Project E: Computer exercise for WI4201 A Comparison of Preconditioned Krylov Subspace Solvers for sequential and parallel problems

GROUP 41
Cihandir Özbek(4448146)
Amadeo Villar Guardia (5377447)

January 15, 2021

1 Problem

The goal of this assignment is to compare the performance of some sparse direct and iterative solution methods applied to the discrete Laplacian in two and three dimensions. In this assignment the Poisson problem on the unit square $\Omega = [0,1]^2$ and the unit cube $\Omega = [0,1]^3$ is considered. 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 $f(\mathbf{x})$

$$-\Delta u = -div \cdot grad(u) = f \; on \; \Omega \tag{1}$$

In this assignment we supply this problem with the non-homogeneous Dirichlet boundary condition:

$$u = u_0 \; on \; \partial\Omega \tag{2}$$

Assume that the computational domain Ω is discretized by a uniform mesh denoted by Ω^h with n elements and meshwidth $h = 1/n$ in each direction. Assume that a central second order accurate finite difference scheme (5-point scheme in 2D and a 7-point scheme in 3D) is applied to discretize the partial differential equation (1) supplied with (2) and assume that the boundaries are not eliminated from the linear system (cfr. Lecture Notes). The discretization then leads to a linear system for the discrete solution u

$$A^h \mathbf{u}^h = \mathbf{f}^h \tag{3}$$

for which coefficient matrix A^h has a number of desirable properties. Let $\mathbf{u}^h_{ex}$ denote the exact solution evaluated in the grid nodes. We will solve this problem with several different solvers. We will first look into Symmetric Successive Overrelaxation (SSOR) as a basic iterative method (BIM). Given a zero initial guess for $\mathbf{u}^h$ and denoting the iteration index as m, solve linear system (3) until convergence criterion

$$\frac{\|\mathbf{r}_m\|}{\|\mathbf{f}^h\|} \leq 10^{-10} \tag{4}$$

is met. Another way is to look into Symmetric SOR as a preconditioner for the conjugate gradient. Lastly, we develop a parallel solver. First of all we divide domain Ω into p subdomains. Suppose that the

number of grid points in the x-direction is given by n_x and in the y-direction by n_y. Domain Ω is then subdivided into p rectangular subdomains Ω_i, $i = 1, \ldots, p$, and each subdomain consists of $\frac{n_x}{n_y}$ gridpoints (where we assume that $\frac{n_x}{p}$ is an integer number). Answer each question in this assignment for both the 2D and 3D problem unless it is stated otherwise.

2 Pen and Paper Assignment

1. *Determine the source function $f(\boldsymbol{x})$ and the boundary data $u_0(\boldsymbol{x})$ such that $u_{ex}(\boldsymbol{x}) = sin(xy)$ in 2D (and $u_{ex}(\boldsymbol{x}) = sin(xyz)$ in 3D) is the exact solution to the problem.*

   - **2D PROBLEM:**

   To determine the source function $f(\mathbf{x})$ we have to introduce our exact solution $u_{ex}(\mathbf{x})$ in the Poisson equation (1). Doing so we obtain:

   $$-\Delta u_{ex}(\mathbf{x}) = -\left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}\right) sin(xy) = \left(x^2 + y^2\right) sin(xy),$$

   and hence:

   $$f(\mathbf{x}) = \left(x^2 + y^2\right) sin(xy). \tag{5}$$

   To determine the boundary data $u_0(\mathbf{x})$ we just have to evaluate the function $u_{ex}(\mathbf{x})$ in $\partial\Omega$. Doing so we obtain:

   $$u_0(\mathbf{x}) = \begin{cases} u_0(0, y) = 0 \\ u_0(1, y) = sin(y) \\ u_0(x, 0) = 0 \\ u_0(x, 1) = sin(x) \end{cases} \tag{6}$$

   - **3D PROBLEM:**

   Proceeding in the same way, we obtain:

   $$-\Delta u_{ex}(\mathbf{x}) = -\left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2}\right) sin(xyz) = \left(x^2y^2 + y^2z^2 + x^2z^2\right) sin(xyz),$$

   and hence:

   $$f(\mathbf{x}) = \left(x^2y^2 + y^2z^2 + x^2z^2\right) sin(xyz). \tag{7}$$

   For the boundary data $u_0(\mathbf{x})$ we obtain:

   $$u_0(\mathbf{x}) = \begin{cases} u_0(0, y, z) = 0 \\ u_0(1, y, z) = sin(yz) \\ u_0(x, 0, z) = 0 \\ u_0(x, 1, z) = sin(xz) \\ u_0(x, y, 0) = 0 \\ u_0(x, y, 1) = sin(xy) \end{cases} \tag{8}$$

2. *Give the size, sparsity structure and upper and lower bandwidth of the matrix A^h for both the 2D and 3D problem.*

To compute an approximate numerical solution of the Poisson equation, we have to discretize our problem. We first analyse the problem in 2D and later we will proceed in 3D.

- **2D PROBLEM:**

The first thing will be to define the grid in which we will work:

$$\Omega^h = \{(x_i, y_j)\ ,\ x_i = (i-1)h\ ,\ y_j = (j-1)h\ ,\ i,j = 1,\dots,n+1\}\,,$$

and we will use the following notation: $u^h_{i,j}$ approximation of $u_{ex}(x_i, y_j)$ and $f^h_{i.j} = f(x_i, y_j)$. Secondly, to discretize the Poisson equation we use the so called Finite difference approximation method. In 2D applying a central second order accurate finite difference scheme, we have:

$$-\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right)\Big|_{i,j} \approx \frac{-u^h_{i-1,j} - u^h_{i+1,j} + 4u^h_{i,j} - u^h_{i,j-1} - u^h_{i,j+1}}{h^2} = f^h_{i,j}\ for\ i,j = 2,\dots,n. \tag{9}$$

We will not eliminate the boundaries form the linear system. For the boundaries we have:

$$u^h_{i,j} = u^h_0(x_i, y_j)\ ,\quad for\ i,j \in \{1, n+1\} \tag{10}$$

To deal with the ordering of $u^h_{i,j}$ in the vector $\mathbf{u}^h$, we use the so-called lexicographic ordering without elimination. This ordering organise the $(n+1)^2$ components $u^h_{i,j}$ according to the order number I:

$$u^h_{i,j} \to u_I\ ,\ I = i + (j-1)(n+1).$$

We consider $A^h \in \mathbb{R}^{(n+1)^2 \times (n+1)^2}$ to be a block matrix and the vectors $\mathbf{u}^h, \mathbf{f}^h \in \mathbb{R}^{(n+1)^2}$ to be block column vectors. The matrix A^h is formed by $(n+1)^2$ square matrices $\in \mathbb{R}^{(n+1)\times(n+1)}$, that is $A^h = \left(A^h_{i,j}\right) i,j = 1,\dots,n+1$.

$$A^h = \begin{bmatrix} A^h_{1,1} & A^h_{1,2} & \cdots & A^h_{1,n+1} \\ A^h_{2,1} & A^h_{2,2} & \cdots & A^h_{2,n+1} \\ \vdots & \vdots & \ddots & \vdots \\ A^h_{n+1,1} & A^h_{n+1,2} & \cdots & A^h_{n+1,n+1} \end{bmatrix} \tag{11}$$

The vector $\mathbf{u}^h$ is defined in terms of $u^h_{i,j}$ and the vector $\mathbf{f}^h$ is defined in terms of $f^h_{i,j}$ for the internal points and $u^h_{0_{i,j}}$ for the contribution of the boundary points, as we will see later. For the sake of simplicity, we consider:

$$\mathbf{u}^h = \begin{bmatrix} u^h_{1,1} \\ u^h_{2,1} \\ \vdots \\ u^h_{n+1,1} \\ u^h_{1,2} \\ u^h_{2,2} \\ \vdots \\ u^h_{n+1,2} \\ \vdots \\ u^h_{1,n+1} \\ u^h_{2,n+1} \\ \vdots \\ u^h_{n+1,n+1} \end{bmatrix} \in \mathbb{R}^{(n+1)^2}, \quad \mathbf{f}^h = \begin{bmatrix} f^h_{1,1} \\ f^h_{2,1} \\ \vdots \\ f^h_{n+1,1} \\ f^h_{1,2} \\ f^h_{2,2} \\ \vdots \\ f^h_{n+1,2} \\ \vdots \\ f^h_{1,n+1} \\ f^h_{2,n+1} \\ \vdots \\ f^h_{n+1,n+1} \end{bmatrix} \in \mathbb{R}^{(n+1)^2}$$

As we said before, these vectors can be expressed as block column vectors.

The vector $\mathbf{u}^h$ is formed by $n+1$ vectors of length $n+1$:

$$\mathbf{u}^h = \begin{bmatrix} (\mathbf{u}^h_1)^t & (\mathbf{u}^h_2)^t & \cdots & (\mathbf{u}^h_{n+1})^t \end{bmatrix}^t, \tag{12}$$

where $\mathbf{u}^h_j = [u_{1,j}, \ldots, u_{n+1,j}]^t$, or using the lexicographic order: $\mathbf{u}^h_j = [u_{1+(j-1)(n+1)}, \ldots, u_{j(n+1)}]^t$.

The vector $\mathbf{f}^h$ is formed by $n+1$ vectors of length $n+1$:

$$\mathbf{f}^h = \begin{bmatrix} (\mathbf{f}^h_1)^t & (\mathbf{f}^h_2)^t & \cdots & (\mathbf{f}^h_{n+1})^t \end{bmatrix}^t \tag{13}$$

where the values of $\mathbf{f}^h_j$ are unknowns that we will solve using (9) and (10).

Using the above block elements, we can express (3) as:

$$\begin{bmatrix} A^h_{1,1} & A^h_{1,2} & \cdots & A^h_{1,n+1} \\ A^h_{2,1} & A^h_{2,2} & \cdots & A^h_{2,n+1} \\ \vdots & \vdots & \ddots & \vdots \\ A^h_{n+1,1} & A^h_{n+1,2} & \cdots & A^h_{n+1,n+1} \end{bmatrix} \begin{bmatrix} \mathbf{u}^h_1 \\ \mathbf{u}^h_2 \\ \vdots \\ \mathbf{u}^h_{n+1} \end{bmatrix} = \begin{bmatrix} \mathbf{f}^h_1 \\ \mathbf{f}^h_2 \\ \vdots \\ \mathbf{f}^h_{n+1} \end{bmatrix} \tag{14}$$

Now we proceed to study the elements of (14).

- To introduce the boundary conditions for $j = 1$, that is the components of $\mathbf{u}^h_1$, we use the first row of A^h. This condition is $\mathbf{u}^h_1 = \mathbf{0}$, as $u^h_{0_{i,1}} = sin(x_i y_1) = sin(0) = 0$. So we take: $A^h_{1,1} = \mathbb{1}_{n+1}$, $A^h_{1,j} = 0$ for $j = 2, \ldots, n+1$ and $\mathbf{f}^h_1 = \mathbf{0}$.
- To introduce the boundary conditions for $j = n+1$, that is the components of $\mathbf{u}^h_{n+1}$, we use the last row of A^h. This condition is $\mathbf{u}^h_{n+1} = sin(\mathbf{x})$, as $u^h_{0_{i,n+1}} = sin(x_i y_{n+1}) = sin(x_i)$. So we take: $A^h_{n+1,n+1} = \mathbb{1}_{n+1}$, $A^h_{n+1,j} = 0$ for $j = 1, \ldots, n$ and $\mathbf{f}^h_{n+1} = sin(\mathbf{x})$.
- For the rest of the vectors $\mathbf{u}^h_2$,...,$\mathbf{u}^h_n$, we proceed as follows:

The blocks $A^h_{2,2}, \ldots, A^h_{n,n}$ in the main diagonal would have the following form:

$$A^h_{j,j} := \hat{T}^h = \frac{1}{h^2}\begin{bmatrix} h^2 & 0 & \ldots & \ldots & \ldots & 0 \\ -1 & 4 & -1 & 0 & \ldots & 0 \\ 0 & \ddots & \ddots & \ddots & \ddots & 0 \\ 0 & \ddots & \ddots & \ddots & \ddots & 0 \\ 0 & \ddots & \ddots & -1 & 4 & -1 \\ 0 & \ldots & \ldots & \ldots & 0 & h^2 \end{bmatrix} \in \mathbb{R}^{(n+1)\times(n+1)} \quad for\ j = 2, \ldots, n \quad (15)$$

Notice that respectively, the first element and the last element of the main diagonal are used to introduce the boundary conditions for $i = 1$ and for $i = n + 1$, so the remaining elements of the first and last row are 0. Hence the first and last elements of $\mathbf{f}^h_j$ will be respectively: $u^h_{0_{1,j}} = sin(x_1 y_j) = sin(0) = 0$ and $u^h_{0_{n+1,j}} = sin(x_{n+1} y_j) = sin(y_j)$.
The rest of the entries are used to specify the interior points. Given an interior point $u^h_{i,j}$, and looking at (9), the only elements with the same value as j (and hence in the same vector $\mathbf{u}^h_j$) that make a contribution are $u^h_{i-1,j}$ and $u^h_{i+1,j}$. That is the reason why we have taken $A^h_{j,j}$ as a tridiagonal matrix, with that specific values. Furthermore the interior points of the vector $\mathbf{f}^h_j$ will be given by $f^h_{2,j}, \ldots, f^h_{n,j}$.
This matrix can be made symmetric by translating the connections of the left most and right most interior points to the left and right boundary point into contributions to the right-hand side vector to obtain:

$$A^h_{j,j} := T^h = \frac{1}{h^2}\begin{bmatrix} h^2 & 0 & \ldots & \ldots & \ldots & 0 \\ 0 & 4 & -1 & 0 & \ldots & 0 \\ 0 & \ddots & \ddots & \ddots & \ddots & 0 \\ 0 & \ddots & \ddots & \ddots & \ddots & 0 \\ 0 & \ddots & \ddots & -1 & 4 & 0 \\ 0 & \ldots & \ldots & \ldots & 0 & h^2 \end{bmatrix} \in \mathbb{R}^{(n+1)\times(n+1)} \quad for\ j = 2, \ldots, n \quad (16)$$

The vector $\mathbf{f}^h_j$ will remain unchanged, except for elements corresponding with $i = 2$ and $i = n$, that we must take into account the contributions of $u^h_{0_{1,j}}$ and of $u^h_{0_{n+1,j}}$, respectively. So these points take the values: $f_{2,j} + \frac{1}{h^2} u^h_{0_{1,j}}$ and $f_{n,j} + \frac{1}{h^2} u^h_{0_{n+1,j}}$.
Looking back at equation (9), we also have to take into account the contributions of the terms $u^h_{i,j-1}$ and $u^h_{i,j+1}$. These terms belongs to the vectors $\mathbf{u}^h_{j-1}$ and $\mathbf{u}^h_{j+1}$. The remaining vectors make no contributions, so therefore only the block matrices $A_{j,j-1}$ and $A_{j,j+1}$ are non-zero. These matrices are of the form:

$$A^h_{j,j-1} = A^h_{j,j+1} = \hat{I}^h = \frac{1}{h^2}\begin{bmatrix} 0 & 0 & 0 \\ 0 & I_{n-1} & 0 \\ 0 & 0 & 0 \end{bmatrix} \in \mathbb{R}^{(n+1)\times(n+1)} \quad (17)$$

where I_{n-1} is the identity matrix $in\ \mathbb{R}^{(n+1)\times(n+1)}$.
Notice that first and last element of the diagonal are 0 because we are only considering the interior points. Moreover to make the matrix A symmetric, we can use the previous procedure. We can take $A^h_{2,1} = 0$ and $A^h_{n,n+1} = 0$, replacing the previous vector $\mathbf{f}^h_2$ with $\mathbf{f}^h_2 + \frac{1}{h^2}\mathbf{u}^h_{0_1}$ and the previous vector $\mathbf{f}^h_n$ with $\mathbf{f}^h_n + \frac{1}{h^2}\mathbf{u}^h_{0_{n+1}}$.

Therefore we can express A^h as:

$$A^h = \begin{bmatrix} I_{n+1} & 0 & \dots & \dots & \dots & \dots & 0 \\ 0 & T^h & -\hat{I}^h & 0 & \dots & \dots & 0 \\ 0 & -\hat{I}^h & T^h & -\hat{I}^h & \dots & \dots & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \dots & \vdots \\ \vdots & \vdots & \ddots & \ddots & \ddots & -\hat{I}^h & \vdots \\ 0 & \dots & 0 & \dots & -\hat{I}^h & T^h & 0 \\ 0 & \dots & \dots & \dots & 0 & 0 & I_{n+1} \end{bmatrix} \in \mathbb{R}^{(n+1)^2 \times (n+1)^2} \tag{18}$$

This matrix A^h is symmetric and positive definite (applying Gershgorin theorem we see that symmetric diagonal dominant matrices are SPD). Therefore it has good properties, that will help us in the following sections and to solve $A^h \mathbf{u}^h = \mathbf{f}^h$.

Furthermore, the Bandwidth of a matrix A is defined as the smallest non-negative integer K such that $A(i,j) = 0\ ,\ for\ |i-j| > K$. The upper bandwidth is $n+1 \rightarrow$ distance between $u^h_{i,j}$ and $u^h_{i,j+1}$ using lexicographic order. The lower bandwidth is $n+1 \rightarrow$ distance between $u^h_{i,j-1}$ and $u^h_{i,j}$ using lexicographic order.

The vector $\mathbf{f}^h$ can be expressed as:

$$\mathbf{f}^h = \begin{bmatrix} u^h_{0_{1,1}} \\ \vdots \\ u^h_{0_{n+1,1}} \\ u^h_{0_{1,2}} \\ f^h_{2,2} + \frac{1}{h^2} u^h_{0_{1,2}} + \frac{1}{h^2} u^h_{0_{2,1}} \\ f^h_{3,2} + \frac{1}{h^2} u^h_{0_{3,1}} \\ \vdots \\ f^h_{n-1,2} + \frac{1}{h^2} u^h_{0_{n-1,1}} \\ f^h_{n,2} + \frac{1}{h^2} u^h_{0_{n,1}} + \frac{1}{h^2} u^h_{0_{n+1,2}} \\ u^h_{0_{n+1,2}} \\ u^h_{0_{1,3}} \\ f^h_{2,3} + \frac{1}{h^2} u^h_{0_{1,3}} \\ f^h_{3,3} \\ \vdots \\ u^h_{0_{1,n+1}} \\ \vdots \\ u^h_{0_{n+1,n+1}} \end{bmatrix} \in \mathbb{R}^{(n+1)^2} \tag{19}$$

where $f^h_{i,j}$ are given by (5) and $u^h_{0_{i,j}}$ are given by (6).

- **3D PROBLEM:**

The grid is the same as we used in the 2D case, but we incorporate a third component (x_i, y_j, z_k), with $z_k = (k-1)h$, $h = 1, \ldots, n+1$. Using the central second accurate finite difference scheme for 3D case, we have:

$$u''_{i,j,k} \approx \frac{-u_{i-1,j,k} - u_{i,j-1,k} - u_{i,j,k-1} + 6u_{i,j,k} - u_{i+1,j,k} - u_{i,j+1,k} - u_{i,j,k+1}}{h^2} = f_{i,j,k}\ ,\ i,j,k = 2, \ldots, n. \tag{20}$$

Following the same procedure that we did in the 2D case we have:

We define the matrices:

$$T^h = \frac{1}{h^2}\begin{bmatrix} h^2 & 0 & \ldots & \ldots & \ldots & 0 \\ 0 & 6 & -1 & 0 & \ldots & 0 \\ 0 & \ddots & \ddots & \ddots & \ddots & 0 \\ 0 & \ddots & \ddots & -1 & 6 & 0 \\ 0 & \ldots & \ldots & \ldots & 0 & h^2 \end{bmatrix} \in \mathbb{R}^{(n+1)\times(n+1)}\ ,\ \hat{I}^h = \frac{1}{h^2}\begin{bmatrix} 0 & 0 & 0 \\ 0 & I_{n-1} & 0 \\ 0 & 0 & 0 \end{bmatrix} \in \mathbb{R}^{(n+1)\times(n+1)} \tag{21}$$

$$S^h = \begin{bmatrix} I_{n+1} & 0 & \ldots & \ldots & \ldots & \ldots & 0 \\ 0 & T^h & -\hat{I}^h & 0 & \ldots & \ldots & 0 \\ 0 & -\hat{I}^h & T^h & -\hat{I}^h & \ldots & \ldots & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \ldots & \vdots \\ \vdots & \vdots & \ddots & \ddots & \ddots & -\hat{I}^h & \vdots \\ 0 & \ldots & 0 & \ldots & -\hat{I}^h & T^h & 0 \\ 0 & \ldots & \ldots & \ldots & 0 & 0 & I_{n+1} \end{bmatrix} \in \mathbb{R}^{(n+1)^2\times(n+1)^2} \tag{22}$$

$$J^h = \begin{bmatrix} \hat{I}^h & 0 & \ldots & 0 \\ 0 & \hat{I}^h & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \ddots & 0 & \hat{I}^h \end{bmatrix} \in \mathbb{R}^{(n+1)^2\times(n+1)^2} \tag{23}$$

Using these matrices we can express A^h as:

$$A^h = \begin{bmatrix} I_{(n+1)^2} & 0 & \ldots & \ldots & \ldots & \ldots & 0 \\ 0 & S^h & -J^h & 0 & \ldots & \ldots & 0 \\ 0 & -J^h & S^h & -J^h & \ldots & \ldots & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \ldots & \vdots \\ \vdots & \vdots & \ddots & \ddots & \ddots & -J^h & \vdots \\ 0 & \ldots & 0 & \ldots & -J^h & S^h & 0 \\ 0 & \ldots & \ldots & \ldots & 0 & 0 & I_{(n+1)^2} \end{bmatrix} \in \mathbb{R}^{(n+1)^3\times(n+1)^3}, \tag{24}$$

where:

- The first and last block matrices in the main diagonal are used to introduce the boundary conditions $k = 1$ and $k = n+1$ respectively.

- The matrices S^h of the main diagonal are the analog of the 2D case $\rightarrow$ for k fixed, contributions of i and j.
- The matrices J^h in the upper and lower diagonals take into account the contributions of $u^h_{i,j,k+1}$ and $u^h_{i,j,k-1}$ respectively.

This matrix A^h, as in the 2D case, is Symmetric Positive Definite (we can apply Gershgorin Theorem to see the positive definiteness). The upper bandwidth is $(n+1)^2 \rightarrow$ distance between $u^h_{i,j,k}$ and $u^h_{i,j,k+1}$ using lexicographic order. The lower bandwidth is also $(n+1)^2 \rightarrow$ distance between $u^h_{i,j,k-1}$ and $u^h_{i,j,k}$ using lexicographic order.

Regarding the vector $\mathbf{f}^h$:

- For the boundary points: $i,j,k \in \{1, n+1\} \rightarrow$ boundary conditions: $u^h_{0_{i,j,k}}$
- For the interior points $i,j,k = 2,\ldots,n \rightarrow$ source function: $f^h_{i,j,k}$
- For points $i,j,k \in \{2,n\} \rightarrow$ contribution of the boundary conditions: $u^h_{0_{i\pm1,j\pm1,k\pm1}}$ (to make the matrix A^h symmetric)

$$\mathbf{f}^h = \begin{bmatrix} u^h_{0_{1,1,1}} \\ \vdots \\ u^h_{0_{n+1,n+1,1}} \\ u^h_{0_{1,1,2}} \\ \vdots \\ u^h_{0_{n+1,1,2}} \\ u^h_{0_{1,2,2}} \\ f^h_{2,2,2} + \frac{1}{h^2}u^h_{0_{1,2,2}} + \frac{1}{h^2}u^h_{0_{1,2,1}} + \frac{1}{h^2}u^h_{0_{1,1,2}} \\ f^h_{3,2,2} + \frac{1}{h^2}u^h_{0_{3,1,2}} + \frac{1}{h^2}u^h_{0_{3,2,1}} \\ \vdots \\ f^h_{n-1,2,2} + \frac{1}{h^2}u^h_{0_{n-1,1,2}} + \frac{1}{h^2}u^h_{0_{n-1,2,1}} \\ f^h_{n,2,2} + \frac{1}{h^2}u^h_{0_{n+12,3}} + \frac{1}{h^2}u^h_{0_{n,1,2}} + \frac{1}{h^2}u^h_{0_{n,2,1}} \\ u^h_{0_{n+1,2,2}} \\ \vdots \\ \vdots \\ u^h_{0_{1,3,3}} \\ f^h_{2,3,3} + \frac{1}{h^2}u^h_{0_{1,3,3}} \\ f^h_{3,3,3} \\ \vdots \\ u^h_{0_{1,1,n+1}} \\ \vdots \\ u^h_{0_{n+1,n+1,n+1}} \end{bmatrix} \in \mathbb{R}^{(n+1)^3} \tag{25}$$

where $f^h_{i,j,k}$ are given by (7) and $u^h_{0_{i,j}}$ are given by (8).

3. *Describe SSOR as a BIM and motivate why it can be used to solve the system considered.*

One way to solve the system $A\mathbf{u}^h = \mathbf{f}^h$ is to use Basic Iterative Methods based on splittings of the matrix A. From splittings of the following form: $A = M - N$, we can define the following iterative scheme:

$$M\mathbf{u}^{k+1,h} = N\mathbf{u}^{k,h} + \mathbf{f}^h$$
$$\mathbf{u}^{k+1,h} = M^{-1}\left(N\mathbf{u}^{k,h} + \mathbf{f}^h\right),$$
$$\mathbf{u}^{k+1,h} = \left(I - M^{-1}A\right)\mathbf{u}^{k,h} + M^{-1}\mathbf{f}^h$$

which correspond to one iteration of a BIM. Introducing the error vector $e^k = u - u^k$, we can see that:

$$\mathbf{e}^{k+1} = \underbrace{(I - M^{-1}A)}_{B,\text{Iteration matrix}}\mathbf{e}^k = B\mathbf{e}^k$$

so it follows that:

$$\mathbf{e}^k = B^k\mathbf{e}^0. \tag{26}$$

The method is convergent if $\mathbf{e}^k \to 0$, as $k \to \infty$ i.e. if $\lim_{k\to\infty} B^k = 0$. That is equivalent using Theorem 2.7.1 of the lecture notes as $\rho(B) < 1$.

From now on, we consider the decomposition: $A = D - E - F$, where A is either the 2d or 3d matrix previously computed, D is the diagonal of A, $-E$ is the lower part of A and $-F$ its the upper part.

We studied during the course, the so-called Jacobi, Gauss-Seidel, Damped Jacobi and the Successive Overrelaxation method (SOR). For the SOR method there were 2 variants:

- Forward SOR, based on the splitting: $M_1 = \frac{1}{w}D - E$
- Backward SOR, based on the splitting: $M_2 = \frac{1}{w}D - F$

We saw that the SOR method generally converges faster than Jacobi's method. However this method has a disadvantage: the matrices M_1 and M_2 that defined the splittings are not symmetric.

If A is symmetric, the SOR method can be made to result in an Symmetric splitting by combining an M_1-forward and M_2-backward sweep. The result method is known as SSOR (Symmetric Successive Overrelaxation). To compute the matrix M that defines the splitting of the SSOR, we do first a forward-step:

$$\hat{\mathbf{e}}^k = (I - M_1^{-1}A)\mathbf{e}^k$$

and then a backward step:

$$\mathbf{e}^{k+1} = (I - M_2^{-1}A)\hat{\mathbf{e}}^k$$

So for one iteration we obtain the following expression:

$$\mathbf{e}^{k+1} = \underbrace{(I - M_2^{-1}A)}_{B_2}\underbrace{(I - M_1^{-1}A)}_{B_1}\mathbf{e}^k = (I - \left(M_1^{-1} + M_2^{-1} - M_2^{-1}AM_1^{-1}\right)A)\mathbf{e}^k$$

Hence:

$$M^{-1} = M_1^{-1} + M_2^{-1} - M_2^{-1}AM_1^{-1} = M_2^{-1}(A - M_1 - M_2)M_1^{-1}$$
$$= \frac{2-w}{w}M_2^{-1}DM_1^{-1} = w(2-w)(D - wF)^{-1}D(D - wE)^{-1}$$

So finally the M-matrix of the SSOR method is:

$$M_{SSOR(w)} = \frac{1}{w(2-w)}(D - wE)D^{-1}(D - wF) \tag{27}$$

Moreover we can use the SSOR method, since is convergent for our problem $A\mathbf{u}^h = \mathbf{f}^h$. As we have seen in the explanation of (26), we have to verify that $\rho(B_{SSOR(w)}) < 1$. This matrix is equal to: $B_{SSOR(w)} = B_2B_1$, where B_1 and B_2 the iteration matrices of the forward and backward SOR respectively. We know that:

$$\rho(B_{SSOR(w)}) = \rho(B_2B_1) \leq \|B_2B_1\| \leq \|B_2\|\|B_1\| \tag{28}$$

and we can verify that both norms are less than 1 (we use the 1-norm and Matlab to check it).

4. *Derive an expression for the complexity of one single iteration of the algorithm. Assuming Niter iterations to be required, derive an estimate for the complexity to iterate until convergence.*

The complexity of the algorithm is different for the case 2d or 3d, since the sparsity structure is different and the length of the vectors is also different (2d case $(n+1)^2$, whereas 3d case $(n+1)^3$). We study the 2d case in detail and then applying the same reasoning we give the results for the 3d problem.

- **2D PROBLEM:**

We have seen that in the 2D case, the vectors $\mathbf{u}^h, \mathbf{f}^f \in \mathbb{R}^{(n+1)^2}$ and the matrix $A^h \in \mathbb{R}^{(n+1)^2 \times (n+1)^2}$. Out of these $(n+1)^2$ points, $2(n+1)+2(n-1) = 4n$ are boundary points and $(n-1)^2$ are interior points.

We have seen in the previous point, that 1 iteration of the SSOR method is equal to 1 iteration of the forward-SOR method plus 1 iteration of the backward-SOR method. So the number of flops per iteration is equal to the sum of flops per iteration of the2 steps. The Algorithm of the SSOR Method is given by:

Algorithm 1: Iteration k+1 of the SSOR algorithm 2d

Input: Matrix A, vectors f and u^k, w
Output: vector u^{k+1}
Forward step:
for *i = 1, . . . , (n+1)*^{2} **do**
  $u^{k+1}(i) = (f(i) - A(i, 1:i-1)u^{k+1}(i) - A(i, i+1:(n+1)^2)u^k(i))/A(i,i)$
  $u^{k+1}(i) = (1-w)u^k(i) + wu^{k+1}(i)$
end
$\sigma^k = u^{k+1}$, {*$\sigma$is an auxiliar vector, that we use to initiate the backward step*}
Backward step:
for *i = (n+1)*$^2, \ldots, 1$ **do**
  $u^{k+1}(i) = (f(i) - A(i, 1:i-1)\sigma^k(i) - A(i, i+1:(n+1)^2)u^{k+1}(i))/A(i,i)$
  $u^{k+1}(i) = (1-w)\sigma^k(i) + wu^{k+1}(i)$
end

Now we are going to analyze it:

SOR-forward Case:

- *For the boundary points*, we have seen that $A(i,i) = 1$, $A(i, 1:i-1) = \mathbf{0}$ and $A(i, i+1 : n+1) = \mathbf{0}$, so $u^{k+1}(i) = f(i)$. Notice that does not depend on k, so we only have to consider it in the 1st iteration $\rightarrow$ No contribution of flops for k+1-th iteration.
- *For the interior points*, in general, given i we have that $A(i,j) = 0$, $, \forall j$ expect for 5 elements $A(i, i-(n+1)), A(i, i-1), A(i,i), A(i, i+1), A(i, i+n+1)$ (note that is not true for every point, in particular is not true for the closest interior points to the boundary $\rightarrow u_{2,j}, u_{n,j}, u_{i,2}, u_{i,n}$). So we have:
  - Matrix vector multiplication $\rightarrow$ 7 flops (4 multiplications and 3 sums)
  - Adding the term $f(i) \rightarrow$ 1 flop and dividing by $A(i,i) \rightarrow$ 1 flop.
  - For the weighted sum $u^k(i)$ and $u^{k+1}(i) \rightarrow$ 3flops.

  So, for a single i we have 12 flops. Multiplying by the number of interior points we obtain, that the number of flops for the forward-SOR is $\approx 12(n-1)^2$.

SOR-backward Case:

- The only difference is the order in which the elements of the vector are visited/updated. So the number of flops is the same as in the forward-SOR, that is $\approx 12(n-1)^2$ flops.

Therefore the number of flops for a single iteration of the SSOR method is $\approx 24(n-1)^2$. In addition, this method has short recurrencies, so the number of flops in each iteration remains constant.

If we assume N_{iter} iterations to be required:

$$\boxed{\text{Number of flops in SSOR 2D} \approx 24 \times (n-1)^2 \times N_{iter} + 4n} \tag{29}$$

where 4n where the flops required in the first iteration to update the boundary points.

- **3D PROBLEM:**

In the 3D case, the vectors $\mathbf{u}^h, \mathbf{f}^f \in \mathbb{R}^{(n+1)^3}$ and the matrix $A^h \in \mathbb{R}^{(n+1)^3 \times (n+1)^3}$. Out of these $(n+1)^3$ points, $(n-1)^3$ are interior points and $6n^2+2$ are boundary points.

SOR-forward Case:

- Reasoning for the boundary points is the same as we previously explained.
- Regarding the interior points $\rightarrow$ in general, given i we have that $A(i,j) = 0, \forall j$ expect for 7 elements $A(i, i-(n+1)^2), A(i, i-(n+1)), A(i, i-1), A(i,i), A(i, i+1), A(i, i+n+1), A(i, i+(n+1)^2)$ (note that is not true for every point, in particular is not true for the closest interior points to the boundary $\rightarrow u_{2,j,k}, u_{n,j,k}, u_{i,2,k}, u_{i,n,k}, u_{i,j,2}, u_{i,j,n}$).

  So we have:

  - Matrix vector multiplication $\rightarrow$ 11flops (6 multiplications and 5 sums)
  - Adding the term $f(i) \rightarrow$ 1 flop, dividing by $A(i,i) \rightarrow$ 1 flop.
  - For the weighted sum $u^k(i)$ and $u^{k+1}(i) \rightarrow$ 3flops.

  So, for a single i we have 16 flops. Multiplying by the number of interior points we obtain, that the number of flops for the forward-SOR is $\approx 16(n-1)^3$.

SOR-backward Case:

- The number of flops for the backward-SOR is also $\approx 16(n-1)^3$.

So if we assume N_{iter} iterations to be required:

$$\boxed{\text{Number of flops in SSOR 3D} \approx 32 \times (n-1)^3 \times N_{iter} + 6n^2 + 2} \tag{30}$$

5. *Repeat the questions 3. and 4. using SSOR as a preconditioner for Conjugate Gradient*

We studied during the lectures the Conjugate gradient method. We saw that this method was always convergent and the convergence rate depended on $\kappa_2(A)$. When $\kappa_2(A)$ is large, the convergence is very slow, so it is not very appropriate for these problems. This is precisely what happens with our matrices A^h in 2d and 3d when h is small.

The way to solve this problem is to introduce so-called preconditioners. A preconditioner is a matrix that transform the system in such a way that: the transformed system has the same solution and the CG method becomes quite powerful to solve this new system. The idea behind the preconditioners is the following:

$$A\mathbf{u}^h = \mathbf{f}^h \Rightarrow M^{-1}A\mathbf{u}^h = M^{-1}\mathbf{f}^h \Rightarrow \tilde{A}\mathbf{u}^h = \tilde{\mathbf{f}^h}$$

The requirements on the M matrix are:

- M should be SPD.
- $\kappa_2(M^{-1}A) << \kappa_2(A)$.
- $M^{-1}\mathbf{r}$ is cheaper to compute.

In this section, we are going to use SSOR as a preconditioner. So the M matrix we use is the $M_{SSOR(w)}$ matrix defined above. Moreover this matrix has the previous required properties:

- Symmetry: follows from the fact that $E^t = F$, in our problem.
- Positive definite: given $\mathbf{u}$, we have to show that:

  $$\mathbf{u}^t M_{SSOR_w}\mathbf{u}^t > 0.$$

  If we call $\mathbf{v} = (D - wF)\mathbf{u}$ and using that $\mathbf{v}^t = \mathbf{u}^t(D - wF)^t = \mathbf{u}^t(D - wE)$, we can rewrite:

  $$\mathbf{u}^t M_{SSOR_w}\mathbf{u}^t = \frac{1}{w(2-w)}\mathbf{v}^t D^{-1}\mathbf{v} > 0,$$

  which follows from the fact that all elements in the main diagonal are positive.
- $\kappa_2(M^{-1}_{SSOR(w)}A) << \kappa_2(A)$. This is because $M_{SSOR(w)}$ has a similar structure to A (same band-width). Hence the structures of M^{-1} and A^{-1} are also similar, and the statement follows from $1 = \kappa_2(Id) = \kappa_2(A^{-1}A) \sim \kappa_2(M^{-1}A)$.
- We can compute $\mathbf{z} = M^{-1}\mathbf{r}$ cheaply. The way to do it is solving the system $M\mathbf{z} = \mathbf{r}$. We can decompose this problem into:

  $$Mz = r \Rightarrow \begin{cases} (D - wF)z = s \\ D^{-1}s = q \\ (D - wE)q = r \end{cases}$$

  But since $(D - wE)$ is a lower triangular matrix, D^{-1} is a diagonal matrix and $(D - wF)$ is an upper triangular matrix, we can use forward and backward substitution and thus the system is quite easy to solve.

We then proceed to derive an expression for the complexity of one iteration of the algorithm. As we previously explained in Section 4 the complexity of the 2d problem is different form the 3d problem. We first study the 2d case in detail and then applying the same reasoning we give the results for the 3d problem.

- **2D PROBLEM:**

We have seen that in the 2D case, the vectors $\mathbf{u}^h, \mathbf{f}^f \in \mathbb{R}^{(n+1)^2}$ and the matrix $A^h \in \mathbb{R}^{(n+1)^2 \times (n+1)^2}$. The Algorithm of the Preconditioned Conjugate Gradient Method is given by:

Algorithm 2: Preconditioned Conjugate Gradient Method for $M = M_{SSOR(w)}$

$\mathbf{u}^0 = \mathbf{0}; \mathbf{r}^0 = \mathbf{f}$
initialization;
for *k=1,2,...* **do**
  $\mathbf{z}^{k-1} = M^{-1}\mathbf{r}^{k-1}$
  if $k = 1$ **then**
    $\mathbf{p}^1 = \mathbf{z}^0$
  end
  else
    $\beta_k = \frac{(\mathbf{r}^{k-1})^T \mathbf{z}^{k-1}}{(\mathbf{r}^{k-2})^T \mathbf{z}^{k-2}}$
    $\mathbf{p}^k = \mathbf{z}^{k-1} - \beta_k \mathbf{p}^{k-1}$
  end
  $\alpha_k = \frac{(\mathbf{r}^{k-1})^T \mathbf{z}^{k-1}}{(\mathbf{p}^k)^T A \mathbf{p}^k}$
  $\mathbf{u}^k = \mathbf{u}^{k-1} + \alpha_k \mathbf{p}^k$
  $\mathbf{r}^k = \mathbf{r}^{k-1} - \alpha_k A \mathbf{p}^k$
end

So in one iteration we have to:

- Compute $\mathbf{z}^{k-1} = M^{-1}\mathbf{r}^{k-1}$.
  As we previously explained we decompose this problem into 3 steps. Moreover, we have to take into account that A^h is pentadiagonal in the 2d case to simplify the calculation and to not make unnecessary flops. We use the forward and backward substitution that are explained in Algorithm 3 and 4 in the part of implementation.
  - Solve: $(D - wE)q = r$.
    - For $i = 1, \ldots, (n+1)^2$
      $q(i) = (r(i) - wA(i, i-1)q(i-1) - wA(i, i-(n+1))q(i-(n+1))/A(i,i)$
      end

    We count 5 flops per fixed i (2multiplication, 2sums and 1 division).
    So the whole loop supposes $5(n+1)^2$ flops.
  - Solve: $D^{-1}s = q$.
    As D is diagonal, we only need 1 flop (1 division) per element of the vector.
    Therefore the whole loop supposes $(n+1)^2$ flops.
  - Solve: $(D - wF)z = s$.
    The whole loop supposes $5(n+1)^2$ as is it the symmetric analogous as the first step.
- Compute β_k.
  Notice that due to the form of β_k the denominator was already computed in the previous iteration k-1. So we only have to take into account the vector-vector product. This calculation means

$2(n+1)^2$ flops (half due to multiplications and the other half by adding up the result of these multiplications).

- Compute $\mathbf{p}^k = \mathbf{z}^{k-1} - \beta_k \mathbf{p}^{k-1}$.
  This vector update implies 2 flops (1 multiplication, 1 substraction) per component. So it supposes $2(n+1)^2$ flops.
- Compute α_k.
  The numerator matches with the numerator of β_k, so it was already computed. For the denominator we have to compute:
  - Matrix-vector product: $A\mathbf{p}^k$, $9(n+1)^2$ flops. As we have 5 multiplications (A pentadiagonal) + 4 summations per element of resulting vector.
  - Inner product $(\mathbf{p}^k)^T(A\mathbf{p}^k)$ which supposes $2(n+1)^2$ flops.
- Compute $\mathbf{u}^k = \mathbf{u}^{k-1} + \alpha_k \mathbf{p}^k$.
  This vector update implies $2(n+1)^2$ flops.
- Compute $\mathbf{r}^k = \mathbf{r}^{k-1} - \alpha_k A\mathbf{p}^k$.
  This vector update implies $2(n+1)^2$ flops, as the matrix-vector product was already computed

Adding all the previous results we obtain that the number of flops in 1 iteration of the SSOR preconditioner for Conjugate gradient for the 2d problem is approximately $30(n+1)^2$.

Regarding the memory, we have to store 5 vector in every iteration: $\mathbf{z}^{k-1}, \mathbf{u}^k, \mathbf{r}^k, \mathbf{p}^k$ and $A\mathbf{u}^k$.

In addition, this method has short recurrencies, so the number of flops in each iteration remains constant. Therefore if we assume N_{iter} iterations to be required:

$$\boxed{\text{Number of flops in SSOR preconditionar CG 2D case} \approx 30 \times (n+1)^2 \times N_{iter}} \tag{31}$$

- **3D PROBLEM:**

The main differences with respect 2d case are that the vectors $\mathbf{u}^h, \mathbf{f}^f \in \mathbb{R}^{(n+1)^3}$, the matrix $A^h \in \mathbb{R}^{(n+1)^3 \times (n+1)^3}$ and this matrix A is heptadiagonal. By applying the same procedures, we obtain that in 1 iteration we have to:

- Compute $\mathbf{z}^{k-1} = M^{-1}\mathbf{r}^{k-1}$.
  - Solve: $(D - wE)q = r$.
    - For $i = 1, \ldots, (n+1)^3$
      $q(i) = (r(i) - wA(i, i-1)q(i-1) - wA(i, i-(n+1))q(i-(n+1))$
      $-wA(i, i-(n+1)^2)q(i-(n+1)^2)/A(i,i)$
      end

    The whole loop supposes $7(n+1)^3$ flops.
  - Solve: $D^{-1}s = q \rightarrow (n+1)^3$ flops.
  - Solve: $(D - wF)z = s \rightarrow 7(n+1)^3$ flops (by symmetry).
- Compute $\beta_k \rightarrow$ inner product $2(n+1)^3$ flops.
- Compute $\mathbf{p}^k = \mathbf{z}^{k-1} - \beta_k \mathbf{p}^{k-1} \rightarrow 2(n+1)^3$ flops.
- Compute α_k.
  - Matrix-vector product: $A\mathbf{p}^k$, $13(n+1)^3$ flops. As we have 7 multiplications (A heptadiagonal) + 6 sums per element of resulting vector.

  - Inner product $(\mathbf{p}^k)^T(A\mathbf{p}^k) \rightarrow 2(n+1)^3$ flops.
- Compute $\mathbf{u}^k = \mathbf{u}^{k-1} + \alpha_k \mathbf{p}^k \rightarrow 2(n+1)^3$ flops.
- Compute $\mathbf{r}^k = \mathbf{r}^{k-1} - \alpha_k A\mathbf{p}^k \rightarrow 2(n+1)^3$ flops.

Adding all the previous results we obtain that the number of flops in 1 iteration of the SSOR preconditioner for Conjugate gradient for the d problem is approximately $38(n+1)^3$.

Regarding the memory, we have to store 5 vector in every iteration: $\mathbf{z}^{k-1}, \mathbf{u}^k, \mathbf{r}^k, \mathbf{p}^k$ and $A\mathbf{u}^k$.

In addition, this method has short recurrencies, so the number of flops in each iteration remains constant. Therefore if we assume N_{iter} iterations to be required:

$$\boxed{\text{Number of flops in SSOR preconditionar CG 3D case} \approx 38 \times (n+1)^3 \times N_{iter}} \tag{32}$$

Brief Conclusion:

Once all the calculations have been made, we can compare the complexity of the SSOR method and the SSOR as a preconditioner for CG. After observing (29), (31) (2d case) and (30), (32) (3d case), we can infer that the number of iterations required in SSOR is slightly less than for the Preconditioning method (same order of magnitude). One may think that as the number of iterations is lower, then it is better to use SSOR. However, the great advantage of the CG method and CG preconditioning is that they are optimal:

$$\|\mathbf{u} - \mathbf{u}_{CG}^k\|_A = \min_{y \in K^k(A, \mathbf{r}^0)} \|\mathbf{u} - \mathbf{y}\|_A \tag{33}$$

In particular considering $\mathbf{u}_{SSOR}^k$ the result of iteration k of SSOR, and as $\mathbf{u}_{SSOR}^k \in K^k(A, \mathbf{r}^0)$ we have that:

$$\|\mathbf{u} - \mathbf{u}_{CG}^k\|_A \leq \|\mathbf{u} - \mathbf{u}_{SSOR}^k\|_A \tag{34}$$

Therefore we conclude that for solving this problem theoretically is better to use SSOR as preconditioner for the Conjugate Gradient.

6. *Describe the Block version of SSOR (BSSOR) based on the decomposition as given above*

Lastly we are going to deal with the BSSOR. In the statement of the problem, it was proposed to divide the domain Ω into p rectangular subdomains Ω_i consisting of $\frac{n_x}{p} \times n_y$ grid points. In order to deal with this problem, it would be highly recommended to have used the y-lexicographic ordering, that is: $I = j + (i-1)(n+1)$ to order the components $u^h_{i,j}$. However during our whole work we have used the x-lexicographic ordering, so dealing with this subdivision of the domain is not the most appropriate and simple way to solve the problem. Therefore we will use the analogous symmetric domain decomposition (the idea is the same). We divide our domain Ω into p rectangular subdomains consisting of $n_x \times \frac{n_y}{p}$ grid points. Notice that using this decomposition, each subdomain is composed by $\frac{n_y}{p}$ rows of the grid. For instance if we take $p = n_y$, each row would be a domain that would correspond to the vectors defined in (12).

Using this decomposition the linear system $A\mathbf{u} = \mathbf{f}$ can be rewritten using block matrices

$$\begin{bmatrix} A_{1,1} & A_{1,2} & \dots & A_{1,p} \\ A_{2,1} & A_{2,2} & \dots & A_{2,p} \\ \vdots & \vdots & \dots & \vdots \\ A_{p,1} & A_{p,2} & \dots & A_{p,p} \end{bmatrix} \begin{bmatrix} U_1 \\ U_2 \\ \vdots \\ U_p \end{bmatrix} = \begin{bmatrix} F_1 \\ F_2 \\ \vdots \\ F_p \end{bmatrix} \tag{35}$$

A regular splitting of matrix A = D - E - F results in

$$D = \begin{bmatrix} A_{1,1} & & \\ & \ddots & \\ & & A_{p,p} \end{bmatrix} E = -\begin{bmatrix} 0 & 0 & 0 & \dots & 0 \\ A_{2,1} & 0 & 0 & \dots & 0 \\ A_{3,1} & A_{3,2} & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \dots & \vdots \\ A_{p,1} & A_{p,2} & A_{p,3} & \dots & 0 \end{bmatrix} F = -\begin{bmatrix} 0 & A_{1,2} & A_{1,3} & \dots & A_{1,p} \\ 0 & 0 & A_{2,3} & \dots & A_{2,p} \\ 0 & 0 & 0 & \dots & A_{3,p} \\ \vdots & \vdots & \vdots & \dots & \vdots \\ 0 & 0 & 0 & \dots & 0 \end{bmatrix} \tag{36}$$

The BSSOR like pointwise SSOR, consists of a forward sweep:

$$D\mathbf{u}^{n+\frac{1}{2}} = \omega(E\mathbf{u}^{n+\frac{1}{2}} + F\mathbf{u}^n + \mathbf{f} + (\mathbf{1} - \omega)\mathbf{u^n} \tag{37}$$

and a backward sweep:

$$D\mathbf{u}^{n+1} = \omega(E\mathbf{u}^{n+\frac{1}{2}} + F\mathbf{u}^{n+1} + \mathbf{f} + (\mathbf{1} - \omega)\mathbf{u^{n+\frac{1}{2}}} \tag{38}$$

This can be rewritten as:

$$A_{i,i}U_i^{(n+\frac{1}{2})} = \omega\left(-\sum_{j=1}^{i-1} A_{i,j}U_j^{(n+\frac{1}{2})} - \sum_{j+1}^{p} A_{i,j}U_j^{(n)} + F_i\right) + (1-\omega)A_{i,i}U_i^{(n)} \quad , i = 1, ..., p \tag{39}$$

and

$$A_{i,i}U_i^{(n+1)} = \omega\left(-\sum_{j=1}^{i-1} A_{i,j}U_j^{(n+\frac{1}{2})} - \sum_{j+1}^{p} A_{i,j}U_j^{(n+1)} + F_i\right) + (1-\omega)A_{i,i}U_i^{(n+\frac{1}{2})} \quad , i = p, .., 1 \tag{40}$$

Suppose $p = 3$ and A is a 3x3 block matrix, lets say we want to calculate $U_2^{n+\frac{1}{2}}$. From Equation 39, we can see that this requires us to have $U_1^{n+\frac{1}{2}}$. This is computed by processor 1, so processor 2 has to wait for this result. Hence, with this ordering of the nodes, BSSOR is really not efficient in parallel compared to the Block Jacobi. In Block Jacobi, no communication between processors is required. Finally we mention that for BSSOR would also be interesting to use the red-black ordering. Using this ordering the block matrices in the main diagonal $A_{i,i}$ become diagonal, so is not quite difficult to compute $A_{i,i}^{-1}$ in (40).

3 Implementation Assignment

1. *Implement a code that constructs the matrix A^h and the right-hand side vector f^h in 2D and 3D.* In both cases we are going to use Kronecker product to carry out this task.

   - **2D PROBLEM**

   For constructing the matrix A^h we looked at (18). The first step is to build the block matrices of the previous expression, that it is: the matrix T^h given by (16) and the matrix $\hat{I}^h$ given by (17). We can construct this matrices using the command diag, as they are tridiagonal/diagonal matrices. Once done it we proceed to the construction of A^h using Kronecker products, which allow us to build these block matrices:

$$A^h = \underbrace{diag[0,1,\ldots,1,0] \otimes T^h}_{\text{Blocks } A_{2,2},\ldots,A_{n,n}} + \underbrace{diag[1,0,\ldots,0,1] \otimes I_{n+1}}_{\text{Blocks} A_{1,1} \text{ and } A_{n+1,n+1}} +$$

$$\underbrace{diag_{\text{1st lower}}[0,-1,\ldots,-1,0] \otimes \hat{I}^h}_{Blocks\ A_{3,2},\ldots,A_{n,n-1}} + \underbrace{diag_{\text{1st upper}}[0,-1,\ldots,-1,0] \otimes \hat{I}^h}_{Blocks\ A_{2,3},\ldots,A_{n-1,n}}$$

   For constructing f^h we looked at (19) and we use the lexicographic order $\rightarrow I = i + (j-1)(n+1)$. We proceed in three steps:

   -First, we introduce the boundary points, that is for $i \text{ or } j \in \{1, n+1\} \rightarrow f(I) = u_0(x_i y_j) = sin(x_i y_j)$

   -Secondly, the interior points, for $i, j = 2, \ldots, n \rightarrow f(I) = f^h_{i,j} = (x_i^2 + y_j^2) sin(x_i y_j)$

   -Lastly we introduce the contributions of the boundary to the pertinent interior points, that is for: $i \text{ or } j \in \{2, n\} \rightarrow f(I) = f(I) + \frac{1}{h^2} sin(x_{i\pm1} y_{j\pm1})$, where the sign $-$ is used when i or j=2, the sign $+$ is used when i or j=n, and if both $i, j \in \{2, n\}$ we have to add 2 contributions.

   - **3D PROBLEM**

   For constructing the matrix A^h we looked at (24). As we have done before, first we construct the matrices T^h and $\hat{I}^h$ given by (21) using the command diag. The following step is to construct both matrices S^h and J_h given by (22) and (23).

$$S^h = \underbrace{diag[0,1,\ldots,1,0] \otimes T^h}_{\text{Blocks } S_{2,2},\ldots,S_{n,n}} + \underbrace{diag[1,0,\ldots,0,1] \otimes I_{n+1}}_{\text{Blocks} S_{1,1} \text{ and } S_{n+1,n+1}} +$$

$$\underbrace{diag_{\text{1st lower}}[0,-1,\ldots,-1,0] \otimes I^h}_{Blocks\ S_{3,2},\ldots,S_{n,n-1}} + \underbrace{diag_{\text{1st upper}}[0,-1,\ldots,-1,0] \otimes I^h}_{Blocks\ S_{2,3},\ldots,S_{n-1,n}}$$

$$J^h = \underbrace{diag[0,1,\ldots,1,0] \otimes \hat{I}^h}_{\text{Blocks } J_{2,2},\ldots,J_{n,n}}$$

   Once these 2 matrices are defined. We proceed to define matrix A^h. Note that S^h,$J^h \in \mathbb{R}^{(n+1)^2 \times (n+1)^2}$, so we have to used another Kronecker product to increase to dimension $(n+1)^3 \times (n+1)^3$. So we finally have:

$$A^h = \underbrace{diag[0,1,\ldots,1,0] \otimes S^h}_{\text{Blocks } A_{2,2},\ldots,A_{n,n}} + \underbrace{diag[1,0,\ldots,0,1] \otimes I_{(n+1)^2}}_{\text{Blocks} A_{1,1} \text{ and } A_{n+1,n+1}} +$$

$$\underbrace{diag_{\text{1st lower}}[0,-1,\ldots,-1,0] \otimes J^h}_{Blocks\ A_{3,2},\ldots,A_{n,n-1}} + \underbrace{diag_{\text{1st upper}}[0,-1,\ldots,-1,0] \otimes J^h}_{Blocks\ A_{2,3},\ldots,A_{n-1,n}}$$

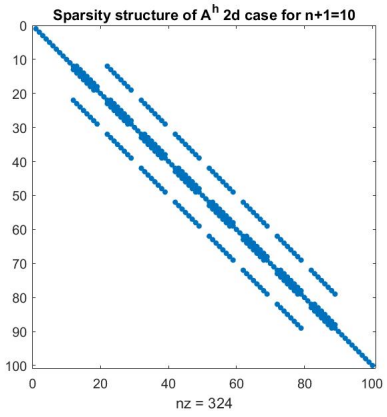
For constructing f^h we looked at (25) and we use the lexicographic order $\rightarrow I = i + (j-1)(n+1) + (k-1)(n+1)^2$. We proceed in the same three steps as before:

-First, we introduce the boundary points, that is for i *or* j *or* $k \in \{1, n+1\} \rightarrow f(I) = u_0(x_i y_j z_k) = sin(x_i y_j z_k)$

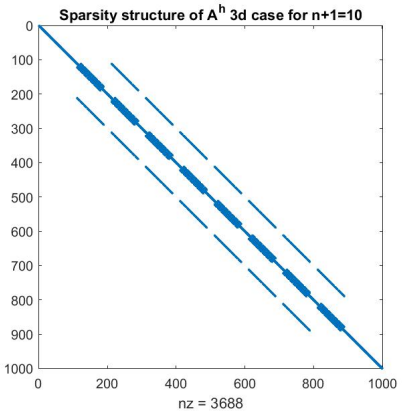
-Secondly, the interior points, for $i, j, k = 2, \ldots, n \rightarrow f(I) = f^h_{i,j} = (x_i^2 y_j^2 + y_j^2 z_k^2 + x_i^2 z_k^2) sin(x_i y_j z_k)$

-Lastly we introduce the contributions of the boundary to the pertinent interior points, that is for: i *or* j *or* $k = \in \{2, n\} \rightarrow f(I) = f(I) + \frac{1}{h^2} sin(x_{i\pm1} y_{j\pm1} z_{k\pm1})$, where the sign $-$ is used when i or j or k=2, the sign $+$ is used when i or j or k=n, and if two of $i, j, k \in \{2, n\}$ we have to add 2 contributions and if the three of them $i, j, k \in \{2, n\}$ we have to add 3 contributions.

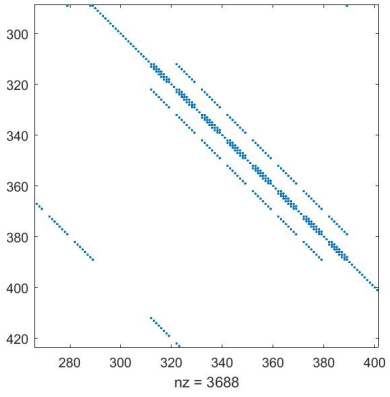
Our code to construct matrix A and f is given in Appendix B.
It is also interesting to use the command spy of Matlab, to see the sparsity structure of matrix A^h.



(a) 2d Case.



(b) 3d Case.



(c) zoom of the 3d Case to appreciate its band structure.

Figure 1: Sparsity of the matrix A^h for dimension n=2 and n=3.

2. Solve the linear problem by a direct solver for $h = 1/2^p$ for p = 2,..., 10 in 2D (and p = 2,..., 8 in 3D) and verify that the discretization scheme is second order accurate by inspecting the max-norm of the discretization error $\|u^h - u^h_{ex}\|_\infty$.

   As the matrix A^h both in 2d and 3d is Symmetric Positive Definite, the most convenient direct solver is Cholesky Decomposition. When A is SPD, we have seen during the lectures that A can be decom-

posed in the form:

$$A = CC^t$$

where $C \in \mathbb{R}^{(n+1)^2 \times (n+1)^2}$ in 2d and $C \in \mathbb{R}^{(n+1)^3 \times (n+1)^3}$ in 3d, is a lower triangular matrix. The fact that only a single lower triangular matrix C needs to be be computed obviously results in savings both in computational cost and in memory storage over the LU-decomposition. It furthermore appears that the SPD property guarantees numerical stability of the factorization algorithm.

To compute the entries of C we use the above expression in its multiplicative element form. As C is lower diagonal $c_{i,j} = 0 \; if \; i < j$ and for C^t we have that $c^t_{i,j} = c_{j,i}$, so

$$a_{i,j} = \sum_{k=1}^{N} c_{i,k} c^t_{k,j} = \sum_{k=1}^{N} c_{i,k} c_{j,k} = \sum_{k=1}^{min(i,j)} c_{i,k} c_{j,k}$$

- We first start with the 1st column of C $\rightarrow$ j=1. for i=1, we have that $a_{1,1} = c^2_{1,1}$, so $c_{1,1} = \sqrt{a_{1,1}}$. for the remaining i, we have $a_{i,1} = c_{i,1} c_{1,1}$ and as $c_{1,1}$ was already computed, the can compute the value of $c_{i,1} = \frac{a_{i,1}}{c_{1,1}}$
- For the 2nd column of C $\rightarrow$ j=2.

  for i=1, as C is lower triangular $c_{1,2} = 0$, for $i = 2$ we can see that $c_{2,2} = \sqrt{a_{2,2} - c^2_{2,1}}$, and for the remaining i we see that $c_{i,2} = \frac{a_{i,2} - c_{2,1} c_{j,1}}{c_{2,2}}$. All these values can be computed, as we already computed the 1st column and the value $c_{2,2}$.
- Following this iterative method, in the j-th step, we will have computed the first j-1 columns of C. So for computing the j-th column:

  For i<j, $c_{i,j} = 0$, as C is lower triangular.

  For i=j, we can show that $c_{j,j} = \sqrt{a_{j,j} - \sum_{k=1}^{j-1} c^2_{i,k}}$, but as k=1,...,j-1 and we know the values of the first j-1 columns, we can compute this value.

  For i>j, we can show that $c_{i,j} = \frac{a_{i,j} - \sum_{k=1}^{j-1} c_{i,k} c_{j,k}}{c_{j,j}}$ but as we computed the fist j-1 columns and the value of $c_{j,j}$, we are able to compute the column j.
- We proceed in this way, until we complete the whole matrix A.

Once C is computed, we have: $Au = CC^t u = f$ that can be transformed into two linear systems:

$$Au = f \Rightarrow \begin{cases} Cw = f \\ C^t u = w \end{cases}$$

First we solve $Cw = f$ and once compute w,we solve $C^t u = w$. As C and C^t are lower diagonal and

upper diagonal respectively, we can use the forward and backward substitution to solve them.

Algorithm 3: Forward Substitution step

Input: Matrix C and vector f
Output: Solution w of the system Cw=f.
for *i = 1, . . . , n* **do**
  w(i)=(f(i)-C(i,1:i-1)w(1:i-1))/C(i,i)
end

Algorithm 4: Backward Substitution step

Input: Matrix C^t and vector w
Output: Solution u of the system $C^t u = w$.
for *i = n, . . . , 1* **do**
  u(i)=(w(i)-$C^t(i, i+1:n)w(i+1:n))/C(i,i)$
end

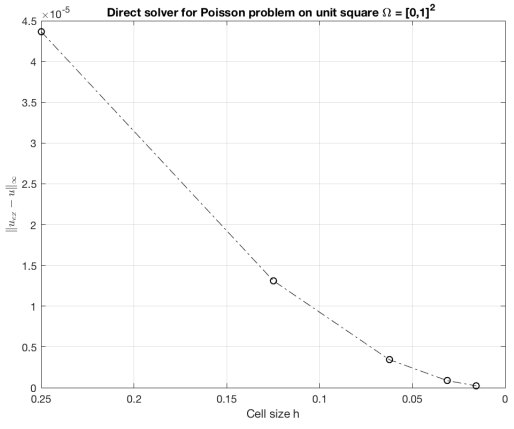
Our code for using Cholesky factorization as a direct solver can be found in Appendix C.
The finite difference scheme converges if $\|\mathbf{u}^h - \mathbf{u}_{ex}\|_\infty \to 0$ as $h \to 0$. Using Taylor Series expansions, one can calculate the order of accuracy of the finite difference scheme. However, since we know our exact solution there is a more practical way for us. If the the method is of order k, it can be said that $\|\mathbf{u}^h - \mathbf{u}_{ex}\|_\infty$ has an upper bound, consisting of a constant C and h^k, where h is the cell size and k the order of accuracy.

$$\|\mathbf{u}^h - \mathbf{u}_{ex}\|_\infty \leq Ch^k \tag{41}$$

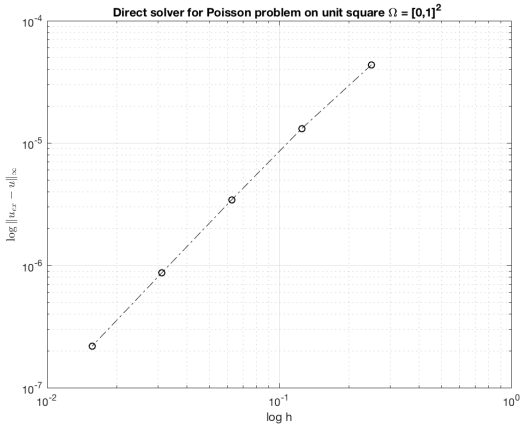
We can rewrite this as

$$\log \|\mathbf{u}^h - \mathbf{u}_{ex}\|_\infty = \log C + k \log h$$

Hence, making a log-log plot can show the order of accuracy of the method by inspecting the slope. From the log-log plot in Figure 2, it can be seen that the slope of the line is equal to 2.



(a) Max norm of the discretization $\|u^h - u^h_{ex}\|_\infty$ against cell-size h



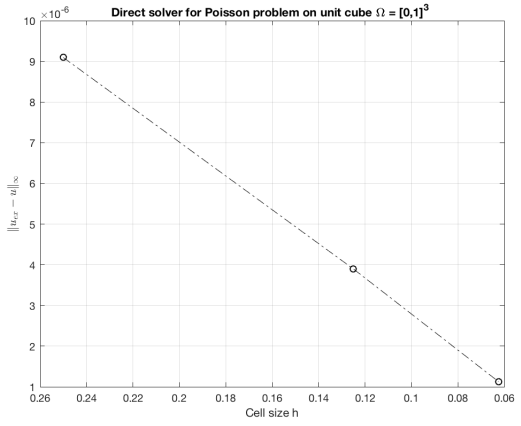
(b) $\log \|u^h - u^h_{ex}\|_\infty$ against $\log h$

Figure 2: Results from using Cholesky decomposition to solve the system $A\mathbf{x} = \mathbf{f}$ for the 2D case

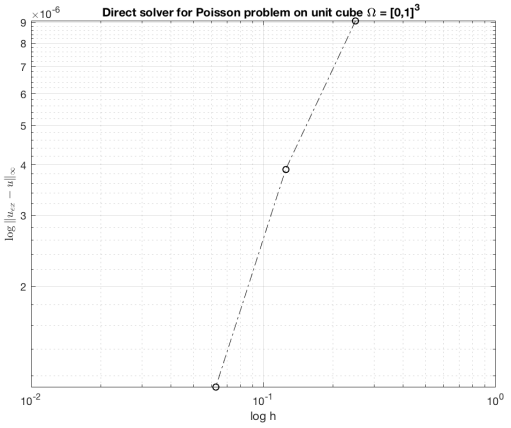
This can be shown clearer in a table. When we calculate $\log\left(\frac{\|u^h - u^h_{ex}\|_{h2}}{\|u^h - u^h_{ex}\|_{h1}}\right) / \log\left(\frac{h2}{h1}\right)$ where h_2 is twice as small as h_1. As one can see as the cell size is decreased, the order of accuracy goes to two, the theoretical value of this finite difference scheme.

Table 1: Rate of convergence of the finite difference scheme for the Poisson problem on unit square $\Omega = [0,1]^2$

n	2^2	2^3	2^4	2^5	2^6
$\log\left(\frac{\|u^h - u^h_{ex}\|_{h2}}{\|u^h - u^h_{ex}\|_{h1}}\right) / \log\left(\frac{h2}{h1}\right)$	0	1.7334	1.9313	1.9826	1.9956



(a) Max norm of the discretization $\|u^h - u^h_{ex}\|_\infty$ against cell-size h



(b) $\log \|u^h - u^h_{ex}\|_\infty$ against $\log h$

Figure 3: Results from using Cholesky decomposition to solve the system $A\mathbf{x} = \mathbf{f}$ for the 3D case

Table 2: Rate of convergence of the finite difference scheme for the Poisson problem on unit cube $\Omega = [0,1]^3$

n	2^2	2^3	2^4
$\log\left(\frac{\|u^h - u^h_{ex}\|_{h2}}{\|u^h - u^h_{ex}\|_{h1}}\right) / \log\left(\frac{h2}{h1}\right)$	0	1.2228	1.7882

The same results for the 3D case can be seen in Figure 3 and Table 2. However, due to long computation times, only n up to 2^4 could be computed for the 3D case. In fact, for both the 2D and 3D case, the CPU times are shown in Table 3 and Table 4, respectively. For the 2D case, with $n = 2^6$, matrix A is of size $4225x4225$. To solve this system using Cholesky decomposition, the program takes around 500 seconds, a bit more than 8 minutes. For the 3D case, with $n = 2^4$, matrix A is of size $4913x4913$. To solve this using Cholesky decomposition it takes around 900 seconds, which is around 15 minutes. So adding around 700 rows and columns to A, the computation time almost dubbeled.

Table 3: CPU time against problem size to solve the Poisson problem on unit square $\Omega = [0,1]^2$

n	2^2	2^3	2^4	2^5	2^6
CPU time [s]	0.0110	0.0372	0.5226	12.0144	497.6235

Table 4: CPU time against problem size to solve the Poisson problem on unit cube $\Omega = [0,1]^3$

n	2^2	2^3	2^4
CPU time [s]	0.0988	4.2044	891.5875

We also include in Figure 4 the three-dimensional graphic solution for the case where the mesh is 2d.

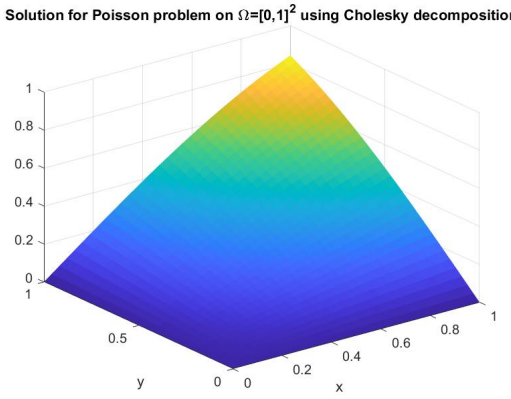


Figure 4: Graphic Solution of the Poisson equation with source function $f(x, y) = (x^2 + y^2)sin(xy)$.

3. *Solve the linear system using SSOR with parameter w= 1.5 used as a BIM. For various values of N, make a logarithmic plot of* $\frac{\|r_m\|_2}{\|f^h\|_2}$ *versus m*

   The SSOR method was studied in greater depth in Sections 3 and 4 of the pen and paper assignment. In addition, the algorithm on which this method is based is explained in Algorithm 1. The main difference is that in our code we include a while loop, with a criterion stop given by (4).

   In this section we are simply showing results obtained with the program we have created with MATLAB.

   We have implemented this code for general matrices. For sparse matrices, in lines 22 and 30 of the code, instead of using the whole row $\rightarrow A(i, 1 : i - 1)$ and $A(i, i + 1 : 1)$, we could specify only the non-zero elements. For example in the 2d case, only include $A(i, i - (n + 1)), A(i, i - 1), A(i, i + 1), A(i, i + (n + 1))$. In this way only the necessary flops would be made, which consequently would decrease the CPU time.

   When we look at the 2D case, in Figure 5, the iterations against $\frac{\|r_m\|_2}{\|f^h\|_2}$ on a log-scale is plotted. As mentioned before, the stopping criteria was when $\frac{\|r_m\|_2}{\|f^h\|_2}$ is less than 10^{-10}. This is done for several values of n, specifically $n = 2^2, 2^3, 2^4, 2^{5,2^6}$.For greater values of n, the program take too long to run.

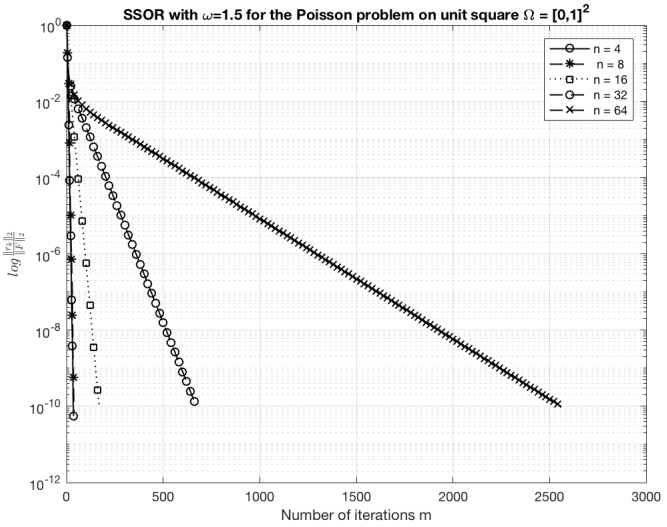
Table 5: Number of iterations m against problem size to solve the Poisson problem on unit square $\Omega = [0, 1]^2$

n	2^2	2^3	2^4	2^5	2^6
m	36	39	169	672	2560

Table 6: Number of iterations m against problem size to solve the Poisson problem on unit cube $\Omega = [0, 1]^3$

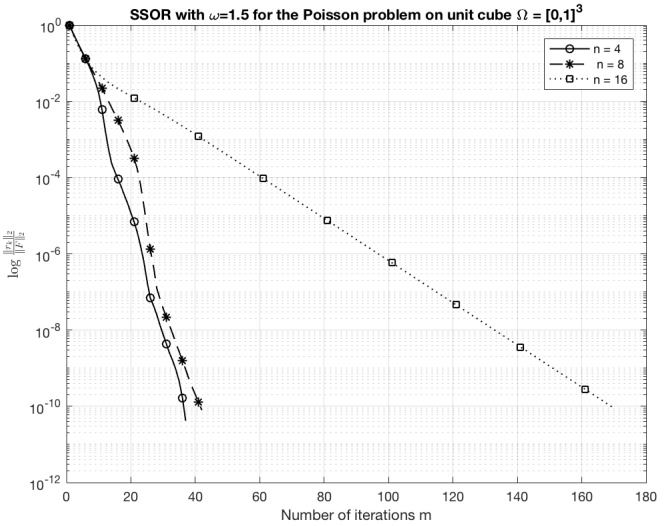
n	2^2	2^3	2^4
m	37	42	170

When we look at the 3D case, in Figure 6, the iterations against $\frac{\|r_m\|_2}{\|f^h\|_2}$ on a log-scale is plotted. As mentioned before, the stopping criteria was when $\frac{\|r_m\|_2}{\|f^h\|_2}$ is less than 10^{-10}. This is done for several values of n, specifically $n = 2^2, 2^3, 2^4$. For greater values of n, the program take too long to run.



(a) The residual $\frac{\|r_m\|_2}{\|f^h\|_2}$ against the number of iterations m

Figure 5: SSOR ($\omega = 1.5$) as BIM to solve the system $A\mathbf{x} = \mathbf{f}$ for the 2D case



(a) The residual $\frac{\|r_m\|_2}{\|f^h\|_2}$ against the number of iterations m

Figure 6: SSOR ($\omega = 1.5$) as BIM to solve the system $A\mathbf{x} = \mathbf{f}$ for the 3D case

When we inspect Figure 5, we can see that for larger n the rate of convergence compared to the amount of iterations becomes linear quite fast. Due to the scaling of the plot, however the rate of convergence for smaller n is not visible. Since for the 3D case, only n up to 2^4 is plotted, it is visible in Figure 3 that for small n the rate of convergence is not the same each iteration.

We can also compare the maximum amount of iterations required with respect to problem size for the 2D case and 3D in Table 5 and Table 6, respectively. It it interesting to see that the 3D case requires around the same amount of iterations as the 2D case (still a single iteration in the 3d case is far more expensive than in the 2d). This would suggest that for SSOR as BIM, the amount of iterations doesn't matter in terms of the dimension of the problem. This would make SSOR as BIM very attractive for the Poisson equation in 3D. Again to compare, for the 3D case with $n = 2^4$, A has a size of $4913x4913$. In the 2D case for $n = 2^6$, A has a size of $4225x4225$. So although matrix A is bigger for the 3D case, it still requires significantly less iterations.

Our code for SSOR as a BIM is given in Appendix D.

4. *For various values of N, report on the asymptotic rate of convergence by tabulating the residual reduction factor $\|r_m\|_2/\|r_{m-1}\|_2$ during the last five iterations*

   When n is large, from Table 7 and Table 8 we can see that the reduction factor becomes constant. When one looks closely to the 2D case in Table 7, one can see that as n is increased the reduction factor approaches 1. From Theorem 5.4.1 of the reader, we known when $\rho(B) < 1$, the solution convergences, and if $\rho(B) \ll 1$ the BIM converges faster. According to (Yousssef Saad, 2003) the asymptotic convergence factor is equal to $\rho(B)$. So when we compare Figure 5 with the values in Table 7, it is can be seen that as the reduction factor increases, the speed of the convergence decreases.

Table 7: Residual reduction factor for the last 5 iterations using SSOR ($\omega = 1.5$) solving the Poisson problem on unit square $\Omega = [0, 1]^2$

n	2^2	2^3	2^4	2^5	2^6
$\|\mathbf{r}_m\|_2/\|\mathbf{r}_{m-1}\|_2$	0.5538	0.4787	0.8804	0.9709	0.9928
$\|\mathbf{r}_{m-1}\|_2/\|\mathbf{r}_{m-2}\|_2$	0.6651	0.5800	0.8804	0.9709	0.9928
$\|\mathbf{r}_{m-2}\|_2/\|\mathbf{r}_{m-3}\|_2$	0.2021	0.4492	0.8804	0.9709	0.9928
$\|\mathbf{r}_{m-3}\|_2/\|\mathbf{r}_{m-4}\|_2$	0.3557	0.5674	0.8804	0.9709	0.9928
$\|\mathbf{r}_{m-4}\|_2/\|\mathbf{r}_{m-5}\|_2$	0.5526	0.6386	0.8804	0.9709	0.9928

Table 8: Residual reduction factor for the last 5 iterations using SSOR ($\omega = 1.5$) solving the Poisson problem on unit cube $\Omega = [0, 1]^3$

n	2^2	2^3	2^4
$\|\mathbf{r}_m\|_2/\|\mathbf{r}_{m-1}\|_2$	0.2504	0.6140	0.8804
$\|\mathbf{r}_{m-1}\|_2/\|\mathbf{r}_{m-2}\|_2$	0.3738	0.6391	0.8804
$\|\mathbf{r}_{m-2}\|_2/\|\mathbf{r}_{m-3}\|_2$	0.4991	0.6355	0.8804
$\|\mathbf{r}_{m-3}\|_2/\|\mathbf{r}_{m-4}\|_2$	0.5772	0.6074	0.8804
$\|\mathbf{r}_{m-4}\|_2/\|\mathbf{r}_{m-5}\|_2$	0.6258	0.5668	0.8804

5. *Report on CPU time required to solve the linear system using SSOR as a BIM.*

   When we discussed the amount of iterations required for the 2D and 3D case, it was very interesting to see that the amount of iterations for the 2D problem and 3D problem where close to each other. When comparing 2D with 3D, the CPU times are also interessting. Again, for the largest n in the 2D case, A is of size $4225x4225$. For the largest n in the 3D case, A is of size $4913x4913$. So although the size of A is bigger for the 3D case, it is still significantly faster then for the 2D case.

Table 9: CPU time against problem size to solve the Poisson problem on unit square $\Omega = [0,1]^2$

n	2^2	2^3	2^4	2^5	2^6
CPU time	0.0157	0.0344	0.5884	17.8612	779.3858

Table 10: CPU time against problem size to solve the Poisson problem on unit cube $\Omega = [0,1]^3$

n	2^2	2^3	2^4
CPU time	0.1195	0.9724	129.6570

6. Solve the linear system using SSOR used as a preconditioner for the CG. For various values of N, make a logarithmic plot of $\frac{\|r_m\|_2}{\|f^h\|_2}$ versus m, compare these graphs with those in the previous section and draw conclusions

   The idea behind preconditioning is to transform the linear system, such that the transformed matrix has a more favourable spectrum. This reduces the iterations required for convergence, compared to the CG method. The CG method is discussed further in Appendix A. The algorithm from the lecture notes is used to write the MATLAB code. We used our own direct solver to solve $M\mathbf{z}^{k-1} = \mathbf{r}^{k-1}$. The results for both the 2D and 3D case are given in Figure 8. There are a several things that can be noted. One that is very obvious is the amount of iterations required. Using SSOR as a BIM, required in the 2D case for $n = 2^6$ around 2500 iterations. When SSOR is used as a preconditioner for the Conjugate-Gradient method, this reduces to around 50 iterations. This is 50 times less then SSOR as BIM. The same behaviour is seen for the 3D case. For instance if we take $n = 2^4$, the amount of iterations are reduced by a factor of 8.

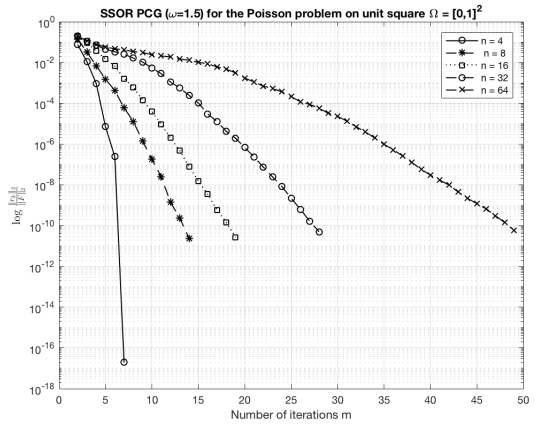
   When we look closely at Figure 8, and compare it to Figure 5 and Figure 6, the rate of convergence differs. For SSOR as a BIM, we concluded that the convergence was linear. For SSOR as preconditioner for the Conjugate-Gradient method, the convergence can be said to bu super-linear. Again when we look at the 2D case for $n = 2^6$, we can see that for the first 20 iterations, the behaviour looks like that for the SSOR as BIM, but after 20 iteration the rate of convergence changes in a favourable way. From the plot, it becomes clear that the residual reduction factor becomes smaller and hence the system is better conditioned.

   Our code for the Preconditioned Conjugate-Gradient method can be found in Appendix E.

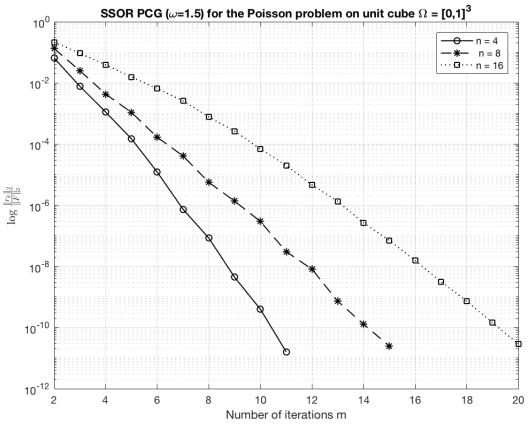
7. Report on CPU time required to solve the linear system using SSOR as a preconditioner and compare these numbers with both theoretical estimates and the CPU time required by SSOR as a basic iterative scheme.

   When we look at the comparison between CPU times for the 2D case in Table 11, we can see a couple of things. When SSOR as BIM is compared to SSOR PCG, we can see that as n increases SSOR PCG becomes significantly faster. Specifically for $n = 2^5$, the SSOR PCG is 50 times faster as SSOR as BIM. This factor however becomes way smaller when $n = 2^6$. Now SSOR PCG is around twice as fast as the BIM variant. It is also interesting to see that for the 2D case, our direct method is faster then SSOR as BIM.

   While for the 2D case SSOR as PCG was faster then it's BIM variant, it is interesting to see that this is completely the opposite for the 3D case. When one looks at Table 12, for $n = 2^4$ SSOR as BIM is around 4.5x faster then its PCG variant. The direct method is the clearly the loser in this comparison.



(a) 2D case



(b) 3D case

Figure 7: SSOR Preconditioned Conjugate-Gradient to solve the system $A\mathbf{x} = \mathbf{f}$. The residual $\frac{\|r_m\|_2}{\|f^h\|_2}$ against the number of iterations m

Table 11: Comparison of CPU times against problem size to solve the Poisson problem on unit square $\Omega = [0, 1]^2$ using PCG

n	2^2	2^3	2^4	2^5	2^6
Cholesky	0.0110	0.0372	0.5226	12.0144	497.6235
SSOR as BIM	0.0157	0.0344	0.5884	17.8612	779.3858
SSOR PCG	0.0241	0.0219	0.1627	0.35330	377.9020

Table 12: Comparison of CPU times against problem size to solve the Poisson problem on unit cube $\Omega = [0, 1]^3$ using PCG

n	2^2	2^3	2^4
Cholesky	0.0988	4.2044	891.5875
SSOR as BIM	0.1195	0.9724	129.6570
SSOR PCG	0.0513	1.0975	596.0785

To compare the CPU times of our code to theoretical estimates, we first need to figure out how to find this theoretical estimate. The theoretical FLOP per second that your machine can calculate is dependent on

(a) Number of cores

(b) Average frequency of CPU

(c) Operations per cycle

Multiplication of these factors will result in the amount of FLOP/s that your machine can do. The MATLAB-scripts were ran on a Intel I7-4870HQ processor. Which has: 4 cores, an average frequency of 3.2 GHz and can do 16 double precision operations per second. This means that this processor can do 204.8 GFLOPS per second.

Table 13: Comparison of number of iterations m against problem size to solve the Poisson problem on unit square $\Omega = [0, 1]^2$

n	2^2	2^3	2^4	2^5	2^6
SSOR as BIM	36	39	169	672	2560
SSOR in PCG	7	14	19	28	49

Table 14: Comparison of number iterations m against problem size to solve the Poisson problem on unit cube $\Omega = [0, 1]^3$

n	2^2	2^3	2^4
SSOR as BIM	37	42	170
SSOR as BIM	11	15	20

From the pen&paper assignments, the amount of FLOPS for SSOR as BIM and PCG are given by Equation 29 and Equation 31, respectively. The same expressions are given for the 3D case in Equation 30 and Equation 32. Using these formulas and the results from Table 13 and Table 14 for $Niter$. One can construct a table for both the 2D and 3D cases, the required flops to solve the problem. This results in tables Table 15 and Table 16.

Table 15: Comparison of number of theoretical number of FLOPS against problem size to solve the Poisson problem on unit square $\Omega = [0, 1]^2$

n	2^2	2^3	2^4	2^5	2^6
SSOR as BIM	7792	45896	912664	15499136	243855616
SSOR in PCG	1890	20580	128250	807240	5834430

Table 16: Comparison of number of theoretical number of FLOPS against problem size to solve the Poisson problem on unit cube $\Omega = [0, 1]^3$

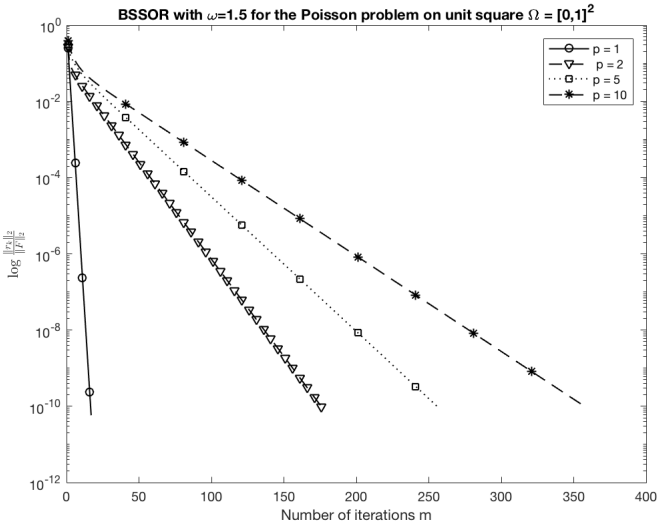
n	2^2	2^3	2^4
SSOR as BIM	32066	461378	18361538
SSOR in PCG	52250	415530	3733880

When we look at tables Table 15 and Table 16, and look at the flops required and the theoretical speed of the CPU, there is a big mismatch between our CPU time results and the theoretical CPU time required. This could be because, in our implementation we don't make use of the sparseness of matrix A. A lot of flops are used for computations for which the result is definitely zero. So a recommendation for improving our implementation is to make use of sparseness. Another mismatch is that in the MATLAB-scripts that we ran, SSOR as BIM in 3D should be way faster then the PCG version. However, when we look at Table 16 you can not see this back. Also when we look at Table 13, for $n = 2^6$ the amount of flops required for SSOR as BIM is around 42x as large compared to SSOR as PG. The same scale is not visible in Table 11, where SSOR PCG is only about twice as fast.

8. Implement BSSOR and give the number of SSOR and BSSOR iterations for nx = 100 and ny = 100. For p we use the values p = 1, 2, 5, 10. Compare the convergence behaviour of SSOR and BSSOR. Only consider the 2D problem.

   Since $n = 100$ was too large, we computed BSSOR for $n = 2^5$.

   Compared to SSOR, the amount of iterations for the same problem size, is in favor of BSSOR. For SSOR this problem took 672 iterations, however when we look at Table 17, the lower p is the less amount of iterations it takes.



(a) 2D case

Figure 8: BSSOR to solve $A\mathbf{x} = \mathbf{f}$ with $n = 2^5$. The residual $\frac{\|r_m\|_2}{\|f^h\|_2}$ against the number of iterations m

Table 17: Comparison of number of iterations m against amount of processors p

p	1	2	5	10
m	17	176	256	358

For the SSOR method the reduction factor for the last 5 iterations was constant at: 0.9709 As you can see from the results in Table 18, as p decreases the convergence becomes faster. We think that this is because how badly BSSOR runs in parallel for lexographic ordering. If for instance a red-black ordering is used, then running $p = 2$, might run faster then $p = 1$. If you want to run BSSOR over multiple processors, one should look in to multicoloring methods.

Our code for the BSSOR can be found in Appendix F.

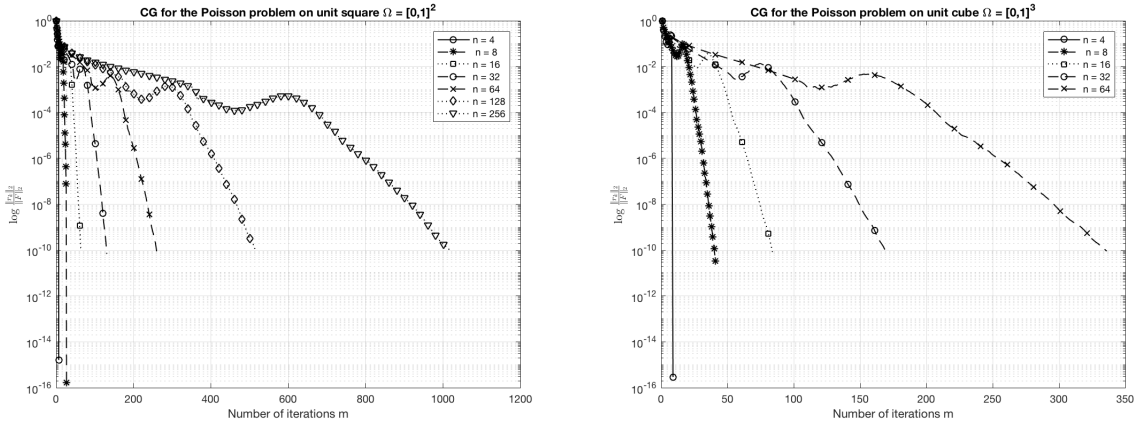
Table 18: Residual reduction factor for the last 5 iterations using BSSOR ($\omega = 1.5$) solving the Poisson problem on unit square $\Omega = [0, 1]^2$ for $n = 2^5$

p	1	2	5	10
$\|\mathbf{r}_m\|_2/\|\mathbf{r}_{m-1}\|_2$	0.2500	0.8893	0.9220	0.9439
$\|\mathbf{r}_{m-1}\|_2/\|\mathbf{r}_{m-2}\|_2$	0.2500	0.8893	0.9220	0.9439
$\|\mathbf{r}_{m-2}\|_2/\|\mathbf{r}_{m-3}\|_2$	0.2500	0.8893	0.9220	0.9439
$\|\mathbf{r}_{m-3}\|_2/\|\mathbf{r}_{m-4}\|_2$	0.2500	0.8893	0.9220	0.9439
$\|\mathbf{r}_{m-4}\|_2/\|\mathbf{r}_{m-5}\|_2$	0.2500	0.8893	0.9220	0.9439

A Conjugate-Gradient method

The Conjugate-Gradient method was not part of this exercise, however due to its similarity we had first programmed the CG method and after that adjusted for the PCG method. It is fascinating how fast the CG method is. The algorithm used is the one that is given in the reader. For the CG method we were able to compute n up to 2^8 for the 2D case and 2^6 for the 3D case. At first we taught it might because we used

our own direct solver to solve $M\mathbf{z}^{k-1} = \mathbf{r}^{k-1}$. However even when we used MATLAB's built in solver: $\mathbf{z}^{k-1} = M\backslash\mathbf{r}^{k-1}$ The CG method was still faster. We think that this has to do that the CG algorithm is implemented with vectorization. For the SSOR as BIM and Cholesky decomposition we used for loops to fill all the elements in a vector. In CG the only for loop is the amount of iterations, the rest is pure vector-vector and matrix-vector multiplication. MATLAB is so fast at this, that even though CG requires more iterations, it still performs better then PCG. So a recommendation if we want to improve our results is to rewrite the code such that for loops are avoided and vectorization is implemented, if this is possible.



(a) 2D case

(b) 3D case

Figure 9: Conjugate-Gradient to solve the system $A\mathbf{x} = \mathbf{f}$. The residual $\frac{\|r_m\|_2}{\|f^h\|_2}$ against the number of iterations m

Table 19: CPU time against problem size to solve the Poisson problem on unit square $\Omega = [0,1]^2$ using CG

n	2^2	2^3	2^4	2^5	2^6	2^7	2^8
CPU time	0.0081	0.0035	0.0057	0.1093	1.5743	25.7835	412.3142

Table 20: CPU time against problem size to solve the Poisson problem on unit cube $\Omega = [0,1]^3$ using CG

n	2^2	2^3	2^4	2^5	2^6
CPU time	0.0085	0.0079	0.1770	5.3239	189.6261

B Source code for construction matrix A and vector f

```
n=input('Number of subintervals=');
d=input('dimension of the problem');
h=1/n; x=0:h:1; y=0:h:1; z=0:h:1;
% the objective is to obtain the symmetric matrix A and vector F
% we are going to use the Kronecker product for carrying out this task
if d==2
   tic
   That=(1/h^2)*(diag([h^2,4*ones(1,n-1),h^2])+diag([0,-1*ones(1,n-2),0],1)+
      diag([0,-1*ones(1,n-2),0],-1));
   Ihat=-(1/h^2)*diag([0,ones(1,n-1),0]);
```

```
    I=eye(n+1);
    % for defining the blocks A_{1,1} and A_{n+1,n+1}
    A=kron(diag([1,zeros(1,n-1),1]),I);
    % for defining the blocks A_{i,i} of the main diagonal i=2,...,n
    A=A+kron(diag([0,ones(1,n-1),0]),That);
    % for defining the blocks A_{i,i-1} of the 1st low diagonal i=2,...,n
    A=A+kron(diag([0,ones(1,n-2),0],-1),Ihat);
    % for defining the blocks A_{i,i+1} of the 1st upper diagonal i=2,...,n
    A=A+kron(diag([0,ones(1,n-2),0],1),Ihat);
    F=[];
    %interior points
    for i=2:n
        for j=2:n
            F(i+(j-1)*(n+1))=(x(i)^2+y(j)^2)*sin(x(i)*y(j));
        end
    end
    % boundaries conditions
    for i=1:n+1
        F(i)=sin(x(i)*y(1));
        F(i+n*(n+1))=sin(x(i)*y(n+1));
    end
    for j=1:n+1
        F(1+(j-1)*(n+1))=sin(x(1)*y(j));
        F(j*(n+1))=sin(x(n+1)*y(j));
    end
    % contribution of the boundaries conditions to the internal points
    for i=2:n
        F(i+(n+1))=F(i+(n+1))+(1/h^2)*sin(x(i)*y(1));
        F(i+(n-1)*(n+1))=F(i+(n-1)*(n+1))+(1/h^2)*sin(x(i)*y(n+1));
    end
    for j=2:n
        F(2+(j-1)*(n+1))=F(2+(j-1)*(n+1))+(1/h^2)*sin(x(1)*y(j));
        F(n+(j-1)*(n+1))=F(n+(j-1)*(n+1))+(1/h^2)*sin(x(n+1)*y(j));
    end
    F=F'; % vertical vector.
  %the exact solution will be
    for i=1:n+1
        for j=1:n+1
            uex(i+(j-1)*(n+1),1)=sin(x(i)*y(j));
        end
    end
    toc
elseif d==3
    tic
    T3=diag([h^2,6*ones(1,n-1),h^2])+diag([0,-1*ones(1,n-2),0],1)+diag([0,-1*
        ones(1,n-2),0],-1);
    Ihat=-diag([0,ones(1,n-1),0]);
    % to define the blocks corresponding to k=1 and k=n+1
    A=kron(kron(diag([1,zeros(1,n-1),1]),eye(n+1)),eye(n+1));
    % to define the blocks of the diagonal
    B=kron(diag([1,zeros(1,n-1),1]),eye(n+1))+(1/h^2)*kron(diag([0,ones(1,n-1)
        ,0]),T3)+(1/h^2)*kron(diag([0,ones(1,n-2),0],-1),Ihat)+(1/h^2)*kron(
        diag([0,ones(1,n-2),0],1),Ihat);
    % to define the diagonal itself (expect k=1 and k=n+1)
    A=A+kron(diag([0,ones(1,n-1),0]),B);
```

```
    % to define the contributions of u_{i,j,k-1} and u_{i,j,k+1}
    C=(1/h^2)*kron(diag([0,ones(1,n-1),0]),Ihat);
    % to define the blocks above and below the diagonal
    A=A+kron(diag([0,ones(1,n-2),0],-1),C)+kron(diag([0,ones(1,n-2),0],1),C);
    %interior points
    F=[];
    for i=2:n
        for j=2:n
            for k=2:n
                F(i+(j-1)*(n+1)+(k-1)*(n+1)^2,1)=(x(i)^2*y(j)^2+x(i)^2*z(k)^2+
                    y(j)^2*z(k)^2)*sin(x(i)*y(j)*z(k));
            end
        end
    end
    % boundary points
    for i=1:n+1
        for j=1:n+1
            %k=1,k=n+1
            F(i+(j-1)*(n+1))=sin(x(i)*y(j)*z(1));
            F(i+(j-1)*(n+1)+n*(n+1)^2)=sin(x(i)*y(j)*z(n+1));
        end
    end
    for i=1:n+1
        for k=1:n+1
            %j=1,j=n+1
            F(i+(k-1)*(n+1)^2)=sin(x(i)*y(1)*z(k));
            F(i+n*(n+1)+(k-1)*(n+1)^2)=sin(x(i)*y(n+1)*z(k));
        end
    end
    for j=1:n+1
        for k=1:n+1
            %i=1,i=n+1
            F(1+(j-1)*(n+1)+(k-1)*(n+1)^2)=sin(x(1)*y(j)*z(k));
            F(n+1+(j-1)*(n+1)+(k-1)*(n+1)^2)=sin(x(n+1)*y(j)*z(k));
        end
    end
    % contributions of the boundary points for the internal points
    for i=2:n
        for j=2:n
            % k=1,k=n+1 contribution to k=2, k=n
            F(i+(j-1)*(n+1)+(n+1)^2)=F(i+(j-1)*(n+1)+(n+1)^2)+(1/h^2)*sin(x(i)
                *y(j)*z(1));
            F(i+(j-1)*(n+1)+(n-1)*(n+1)^2)=F(i+(j-1)*(n+1)+(n-1)*(n+1)^2)+(1/h
                ^2)*sin(x(i)*y(j)*z(n+1));
        end
    end
    for i=2:n
        for k=2:n
            % j=1,j=n+1 contribution to j=2 j=n
            F(i+(n+1)+(k-1)*(n+1)^2)=F(i+(n+1)+(k-1)*(n+1)^2)+(1/h^2)*sin(x(i)
                *y(1)*z(k));
            F(i+(n-1)*(n+1)+(k-1)*(n+1)^2)=F(i+(n-1)*(n+1)+(k-1)*(n+1)^2)+(1/h
                ^2)*sin(x(i)*y(n+1)*z(k));
        end
    end
```

```
    for j=2:n
        for k=2:n
            %i=1,i=n+1 contribution to i=2 i=n
            F(2+(j-1)*(n+1)+(k-1)*(n+1)^2)=F(2+(j-1)*(n+1)+(k-1)*(n+1)^2)+(1/h
                ^2)*sin(x(1)*y(j)*z(k));
            F(n+(j-1)*(n+1)+(k-1)*(n+1)^2)=F(n+(j-1)*(n+1)+(k-1)*(n+1)^2)+(1/h
                ^2)*sin(x(n+1)*y(j)*z(k));
        end
    end
    %the exact solution will be
   for i=1:n+1
       for j=1:n+1
           for k=1:n+1
               uex(i+(j-1)*(n+1)+(k-1)*(n+1)^2,1)=sin(x(i)*y(j)*z(k));
           end
       end
   end
   toc
else
    disp('Dimension must be d=2 or d=3')
end
```

C Source Code Direct solver (using Cholesky Decomposition)

```
% this scrip solve the linear system A*u=F,when A is SPD
% using Cholesky decomposition method.
% it only works properly if A is SPD
A=input('Introduce your matrix A');
F=input('Introduce your vector F');
tf=issymmetric(A);
C=[];
if tf==1
    C(1,1)=sqrt(A(1,1));
    C(2:length(A),1)=A(2:length(A),1)/C(1,1);
    for k=2:size(A)
        C(k,k)=sqrt(A(k,k)-sum(C(k,1:k-1).*C(k,1:k-1)));
        for i=k+1:size(A)
            C(i,k)=(1/C(k,k))*(A(i,k)-sum(C(i,1:k-1).*C(k,1:k-1)));
        end
    end
else
    disp('A must be symmetric')
end
U=transpose(C);
w=[];
w(1,1)=F(1,1)/C(1,1);
for i=2:length(F)
    w(i,1)=(F(i,1)-C(i,1:i-1)*w(1:i-1,1))/C(i,i);
end
u=[];
u(length(F),1)=w(length(F),1)/C(length(F),length(F));
for i=length(F)-1:-1:1
```

```
    u(i,1)=(w(i,1)-U(i,i+1:length(F))*u(i+1:length(F)))/C(i,i);
end
```

D Source code SSOR as Basic Iterative Method

```
function [u,r,mu]=SSOR(A,F,w,tol)
tic
% This function uses the symmetric successive overrelaxation method
% to solve the system A*u=f
% it also gives us the vector r to measure the errors
% the asymptotic rate of convergencem mu
% and the CPU time required to solve the linear system using SSOR as a BIM.
n=length(F);
U=[]; %matrix colum k represent iteration vector u^{k+1}
U(:,1)=zeros(length(F),1); %zero initial guess for u^{h}.
R(:,1)=F; %matrix -> each column
k=1;
r(1)=1;
phi=[];
while norm(R(:,k))/norm(F)>tol
    k=k+1;
    % the forward part is:
    U(1,k)=F(1,1)-A(1,2:n)*U(2:n,k-1);
    U(1,k)=(1-w)*U(1,k-1)+w*U(1,k);
    for i=2:length(F)
        % the damping is performed component wise as well
        U(i,k)=(1-w)*U(i,k-1)+w*(F(i,1)-A(i,1:i-1)*U(1:i-1,k)-A(i,i+1:n)*U(i
            +1:n,k-1))/A(i,i);
    end
    % the backward part is:
    phi(:,k)=U(:,k);
    U(n,k)=F(n,1)-A(n,1:n-1)*phi(1:n-1,k);
    U(n,k)=(1-w)*phi(n,k)+w*U(n,k);
    for i=n-1:-1:1
        % the damping is performed component wise as well
        U(i,k)=(1-w)*phi(i,k)+w*(F(i,1)-A(i,1:i-1)*phi(1:i-1,k)-A(i,i+1:n)*U(i
            +1:n,k))/A(i,i);
    end
    R(:,k)=F-A*U(:,k);
    r(k)=norm(R(:,k))/norm(F);
end
r=r';
% to compute the  asymptotic rate of convergence we proceed as follows:
counter=0;
for i=length(r)-5:length(r)-1
    counter=counter+(r(i+1)/r(i));
end
mu=(counter/5);
scatter(1:k,r,'b')
set(gca,'yscale','log')
hold on
semilogy(1:k,r,'b')
```

```
ylabel('$ \frac{\|r_{k}\|_{2}}{\|F\|_{2}}$','Interpreter','Latex')
xlabel('Number of iterations')
grid
u=U(:,end);
toc
end
```

E Source code PCG

```
function [u,r,k] = SSORpreconditionerCG(A,F,L,LT,tol)

% This function uses the SSOR BIM as a preconditioner for
% the Conjugate Gradient method.
n=length(F);
% Column 1 correspond to iteration 0

U(:,1)=zeros(n,1); R(:,1)=F;
k=1; Z=[]; P=[]; beta=[]; alpha=[];
r = zeros(1,500);
while (norm(R(:,k))/norm(F))>tol
    Z(:,k) = cholesky_solve(L,LT,R(:,k));
    if k==1
        P(:,2)=Z(:,1); %P=[0,p1,p2,..]
    else
        beta(k+1)=(R(:,k)'*Z(:,k))/(R(:,k-1)'*Z(:,k-1));
        P(:,k+1)=Z(:,k)+beta(k+1)*P(:,k);
    end
    alpha(k+1)=(R(:,k)'*Z(:,k))/(P(:,k+1)'*A*P(:,k+1));
    U(:,k+1)=U(:,k)+alpha(k+1)*P(:,k+1);
    R(:,k+1)=R(:,k)-alpha(k+1)*A*P(:,k+1);
    r(k+1)=norm(R(:,k+1))/norm(F);
    k=k+1;
end

r = r(1,1:k);
r=r';

 u=U(:,end);

end
```

F Source code BSSOR as BIM

```
function [u, r, k, mu] = BSSOR(A,F,w, p,tol)

n=length(F);
loc = zeros(p,2);

nd = size(A,1);
```

```
q = fix(nd/p);
kmax = 1000;
% Create matrix with start and end rows for each processor
for i=1:p
    loc(i,1) = (i-1)*q + 1;
    if mod(nd,p)~=0 && i==p
        loc(i,2) = nd;
    else
        loc(i,2) = i*q;
    end
end

U = zeros(size(F));
r = zeros(1,kmax);
R = zeros(size(F,1), kmax);

for k=1:kmax
    %Forward solve
    for i=1:p
        si = loc(i,1);
        ei = loc(i,2);

        Ui = U(si:ei,1);
        Aii = A(si:ei,si:ei);
        sigma = zeros(size(F(si:ei)));

        for j=1:(i-1)
            sj = loc(j,1);
            ej = loc(j,2);
            sigma = sigma + A(si:ei,sj:ej)*U(sj:ej,1);
        end
        for j=(i+1):p
            sj = loc(j,1);
            ej = loc(j,2);
            sigma = sigma + A(si:ei,sj:ej)*U(sj:ej,1);
        end
        B = w.*(F(si:ei,1) - sigma) + (1-w).*(Aii*Ui);
        U(si:ei,1) = Aii\B;
        %U(si:ei,1) = Cholesky_decomp(Aii,B);
    end
    %Backward solve
    for i=p:-1:1
        si = loc(i,1);
        ei = loc(i,2);

        Ui = U(si:ei,1);
        Aii = A(si:ei,si:ei);
        sigma = zeros(size(F(si:ei)));
        for j=1:(i-1)
            sj = loc(j,1);
            ej = loc(j,2);
            sigma = sigma + A(si:ei,sj:ej)*U(sj:ej,1);
        end
        for j=(i+1):p
            sj = loc(j,1);
```

```
            ej = loc(j,2);
            sigma = sigma + A(si:ei,sj:ej)*U(sj:ej,1);
        end
        B = w.*(F(si:ei,1) - sigma) + (1-w).*(Aii*Ui);
        U(si:ei,1) = Aii\B;
        %U(si:ei,1) = Cholesky_decomp(Aii,B);

    end
     R(:,k)=F-A*U(:,1);
     r(1,k)=norm(R(:,k))/norm(F);
    if r(1,k) < tol
        break
    end

%    legend('n = 2', ' n = 4', 'n = 8', 'n = 16', 'n = 32')

end

R = R(:,1:k);
r = r(1,1:k);
      counter=1;
      mu=[];
      for i=length(r):-1:(length(r)-4)
          mu_i = norm(R(:,i)) / norm(R(:,i-1));
          mu(counter) = mu_i;
          counter = counter+1;

      end

u=U(:,1);

end
```