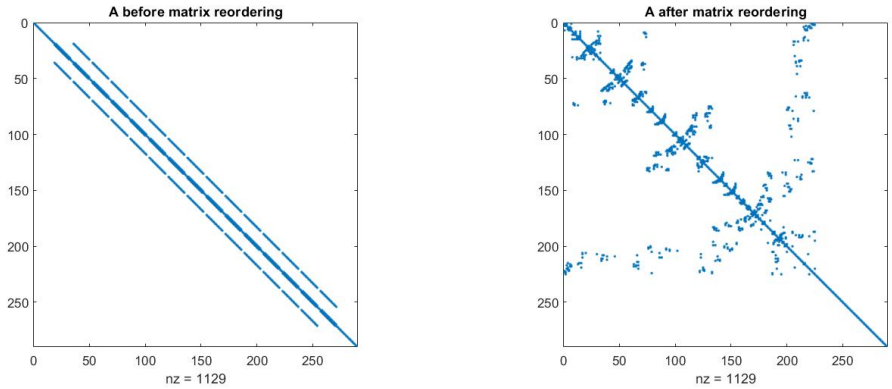
Figure 7: Fill-in ratio Cholesky decomposition in 2D (a) and 3D (b)

3.5 Matrix reordering scheme

In order to reduce the number of nonzeros created in the Cholesky decomposition, we introduce a matrix reordering scheme to reduce the bandwidth of matrix A. With this bandwidth reduction scheme the two previous tests regarding the CPU time and the fill-in ratio are repeated.

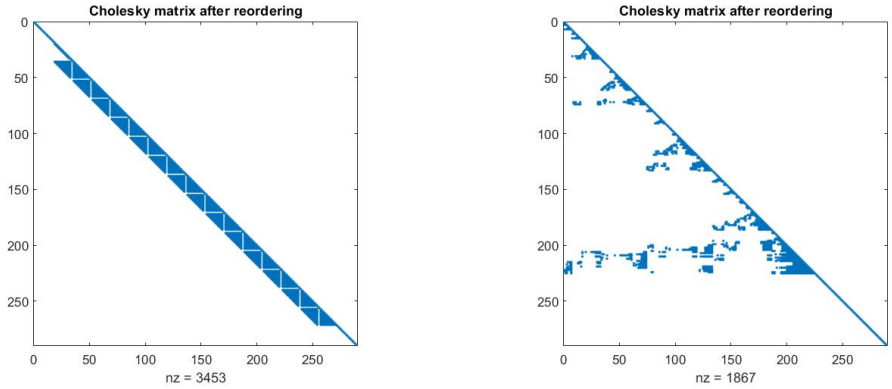
The matrix reordering scheme used is the Minimum Degree Ordering. The reordering scheme is implemented by use of the MATLAB commands $s = symamd(A)$. The matrix A, the vector $\mathbf{f}$ and $\mathbf{u}$ are reordered according to $\mathbf{s}$.

For n = 16 in 2D, the matrices A and C are shown in figures 8 and 9 respectively. The figures show the ordering and the number of nonzeros before and after the reordering scheme. It should be noted that the number of nonzeros of the Cholesky decomposition decreased from 3453 to 1867. The decrease of number of nonzeros is 46%, which is significant. For the three dimensional case shows the same behaviour and is therefore not shown.



(a) Before reordering (b) After reordering

Figure 8: Ordering matrix A before and after reordering scheme



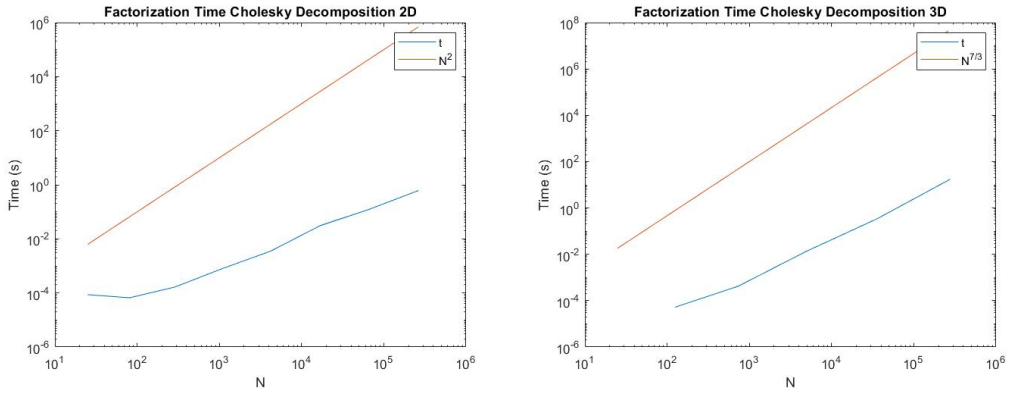
(a) Before reordering (b) After reordering

Figure 9: Ordering matrix C before and after applied reordering scheme on A

3.5.1 CPU time factorization and solving with matrix reordering

Now the tests from before are repeated to show the CPU time required for the Cholesky factorization and the forward/backward solving using the reordering scheme on matrix A. In figure 10 the time needed for the Cholesky decomposition is shown as a function of the problem size N for both 2D and 3D. From the figures

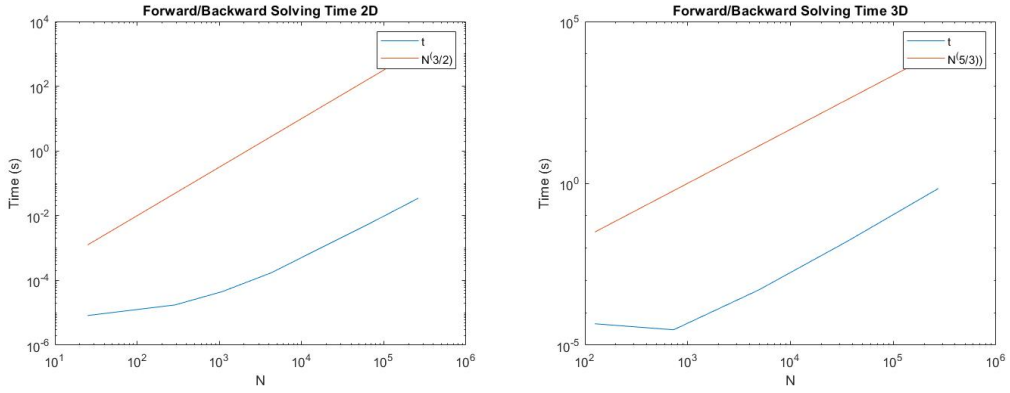
it can be noticed that the factorization time after reordering increases slightly less fast with an increase of N than without ordering.



(a) 2D (b) 3D

Figure 10: Factorization time Cholesky decomposition after matrix reordering

In the forward backward solving time a large improvement can be seen. The total CPU time decreased with more than factor 10 in 2D. Further the solving time seems to increase less fast with an increase of N for large N, which is positive. It can be concluded that the reordering of the matrix has a positive influence on the CPU time for both factorization and the forward/backward solving.

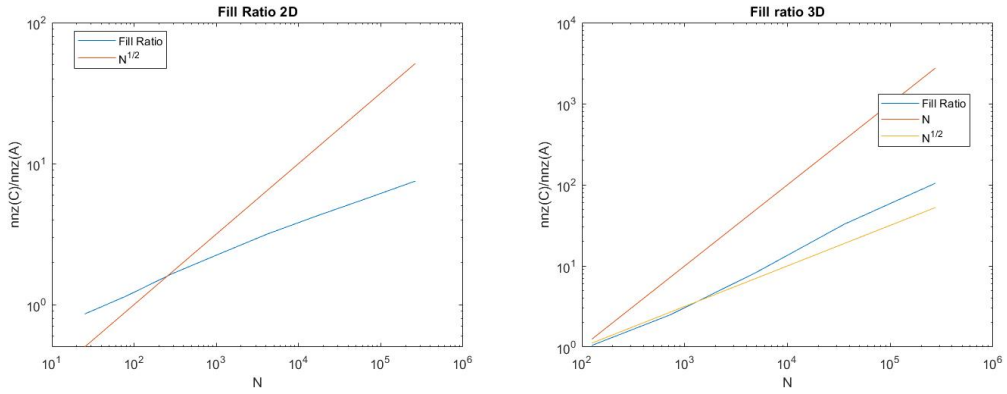


(a) 2D (b) 3D

Figure 11: Forward backward solving time with matrix reordering

3.5.2 Fill-in ratio with matrix reordering

After reordering the fill in ratio is much smaller than before in 2D it is growing much less than $\mathcal{O}(N^{1/2})$ and in 3D it has reduced to about $\mathcal{O}(N^{1/2})$ as can be seen in figure 12.



(a) 2D (b) 3D

Figure 12: Fill-in ratio Cholesky decomposition after reordering in 2D (a) and 3D (b)

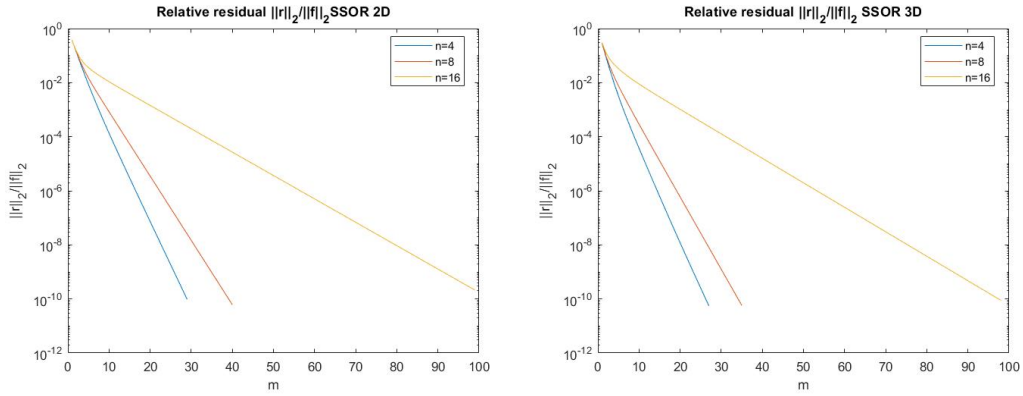
3.6 Symmetric Successive Overrelaxation

3.6.1 Implementation and solution

In this section we will cover the implementation and results of using SSOR as a BIM to solve the problem. We use value $\omega = 1.5$ as the relaxation coefficient. In algorithm 8 the implementation of SSOR can be seen. To show the performance of the solver a logarithmic plot of $\frac{||\mathbf{r}||_2}{||\mathbf{f}||_2}$ against the number of iterations m is shown in figure 13.

Algorithm 8 SSOR implementation

procedure SSOR(A,**f**) ▷ solving Au = f
    omega = 1.5
    while $m < m_{max}$ & $||\mathbf{r}||_2/||\mathbf{f}||_2 > 10^{-10}$ **do**
        for i=1:length(**u**) **do**
            σ = **u**(i)
            u(i) = 0
            u(i) = (**f**(i)-A(i,:)**u**)/A(i,i)
            u(i) = (1-ω)$\sigma + \omega$**u**(i)
        end for
        for i=length(**u**):1 **do**
            σ = **u**(i)
            u(i) = 0;
            u(i) = (**f**(i)-A(i,:)**u**)/A(i,i)
            u(i) = $(1-\omega)\sigma + \omega$**u**($i$);
        end for
        r = **f** - A*__u__;
    end while
end procedure



(a) 2D (b) 3D

Figure 13: Relative residual $\frac{||\mathbf{r}||_2}{||\mathbf{f}||_2}$ of SSOR solver.

From the figures it can be seen that a larger number of elements increases the number of iterations necessary to reach a certain accuracy. Further it can be seen that the after a few iterations the convergence speed is constant. Every iteration the relative residual is multiplied by some consant smaller than one.

3.6.2 Asymptotic rate of convergence

Next the asymptotic rate of convergence of the SSOR method is inspected. In tables 1 and 2 the residual reduction factor $\frac{||\mathbf{r}_m||_2}{||\mathbf{r}_{m-1}||_2}$ for the last five iterations is shown for three different values of n. In the table the total number of iterations until convergence is shown as well. It can be seen that the residual reduction factors are approximately equal for the last five iterations. It can also be seen that with more iterations the resiudal reduction factor becomes more 'constant'. Whereas for $n = 4, m = 29$ there are still slight increases in the 4th decimal, for $n = 8$ the changes are in the 5th decimal and for $n = 16$ the changes is past the 5th decimal.

Further it can be seen that for the small number of elements the convergence is still very fast at the end as the residual is more than halved with each iteration. For larger n the rate of convergence slows down.

n	#iterations m	m -4	m-3	m-2	m-1	m
4	29	0.47869	0.47891	0.47912	0.47931	0.47950
8	40	0.57794	0.57793	0.57791	0.57790	0.57789
16	99	0.81947	0.81947	0.81947	0.81947	0.81947

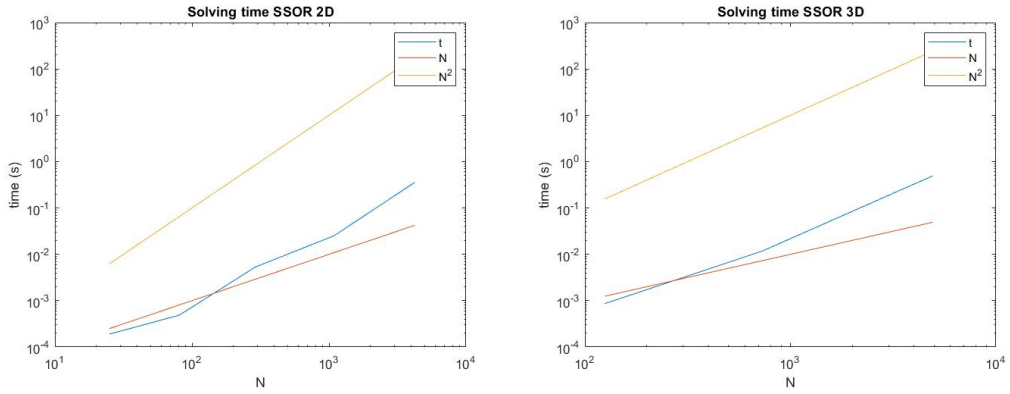
Table 1: Residual reduction factor 2D for the last 5 iterations

n	#iterations m	m -4	m-3	m-2	m-1	m
4	27	0.46262	0.46413	0.46543	0.46655	0.46750
8	35	0.54022	0.54020	0.54018	0.54016	0.54014
16	98	0.81182	0.81182	0.81182	0.81182	0.81182

Table 2: Residual reduction factor 3D for the last 5 iterations

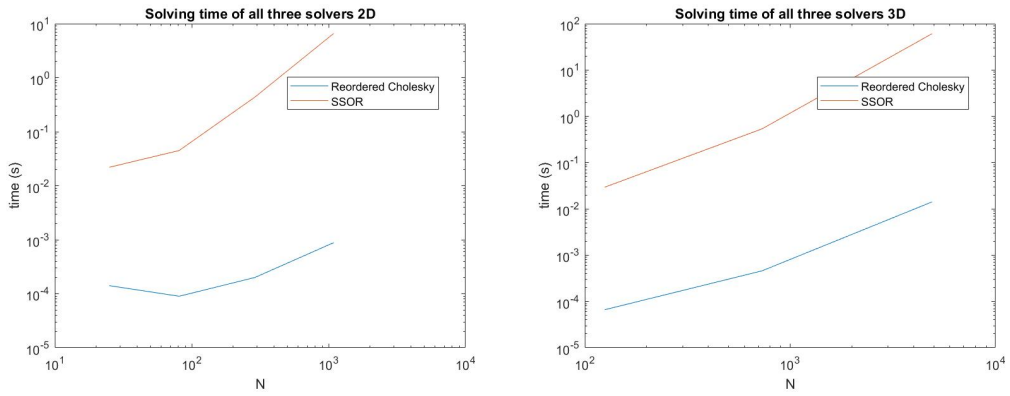
3.6.3 CPU time

The CPU times for one iteration of the SSOR solver are shown in figure 14 and compared with the theoretical estimates found in 2.6. We see that the efficiency of $\mathcal{O}(N)$ is not met, but instead one iteration takes about $\mathcal{O}(N^2)$ flops.



(a) 2D (b) 3D

Figure 14: CPU time of one iteration of the SSOR solver.



(a) 2D (b) 3D

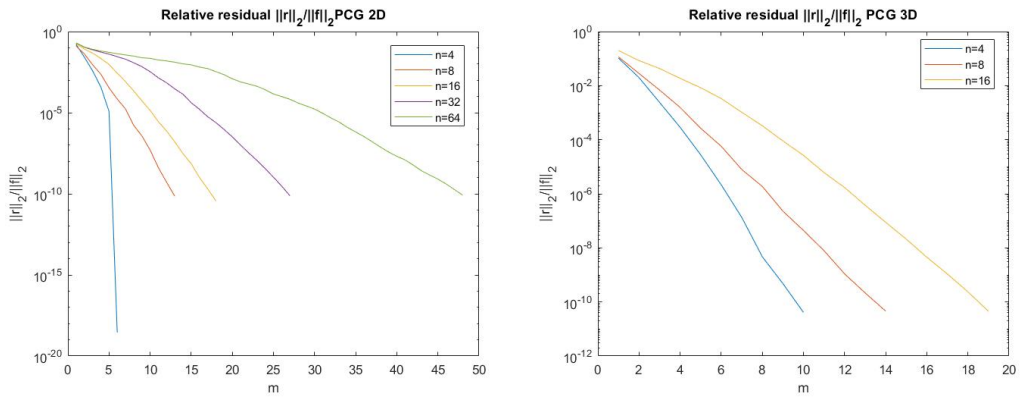
Figure 15: CPU time of SSOR solver untill convergence and Cholesky using a reordered matrix scheme.

As can be seen in figure 15 solving the problem with SSOR untill convergence takes much longer than with the direct solver Cholesky.

3.7 Preconditioned CG

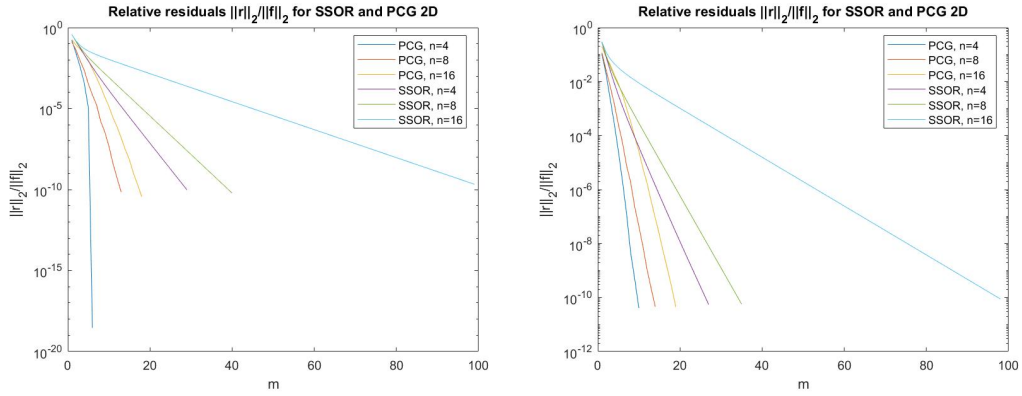
The last solver used is the Preconditioned Conjugate Gradient method with SSOR as a preconditioner.

3.7.1 Implementation and solution



(a) 2D (b) 3D

Figure 16: Relative residuals for different values of n versus the iteration m using PCG in 2D (a) and 3D (b).



(a) 2D (b) 3D

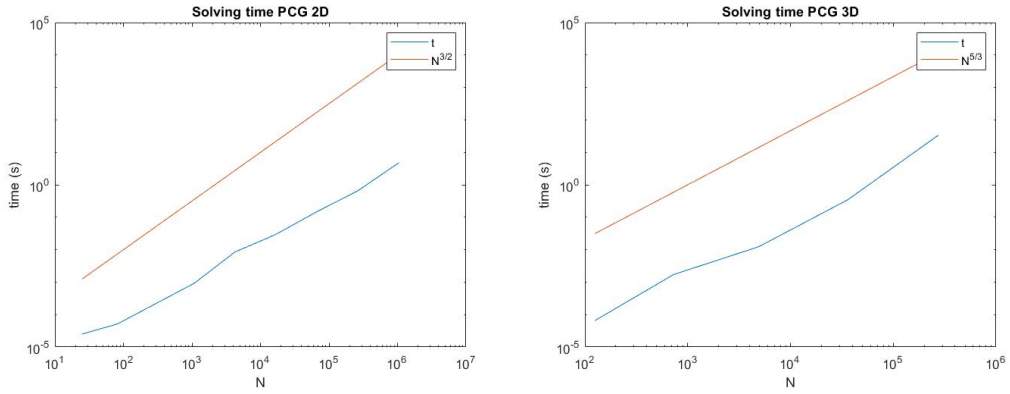
Figure 17: Relative residuals for different values of n versus the iteration m using PCG in 2D (a) and 3D (b).

In Figure 16 the relative residuals $\frac{||\mathbf{r}_m||_2}{||\mathbf{f}||_2}$ plotted against the iteration step. It can be seen that the solver converges in less iterations for a smaller number of n. It seems that the convergence rate keeps improving, since the slopes of the lines are steeper for larger number of m.

In Figure 17 the relative residuals for both the SSOR and PCG solver are plotted. It can be seen that the PCG method converges in much less iterations.

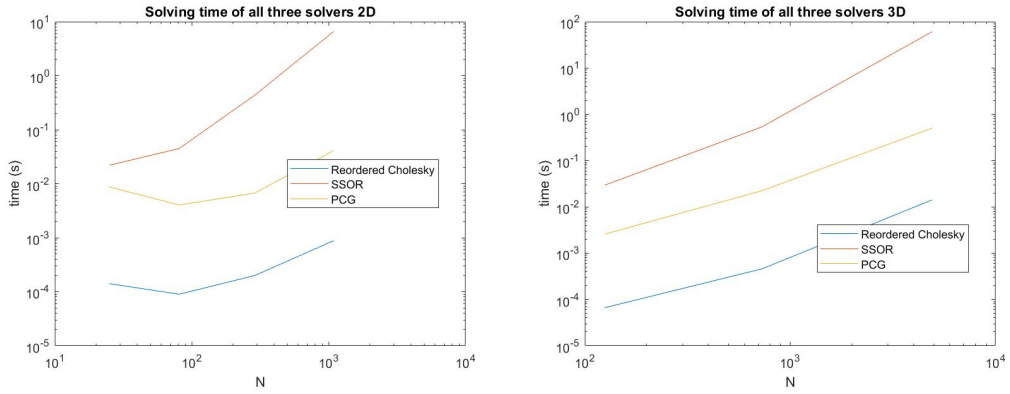
3.7.2 CPU time

In Figure 18 the CPU times of one iteration of the PCG algorithm are plotted as a function of the problem size N. It can be seen that the these increase in the same speed as we calculated in Eq. (6) since the slopes of the two functions are approximately the same. In 2D PCG might be faster than expected because we used the \-operator for line 3 of Algorithm 6.



(a) 2D (b) 3D

Figure 18: CPU time (t, blue) of one iteration of PCG in 2D (a) and 3D (b) and the expected order of increasement as function of N (red).



(a) 2D (b) 3D

Figure 19: CPU time of Cholesky using reordering, SSOR and PCG untill convergence for 2D (a) and 3D (b).

In figure 19 all CPU times for the different methods are plotted. We can observe the Cholesky using a reordered matrix scheme is the fastest method and PCG is much faster than SSOR. Not only do the computation times per iteration grow slower in terms of N, but the method also needs less iterations to converge than SSOR.

4 Matlab Code

```
%% Check order of convergence for
clear;
close all;
%%
%SolveProblem(p,dimension,iter,solver,reduction scheme,m_max)
%%
solver = 'SSOR';                    % Options: 'Cholesky','SSOR','PCG'
m_max = 200;
```

```
p2 = 2:1:6;                              %
n2 = 2.^p2;                              %
N2 = (n2+ones(size(n2))).^2;             %
h2 = 1./n2;                              %
%err2D = ones(size(p2));                 % Maxnorm error
tF2 = ones(size(p2));                    % Factorization time
tS2 = ones(size(p2));                    % Solving time
%fill_ratio2 = ones(size(p2));           % fill in ratio
resid2 = ones(size(p2,1),m_max);         % residual SSOR
rrf2 = ones(size(p2,1),5);               % residual reduction factor

for p = p2
    p
    [u2, u_ex2, err2D(p-1),tF2(p-1),tS2(p-1), fill_ratio2(p-1),resid2(p-1,:),
        rrf2(p-1,:),M2(p-1)] = SolveProblem(p,2,3,solver,0,m_max);
end


p3 = 2:1:4;                              %
n3 = 2.^p3;                              %
N3 = (n3 + ones(size(n3))).^3;
h3 = 1./n3;
err3D = ones(size(p3));                  % Maxnorm error
tF3 = ones(size(p3));                    % Factorization time
tS3 = ones(size(p3));                    % Solving time
fill_ratio3 = ones(size(p3));            % fill in ratio
resid3 = ones(size(p3,1),m_max);         % residual SSOR
rrf3 = ones(size(p3,1),5);               % residual reduction factor

for p= p3
    p
    [u3, u_ex3, err3D(p-1),tF3(p-1),tS3(p-1),fill_ratio3(p-1), resid3(p-1,:),
        rrf3(p-1,:),M3(p-1)] = SolveProblem(p,3,3,solver,0,m_max);
end


%%
if strcmp(solver,'Cholesky')
    figure;
    plot(h2,err2D);
    title('Maxnorm |u^h - u^h_{ex}| for 2D')
    xlabel('h')
    ylabel('maxnorm')
    set(gca, 'XScale','log')
    set(gca, 'YScale', 'log')
    hold on;
    plot(h2, h2.^2);
    legend('|e|_2','h^2')
    hold off;

    figure;
    set(gca, 'YScale', 'log')
    plot(h3,err3D);
    title('Maxnorm |u^h - u^h_{ex}| for 3D')
```

```
61      xlabel('h')
62      ylabel('maxnorm')
63      set(gca, 'XScale','log')
64      set(gca, 'YScale', 'log')
65      hold on;
66      plot(h3, h3.^2);
67      legend('|e|_2','h^2')
68      hold off;
69
70      %% Factorization and solving time as function of problem size
71      figure;
72      plot(N2,tF2)
73      title('Factorization Time Cholesky Decomposition 2D')
74      xlabel('N')
75      ylabel('Time (s)')
76      set(gca, 'XScale','log')
77      set(gca, 'YScale', 'log')
78      hold on;
79      plot(N2, 10^-5*N2.^(2));
80      legend('t','N^{2}')
81      hold off;
82
83      figure;
84      plot(N3,tF3)
85      title('Factorization Time Cholesky Decomposition 3D')
86      xlabel('N')
87      ylabel('Time (s)')
88      set(gca, 'XScale','log')
89      set(gca, 'YScale', 'log')
90      hold on;
91      plot(N2, 10^-5*N2.^(7/3));
92      legend('t','N^{7/3}')
93      hold off;
94
95      figure;
96      plot(N2,tS2)
97      title('Forward/Backward Solving Time 2D')
98      xlabel('N')
99      ylabel('Time (s)')
100     set(gca, 'XScale','log')
101     set(gca, 'YScale', 'log')
102     hold on;
103     plot(N2, 10^-5*N2.^(3/2));
104     legend('t','N^(3/2)')
105     hold off;
106
107     figure;
108     plot(N3,tS3)
109     title('Forward/Backward Solving Time 3D')
110     xlabel('N')
111     ylabel('Time (s)')
112     set(gca, 'XScale','log')
113     set(gca, 'YScale', 'log')
114     hold on;
```

```
115         plot(N3, 10^-5*N3.^(5/3));
116         legend('t','N^(5/3)')
117         hold off;
118         %% Fill ratio analysis
119         figure;
120         plot(N2,fill_ratio2)
121         title('Fill ratio 2D')
122         xlabel('N')
123         ylabel('nnz(C)/nnz(A)')
124         set(gca, 'XScale','log')
125 
126         figure;
127         plot(N3,fill_ratio3)
128         title('Fill ratio 3D')
129         xlabel('N')
130         ylabel('nnz(C)/nnz(A)')
131         set(gca, 'XScale','log')
132 
133     end
134     %%
135     figure;
136     plot(N2,tS2);
137     title(['Solving time ',solver,' 2D'])
138     xlabel('N')
139     ylabel('time (s)')
140     set(gca, 'XScale','log')
141     set(gca, 'YScale', 'log')
142     hold on;
143     %plot(N2, 10^-5*N2.^2);
144     if strcmp(solver,'Cholesky')
145         plot(N2,10^-5*N2.^(3/2));
146         legend('t','N^{3/2}')
147     elseif strcmp(solver, 'SSOR')
148         plot(N2,10^-5*N2*M2)
149         %plot(N2,10^-5*N2.^2)
150         legend('t','N*N_{iter}')
151         %legend('t','N','N^2')
152     elseif strcmp(solver,'PCG')
153         plot(N2,10^-5*N2.^(3/2).*M2);
154         legend('t','N^{3/2}*N_{iter}')
155     end
156     hold off;
157 
158     figure;
159     plot(N3,tS3);
160     title(['Solving time ',solver,' 3D'])
161     xlabel('N')
162     ylabel('time (s)')
163     set(gca, 'XScale','log')
164     set(gca, 'YScale', 'log')
165     hold on;
166     %plot(N3, 10^-5*N3.^2);
167     %plot(N3,10^-5*N3.^(5/3));
168     if strcmp(solver,'Cholesky')
```

```
    plot(N3,10^-5*N3.^(5/3));
    legend('t','N^{5/3}')
elseif strcmp(solver, 'SSOR')
    plot(N3,10^-5*N3*M3);
    %plot(N3,10^-5*N3.^2);
    legend('t','N*N_{iter}')
    %legend('t','N','N^2')
elseif strcmp(solver,'PCG')
    plot(N3,10^-5*N3.^(5/3).*M3);
    legend('t','N^{5/3}*N_{iter}')
end

hold off;

%% Plot relative residuals SSOR
if strcmp(solver,'SSOR') || strcmp(solver,'PCG')
    figure;
    plot(resid2');
    set(gca, 'YScale', 'log')
    title(['Relative residual ||r||_2/||f||_2', solver,' 2D'])
    for i=1:length(n2)
        legendn{i} = sprintf('n=%s',num2str(n2(i)));
    end
    legend(legendn,'Location','best')
    xlabel('m')
    ylabel('||r||_2/||f||_2')
    %%
    %
    figure;
    plot(resid3');
    set(gca, 'YScale', 'log')
    title(['Relative residual ||r||_2/||f||_2 ',solver,' 3D'])
    for i=1:length(n3)
        legendn{i} = sprintf('n=%s',num2str(n3(i)));
    end
    legend(legendn,'Location','best')
    xlabel('m')
    ylabel('||r||_2/||f||_2')
end

%latex_table = latex(vpa(sym(rrf2),3))
```

```
%% Scientific Computing
% Group assignment for course Scientific Computing
% By: Nerine Usman & Marianne Schaaphok
% Date: 25-11-2018

function [u,u_ex,err,tF,tS, fill_ratio, resid,rrf,m] = SolveProblem(p,
    dimension,iter, solver,redsc,m_max)
%% Parameters

%dimension = 3;
tol = 1e-4;
maxnorm = size(1,9);
```

```
%p = 2;
n = 2^p;
h = 1/n;

%% Construct matrices A

% Help matrices for construction of A
H_1 = spdiags([h^2; zeros(n-1,1); h^2],0,n+1,n+1);
D_1 = spdiags([0; ones(n-1,1);0], 0, n+1,n+1);
D_2 = kron(D_1,D_1);
if dimension == 3
    D_3 = kron(D_1,D_2);
end
T_1 = spdiags(-1*[0;ones(n-2,1); 0; 0],-1,n+1,n+1) ...
    + spdiags(-1*[0;0;ones(n-2,1);0],1,n+1,n+1);
I_1 = spdiags(ones((n+1),1),0,(n+1),(n+1));
I_2 = spdiags(ones((n+1)^2,1),0,(n+1)^2,(n+1)^2);


% Boundary neighbours of interior points
B_1 = sparse(zeros(n+1));
B_1(2,1) = 1;
B_1(n,n+1) = 1;
B_2 = kron(D_1,B_1) + kron(B_1,D_1);

if dimension == 3
    B_3 = kron(D_1,B_2) + kron(B_1,D_2);
end

% 1D matrix A
A_1 = H_1+2*D_1 + T_1;

% 2D matrix A
A_2 = kron(H_1,I_1) + kron(D_1,A_1+2*D_1) + kron(T_1,D_1);

% 3D matrix A
if dimension == 3
    A_3 = kron(H_1,I_2) + kron(D_1,A_2+2*kron(D_1,D_1)) + kron(T_1,D_2);
end

A_1 = 1/h^2 * A_1;
A_2 = 1/h^2 * A_2;
if dimension == 3
    A_3 = 1/h^2 * A_3;
end
%imagesc(A_2)

%% Construct vector f 2D
if (dimension == 2)
    % Construct mesh
    x = 0:h:1;
    y = 0:h:1;
    [X,Y] = meshgrid(x,y);

```

```
66
67      % Construct f for internal points 2D problem
68      F2 = f2(X,Y);
69      F2 = reshape(F2,[(n+1)^2,1]);
70      f_int2 = D_2*F2;
71
72      % Add f for boundary points
73      U2 = exact(X,Y,1);
74      U2 = reshape(U2, [(n+1)^2,1]);
75      f_boun2 = (I_2-D_2)*U2;
76
77      % Correct f for symmetric A
78      f_nb2 = B_2*(U2/h^2);
79
80      f_2 = f_int2 + f_boun2 + f_nb2;
81
82      A = A_2;
83      f = f_2;
84      u_ex = U2;
85  %% Construct vector f 3D
86  elseif (dimension == 3)
87      I_3 = spdiags(ones((n+1)^3,1),0,(n+1)^3,(n+1)^3);
88
89      % Construct mesh
90      x = 0:h:1;
91      y = 0:h:1;
92      z = 0:h:1;
93      [X,Y,Z] = meshgrid(x,y,z);
94
95
96      % Construct f for internal points 2D problem
97      F3 = f3(X,Y,Z);
98      F3 = reshape(F3,[(n+1)^3,1]);
99      f_int3 = D_3*F3;
100
101     % Add f for boundary points
102     U3 = exact(X,Y,Z);
103     U3 = reshape(U3, [(n+1)^3,1]);
104     f_boun3 = (I_3-D_3)*U3;
105
106     % Correct f for symmetric A
107     f_nb3 = B_3*(U3/h^2);
108
109     f_3 = f_int3 + f_boun3 + f_nb3;
110
111     A = A_3;
112     f = f_3;
113     u_ex = U3;
114 else
115     fprintf('Please choose dimension 2 or 3')
116 end
117
118 %% Solve the system
119 % Use of bandwith reduction scheme
```

```
120 if redsc == 1
121       s = symamd(A);
122 %       figure;
123 %       spy(A);
124 %       title('A before matrix reordering')
125       A = A(s,s);
126 %       figure
127 %       spy(A)
128 %       title('A after matrix reordering')
129       f = f(s);
130 end
131
132  % Define variables
133  R = sparse(size(A));
134  %R = zeros(size(A));
135  u = zeros(size(f));
136  resid = zeros(m_max,1);
137  normR = zeros(m_max,1);
138  rrf = zeros(5,1);
139  tF = 0;
140  tS = 0;
141  r = 1;
142  nf = norm(f);
143 m=1;
144
145  % Use given solver
146  if strcmp(solver,'Cholesky')
147     tic;
148     R = chol(A,'lower');
149     tF = toc;
150
151 %      Cholesky Decomposition
152 %      for k = 1:size(A,2)
153 %          j = max(k-(n+1),1);
154 %          A(k,k) = sqrt(A(k,k) - A(k,j:k-1)*A(k,j:k-1)');
155 %          for i = k+1:min(k+n+1,size(A,2))
156 %              l = max(i-(n+1),1);
157 %             A(i,k) = 1/(A(k,k))*(A(i,k) - A(i, l:k-1)*A(k,l:k-1)');
158 %          end
159 %      end
160 %      R = tril(A);
161
162
163     % Direct solving algorithm
164     tic;
165     y = R\f;
166     u = R'\y;
167     tS = toc;
168
169     if redsc == 1
170     u(s) = u;
171     end
172
173 elseif strcmp(solver,'SSOR')
```

```
174         omega = 1.5;
175         resid(1) = 1;
176         u = zeros(size(u_ex));
177         tic;
178         while m<m_max && norm(r)/nf>10^-10
179             %tic;
180             for i = 1:length(u)
181                 sigma = u(i);
182                 %u(i) = (f(i) - A(i,1:i-1)*u(1:i-1) - A(i,i+1)*u(i+1,n))/A(i,i);
183                 u(i) = 0;
184                 u(i) = (f(i)-A(i,:)*u)/A(i,i);
185                 u(i) = (1-omega)*sigma + omega*u(i);
186             end
187
188             for i = length(u):-1:1
189                 sigma = u(i);
190                 u(i) = 0;
191                 u(i) = (f(i)-A(i,:)*u)/A(i,i);
192                 u(i) = (1-omega)*sigma + omega*u(i);
193             end
194             %tS = toc;
195             r = f-A*u;
196             normR(m) = norm(r);
197             resid(m) = normR(m)/nf;
198             m=m+1;
199
200         end
201
202         tS = toc;
203         rrf = normR(max(m-5,1):m-1)./normR(max(m-6,1):m-2);
204
205
206     elseif strcmp(solver,'PCG')
207         omega = 1.5;
208         % Initialize
209         u = zeros(size(u_ex));
210
211         % Decompose A
212         D = diag(diag(A));
213         E = D - tril(A);
214         F = E';
215
216         % Preconditioning matrix
217         M = (D-omega*E)*spdiags(1./diag(D),0,size(D,1),size(D,2))*(D-omega*F)/(
                omega*(2-omega));
218
219         % Algorithm
220         tic;
221         r = f;
222
223         while m<=m_max && norm(r)/nf>10^-10
224             z = M\r;
225             if m==1
226                 p = z;
```

```
227             current = r'*z;
228         else
229             current = r'*z;
230             beta = current/prev;
231             p = z + beta*p;
232         end
233         temp = A*p;
234         alpha = current/(p'*temp);
235         u = u + alpha*p;
236         r = r - alpha*temp;
237         prev = current;
238 
239         resid(m) = norm(r)/nf;
240         m = m+1;
241         %tS = toc;
242       end
243       tS = toc;
244 
245   end
246 
247 %   figure
248 %   spy(R)
249 %   title('Cholesky matrix after reordering')
250 %   hold off;
251    fill_ratio = nnz(R)/nnz(A);
252    err = norm(u-u_ex, 'inf');
253 
254 % if dimension==2
255 %       u_pl = reshape(u,[(n+1),(n+1)]);
256 %       u_ex1 = reshape(U2,[n+1,n+1]);
257 %
258 %       figure;
259 %       surf(X,Y,u_pl)
260 %       hold on;
261 %       surf(X,Y,u_ex1)
262 %       hold off;
263 % end
264 
265 
266 %% Functions
267 function [f] = f2(x,y)
268     f = (x.^2 + y.^2).*sin(x.*y);
269 end
270 
271 function [f] = f3(x,y,z)
272     f = ((x.*y).^2 + (y.*z).^2 + (x.*z).^2).*sin(x.*y.*z);
273 end
274 
275 function [ex] = exact(x,y,z)
276     ex = sin(x.*y.*z);
277 end
278 end
```

```
1 clear all
```

```
close all

load('PCG_RelRes.mat')
PCG_resid2 = resid2;
PCG_resid3 = resid3;
load('SSOR_RelRes.mat')
SSOR_resid2 = resid2;
SSOR_resid3 = resid3;

figure;
plot(PCG_resid2');
hold on
plot(SSOR_resid2');
set(gca, 'YScale', 'log')
title(['Relative residuals ||r||_2/||f||_2 for SSOR and PCG 2D'])
for i=1:length(n2)
    legendn{i} = sprintf('PCG, n=%s',num2str(n2(i)));
end
for i=1:length(n2)
    legendn{i+length(n2)} = sprintf('SSOR, n=%s',num2str(n2(i)));
end
legend(legendn,'Location','best')
xlabel('m')
ylabel('||r||_2/||f||_2')


%% 3D %%
figure;
plot(PCG_resid3');
hold on
plot(SSOR_resid3');
set(gca, 'YScale', 'log')
title(['Relative residuals ||r||_2/||f||_2 for SSOR and PCG 2D'])
for i=1:length(n2)
    legendn{i} = sprintf('PCG, n=%s',num2str(n2(i)));
end
for i=1:length(n2)
    legendn{i+length(n2)} = sprintf('SSOR, n=%s',num2str(n2(i)));
end
legend(legendn,'Location','best')
xlabel('m')
ylabel('||r||_2/||f||_2')
```

```
clear all
close all

load('14-01-2019/SSOR_mmax200_3D.mat')
load('14-01-2019/SSOR_mmax200_2D.mat')
SSOR_time_2D = tS2;
SSOR_time_3D = tS3;
N_2 = N2;
N_3 = N3;

load('14-01-2019/Cholesky_red_2D.mat')
```

```
load('14-01-2019/Cholesky_red_3D.mat')
Cholesky_time_2D = tS2(1:size(N_2,2)) + tF2(1:size(N_2,2));
Cholesky_time_3D = tS3(1:size(N_3,2)) + tF3(1:size(N_3,2));

load('14-01-2019/PCG_mmax200_2D.mat')
load('14-01-2019/PCG_mmax200_3D.mat')
PCG_time_2D = tS2(1:size(N_2,2));
PCG_time_3D = tS3(1:size(N_3,2));

figure;
plot(N_2,Cholesky_time_2D);
hold on;
plot(N_2,SSOR_time_2D);
plot(N_2,PCG_time_2D);
title(['Solving time of all three solvers 2D'])
xlabel('N')
ylabel('time (s)')
set(gca, 'XScale','log')
set(gca, 'YScale', 'log')
legend('Reordered Cholesky', 'SSOR', 'PCG', 'location', 'best')
hold off;

figure;
plot(N_3,Cholesky_time_3D);
hold on;
plot(N_3,SSOR_time_3D);
plot(N_3,PCG_time_3D);
title(['Solving time of all three solvers 3D'])
xlabel('N')
ylabel('time (s)')
set(gca, 'XScale','log')
set(gca, 'YScale', 'log')
legend('Reordered Cholesky', 'SSOR', 'PCG', 'location', 'best')
hold off;
```

References

[1] C. Vuik and D. Lahaye, "Course wi4201: Scientific computing," Sep 2017.