Bachelor Thesis

The algebraic preconditioner for saddle point systems based on the properties of a PDE

By

Joris van der Wagt
5355176

Supervisor:	Dr. Carolina A. Urzúa Torres
Graduation Committee:	Dr. C.A. Urzúa Torres & Prof.dr. H.M. Schuttelaars & Dr. M.E. Kootte
Institution:	Delft University of Technology
Place:	Faculty of Electrical Engineering, Mathematics and Computer Science Delft
Thesis completion:	July 1, 2024

Laymen's summary

In daily life, everything changes constantly, for instance change in temperature, movement or financial markets. This change can be described using differential equations. However, differential equations are very difficult to solve and therefore, we often use a computer algorithm to get an approximation. In order to set up an algorithm, the region, where the differential equation holds, needs to be chopped into segments, or discretised, and for each segment we calculate the solution. If we make the segments really small, we get a better approximation, but the computer algorithm takes longer to find a good approximation.

We still want to find a good approximation for very small segments, so we use a preconditioner in the computer algorithm. The result, the approximation, can be found faster. To find an effective preconditioner, two approaches are known. The first type uses properties of the discretisation and the other uses properties of the differential equation. This thesis found a connection, between the first approach and why it works, based on the second approach.

Summary

Differential equations are commonly solved by, first, discretising the domain and then using an iterative method on the resulting system of equations. Refining the mesh gives a more accurate solution. However, the iterative method does not necessarily converge quickly to a good approximate solution anymore. To deal with this issue, we can add a preconditioner to the system. A good preconditioner enhances the speed of convergence.

Two ways of finding a good preconditioner are known. The first uses the properties of the matrix of the system, that is an algebraic preconditioner. The second uses the properties of the differential equation that give rise to the system, which is an operator preconditioner. This thesis makes a connection between both types of preconditioning.

First, we discretise a differential equation and perform numerical test on the system to see if the algebraic preconditioning works. Then, the domains on which the matrix and preconditioner act are defined in terms of the differential equations.

We conclude that the matrix containing the differential operators acts on $H(\text{div}, \Omega) \times H^1(\Omega)$ and the preconditioner acts on $\left[L^2(\Omega)\right]^2 \times H^2(\Omega)$. So we need to constrain the domains in such a way that the operators both act on the same domain: that is, $H(\text{div}, \Omega) \times H^2(\Omega)$. If the function is in this domain, then we know that the algebraic preconditioner is an effective preconditioner.

Contents

Laymen's summary **i**

Summary **i**

1 Introduction **1**

2 Literature review **2**
2.1 Saddle point systems . . . 2
2.2 Preconditioning . . . 3
2.2.1 Algebraic preconditioning . . . 3
2.2.2 Operator preconditioning . . . 4
2.3 Support-operator finite difference algorithm . . . 5
2.3.1 Discretisation . . . 6
2.3.2 Operator discretisation . . . 7

3 Application of the support-operator finite difference algorithm **9**
3.1 Boundary operator . . . 10
3.2 Gradient operator . . . 10
3.3 Divergence operator . . . 11
3.4 Functionality tests . . . 11

4 Numerical tests **13**
4.1 *GMRES* method without preconditioning . . . 14
4.2 *GMRES* method with algebraic preconditioning . . . 15

5 Properties of the PDE **18**
5.1 Hilbert space of the Laplace operator . . . 19
5.2 Hilbert space of the preconditioner operator . . . 20
5.3 Condition number . . . 21

6 Conclusion and discussion **24**

References **25**

A Spectral equivalence of the LDLT-decomposition **26**

B Python code for functionality tests **27**

C Python code for performing *GMRES* **31**

1 Introduction

Differential equations can be quite difficult to solve, if they can even be solved at all. However, to calculate and predict change in daily life, like the change in temperature, they are essential. Therefore, we solve them by discretising the differential equations and using an iterative method until we are close enough to a good approximate solution. When this happens, we have convergence. The discretisation of the domain in two dimensions can be done by using a finite difference algorithm. The result is a grid with rectangles which will be referred to as grid cells. By setting up an equation for each grid cell, it can be written in matrix form $A\vec{x} = \vec{b}$. Since the solution is only calculated at certain grid cells, one would assume to make the length of the grid cells really small, so we approach an almost continuous solution on the domain. Using very small grid cells will result in more grid cells on the domain, and thus more equations and a larger matrix A. The problem that arises when doing this is an unstable system $A\vec{x} = \vec{b}$.

When using Krylov subspace iterative methods for finding a solution for $A\vec{x} = \vec{b}$, a matrix with a large condition number is not desirable. Convergence bounds for *CG*, which are described in (Wathen, 2015, Ch. 3), *MINRES* in (Wathen, 2015, Ch. 5) and *GMRES* in (Wathen, 2015, Ch. 6), are greater for large condition numbers. According to (Mardal & Winther, 2011, Ch. 7), the condition number $\kappa(A) = \lambda_{\max}/\lambda_{\min}$ increases as the step size decreases, where $\lambda_{\max}$ and $\lambda_{\min}$ are defined as the largest and smallest eigenvalue of matrix A, respectively. Thus, a lot of iterations are required until we converge to a good approximate solution. A system with $\kappa(A) \approx 1$, will have convergence after only a few iterations, because the convergence bounds are really small. So, by setting up a preconditioning matrix P, such that $P^{-1}A$ has $\kappa(P^{-1}A) << \kappa(A)$, the new system $P^{-1}A\vec{x} = P^{-1}\vec{b}$ can be solved a lot quicker.

This thesis takes a closer look at the preconditioning for saddle point systems. (Boffi et al., 2013, Ch. 3) describes saddle point systems as systems that can be written in the form

$$A\vec{x} = \begin{bmatrix} K & B^T \\ B & 0 \end{bmatrix} \begin{bmatrix} y \\ z \end{bmatrix} = \begin{bmatrix} f \\ g \end{bmatrix} = \vec{b}, \tag{1.1}$$

where $K \in \mathbb{R}^{n\times n}$, $B \in \mathbb{R}^{m\times n}$, $0 \in \mathbb{R}^{m\times m}$, and thus $A \in \mathbb{R}^{(n+m)\times(n+m)}$. Preconditioners for these types of problems can be established in two ways, algebraic and operator preconditioning. Algebraic preconditioning is based on the algebraic properties of matrix A as stated in (Axclsson & Vassilevski, 1989, Ch. 1). Whereas operator preconditioning is based on the properties of the differential equation that gives rise to the linear operator blocks in matrix A from equation 1.1 as stated in (Hiptmair, 2006, Sect. 1).

The aim in this thesis is to understand why, in the context of saddle point problems, the algebraic preconditioner P from (Wathen, 2015, Ch. 5) is considered to be a good preconditioner, based on the properties of the partial differential equation. This is done by using Poisson's equation with Dirichlet boundary conditions in two dimensions. That is,

$$-\Delta u = -\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} = f, \tag{1.2}$$

on a rectangular domain $\Omega \subset \mathbb{R}^2$. We approach this problem, by first discretising the Laplace operator using a finite difference algorithm. (Shashkov & Steinberg, 1995, Ch. 2-3) set up a schema for general elliptic problems, which we will apply on equation 1.2. This scheme dissects the Laplace operator into the gradient and divergence operator. Using the matrices representing the gradient and divergence operator, separately, we can transform this into a saddle point system. Numerical tests are performed on the saddle point system to see how fast it converges using the *GMRES* iterative method. This is firstly done without preconditioning and then with preconditioning. Once we have established that the algebraic preconditioning from (Wathen, 2015, Ch. 5) improves the rate of convergence, we approach the saddle point system from the perspective of the operators that give rise to the saddle point system.

2 Literature review

To get a clear image of the research previously done on this topic, we take a closer look on the most important aspects of preconditioning for saddle point systems. First, what saddle point systems are, their properties and how they can be solved will be discussed. Secondly, algebraic preconditioning for saddle point systems and operator preconditioning in the general sense will be presented. Lastly, we delve into how general elliptic PDE's can be discretised using a finite difference algorithm following the approach from (Shashkov & Steinberg, 1995, Ch. 2-3). This discretisation can, in turn, be written as a saddle point system.

2.1 Saddle point systems

A good description where the name of saddle point systems comes from is given by (Rozložník, 2018, Ch. Pref). He states that the name is given because the possible solutions form a horse saddle. According to (Benzi et al., 2005, Ch. 2), this is the result of a minimisation problem subject to certain linear constraints, which can arise in a lot of different fields, such as electromagnetism, fluid dynamics but also economics. The formulation for saddle point problems is given in (Boffi et al., 2013, Ch. 3),

$$\begin{bmatrix} K & B^T \\ B & 0 \end{bmatrix} \begin{bmatrix} y \\ z \end{bmatrix} = \begin{bmatrix} f \\ g \end{bmatrix}. \tag{2.1}$$

In equation 2.1, K and B are matrices derived from discretising the differential equation. K is a matrix in $\mathbb{R}^{n\times n}$, B is a matrix in $\mathbb{R}^{m\times n}$, and thus y, f are vectors in $\mathbb{R}^n$ and z, g in $\mathbb{R}^m$.

An important property of equation 2.1, is the invertibility of the block matrix, which is necessary for a unique solution. (Boffi et al., 2013, Ch. 3) states that the invertibility of A is dependent on the properties of matrix K and B. Recall, that any matrix is invertible, if and only if the matrix is non-singular, if and only if the determinant of the matrix is non-zero. We want to know when this is the case for

$$A = \begin{bmatrix} K & B^T \\ B & 0 \end{bmatrix}. \tag{2.2}$$

Following the approach of (Boffi et al., 2013, Sect. 3.2), equation 2.1 can be rewritten as

$$Ky + B^T z = f, \tag{2.3}$$
$$By = g. \tag{2.4}$$

This has a unique solution if the homogeneous system,

$$Ky + B^T z = 0, \tag{2.5}$$
$$By = 0, \tag{2.6}$$

only has the solution $x = 0$ and $y = 0$. (Boffi et al., 2013, Thm. 3.2.1) gives the following conditions for when this holds. That is,

- the mapping $K_{LL} : L \mapsto L$ is surjective, where $L := Ker(B)$ and $K_{LL} := \pi_k K E_L$, in which π_k is a projection of equation 2.5 and E_L is the extension operator to the subspace L,
- the mapping $B : \mathbb{R}^n \mapsto \mathbb{R}^m$ is surjective.

So if these conditions hold, equation 2.1 is solvable and we can find the unique solutions numerically.

Finding the unique solution for equation 2.1, (Rozložník, 2018, Ch. 5-6) takes two approaches, the direct approach and the iterative approach. The direct approach is only possible for a symmetric positive definite matrix K and full-column rank matrix B. If that holds, (Rozložník, 2018, Sect. 5.2)

uses the Schur complement matrix $S = BK^{-1}B^T$, defined in (Wathen, 2015, Ch. 5), and Cholesky factorisation of matrix K. However, this is computationally expensive and iterative methods are preferred. (Rozložník, 2018, Sect. 6.2) discusses the iterative methods, that we are interested in. This is the Krylov subspace method, which uses preconditioning for fast convergence. The three subclasses of Krylov subspaces are orthogonal residual methods, which uses the conjugate gradient (*CG*) method; minimal residual methods, which uses the minimal residual (*MINRES*) method or general minimal residual (*GMRES*) method; and biorthogonalisation methods, which uses the biconjugate gradient (*Bi-CG*) method or quasi-minimal residual (*QMR*) method.

However, not every iterative method can be used in every situation. *CG* is only applicable for a matrix A that is symmetric and positive definite, *MINRES* is only applicable for a matrix A that is symmetric and indefinite and *GMRES*, *Bi-CG* and *QMR* are applicable for a general matrix A. Therefore, it is necessary to check what type of system we are dealing with, when choosing the iterative method.

2.2 Preconditioning

Krylov subspace iterative methods, used on a general matrix equation $A\vec{x} = \vec{b}$, converge to the solution in a number of steps. However, in order to find a solution such that the error is very small, it can take a lot of iterations. This is where preconditioning comes in. The idea of preconditioning, as in (Wathen, 2015, Ch. 1), is to find a matrix P, such that $P^{-1}A\vec{x} = P^{-1}\vec{b}$ converges faster to the solution than the original equation $A\vec{x} = \vec{b}$.

The speed of convergence determines the number of steps needed to get an approximation that is close enough to the actual solution. An upper bound for the relative error in the kth iterative determines the speed of convergence. For *CG*, for example, (Wathen, 2015, Ch. 3) gives the upper bound for the relative error as

$$\frac{\|\vec{x} - \vec{x}_k\|_A}{\|\vec{x} - \vec{x}_0\|_A} \leq 2\left(\frac{\sqrt{\kappa(A)} - 1}{\sqrt{\kappa(A)} + 1}\right)^k, \tag{2.7}$$

where $\kappa(A) = \lambda_{\max}(A)/\lambda_{\min}(A)$, where $\lambda(A)$ are the eigenvalues of matrix A, and k is the kth iterative. If $\kappa \approx 1$, the upper bound would be very small, thus, the relative error would very small. So if we could find a matrix P, such that

$$\kappa(A) = \frac{\lambda_{max}(A)}{\lambda_{min}(A)} >> \frac{\lambda_{max}(P^{-1}A)}{\lambda_{min}(P^{-1}A)} = \kappa(P^{-1}A), \tag{2.8}$$

the Krylov subspace iterative method would take less iterations until convergence, and consequently less computations.

For equation 2.1, (Wathen, 2015, Ch. 5) recommends to use the *MINRES* method, since A is symmetric, but not necessarily positive definite. To find a preconditioning matrix P, there are two approaches. The first is the algebraic approach, which uses the properties of matrix A. The second is the operator approach, which uses the properties of the differential equation.

2.2.1 Algebraic preconditioning

By the properties of matrix A in equation 2.2, A is non-singular. (Wathen, 2015, Ch. 5) gives the matrix

$$P = \begin{bmatrix} K & 0 \\ 0 & S \end{bmatrix}, \tag{2.9}$$

in which $S = BK^{-1}B^T$ or the Schur complement, as a good preconditioner. (Wathen, 2015, Ch. 5) also states that *MINRES* or *GMRES* terminates after k iterations if the system only has k distinct eigenvalues. By (Murphy et al., 2000, Prop. 1), $P^{-1}A$ only has 3 distinct eigenvalues. So the *MINRES* and *GMRES* method terminates after 3 iterations.

However, the matrix P is practically not that useful, since matrices K and B arise from discretisations. Inverting K and B may be as costly as solving the original system. A better preconditioner could then be constructed by using approximations of K and S, which results in

$$\widehat{P} = \begin{bmatrix} \widehat{K} & 0 \\ 0 & \widehat{S} \end{bmatrix}. \tag{2.10}$$

According to (Wathen, 2015, Ch. 5), the *MINRES* iterative method converges quickly, using the preconditioner from equation 2.10, if

$$\gamma \leq \frac{\vec{x}^T K \vec{x}}{\vec{x}^T \widehat{K} \vec{x}} \leq \Gamma \text{ and } \upsilon \leq \frac{\vec{y}^T S \vec{y}}{\vec{y}^T \widehat{S} \vec{y}} \leq \Upsilon, \tag{2.11}$$

for positive constant γ, Γ, υ and Υ and for all vectors $\vec{x}$, $\vec{y} \in \mathbb{R} \setminus \{0\}$. By (Tyrtyshnikov & Chan, 2000, Lem. 2.1), two Hermitian positive definite matrices A and B are spectrally equivalent if

$$c_1 \left\|\vec{x}\right\|_B^2 \leq \left\|\vec{x}\right\|_A^2 \leq c_2 \left\|x\right\|_B^2, \quad \text{for } 0 < c_1 \leq c_2 \text{ and } \vec{x} \in \mathbb{C}^n, \tag{2.12}$$

$\Longleftrightarrow$

$$c_1 \leq \frac{\vec{x}^T A \vec{x}}{\vec{x}^T B \vec{x}} \leq c_2, \quad \text{for } 0 < c_1 \leq c_2 \text{ and } \vec{x} \in \mathbb{C}^n. \tag{2.13}$$

The bounds from equation 2.11 actually give that, if $\widehat{K}$ and $\widehat{S}$ are approximations of matrices K and S respectively, then $\widehat{K}$ is spectrally equivalent to K and $\widehat{S}$ is spectrally equivalent to S.

2.2.2 Operator preconditioning

Since the operator preconditioning uses information given from the differential equation, it is important to note what space the operators are in. (Hiptmair, 2006, Sect. 1) states that the differential equation gives rise to a linear operator $A : V \mapsto W$, that maps elements from V to W. Now let $B : W \mapsto V$ be another linear operator, then BA maps elements from V to V. (Hiptmair, 2006, Sect. 1) also states that the discretisation of BA gives rise to a well-conditioned matrix, that is $\kappa(B_h A_h)$ is close to 1. And thus, B_h could be a good preconditioner. However, we will get a discrete approximation $A_h : V_h \mapsto W_h$ of A, that connects V_h and W_h. This has to be taken into account for finding a good preconditioner.

Next, (Hiptmair, 2006, Thm. 2.1) defines an upper bound for the smallest and largest eigenvalue. This is done by first defining two reflexive Banach spaces V and W. Since V and W are Banach spaces, (Hiptmair, 2006, Sect. 2) defines the continuous sesquilinear forms $a \in L(V \times V, \mathbb{C})$ and $b \in L(W \times W, \mathbb{C})$. From (Ouhabaz, 2005, Def. 1.1), we know that if a and b are continuous, then

$$|a(u, v)| \leq \alpha_A \left\|u\right\| \left\|v\right\|, \qquad \forall u, v \in V, \tag{2.14}$$

$$|b(q, w)| \leq \alpha_B \left\|q\right\| \left\|w\right\|, \qquad \forall q, w \in W, \tag{2.15}$$

for $\alpha_A, \alpha_B > 0$. So (Hiptmair, 2006, Sect. 2) defines the finite-dimensional subspaces $V_h \subset V$ and $W_h \subset W$, such that a and b satisfy

$$\sup_{v_h \in V_h} \frac{|a(u_h, v_h)|}{\left\|v_h\right\|_V} \geq c_A \left\|u_h\right\|_V, \qquad \forall u_h \in V_h, \tag{2.16}$$

$$\sup_{w_h \in W_h} \frac{|b(q_h, w_h)|}{\left\|w_h\right\|_W} \geq c_B \left\|q_h\right\|_W, \qquad \forall q_h \in W_h, \tag{2.17}$$

for $c_A, c_B > 0$. Then, to connect the subspaces V_h and W_h, (Hiptmair, 2006, Sect. 2) assumes the existence of a continuous sesquilinear form $d \in L(V \times W, \mathbb{C})$ that satisfies

$$\sup_{w_h \in W_h} \frac{|d(v_h, w_h)|}{\left\|w_h\right\|_W} \geq c_D \left\|v_h\right\|_V, \qquad \forall v_h \in V_h \text{ and } c_D > 0. \tag{2.18}$$

And lastly, (Hiptmair, 2006, Sect. 2) introduces the Galerkin-matrices

$$\mathbf{A} = (a(k_i, k_j))_{i,j=1}^{N}, \tag{2.19}$$

$$\mathbf{B} = (b(l_i, l_j))_{i,j=1}^{M}, \tag{2.20}$$

$$\mathbf{D} = (d(k_i, l_j))_{i,j=1}^{N,M}. \tag{2.21}$$

Here $\{k_1, ..., k_N\}$ is a basis for V_h with $N = \dim(V_h)$ and $\{l_1, ..., l_M\}$ is a basis for W_h with $M = \dim(W_h)$. He also defines $A_h : V_h \mapsto V_h'$, $B_h : W_h \mapsto W_h'$ and $D_h : V_h \mapsto W_h'$ as the bounded linear operators associated with the sesquilinear forms a, b and d. From this we can deduce, that

$$\|A_h\|_{V_h \mapsto V_h'} = \alpha_A, \text{ thus } \left\|A_h^{-1}\right\|_{V_h' \mapsto V_h} \leq c_A^{-1}, \tag{2.22}$$

$$\|B_h\|_{W_h \mapsto W_h'} = \alpha_B, \text{ thus } \left\|B_h^{-1}\right\|_{W_h' \mapsto W_h} \leq c_B^{-1}, \tag{2.23}$$

$$\|D_h\|_{V_h \mapsto W_h'} = \alpha_D, \text{ thus } \left\|D_h^{-1}\right\|_{W_h' \mapsto V_h} \leq c_D^{-1}. \tag{2.24}$$

Then by (Hiptmair, 2006, Thm. 2.1), if $\dim(V_h) = \dim(W_h)$, the condition number is bounded by

$$\kappa(\mathbf{D}^{-1}\mathbf{B}\mathbf{D}^{-T}\mathbf{A}) \leq \frac{\alpha_A \alpha_B \alpha_D^2}{c_A c_B c_D^2}. \tag{2.25}$$

The matrix $\mathbf{D}^{-1}\mathbf{B}\mathbf{D}^{-T}\mathbf{A}$ is an endomorphism on V_h, since $\mathbf{D}^{-T}\mathbf{A} : V_h \mapsto W_h$ and $\mathbf{D}^{-1}\mathbf{B} : W_h \mapsto V_h$. The linear operators A and B can thus be discretised by $\mathbf{A}$ and $\mathbf{D}^{-1}\mathbf{B}\mathbf{D}^{-T}$ respectively. The linear operator B is a preconditioner since the condition number is bounded. If the condition number is close to one, it can be considered a good preconditioner.

In this thesis, we do not pursue the Galerkin discretisation. Instead, we will build our matrices with a finite difference algorithm.

2.3 Support-operator finite difference algorithm

Poisson's equation on the domain $\Omega \subset \mathbb{R}^2$ with Dirichlet boundary conditions, as given below,

$$-\Delta u = f \text{ in } \Omega, \tag{2.26}$$

$$u = 0 \text{ on } \partial\Omega, \tag{2.27}$$

is an example of a second order elliptic partial differential equation defined in (Pinchover & Jacob, 2005, Sect. 7.1). A general second order elliptic partial differential equation can be written as

$$-\text{div } \mathcal{K} \text{ grad } u = f, \tag{2.28}$$

in the domain Ω and on the boundary $\partial\Omega$ as

$$\beta(\vec{n}, \mathcal{K} \text{ grad } u) + \alpha u = \gamma, \tag{2.29}$$

given in (Shashkov & Steinberg, 1995, Ch. 1). 2.28 and 2.29 can also be written in operator form, i.e., $\mathcal{A}u = \mathcal{F}$. Thus

$$\mathcal{A}u = \begin{cases} -\text{div } \mathcal{K} \text{ grad } u, & \text{in } \Omega \\ \beta(\vec{n}, \mathcal{K} \text{ grad } u) + \alpha u, & \text{on } \partial\Omega \end{cases}, \tag{2.30}$$

and

$$\mathcal{F} = \begin{cases} f, & \text{in } \Omega, \\ \gamma, & \text{on } \partial\Omega. \end{cases} \tag{2.31}$$

The operator $\mathcal{A}$ can then also be written as $\mathcal{A} = \mathcal{BKC} + \mathcal{D}$, such that $\mathcal{A}u = \mathcal{F}$, where

$$\mathcal{B}\vec{w} = \begin{cases} \text{div } \vec{w}, & \text{in } \Omega, \\ -\beta(\vec{w}, \vec{n}), & \text{on } \partial\Omega, \end{cases} \tag{2.32}$$

$$\mathcal{K}\vec{w} = K\vec{w}, \text{ in } \Omega, \tag{2.33}$$

$$\mathcal{C}u = -\text{grad } u, \text{ in } \Omega, \tag{2.34}$$

$$\mathcal{D}u = \begin{cases} 0, & \text{in } \Omega, \\ \alpha u, & \text{on } \partial\Omega, \end{cases} \tag{2.35}$$

where $\mathcal{C}$ is the adjoint operator of $\mathcal{B}$. (Shashkov & Steinberg, 1995, Ch. 2) uses the Divergence Theorem, as in (Stolze, 1978, Eq. 3), that the following integral identity holds,

$$\int_V \phi \text{ div } \vec{w} \, \mathrm{d}V + \int_V (\vec{w} \text{ grad } \phi) \, \mathrm{d}V = \oint_S \phi(\vec{w}, \vec{n}) \, \mathrm{d}S, \tag{2.36}$$

and if the function u and v are in an arbitrary space H, then the inner product is defined as

$$(u, v)_H = \int_V uv \, \mathrm{d}V + \oint_{\partial V} uv \, \partial V. \tag{2.37}$$

Then it can be shown that

$$(\mathcal{B}\vec{w}, u)_H = \int_V u \text{ div } \vec{w} \, \mathrm{d}V - \oint_{\partial V} u(\vec{w}, \vec{n}) \, \mathrm{d}S, \tag{2.38}$$

$$= -\int_V (\vec{w}, \text{grad } u) \, \mathrm{d}V, \tag{2.39}$$

$$= (\vec{w}, \mathcal{C}u)_H. \tag{2.40}$$

Thus, $\mathcal{C}$ is the adjoint operator of $\mathcal{B}$.

Next, we define

$$\vec{w} = -\mathcal{K} \text{ grad } u, \tag{2.41}$$

where $\vec{w}$ is a vector field, since it is the gradient of u. Then equations 2.28 and 2.29 can be rewritten as a saddle point system of equations,

$$\vec{w} + \mathcal{K} \text{ grad } u = 0 \quad \text{in } \Omega, \tag{2.42}$$

$$\text{div } \vec{w} = f \quad \text{in } \Omega, \tag{2.43}$$

$$-\beta(\vec{w}, \vec{n}) + \alpha u = \gamma \quad \text{on } \partial\Omega. \tag{2.44}$$

2.3.1 Discretisation

To solve these types of partial differential equations numerically, the domain needs to be discretised. Using the approach as mentioned in the previous paragraph, (Shashkov & Steinberg, 1995, Ch. 3), the finite difference scheme can be applied on a two-dimensional rectangular domain, thus $u = u(x, y)$ for $(x, y) \in \Omega$. In the x and y direction, we take uniform steps of size hX and hY, respectively. The grid points are given by indices (i, j), for $1 \leq i \leq N$ and $1 \leq j \leq M$. Given that u is a scalar function, u is discretised in one component, denoted as U. Furthermore, $\vec{w}$ is a vector field and is discretised in two components: one in the x-direction, WX, and one in the y-direction, WY. The discretisation of u is done by setting the value for $U_{(i,j)}$ to be the cell of which the grid point (i, j) is the bottom left vertex, see Figure 2.1.

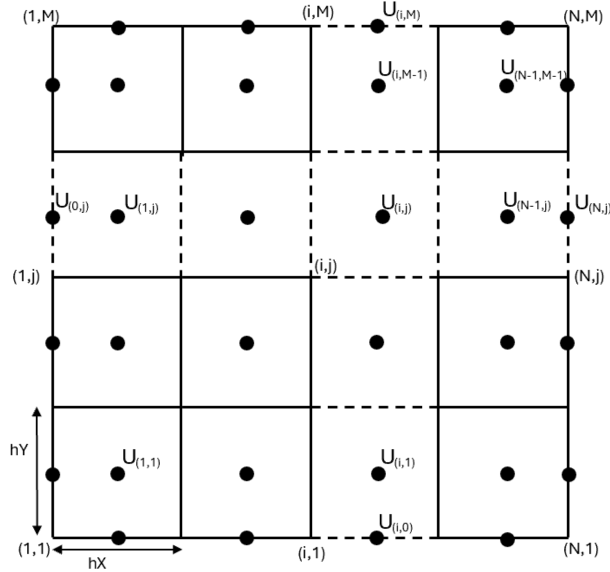

Figure 2.1: Discretisation of scalar function u for $1 \leq i \leq N$ and $1 \leq j \leq M$.

The same thing can be done for vector function $\vec{W} = (WX, WY)$, where WX is located halfway across the edge above grid point (i, j), and WY is located halfway across the edge to the right of grid point (i, j), see Figure 2.2.

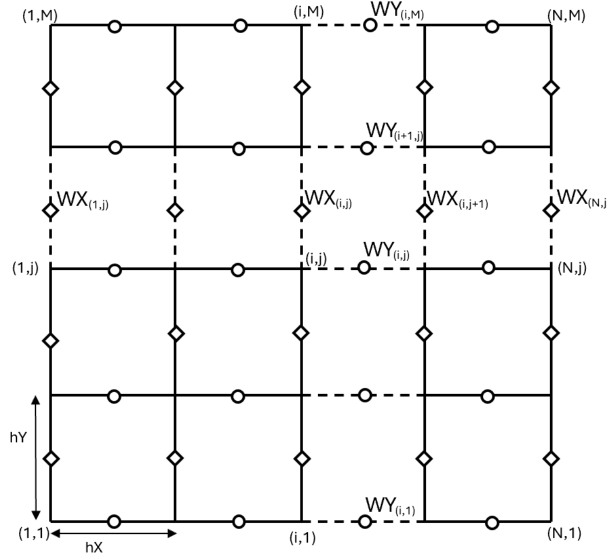

Figure 2.2: Discretisation of vector function $\vec{w}$ for $1 \leq i \leq N$ and $1 \leq j \leq M$.

2.3.2 Operator discretisation

Next, (Shashkov & Steinberg, 1995, Ch. 3) continues to define discrete representations of the operators $\mathcal{B}$, $\mathcal{K}$, $\mathcal{C}$ and $\mathcal{D}$. In equation 2.32, $\mathcal{B}\vec{w} = \text{div } \vec{w}$ in the domain and $\mathcal{B}\vec{w} = -\beta(\vec{w}, \vec{n})$ on the boundary of the domain. Since the divergence of $\vec{w}$ in two dimensions is defined as

$$\text{div } \vec{w} = \begin{bmatrix} \frac{\partial}{\partial x} & \frac{\partial}{\partial y} \end{bmatrix} \begin{bmatrix} w_x \\ w_y \end{bmatrix}, \tag{2.45}$$

$$= \frac{\partial w_x}{\partial x} + \frac{\partial w_y}{\partial y} \tag{2.46}$$

where w_x is the x-component of $\vec{w}$ and w_y is the y-component of $\vec{w}$. (Shashkov & Steinberg, 1995, Sect. 3.3) uses the forward difference for the first order derivative of w_x and w_y. The boundary is defined as $-\beta$-times the boundary vector. This results in

$$\begin{aligned}
(B_h\vec{W})_{(i,j)} &= \frac{WX_{(i+1,j)} - WX_{(i,j)}}{hX} + \frac{WY_{(i,j+1)} - WY_{(i,j)}}{hY}, && \text{for } i = 1, ..., N-1 \text{ and } j = 1, ..., M-1,\\
(B_h\vec{W})_{(i,0)} &= -\beta WY_{(i,1)}, && \text{for } i = 1, ..., N-1,\\
(B_h\vec{W})_{(i,M)} &= \beta WY_{(i,M)}, && \text{for } i = 1, ..., N-1,\\
(B_h\vec{W})_{(0,j)} &= -\beta WX_{(1,j)}, && \text{for } j = 1, ..., M-1,\\
(B_h\vec{W})_{(N,j)} &= \beta WX_{(N,j)}, && \text{for } j = 1, ..., M-1.
\end{aligned} \tag{2.47}$$

The discretisation K_h of $\mathcal{K}$ in 2 dimensions is, as in (Shashkov & Steinberg, 1995, Sect. 3.5), defined as

$$K_h = \begin{bmatrix} KXX & KXY \\ KXY & KYY \end{bmatrix}, \tag{2.48}$$

where the values for $KXX_{(i,j)}$ and $KYY_{(i,j)}$ are defined at the same points as $WX_{(i,j)}$ and $WY_{(i,j)}$ in Figure 2.2. The value for $KXY_{(i,j)}$ is defined at the centre of a cell. If the grid point (i, j) is the bottom left vertex of the cell, then $KXY_{(i+\frac{1}{2},j+\frac{1}{2})}$ is defined at that cell centre.

Since operator $\mathcal{K}$ is multiplied with a vector field, we need to define the matrix-vector product with an arbitrary vector $\vec{v} = (vx, vy)$. This will, for the x-component, be done as

$$\begin{aligned}
(K_h\vec{v})^x_{(i,j)} =& KXX_{(i,j+\frac{1}{2})} vx_{(i,j)} + \frac{1}{2} KXY_{(i-\frac{1}{2},j+\frac{1}{2})} vy_{(i-1,j+\frac{1}{2})} && \text{for } i = 2, ..., N-1\\
&+ \frac{1}{2} KXY_{(i+\frac{1}{2},j+\frac{1}{2})} vy_{(i,j+\frac{1}{2})}, && \text{and } j = 1, ..., M-1,\\
\\
(K_h\vec{v})^x_{(1,j)} =& KXX_{(1,j+\frac{1}{2})} vx_{(1,j)} + KXY_{(\frac{3}{2},j+\frac{1}{2})} vy_{(1,j+\frac{1}{2})}, && \text{for } j = 1, ..., M-1,\\
(K_h\vec{v})^x_{(N,j)} =& KXX_{(N,j+\frac{1}{2})} vx_{(N,j)} + KXY_{(N-\frac{1}{2},j+\frac{1}{2})} vy_{(N-1,j+\frac{1}{2})}, && \text{for } j = 1, ..., M-1,
\end{aligned} \tag{2.49}$$

and for the y-component as

$$\begin{aligned}
(K_h\vec{v})^y_{(i,j)} =& KYY_{(i+\frac{1}{2},j)} vy_{(i,j)} + \frac{1}{2} KXY_{(i+\frac{1}{2},j+\frac{1}{2})} vx_{(i+\frac{1}{2},j)} && \text{for } i = 1, ..., N-1\\
&+ \frac{1}{2} KXY_{(i+\frac{1}{2},j-\frac{1}{2})} vx_{(i+\frac{1}{2},j-1)}, && \text{and } j = 2, ..., M-1,\\
\\
(K_h\vec{v})^y_{(i,1)} =& KYY_{(i+\frac{1}{2},1)} vy_{(i,1)} + KXY_{(i,\frac{3}{2})} vy_{(i+\frac{1}{2},1)}, && \text{for } i = 1, ..., N-1,\\
(K_h\vec{v})^y_{(i,M)} =& KYY_{(i+\frac{1}{2},M)} vy_{(i,M)} + KXY_{(i+\frac{1}{2},M-\frac{1}{2})} vx_{(i+\frac{1}{2},M-1)}, && \text{for } j = 1, ..., M-1.
\end{aligned} \tag{2.50}$$

Next, from equation 2.34 we have $\mathcal{C}u = -\text{grad } u$, then operator $\mathcal{C} = -\text{grad}$. Since

$$\text{grad } u = \begin{bmatrix} \frac{\partial u}{\partial x} \\ \frac{\partial u}{\partial y} \end{bmatrix}, \tag{2.51}$$

$\mathcal{C}u$ produces again a x- and y-component. (Shashkov & Steinberg, 1995, Sect. 3.4) chooses to use the backwards difference for the first order derivative, because we want the derivative of the grid point (i, j) for $1 \leq i \leq N$ and $1 \leq j \leq M$. However, the discretisation of scalar function u, as done in Figure 2.1, creates extra grid points for $i = 0$ and $j = 0$. So, we do not need the derivative for those points. The backwards difference is therefore an obvious choice.

The discretisation C_h of $\mathcal{C}$ can then be established using the following equations for the x-component:

$$(C_hU)^x_{(i,j)} = \frac{U_{(i,j)} - U_{(i-1,j)}}{hX}, \qquad \text{for } i = 2, ..., N-1 \text{ and } j = 1, ..., M,$$
$$(C_hU)^x_{(1,j)} = \frac{U_{(1,j)} - U_{(0,j)}}{hX/2}, \qquad \text{for } j = 1, ..., M,$$
$$(C_hU)^x_{(N,j)} = \frac{U_{(N,j)} - U_{(N-1,j)}}{hX/2}, \qquad \text{for } j = 1, ..., M, \tag{2.52}$$

and the following equations for the y-component:

$$(C_hU)^y_{(i,j)} = \frac{U_{(i,j)} - U_{(i,j-1)}}{hY}, \qquad \text{for } i = 1, ..., N \text{ and } j = 2, ..., M-1,$$
$$(C_hU)^y_{(i,1)} = \frac{U_{(i,1)} - U_{(i,0)}}{hY/2}, \qquad \text{for } i = 1, ..., N,$$
$$(C_hU)^y_{(i,M)} = \frac{U_{(i,M)} - U_{(i,M-1)}}{hY/2}, \qquad \text{for } i = 1, ..., N. \tag{2.53}$$

The last operator $\mathcal{D}$ is defined as 0 in the domain and α on the boundary of the domain. Hence, its discretisation D_h is

$$(D_hU)_{(0,j)} = \alpha_{(0,j)}U_{(0,j)}, \qquad \text{for } j = 1, ..., M,$$
$$(D_hU)_{(N,j)} = \alpha_{(N,j)}U_{(N,j)}, \qquad \text{for } j = 1, ..., M,$$
$$(D_hU)_{(i,0)} = \alpha_{(i,0)}U_{(i,0)}, \qquad \text{for } i = 1, ..., N,$$
$$(D_hU)_{(i,M)} = \alpha_{(i,M)}U_{(i,M)}, \qquad \text{for } i = 1, ..., N. \tag{2.54}$$

Using the matrices B_h, K_h, C_h and D_h, we get that we can approximate $\mathcal{A} = \mathcal{BKC} + \mathcal{D}$ by

$$(B_hK_hC_h + D_h)U = \mathcal{F}. \tag{2.55}$$

3 Application of the support-operator finite difference algorithm

In this thesis, we want to find a connection between the algebraic preconditioner for saddle point systems given in (Wathen, 2015, Ch. 5) and the operator preconditioner that arises from the PDE. To make this connection, we first have to check, if the preconditioner in (Wathen, 2015, Ch. 5) is indeed an effective preconditioner, since we are using a finite difference discretisation, instead of finite elements. By (Pinchover & Jacob, 2005, Sect. 7.1), Poisson's equation with Dirichlet boundary conditions

$$-\Delta u = f \text{ in } \Omega, \tag{3.1}$$
$$u = 0 \text{ on } \partial\Omega, \tag{3.2}$$

is an elliptic PDE. So with the approach from (Shashkov & Steinberg, 1995, Ch. 3), we can discretise Poisson's equation 3.1 using the support-operator finite difference algorithm. This discretisation can then be written as a saddle point problem.

For the discretisation from (Shashkov & Steinberg, 1995, Ch. 3), equation 3.1 must be written in the form of equation 2.28. Note that the divergence is defined in equation 2.45 and the gradient in equation 2.51. Then,

$$\text{div (grad } u) = \begin{bmatrix} \frac{\partial}{\partial x} & \frac{\partial}{\partial y} \end{bmatrix} \begin{bmatrix} \frac{\partial u}{\partial x} \\ \frac{\partial u}{\partial y} \end{bmatrix}, \tag{3.3}$$
$$= \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^x}, \tag{3.4}$$
$$= \Delta u. \tag{3.5}$$

Thus, equation 3.1 is of the form

$$-\text{div}\ (\text{grad}\ u) = f, \tag{3.6}$$

which is the same as equation 2.28 where $\mathcal{K}$ is the identity. If we discretise $\mathcal{K}$, it just stays the identity matrix so we can neglect this. The boundary equation 3.2 is already of the form of equation 2.29, where $\alpha = 1$, $\beta = 0$ and $\gamma = 0$.

3.1 Boundary operator

The matrix representation D_h of operator $\mathcal{D}$ is done by the equations from 2.54. Since $\alpha = 1$, we get a matrix which is all zero except for the boundary grid points, those get value 1. However, we know from equation 3.2, that u is zero on the boundary and those values of u are known. Therefore, $U_{(i,j)}$, for $i = 0$, $i = N$, $j = 0$ or $j = M$, are not used in the discretised u. The operator $\mathcal{D}$ will be represented as a matrix, so it will be more practical to transform the discretised u into a vector, $\overrightarrow{U}$. Vector $\overrightarrow{U}$ consists only of interior points. That is,

$$\overrightarrow{U} = \begin{bmatrix} U_{(1,1)} \\ U_{(2,1)} \\ \vdots \\ U_{(N-1,1)} \\ U_{(1,2)} \\ \vdots \\ U_{(N-1,2)} \\ \vdots \\ U_{(N-1,M-1)} \end{bmatrix}. \tag{3.7}$$

One can transform this vector back in Figure 2.1. The first $N-1$ indiceds are the bottom row from left to right. The next $N-1$ indices are the next row etc. Then $(D\overrightarrow{U})_{(i,j)} = 0$ for $1 \leq i \leq N$ and $1 \leq j \leq M$ and the matrix D_h representing the operator $\mathcal{D}$ is a zero matrix. Thus we can neglect it.

3.2 Gradient operator

The matrix C_h, representing gradient operator $\mathcal{C}$, stems from equations 2.52 for the x-component and 2.53 for the y-component for $1 \leq i \leq N$ and $1 \leq j \leq M$. In the equations where $i = N$ or $j = M$, we have to use boundary values of U. We know these are zero, therefore they can be neglected. Thus, the same vector $\overrightarrow{U}$ as in equation 3.7 is used. The result is 2 matrices, C_h^x and C_h^y, of size $[N*M] \times [(N-1)*(M-1)]$. By stacking C_h^x and C_h^y on top of each other, matrix C_h is created, which is a matrix of size $[2*N*M] \times [(N-1)*(M-1)]$, $C_h \in \mathbb{R}^{[2*N*M]\times[(N-1)*(M-1)]}$.

$C_h\overrightarrow{U}$ returns a column vector $\overrightarrow{C_U}$ of length $2*N*M$. In this vector, the first $N*M$ elements are the x-derivative of the point $U_{(i,j)}$ for $1 \leq i \leq N$ and $1 \leq j \leq M$. It follows the same convention from equation 3.7, where i first goes from 1 to N as j stays 1. Then j increases by one and i goes again from 1 to N etc. up until the last element, which is the derivative at the point (N, M). The second $N*M$ elements are the y-derivative of the point $U_{(i,j)}$ for $1 \leq i \leq N$ and $1 \leq j \leq M$.

In order to find the gradient of the point $U_{(i,j)}$, one must use the $[i+(j-1)*(M-1)]$th element for the x-derivative and the $[((N-1)*(M-1))*(i+(j-1)*(M-1))]$ for the y-derivative of the column

vector $\overrightarrow{C_U}$. Thus,

$$-\text{grad } u(x_i, y_j) = \begin{bmatrix} \frac{\partial u(x_i,y_j)}{\partial x} \\ \frac{\partial u(x_i,y_j)}{\partial y} \end{bmatrix}, \tag{3.8}$$

$$= (C_h \overrightarrow{U})_{(i,j)}, \tag{3.9}$$

$$= \begin{bmatrix} \overrightarrow{C_U}_{[i+(j-1)*(M-1)]} \\ \overrightarrow{C_U}_{[((N-1)*(M-1))*(i+(j-1)*(M-1))]} \end{bmatrix}. \tag{3.10}$$

3.3 Divergence operator

The matrix B_h, representing divergence operator $\mathcal{B}$, is used for a matrix-vector product $B_h\overrightarrow{W}$. From equations 2.47, we see that that the x- and y-component of $\overrightarrow{W} = (WX, WY)$ are, for the interior points of the domain, used in the same equation. To be able to represent the divergence as a matrix, we set

$$\overrightarrow{W} = \begin{bmatrix} WX_{(1,1)} \\ \vdots \\ WX_{(N,M)} \\ WY_{(1,1)} \\ \vdots \\ WY_{(N,M)} \end{bmatrix}, \tag{3.11}$$

where the same order of numbering has been used as in equation 3.7. (Shashkov & Steinberg, 1995, Sect. 3.3) does have equations for the divergence on the boundary i.e. $i = 0$, $i = N$, $j = 0$ or $j = M$, since the discretisation of u creates the boundary values of U. However, in our case $\beta = 0$, thus the equation if $i = 0$, $i = N$, $j = 0$ or $j = M$, becomes zero and we can neglect these cases. Therefore, the only non-zero equations are at the interior grid points. The representation of operator $\mathcal{B}$ becomes matrix B_h of size $[(N-1)*(M-1)] \times [2*N*M]$, thus $B_h \in \mathbb{R}^{[(N-1)*(M-1)]\times[2*N*M]}$.

3.4 Functionality tests

Before we can perform numerical tests with this discretisation, we need to make sure that the matrices B_h and C_h produce the same result as the operator that they represent. The Python script can be found in Appendix B. We choose two very simple functions of which we know what the result is of applying the minus Laplace operator. That is, $u_1(x,y) = 1$ and $u_2(x,y) = xy$. Since the Laplace operator takes the second order derivative in the x- and y-direction, both $-\Delta u_1 = 0$ and $-\Delta u_2 = 0$. Thus, by equation 2.55, both

$$B_h * (C_h * U_1) = 0, \tag{3.12}$$

$$B_h * (C_h * U_2) = 0. \tag{3.13}$$

This can be seen below, in Figure 3.1.

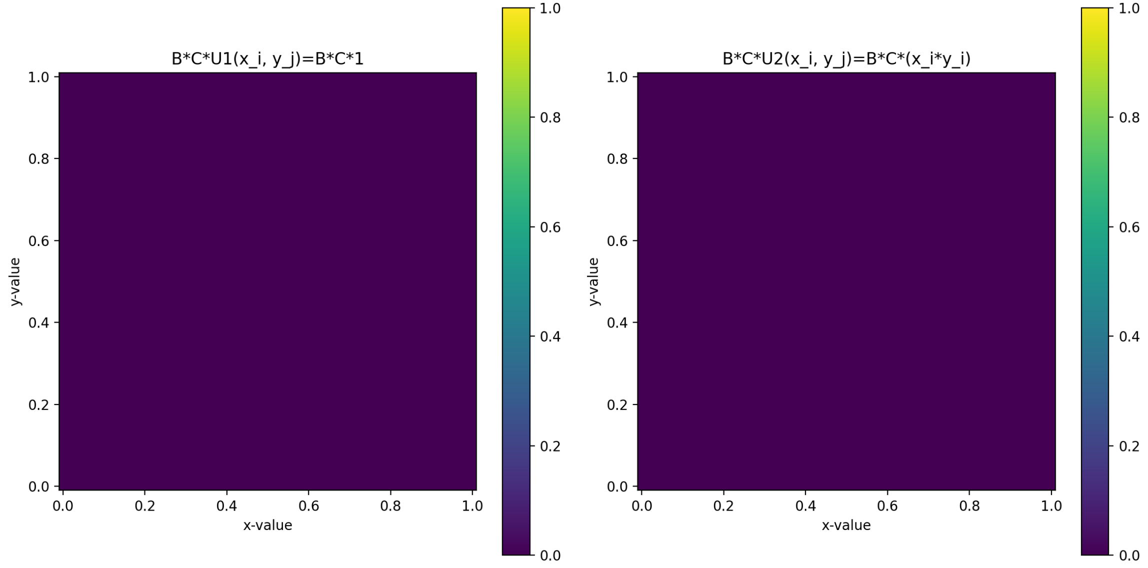

Figure 3.1: $B_h{*}(C_h{*}U_1(x_i, y_i)) = \overrightarrow{F_1}$ (left) and $B_h{*}(C_h{*}U_2(x_i, y_i)) = \overrightarrow{F_2}$ (right) with $hX = hY = 1/50$.

The resulting $\overrightarrow{F}$ is zero everywhere, which is what we wanted. The discretisation of the gradient and divergence operator act how we wanted them to work.

Finally, we also test it on a function that is zero on the boundary. So it can be used on the domain Ω. That is,

$$u_3(x, y) = \sin(\pi x)\sin(\pi y). \tag{3.14}$$

Applying the minus Laplace operator gives,

$$-\Delta u_3(x, y) = -\frac{\partial^2 u_3}{\partial x^2} - \frac{\partial^2 u_3}{\partial y^2}, \tag{3.15}$$

$$= \pi^2 \sin(\pi x)\sin(\pi y) + \pi^2 \sin(\pi x)\sin(\pi y), \tag{3.16}$$

$$= 2\pi^2 \sin(\pi x)\sin(\pi y). \tag{3.17}$$

So $B_h * (C_h * U_3(x_i, y_j))$ should be zero on the boundary and $2\pi^2$ as high as the original function $u_3(x, y)$. The result is given below in Figure 3.2.

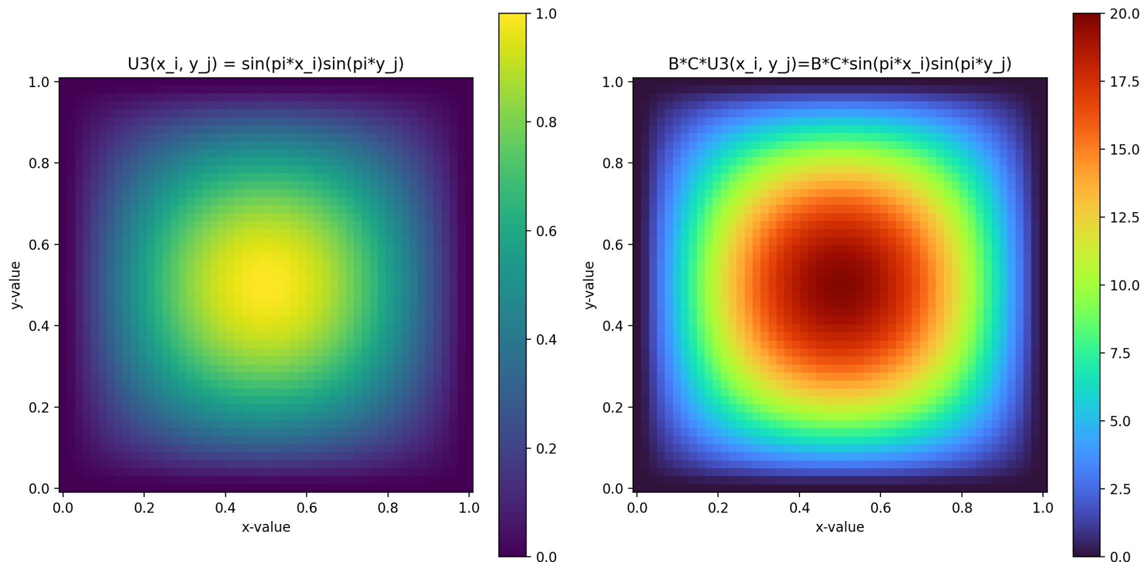

Figure 3.2: $u_3(x, y) = \sin(\pi x)\sin(\pi y)$ (left) and $B_h * (C_h * U_3(x_i, y_i)) = \overrightarrow{F_3}$ (right) with $hX = hY = 1/50$.

We can see that the function $u_3(x, y)$ grows in the centre but does not get a value larger than 1. $\overrightarrow{F_3}$ almost reaches a value of 20, which is indeed, $2\pi^2$ larger than the original function. From this, we can conclude that the matrix $A_h = B_h * -C_h$ represents the Laplace operator.

The system of equations in the domain, 2.42 and 2.43, can be written in matrix form. That is,

$$A_h\vec{x} = \begin{bmatrix} I_h & -C_h \\ B_h & 0 \end{bmatrix} \begin{bmatrix} \overrightarrow{W_h} \\ \overrightarrow{U_h} \end{bmatrix} = \begin{bmatrix} \vec{0} \\ \vec{F} \end{bmatrix} = \vec{b}, \tag{3.18}$$

where I_h is the identity matrix, 0 is a zero matrix and $\overrightarrow{W_h}$ and $\overrightarrow{U_h}$ are the result of solving the matrix equation. The block matrices are elements defined in

$$I_h \in \mathbb{R}^{[2*N*M]\times[2*N*M]}, \tag{3.19}$$

$$C_h \in \mathbb{R}^{[2*N*M]\times[(N-1)*(M-1)]}, \tag{3.20}$$

$$B_h \in \mathbb{R}^{[(N-1)*(M-1)]\times[2*N*M]}, \tag{3.21}$$

$$0 \in \mathbb{R}^{[(N-1)*(M-1)]\times[(N-1)*(M-1)]}. \tag{3.22}$$

The column vectors are elements in

$$\overrightarrow{W_h} \in \mathbb{R}^{[2*N*M]}, \tag{3.23}$$

$$\vec{0} \in \mathbb{R}^{[2*N*M]}, \tag{3.24}$$

$$\overrightarrow{U_h} \in \mathbb{R}^{[(N-1)*(M-1)]}, \tag{3.25}$$

$$\vec{F} \in \mathbb{R}^{[(N-1)*(M-1)]}. \tag{3.26}$$

We want matrix A_h to comply with the condition from (Boffi et al., 2013, Thm. 3.2.1). That is, I_h is a surjective mapping, which it obviously is and B_h is a surjective mapping. By the discretisation, the matrix B_h is of full rank, so this also holds. Note that, in (Boffi et al., 2013, Thm. 3.2.1), the upper right block is the transpose of B_h. By the discretisation from (Shashkov & Steinberg, 1995, Ch. 2-3), this is not the case. So we need to check that C_h is an injective mapping. By the discretisation, the matrix C_h is of full rank, so this also holds. Thus, we can find the unique solutions numerically.

4 Numerical tests

In the previous section, we established the matrix equation 3.18 by using the support-operator finite difference algorithm from (Shashkov & Steinberg, 1995, Ch. 2-3). However, the matrix A_h is not a symmetric matrix since $B_h \neq -C_h^T$. The *MINRES* iterative method can not be used to solve this system, so we use the *GMRES* iterative method.

First, we try to solve equation 3.18 by using the GMRES method. This equation is set up in Python, which can be found in appendix C. First, the matrices I_h, C_h, B_h and 0 are made with the shapes as defined in section 3.4. Then, the matrices are put together to create matrix A_h. Vector $\vec{b}$ is created by setting up a zero column vector of length $2*N*M$ and column vector $\vec{F}$ of length $(N-1)*(M-1)$ and stacking the first onto the second. Column vector $\vec{F}$ is determined by defining a function $u(x, y)$ on a square domain that is zero on the boundary. Then, $-\Delta u = f$, so $\vec{F}$ is the discretised representation of f.

For the remainder of this thesis, we choose the domain Ω to be a subset of $\mathbb{R}^2$ defined by

$$\Omega = \left\{(x, y) \in \mathbb{R}^2 : 0 \leq x \leq 1 \text{ and } 0 \leq y \leq 1\right\}. \tag{4.1}$$

Let the function u be defined as

$$u(x, y) = \sin(\pi x)\sin(\pi y), \text{ for } 0 \leq x \leq 1,\ 0 \leq y \leq 1. \tag{4.2}$$

Then $u(x,y) = 0$, if $x = 0$, $x = 1$, $y = 0$ or $y = 1$. We take a uniform step size hX in the x-direction and hY in the y-direction. For simplicity, we choose $hX = hY$, so we get $N = M$. The discretisation of u is then defined as $u(x_i, y_j) = U(i,j)$ for $1 \leq i \leq N$ and $1 \leq j \leq M$, where $U(1,1)$ is in the middle of the cell in the left bottom of Figure 2.1. Thus $U(1,1) = u(x_1, y_1) = u(hX/2, hY/2)$. In general,

$$
\begin{aligned}
U(i,j) &= u(x_i, y_j), && \text{for } 1 \leq i \leq N-1,\ 1 \leq j \leq M-1, && (4.3)\\
&= u(hX/2 + (i-1) * hX, hY/2 + (j-1) * hY), && \text{for } 1 \leq i \leq N-1,\ 1 \leq j \leq M-1. && (4.4)
\end{aligned}
$$

We can approximate $\overrightarrow{F}$ by,

$$
\begin{aligned}
f =& -\Delta u, && (4.5)\\
=& 2\pi^2 \sin(\pi x) \sin(\pi y). && (4.6)
\end{aligned}
$$

This derivation is given in Section 3.4. So

$$
\begin{aligned}
F_{(i,j)} &= 2\pi^2 \sin(\pi x_i) \sin(\pi y_j), && (4.7)\\
&= 2\pi^2 U(i,j). && (4.8)
\end{aligned}
$$

4.1 *GMRES* method without preconditioning

Now that everything is set up, we can use the *GMRES* method from the package *Scipy* version 1.7.3. Matrix A_h and column vector $\vec{b}$ are, naturally, set as matrix A and column vector $\vec{b}$ from equation 3.18. By not defining a starting guess, the initial guess will be set to a zero column vector. Furthermore, the function *counter* is made for callback, which stores the iteration number and the relative residual norm for each iteration. The standard tolerance for the absolute error is set to 10^{-5} with a maximum number of iterations of $10,000$. And finally, restart is set to *None*, so that we have a higher chance of convergence to an approximate solution.

For the step sizes $hX = hY = 1/400$, $1/800$, $1/1,600$ and $1/3,200$, we perform the *GMRES* iterative method which yields the following results shown in Table 4.1.

Step size hX	Step size hY	# of iterations	Relative residual norm $\|r_k\|$	Measurement error $\|e_k\|$
1/400	1/400	2	$5.284843388958746 * 10^{-12}$	0.007853719213363829
1/800	1/800	2	$1.87561851480186 * 10^{-11}$	0.003926958014194687
1/1,600	1/1,600	12	$6.0398347298582225 * 10^{-6}$	0.001963491308030413
1/3,200	1/3,200	18	$6.831171635516434 * 10^{-6}$	0.0009817471916336502

Table 4.1: *GMRES* iterative method for the system $A_h\vec{x} = \vec{b}$ without preconditioning.

We observe that for all the step sizes, we converge to an approximate solution. The measurement error $\|e_k\|$, defined as

$$
\begin{aligned}
\|e_k\| &= \left\| \overrightarrow{U} - \overrightarrow{U_h} \right\|_\infty, && (4.9)\\
&= \max\left(\left| \overrightarrow{U} - \overrightarrow{U_h} \right| \right). && (4.10)
\end{aligned}
$$

is halved, if the step sizes is also halved. So as we define more grid cells, the solution becomes more accurate. The relative residual norm, defined as

$$
\|r_k\| = \frac{\left\| \vec{b} - A_h \vec{x_k} \right\|_2}{\left\| \vec{b} \right\|_2}, \tag{4.11}
$$

does become larger, especially for the step sizes $1/1,600$ and $1/3,200$. The number of iterations needed to converge to a good approximate solution for the step sizes $1/1,600$ and $1/3,200$ is significantly higher. Therefore, if we continue to decrease the step size, eventually we will not converge any longer to an approximate solution within 10,000 iterations. Thus, preconditioning is needed to achieve convergence to an approximate solution.

4.2 *GMRES* method with algebraic preconditioning

Now that we have established that preconditioning is necessary for small step sizes, we return to the paper of (Wathen, 2015, Ch. 5). For the system

$$A\vec{x} = \begin{bmatrix} K & B^T \\ B & 0 \end{bmatrix} \begin{bmatrix} \vec{w} \\ u \end{bmatrix} = \begin{bmatrix} 0 \\ f \end{bmatrix} = \vec{b} \tag{4.12}$$

as in equation 2.1, he gives a preconditioner if A is nonsingular. That is,

$$P = \begin{bmatrix} K & 0 \\ 0 & S \end{bmatrix}, \text{ where } S = BK^{-1}B^T. \tag{4.13}$$

To put this preconditioner P to use, we must invert P. The inverse of K and S can be computationally expensive for small step sizes. Therefore, we use approximate matrices $\widehat{K}$ and $\widehat{S}$ for matrices K and S and the approximate matrices $\widehat{K}$ and $\widehat{S}$ are easy to invert.

Since we used a finite difference algorithm, instead of a finite elements algorithm that is used in (Wathen, 2015, Ch. 5), we must check if the same algebraic preconditioner is also valid for a finite differences algorithm. Since the system resulting from the finite difference algorithm is given by

$$A_h\vec{x} = \begin{bmatrix} I_h & -C_h \\ B_h & 0 \end{bmatrix} \begin{bmatrix} \vec{W} \\ \vec{U} \end{bmatrix} = \begin{bmatrix} \vec{0} \\ \vec{F} \end{bmatrix} = \vec{b}, \tag{4.14}$$

where $-C_h \neq B_h^T$ as equation 4.12, so A_h is not symmetric. The preconditioner for this system is then given by

$$P_h = \begin{bmatrix} I_h & 0 \\ 0 & S_h \end{bmatrix}, \text{ where } S_h = B_h * -C_h. \tag{4.15}$$

To make sure P_h is a good preconditioner, we perform the *GMRES* for large step sizes, since the matrices I_h and S_h can still be easily inverted. The inverse of I_h is the identity matrix, so we only need to invert S_h. We choose step sizes 1/12.5, 1/25, 1/50 and 1/100, because an even smaller step size is computationally too expensive. The results of the system in 4.14, can be found in Table 4.2 and the results with preconditioner, from 4.15, can be found in Table 4.3.

Step size hX	Step size hY	# of iterations	Relative residual norm $\|r_k\|$	Measurement error $\|e_k\|$
1/12.5	1/12.5	60	$8.750497942317271 * 10^{-6}$	0.24682060020751093
1/25	1/25	2	$2.0254961840853918 * 10^{-14}$	0.12483757379962887
1/50	1/50	2	$6.23641198379237 * 10^{-14}$	0.06269756800845525
1/100	1/100	2	$3.325572579050993 * 10^{-13}$	0.03139913382814065

Table 4.2: *GMRES* iterative method for the system $A_h\vec{x} = \vec{b}$ with large step sizes without preconditioning.

Step size hX	Step size hY	# of iterations	Relative residual norm $\|r_k\|$	Measurement error $\|e_k\|$
1/12.5	1/12.5	2	$3.397249763949668 * 10^{-17}$	0.24682043591023792
1/25	1/25	2	$1.2291530700576463 * 10^{-16}$	0.12483757379962904
1/50	1/50	2	$2.756916605462261 * 10^{-16}$	0.06269756800845519
1/100	1/100	2	$6.727883399682107 * 10^{-16}$	0.031399133828140455

Table 4.3: *GMRES* iterative method for the preconditioned system $P_h^{-1} A_h \vec{x} = P_h^{-1} \vec{b}$ with large step sizes, where P_h^{-1} is the exact inverse of P_h.

The first thing that pops out, is the number of iterations necessary in Table 4.2 for a step size of 1/12.5. This is because the step size is too large. Furthermore, we see in Table 4.3 that, even though we still have the same number of iterations, the relative residual norm is much smaller. Thus, the preconditioner P_h as in equation 4.15, is indeed a good preconditioner.

Matrix S_h is a product of matrices B_h and C_h, which arise from the discretisation of the divergence and the gradient. Therefore, they are both sparse matrices. An approximation of matrix S_h that solves systems of equations quickly, is the incomplete LU-decomposition. We take the incomplete, instead of the complete LU-decomposition, since S_h is sparse. The resulting lower and upper triangular matrices L and U are not necessarily sparse, which can really slow down computations. In the incomplete LU, we can specify a drop tolerance and a fill-in factor. The drop tolerance specifies when a value is too small, then it will be set to 0. The fill-in factor gives the maximum growth of non-zero entries in the matrices L and U. We choose the drop tolerance to be 10^{-4} and the fill-in factor to be 10. The incomplete LU-decomposition can not be used to speed up the calculation of $A\vec{x} = LU\vec{x} = \vec{b}$, since this is not exact. However, it is close enough to the real LU, to be used in a approximation of $A\vec{x} = \vec{b}$ from an iterative method, like *GMRES*. From (Chow & Saad, 1997, Ch. 6), it is known that the incomplete LU-decomposition of P_h should be a good preconditioner.

The package *Scipy* has a built-in command for the incomplete LU-decomposition, which returns a linear operator. From this linear operator, we can easily extract the matrices L and U, but then we still need to invert L and U. To be able to do this, we first establish the matrix P_h and take the incomplete LU-decomposition of the matrix P_h. Then, the resulting linear operator can be entered as preconditioner in the *GMRES* from *Scipy*. We use the same input for *GMRES* as defined in section 4.1. The result is given in Table 4.4

Step size hX	Step size hY	# of iterations	Relative residual norm $\|r_k\|$	Measurement error $\|e_k\|$
1/400	1/400	1,040	$1.26927754724922 * 10^{-10}$	0.007853707163769371
1/800	1/800	4,246	$2.969129032935652 * 10^{-11}$	0.003926951060014776
1/1,600	1/1,600	10,000	$2.4546008870070745 * 10^{-6}$	0.7631598991312758
1/3,200	1/3,200	...	...	...

Table 4.4: *GMRES* iterative method for the preconditioned system $\widehat{P_h^{-1}} A_h \vec{x} = \widehat{P_h^{-1}} \vec{b}$, where $\widehat{P_h}$ is the incomplete LU-decomposition of P_h.

For the step sizes 1/400 and 1/800, we converge to an approximate solution, but only after around 1,000 and around 4,000 iterations, respectively. For the step size 1/1,600, we did not converge before 10,000 iterations. Note, that this measurement error is also way bigger than the measurement error in Table 4.1. This is, because we dit not reach convergence. Our relative residual norm is smaller than the tolerance, but the tolerance is set for the absolute residual norm. For the step size 1/3,200, our script crashes, because it could not store the linear operator of the incomplete LU-decomposition for P_h.

Since our script crashes for a step size of 1/3,200, we look for a different way to invert matrix S_h. If we can approximate S_h by a diagonal matrix, then this $\widehat{S_h}$ is easy to invert. By numerical testing, we

see that S_h is symmetric with all positive eigenvalues. So we can apply the $LDLT$-decomposition for S_h. We want

$$S_h = L\widehat{D}L^T, \tag{4.16}$$

where L is a lower triangular matrix with ones on the diagonal and $\widehat{D}$ is a diagonal matrix. First, we take the incomplete LU-decomposition for sparse matrices such that

$$S_h \approx LU. \tag{4.17}$$

Then L is already a lower triangular matrix with ones on the diagonal and U is an upper triangular matrix. Now let

$$U = \widehat{D}L^T, \tag{4.18}$$

multiplying both sides on the right side by the inverse of L^T gives

$$\widehat{D} = UL^{-T}. \tag{4.19}$$

However, we still need to take an inverse of L^T, which is not desirable since it is computationally expensive.

Since $\widehat{D}$ is a diagonal matrix, U an upper triangular matrix and L^{-T} an upper triangular matrix, the diagonal element $\widehat{D}_{(i,i)}$ is determined by multiplying the diagonal element $U_{(i,i)}$ by the diagonal element $L^{-T}_{(i,i)}$. So we only need to find the inverse of the diagonal elements, which is done by dividing the diagonal elements by itself. The inverse of matrix $\widehat{D}$ can be computed rather easily.

Since the diagonal of L^{-T} already exists of ones, the diagonal elements of $\widehat{D}$ are given by the diagonal elements of U. An approximation of S_h is given by

$$\widehat{S_h} \approx \widehat{D} = \begin{cases} U_{(i,j)} & \text{for } i = j = 1, ..., (N-1)*(M-1), \\ 0 & \text{for } i \neq j, \end{cases} \tag{4.20}$$

where U is an upper triangular matrix in $\mathbb{R}^{(N-1)*(M-1)\times(N-1)*(M-1)}$ from the incomplete LU-decomposition of S_h.

An approximation of our preconditioner P_h is then given by

$$\widehat{P_h} = \begin{bmatrix} I_h & 0 \\ 0 & \widehat{D} \end{bmatrix}. \tag{4.21}$$

Using this preconditioner with the *GMRES* method from *Scipy* yields the results in Table 4.5.

Step size hX	Step size hY	# of iterations	Relative residual norm $\|r_k\|$	Measurement error $\|e_k\|$
1/400	1/400	10,000	$9.755737318079203 * 10^{-8}$	0.052486945147828146
1/800	1/800	10,000	$4.646783984037046 * 10^{-7}$	0.9994713910433102
1/1,600	1/1,600	10,000	$1.1647152655330881 * 10^{-7}$	0.9999036299529601
1/3,200	1/3,200	10,000	$2.91394828943372 * 10^{-8}$	0.9999738398465304

Table 4.5: *GMRES* iterative method for the preconditioned system $\widehat{P_h^{-1}}A_h\vec{x} = \widehat{P_h^{-1}}\vec{b}$, where P_h is defined in equation 4.21.

We never converge to an approximate solution within $10,000$ iterations. Therefore, the preconditioner $\widehat{P_h}$ from equation 4.21 is not a good preconditioner. (Wathen, 2015, Ch. 5) gives bounds for the approximation S_h, for which it holds that the $\widehat{S_h}$ should be spectrally equivalent to S_h. In that case, P_h is a good preconditioner. This does not always hold for $S_h = L\widehat{D}L^T$ and $\widehat{D}$, which can be read in appendix A.

We check that S_h and $\widehat{D}$ are, in fact, not spectrally equivalent. We set hX and hY to be $1/10$, so that we get rather small matrices. We sort the eigenvalues of S_h and $\widehat{D}$ and plot them as seen in Figure 4.1.

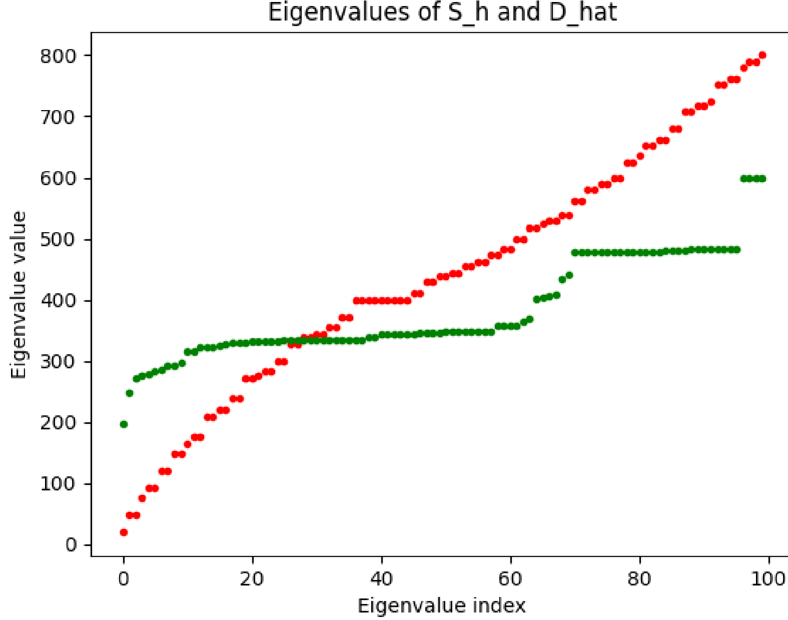

Figure 4.1: Eigenvalues of S_h (red) and $\widehat{D}$ (green) with $hX = hY = 1/10$.

The eigenvalues of S_h are given in red and the eigenvalues of $\widehat{D}$ are given in green. It can be seen, that the eigenvalues of $\widehat{D}$ are not the same as S_h.The smallest and largest eigenvalue of $\widehat{D}$ do not come close to the smallest and largest eigenvalue of S_h. Thus the spectrum is different. So a better approximation of S_h must be found that is spectrally equivalent to S_h and easy to invert.

5 Properties of the PDE

The matrices I_h, B_h and C_h from Section 3 together form the matrix A_h as in equation 4.14. The matrices arise from the discretisation of the operators from Poisson's equation using a finite difference algorithm. However, to find a preconditioner that uses the properties of the PDE, i.e., an operator preconditioner, we have to use the exact form of the saddle point system. Poisson's equation 3.1 with Dirichlet boundary conditions, can be rewritten as a saddle point system in block form. Once again, we let $\vec{w} = -\text{grad } u$, such that

$$-\Delta u = f, \tag{5.1}$$

can be written as the system of equations

$$\begin{aligned} \vec{w} \quad + \quad \text{grad } u &= 0, \\ \text{div } \vec{w} \qquad\qquad &= f. \end{aligned} \tag{5.2}$$

We can write this in block form as

$$A\vec{x} = \begin{bmatrix} I & \text{grad} \\ \text{div} & 0 \end{bmatrix} \begin{bmatrix} \vec{w} \\ u \end{bmatrix} = \begin{bmatrix} 0 \\ f \end{bmatrix} = \vec{b}, \tag{5.3}$$

where I is the identity operator. Note that, (Arnold et al., 1997, Ch. 2) defines the two-dimensional operators of the gradient and the divergence as,

$$\text{grad} = \begin{bmatrix} \partial/\partial x \\ \partial/\partial y \end{bmatrix}, \tag{5.4}$$

$$\text{div} = \begin{bmatrix} \partial/\partial x & \partial/\partial y \end{bmatrix}. \tag{5.5}$$

So the gradient operator is the transpose of the divergence operator. Equation 5.3 is equivalent to equation 2.1, where $I = \mathcal{K}$, $\text{div} = \mathcal{B}$ and $\text{grad} = \mathcal{B}^T$, thus A is symmetric.

In finite dimensional space, we know from (Horn & Johnson, 1985, Thm. 4.2.2) that the smallest and largest eigenvalue of A are given by

$$\lambda_{\min}(A) = \min_{\vec{x} \in \mathbb{R}^n \setminus \{0\}} \frac{\vec{x}^T A \vec{x}}{\vec{x}^T \vec{x}}, \tag{5.6}$$

$$\lambda_{\max}(A) = \max_{\vec{x} \in \mathbb{R}^n \setminus \{0\}} \frac{\vec{x}^T A \vec{x}}{\vec{x}^T \vec{x}}. \tag{5.7}$$

Thus,

$$\lambda_{\min}(A)\vec{x}^T\vec{x} \leq \vec{x}^T A \vec{x} \leq \lambda_{\max}(A)\vec{x}^T\vec{x}, \qquad \forall \vec{x} \in \mathbb{R}^n \setminus \{0\}. \tag{5.8}$$

Since $\vec{x} \in \mathbb{R}^n \setminus \{0\}$, $\mathbb{R}^n$ is a real normed vector space, with the Euclidean norm. Thus, $\vec{x}^T\vec{x} = \|\vec{x}\|_2^2 = (\vec{x}, \vec{x})_2$ or in general, $(\vec{x}, \vec{y})_2 = \vec{x}^T\vec{y}$. Now we can rewrite equation 5.8 as

$$\lambda_{\min}(A)(\vec{x}, \vec{x})_2 \leq (\vec{x}, A\vec{x})_2 \leq \lambda_{\max}(A)(\vec{x}, \vec{x})_2, \qquad \forall \vec{x} \in \mathbb{R}^n \setminus \{0\}. \tag{5.9}$$

However, in equation 5.3 we are working in infinite dimensional space. Therefore, we introduce Hilbert spaces and we try to find bounds as in equation 5.9.

5.1 Hilbert space of the Laplace operator

The goal of this thesis, is to understand why the algebraic preconditioner is a good preconditioner in terms of the properties of the PDE. In order to do this, we want to find the Hilbert spaces, that the operators A and P map between. So we define a Hilbert space H, such that $A : H \mapsto H'$, where H' is the dual space of H. The properties of a Hilbert space H from (Rudin, 1921, Ch. 4) are:

- We can define an inner product on space H which has the properties as in (Rudin, 1921, Def. 4.1)
- H is a complete metric space.

Then we can write equation 5.9 with $x \in H$ as

$$c_1(x, x)_H \leq (x, Ax)_H \leq c_2(x, x)_H. \tag{5.10}$$

From equation 5.1, we see that the operator A represents the $-\Delta$ operator. Thus, from equation 5.3,

$$A : \begin{bmatrix} \vec{w} \\ u \end{bmatrix} \mapsto \begin{bmatrix} \vec{w} + \text{grad } u \\ \text{div } \vec{w} \end{bmatrix}. \tag{5.11}$$

We check the conditions for when the integrals on the right hand side make sense, so possible outliers in measurements are flattened out. The integral

$$\int_\Omega \vec{w} \, \mathrm{d}\Omega + \int_\Omega \text{grad } u \, \mathrm{d}\Omega, \tag{5.12}$$

makes sense if $\vec{w} \in \left[L^2(\Omega)\right]^2$ and if $\partial u/\partial x \in L^2(\Omega)$ and $\partial u/\partial y \in L^2(\Omega)$. The integral

$$\int_\Omega \text{div } \vec{w} \, \mathrm{d}\Omega, \tag{5.13}$$

makes sense if div $\vec{w} \in L^2(\Omega)$. So from this, we can conclude that

$$\vec{w} \in H(\text{div}, \Omega) := \left\{ \vec{w} \in \left[L^2(\Omega)\right]^2 \text{ such that div } \vec{w} \in L^2(\Omega) \right\}, \tag{5.14}$$

$$u \in H^1(\Omega) := \left\{ u \in L^2(\Omega) \text{ such that } \frac{\partial u}{\partial x} \in L^2(\Omega) \text{ and } \frac{\partial u}{\partial y} \in L^2(\Omega) \right\}. \tag{5.15}$$

Thus, $A : H(\text{div}, \Omega) \times H^1(\Omega) \mapsto \left[L^2(\Omega)\right]^2 \times L^2(\Omega)$. Note that, $H(\text{div}, \Omega)$ is a Hilbert space by (Tartar, 2007, Ch. 20), $H^1(\Omega)$ is a Hilbert space by (Braess, 2007, Sect. 2.1) and L^2-space is a Hilbert space by (Christensen, 2010, Thm. 6.1.1).

5.2 Hilbert space of the preconditioner operator

In equation 5.10, we now add the preconditioner P for saddle point systems from (Wathen, 2015, Ch. 5). In block form, this will translate to

$$P = \begin{bmatrix} I & 0 \\ 0 & \Delta \end{bmatrix}. \tag{5.16}$$

Note that, P is not positive definite, since we do not have a negative sign in front of the Laplace operator. We want that A and P map between the same spaces. So $P : H \mapsto H'$, where H' is again the dual space of H. Let $x \in H$, then

$$\tilde{c_1}(x, x)_H \leq (x, P^{-1}Ax)_H \leq \tilde{c_2}(x, x)_H, \tag{5.17}$$

with $\tilde{c_2}/\tilde{c_1} \approx 1$. The discretisation of the operators A and P will result in

$$\tilde{c_1}(\vec{\tilde{x}}, \vec{\tilde{x}})_2 \leq (\vec{\tilde{x}}, P_h^{-1} A_h \vec{\tilde{x}})_2 \leq \tilde{c_2}(\vec{\tilde{x}}, \vec{\tilde{x}})_2, \tag{5.18}$$

where A_h, P_h and $\vec{\tilde{x}}$ are as defined in equation 4.12 and 4.15. From Table 4.3, we know this is a good preconditioner. Then P maps as

$$P : \begin{bmatrix} \vec{w} \\ u \end{bmatrix} \mapsto \begin{bmatrix} \vec{w} \\ \Delta u \end{bmatrix}. \tag{5.19}$$

Again, checking the conditions for when the integrals on the right hand side make sense. This gives,

$$\int_\Omega \vec{w} \, \mathrm{d}\Omega, \tag{5.20}$$

is valid if $\vec{w} \in \left[L^2(\Omega)\right]^2$ and

$$\int_\Omega \Delta u \, \mathrm{d}\Omega, \tag{5.21}$$

is valid if $\partial^2 u/\partial x^2 \in L^2(\Omega)$ and $\partial^2 u/\partial y^2 \in L^2(\Omega)$. Thus,

$$\vec{w} \in \left[L^2(\Omega)\right]^2, \tag{5.22}$$

$$u \in H^2(\Omega) := \left\{ \begin{array}{lll} u \in L^2(\Omega) & \text{such that} & \frac{\partial u}{\partial x} \in L^2(\Omega), \frac{\partial u}{\partial y} \in L^2(\Omega) \\ & \text{and} & \frac{\partial^2 u}{\partial x^2} \in L^2(\Omega), \frac{\partial^2 u}{\partial x \partial y} \in L^2(\Omega), \frac{\partial^2 u}{\partial y^2} \in L^2(\Omega) \end{array} \right\}. \tag{5.23}$$

So $P : \left[L^2(\Omega)\right]^2 \times H^2(\Omega) \mapsto \left[L^2(\Omega)\right]^2 \times L^2(\Omega)$, where $H^2(\Omega)$ is also a Hilbert space by (Braess, 2007, Sect. 2.1).

Note that, $H(\text{div}, \Omega) \subset \left[L^2(\Omega)\right]^2$ and $H^2(\Omega) \subset H^1(\Omega)$. From equation 5.17, we want that the spaces of A and P line up. So, if we restrict the operators A and P to map as follows:

$$A : H(\text{div}, \Omega) \times H^2(\Omega) \subset H(\text{div}, \Omega) \times H^1(\Omega) \mapsto \left[L^2(\Omega)\right]^2 \times L^2(\Omega), \tag{5.24}$$

$$P : H(\text{div}, \Omega) \times H^2(\Omega) \subset \left[L^2(\Omega)\right]^2 \times H^2(\Omega) \mapsto \left[L^2(\Omega)\right]^2 \times L^2(\Omega). \tag{5.25}$$

Then,

$$P^{-1}A : H(\text{div}, \Omega) \times H^2(\Omega) \mapsto H(\text{div}, \Omega) \times H^2(\Omega). \tag{5.26}$$

We notice two things. $H(\text{div}, \Omega) \times H^2(\Omega)$ and $\left[L^2(\Omega)\right]^2 \times L^2(\Omega)$ are Hilbert spaces and thus, Banach spaces and $P^{-1}A$ is an endomorphism on $H(\text{div}, \Omega) \times H^2(\Omega)$. By (Hiptmair, 2006, Thm. 2.1), if we discretise wisely (so the matrices are squares), then our condition number is bounded.

Now that we have restricted A and P to have the same domain, we can see what this means for our original function u. Previously stated in equation 5.1, we use the Laplace operator on u. That is, take the second derivative in all directions and sum them. Therefore, u must be twice differentiable in all directions. From the restriction of matrix A, we see that the bottom part of the vector, which A is multiplied with, should belong to the H^2-space. This part is the discretised function u and thus, by its initial requirements, is already twice differentiable. The same reasoning can be made for the restriction on the space of matrix P. Here, we take a subset of the L^2-space, which is, the $H(\text{div})$-space. The divergence of the upper part of vector Au, multiplied by the inverse of P, should be in the L^2-space. By definition, this is the divergence of the gradient of u, which is Δu, our original function.

5.3 Condition number

Since the mapping of $P^{-1}A$ gives an endomorphism, the condition number is bounded. This is given by the constants $\tilde{c_1}$ and $\tilde{c_2}$ in equation 5.17. Now that we have defined our Hilbert space, we can define the constants. So, let $x \in H(\text{div}, \Omega) \times H^2(\Omega)$ such that

$$x = \begin{bmatrix} \vec{w} \\ u \end{bmatrix}, \text{ where } \vec{w} \in H(\text{div}, \Omega) \text{ and } u \in H^2\Omega. \tag{5.27}$$

Equation 5.17 becomes

$$\tilde{c_1}(x, x)_{H(\text{div},\Omega)\times H^2(\Omega)} \le (x, P^{-1}Ax)_{H(\text{div},\Omega)\times H^2(\Omega)} \le \tilde{c_2}(x, x)_{H(\text{div},\Omega)\times H^2(\Omega)}. \tag{5.28}$$

Furthermore, we also define the inner product of $H(\text{div}, \Omega) \times H^2(\Omega)$ to be

$$(\alpha, \alpha)_{H(\text{div},\Omega)\times H^2(\Omega)} = (\vec{\beta}, \vec{\beta})_{H(\text{div},\Omega)} + (\gamma, \gamma)_{H^2(\Omega)}, \text{ for } \alpha = \begin{bmatrix} \vec{\beta} \\ \gamma \end{bmatrix}, \tag{5.29}$$

where $\vec{\beta} \in H(\text{div}, \Omega)$, and $\gamma \in H^2(\Omega)$. The inner products on $H(\text{div}, \Omega)$ and $H^2(\Omega)$ are defined as

$$(\vec{\beta}, \vec{\chi})_{H(\text{div},\Omega)} = (\vec{\beta}, \vec{\chi})_{L^2(\Omega)} + (\text{div } \vec{\beta}, \text{div } \vec{\chi})_{L^2(\Omega)}, \qquad \text{for } \vec{\beta}, \vec{\chi} \in H(\text{div}, \Omega), \tag{5.30}$$

$$(\gamma, \psi)_{H^2(\Omega)} = (\gamma, \psi)_{L^2(\Omega)} + (\text{grad } \gamma, \text{grad } \psi)_{L^2(\Omega)} + (\Delta\gamma, \Delta\psi)_{L^2(\Omega)}, \qquad \text{for } \gamma, \psi \in H^2(\Omega). \tag{5.31}$$

So,

$$\begin{aligned}(x, x)_{H(\text{div},\Omega)\times H^2(\Omega)} =& (\vec{w}, \vec{w})_{H(\text{div},\Omega)} + (u, u)_{H^2(\Omega)}, & (5.32)\\ =& (\vec{w}, \vec{w})_{L^2(\Omega)} + (\text{div } \vec{w}, \text{div } \vec{w})_{L^2(\Omega)} \\ &+ (u, u)_{L^2(\Omega)} + (\text{grad } u, \text{grad } u)_{L^2(\Omega)} + (\Delta u, \Delta u)_{L^2(\Omega)}. & (5.33)\end{aligned}$$

We will now deduce an upper and lower bound in the general case for $(x, P^{-1}Ax)_{H(\text{div},\Omega)\times H^2(\Omega)}$. Recall,

$$P = \begin{bmatrix} I & 0 \\ 0 & \Delta \end{bmatrix} \mapsto P^{-1} = \begin{bmatrix} I & 0 \\ 0 & \Delta^{-1} \end{bmatrix}, \tag{5.34}$$

$$A = \begin{bmatrix} I & \text{grad} \\ \text{div} & 0 \end{bmatrix}. \tag{5.35}$$

So $P^{-1}Ax$ is given by

$$P^{-1}Ax = \begin{bmatrix} I & 0 \\ 0 & \Delta^{-1} \end{bmatrix} \begin{bmatrix} I & \text{grad} \\ \text{div} & 0 \end{bmatrix} \begin{bmatrix} \vec{w} \\ u \end{bmatrix} \tag{5.36}$$

$$= \begin{bmatrix} \vec{w} + \text{grad } u \\ \Delta^{-1}\text{div } \vec{w} \end{bmatrix}. \tag{5.37}$$

So the inner product, for which we want to find bounds for, can be written as

$$(x, P^{-1}Ax)_{H(\text{div},\Omega)\times H^2(\Omega)} = (\vec{w}, \vec{w} + \text{grad } u)_{H(\text{div},\Omega)} + (u, \Delta^{-1}\text{div } \vec{w})_{H^2(\Omega)}, \tag{5.38}$$

$$= (\vec{w}, \vec{w})_{H(\text{div},\Omega)} + (\vec{w}, \text{grad } u)_{H(\text{div},\Omega)} + (u, \Delta^{-1}\text{div } \vec{w})_{H^2(\Omega)}, \tag{5.39}$$

$$\begin{aligned} = &(\vec{w}, \vec{w})_{H(\text{div},\Omega)} + (\vec{w}, \text{grad } u)_{L^2(\Omega)} + (\text{div } \vec{w}, \Delta u)_{L^2(\Omega)} \\ &+ (u, \Delta^{-1}\text{div } \vec{w})_{L^2(\Omega)} + (\text{grad } u,\ \text{grad } \Delta^{-1}\text{div } \vec{w})_{L^2(\Omega)} \\ &+ (\Delta u, \text{div } \vec{w})_{L^2(\Omega)}. \end{aligned} \tag{5.40}$$

From the divergence theorem of (Stolze, 1978, Eq. 3) with $F = u\vec{w}$, we know that

$$\int_\Omega u \text{ div } \vec{w} \, \mathrm{d}\Omega = \int_{\partial\Omega} (u\vec{w}) \cdot \vec{n} \, \mathrm{d}\partial\Omega - \int_\Omega \vec{w} \text{ grad } u \, \mathrm{d}\Omega, \tag{5.41}$$

Therefore, we can write

$$(\text{grad } u, \text{grad } \Delta^{-1}\text{div } \vec{w})_{L^2(\Omega)} = -(u, \text{div grad } \Delta^{-1}\text{div } \vec{w})_{L^2(\Omega)}, \tag{5.42}$$

$$= -(u, \text{div } \vec{w})_{L^2(\Omega)}. \tag{5.43}$$

Reapplying the divergence theorem, gives

$$-(u, \text{div } \vec{w})_{L^2(\Omega)} = (\text{grad } u, \vec{w})_{L^2(\Omega)}. \tag{5.44}$$

Plugging this into equation 5.40,

$$\begin{aligned} (x, P^{-1}Ax)_{H(\text{div},\Omega)\times H^2(\Omega)} = &(\vec{w}, \vec{w})_{H(\text{div},\Omega)} + 2(\vec{w}, \text{grad } u)_{L^2(\Omega)} + 2(\text{div } \vec{w}, \Delta u)_{L^2(\Omega)} \\ &+ (u, \Delta^{-1}\text{div } \vec{w})_{L^2(\Omega)}, \end{aligned} \tag{5.45}$$

$$= (\vec{w}, \vec{w})_{H(\text{div},\Omega)} + 2(\vec{w}, \text{grad } u)_{H(\text{div},\Omega)} + (u, \Delta^{-1}\text{div } \vec{w})_{L^2(\Omega)}. \tag{5.46}$$

By the Cauchy-Schwarz inequality, that is

$$|(\alpha, \beta)_H| \leq \|\alpha\|_H \|\beta\|_H, \text{ for } \alpha, \beta \in H, \tag{5.47}$$

we can bound the absolute value the inner product by

$$\left|(x, P^{-1}Ax)_{H(\text{div},\Omega)\times H^2(\Omega)}\right| = \left|(\vec{w}, \vec{w})_{H(\text{div},\Omega)} + 2(\vec{w}, \text{grad } u)_{H(\text{div},\Omega)} + (u, \Delta^{-1}\text{div } \vec{w})_{L^2(\Omega)}\right|, \tag{5.48}$$

$$\leq \left|(\vec{w}, \vec{w})_{H(\text{div},\Omega)}\right| + 2\left|(\vec{w}, \text{grad } u)_{H(\text{div},\Omega)}\right| + \left|(u, \Delta^{-1}\text{div } \vec{w})_{L^2(\Omega)}\right|, \tag{5.49}$$

$$\begin{aligned} \leq &\|\vec{w}\|^2_{H(\text{div},\Omega)} + 2\|\vec{w}\|_{H(\text{div},\Omega)} \|\text{grad } u\|_{H(\text{div},\Omega)} \\ &+ \|u\|_{L^2(\Omega)} \left\|\Delta^{-1}\text{div } \vec{w}\right\|_{L^2(\Omega)}. \end{aligned} \tag{5.50}$$

By Young's inequality (Carothers, 2000, Lem. 3.6), that is,

$$ab \leq \frac{a^p}{p} + \frac{b^q}{q}, \text{ for any } a, b \geq 0 \text{ and } \frac{1}{p} + \frac{1}{q} = 1 \text{ for } 1 < p, q < \infty, \tag{5.51}$$

we have that

$$\begin{aligned} &\leq 2\left\|\vec{w}\right\|^2_{H(\text{div},\Omega)} + \left\|\text{grad } u\right\|^2_{L^2(\Omega)} + \left\|\Delta u\right\|^2_{L^2(\Omega)} + \frac{\left\|u\right\|^2_{L^2(\Omega)}}{2} \\ &\quad + \frac{\left\|\Delta^{-1}\text{div } \vec{w}\right\|^2_{L^2(\Omega)}}{2}, \qquad (5.52)\\ &=2(\vec{w},\vec{w})_{H(\text{div},\Omega)} + \frac{1}{2}(u,u)_{H^2(\Omega)} + \frac{1}{2}(\text{grad } u, \text{grad } u)_{L^2(\Omega)} \\ &\quad + \frac{1}{2}(\Delta u, \Delta u)_{L^2(\Omega)} + \frac{1}{2}\left\|\Delta^{-1}\text{div } \vec{w}\right\|^2_{L^2(\Omega)}. \qquad (5.53) \end{aligned}$$

By the absolute values of equation 5.48, we can find a lower and upper bound. For the general case, we can not find the constant $\tilde{c_1}$ and $\tilde{c_2}$ in equation 5.17. The last term of the lower and upper bound, $\left\|\Delta^{-1}\text{div } \vec{w}\right\|^2_{L^2(\Omega)}$, can not be written as $(\vec{w},\vec{w})_{H(\text{div},\Omega)}$ or $(u,u)_{H^2(\Omega}$. If this term is of the same size as $\left\|u\right\|^2_{L^2(\Omega)}$, then we are able to find the constants.

In the specific case of Poisson's equation as in equation 5.2, we have that $\vec{w} = -\text{grad } u$. Continuing from equation 5.46, we get

$$\begin{aligned} (x, P^{-1}Ax)_{H(\text{div},\Omega)\times H^2(\Omega)} &= (\vec{w},\vec{w})_{H(\text{div},\Omega)} + 2(\vec{w}, \text{grad } u)_{H(\text{div},\Omega)} + (u, \Delta^{-1}\text{div } \vec{w})_{L^2(\Omega)}, \qquad (5.54)\\ &= (\text{grad } u, \text{grad } u)_{H(\text{div},\Omega)} - 2(\text{grad } u, \text{grad } u)_{H(\text{div},\Omega)} - (u,u)_{L^2(\Omega)}, \qquad (5.55)\\ &= -(\text{grad } u, \text{grad } u)_{H(\text{div},\Omega)} - (u,u)_{L^2(\Omega)}, \qquad (5.56)\\ &= -(\text{grad } u, \text{grad } u)_{L^2(\Omega)} - (\Delta u, \Delta u)_{L^2(\Omega)} - (u,u)_{L^2(\Omega)}, \qquad (5.57)\\ &= -(u,u)_{H^2(\Omega)}. \qquad (5.58) \end{aligned}$$

The upper and lower bound as in equation 5.32, then becomes

$$\begin{aligned} (x,x)_{H(\text{div},\Omega)\times H^2(\Omega)} &= (\vec{w},\vec{w})_{H(\text{div},\Omega)} + (u,u)_{H^2(\Omega)}, \qquad (5.59)\\ &= (\text{grad } u, \text{grad u})_{L^2(\Omega)} + (\Delta u, \Delta u)_{L^2(\Omega)} + (u,u)_{L^2(\Omega)} + (\text{grad } u, \text{grad } u)_{L^2(\Omega)} \\ &\quad + (\Delta u, \Delta u)_{L^2(\Omega)}, \qquad (5.60)\\ &= (u,u)_{H^2(\Omega)} + (\text{grad } u, \text{grad } u)_{L^2(\Omega)} + (\Delta u, \Delta u)_{L^2(\Omega)}. \qquad (5.61) \end{aligned}$$

Note that, since an inner product is always greater or equal to 0, we have that

$$\begin{aligned} (u,u)_{H^2(\Omega)} &= (u,u)_{L^2(\Omega)} + (\text{grad } u, \text{grad } u)_{L^2(\Omega)} + (\Delta u, \Delta u)_{L^2(\Omega)}, \qquad (5.62)\\ &\geq (\text{grad } u, \text{grad } u)_{L^2(\Omega)} + (\Delta u, \Delta u)_{L^2(\Omega)}. \qquad (5.63) \end{aligned}$$

Thus for the lower bound,

$$\begin{aligned} (x, P^{-1}Ax)_{H(\text{div},\Omega)\times H^2(\Omega)} &= -(u,u)_{H^2(\Omega)}, \qquad (5.64)\\ &\geq -(u,u)_{H^2(\Omega)} - (\text{grad } u, \text{grad } u)_{L^2(\Omega)} - (\Delta u, \Delta u)_{L^2(\Omega)}, \qquad (5.65)\\ &= -1 * (x,x)_{H(\text{div},\Omega)\times H^2(\Omega)}. \qquad (5.66) \end{aligned}$$

And for the upper bound,

$$(x, P^{-1}Ax)_{H(\text{div},\Omega)\times H^2(\Omega)} = -(u,u)_{H^2(\Omega)}, \tag{5.67}$$

$$= -\frac{1}{2}(u,u)_{H^2(\Omega)} - \frac{1}{2}(u,u)_{H^2(\Omega)}, \tag{5.68}$$

$$\leq -\frac{1}{2}(u,u)_{H^2(\Omega)} - \frac{1}{2}(\text{grad } u, \text{grad } u)_{L^2(\Omega)} - \frac{1}{2}(\Delta u, \Delta u)_{L^2(\Omega)}, \tag{5.69}$$

$$= -\frac{1}{2} * (x,x)_{H(\text{div},\Omega)\times H^2(\Omega)}. \tag{5.70}$$

So, if we let $\tilde{c_1} = -1$ and $\tilde{c_2} = -1/2$, we have that

$$-(x,x)_{H(\text{div},\Omega)\times H^2(\Omega)} \leq (x, P^{-1}Ax)_{H(\text{div},\Omega)\times H^2(\Omega)} \leq -\frac{1}{2}(x,x)_{H(\text{div},\Omega)\times H^2(\Omega)}, \tag{5.71}$$

or

$$\frac{1}{2}(x,x)_{H(\text{div},\Omega)\times H^2(\Omega)} \leq -(x, P^{-1}Ax)_{H(\text{div},\Omega)\times H^2(\Omega)} \leq (x,x)_{H(\text{div},\Omega)\times H^2(\Omega)} \tag{5.72}$$

The negative sign flips the sign inside P, such that we get the negative Laplace operator. So the condition number is $\tilde{c_2}/\tilde{c_1} = 2$, which is rather close to 1. So P is a good preconditioner.

The algebraic preconditioner from (Wathen, 2015, Ch. 5) for saddle point systems resulting from a elliptic PDE is a good preconditioner, since the function used in the PDE already meets all the requirements. We have also shown that, the inner product is always bounded from below and above and that the condition number for Poisson's equation is equal to 2.

6 Conclusion and discussion

The goal of this thesis, is to understand when and why the algebraic preconditioner for saddle point systems is considered to be a good preconditioner, based on the properties of the partial differential equation. The block representation of the Laplace operator in saddle point formulation, maps elements from $H(\text{div},\Omega) \times H^1(\Omega)$ to $\left[L^2(\Omega)\right]^2 \times L^2(\Omega)$. The block representation of the preconditioner operator, maps elements from $\left[L^2(\Omega)\right]^2 \times H^2(\Omega)$ to $\left[L^2(\Omega)\right]^2 \times L^2(\Omega)$. If we restrict the domains to align, such that they both map elements from $H(\text{div},\Omega) \times H^2(\Omega)$ to $\left[L^2(\Omega)\right]^2 \times L^2(\Omega)$, the product of the inverse preconditioning operator and the Laplace operators is an endomorphism. The condition number of the corresponding matrix will then be bounded and of order $\mathcal{O}(1)$. The restriction does not constrain our solution whatsoever, since the solution meets the requirements by definition. So, the algebraic preconditioner is a good preconditioner. What must be noted, is that the sign inside our preconditioner is positive, which results in a negative definite block. For operator preconditioning, we want the blocks to be positive definite. Future research could look into this problem.

We came to this restriction, by first checking numerically, if the algebraic preconditioner for saddle point systems from (Wathen, 2015, Ch. 5) is a good preconditioner. This was done, by first discretising Poisson's equation using the finite difference algorithm from (Shashkov & Steinberg, 1995, Ch. 2-3). We only discretised Poisson's equation for a rectangular domain, however, one can use the same paper to find a discretisation of general second order elliptic PDE and a general domain. This should, in fact, provide the same result, if applied on a rectangular domain. The discretisation of Poisson's equation returns matrices that represent gradient and divergence operator. A function, which is in the discretised domain, multiplied by these operators, does indeed return the output of Poisson's equation.

These matrix representations, together, can form a saddle point system, which can be solved using the *GMRES* iterative method. For the step size 1/1,600, we do reach a good approximate solution, but it takes more iterations. If we take even smaller step sizes, the number of iterations will then grow bigger. Applying the inverted preconditioner matrix for large step sizes on the saddle point system, results in roughly the same number of iterations and a smaller relative residual norm. From this, we

can conclude that the algebraic preconditioner is, indeed, an effective preconditioner. However, future research could look into finding an approximation of this inverse for very small step sizes. We first tried the incomplete LU-decomposition of the whole preconditioner matrix. This approach resulted in too much complication, so we tried to approach the non-identity block in the preconditioner matrix by a diagonal matrix. This matrix would then, in turn, be easy to invert. However, this also did not result in less iterations needed. By (Wathen, 2015, Ch. 5), an approximation could work, if this approximation was spectrally equivalent. The diagonal matrix that we used was clearly not spectrally equivalent. Further research could look into spectrally equivalent approximations of symmetric positive definite matrices, for which the inversion should not be too computationally expensive.

Once we established that the algebraic preconditioner is a good preconditioner, the Hilbert spaces for both the matrix containing the operators and the preconditioner matrix are defined, indicating the domains and co-domains they map between. These Hilbert spaces did not align, so we could not combine the two. However, if we take a subset of the domains of both Hilbert spaces, the domains align. The product of the inverse of the preconditioner matrix and the operator matrix forms an endomorphism, from which we can conclude that this is a good preconditioner.

References

Arnold, D. N., Falk, R., & Winther, R. (1997). Preconditioning in H(div) and applications. *Mathematics of Computations*, *66*(219), 957-984.

Axclsson, O., & Vassilevski, P. S. (1989). Algebraic multilevel preconditioning methods. I. *Numerische Mathematik*, *56*, 157-177. doi: https://doi.org/10.1007/BF01409783

Benzi, M., Golub, G. H., & Liesen, J. (2005). Numerical solutions of saddle point problems. *Acta Numerica*, *14*, 1-137. doi: https://doi.org/10.1017/s0962492904000212

Boffi, D., Brezzi, F., & Fortin, M. (2013). *Mixed Finite Element Methods and Applications.* Springer. doi: https://doi.org/10.1007/978-3-642-36519-5

Braess, D. (2007). *Finite elements: Theory, fast solvers, and applications in solid mechanics* (3th ed. ed.). Cambridge University Press.

Carothers, N. (2000). *Real analysis.* New York, United States: Cambrigde University press.

Chow, E., & Saad, Y. (1997). Experimental study of ILU preconditioners for indefinite matrices. *Journal of Computational and Applied Mathematics*, *86*(2), 387-414. doi: https://doi.org/10.1016/s0377-0427(97)00171-4

Christensen, O. (2010). *Functions, Spaces and Expansions.* Boston, MA: Birkhäuser. doi: https://doi.org/10.1007/978-0-8176-4980-7

Hiptmair, R. (2006). Operator Preconditioning. *Computers & Mathematics with Applications*, *52*(5), 699-706. doi: https://doi.org/10.1016/j.camwa.2006.10.008

Horn, R. A., & Johnson, C. R. (1985). *Matrix analysis.* Cambridge, UK: Cambridge Univeristy Press.

Mardal, K.-A., & Winther, R. (2011). Preconditioning discretizations of systems of partial differential equations. *Numerical Linear Algebra with applications*, *18*(1), 1-40. doi: https://doi.org/10.1002/nla.716

Murphy, M., Golub, G., & Wathen, A. (2000). A Note on Preconditioning for Indefinite Linear Systems. *SIAM Journal on Scientific Computing*, *21*(6), 1969-1972. doi: https://doi.org/10.1137/S1064827599355153

Ouhabaz, E. M. (2005). *Analysis of Heat Equations on Domains.* Princeton, New Jersey: Princeton University Press.

Pinchover, Y., & Jacob, R. (2005). *An Introduction to Partial Differential Equations.* Cambrdige, United Kingdom: Cambridge University Press.

Rozložník, M. (2018). *Saddle-Point Problems and Their Iterative Solution.* Cham, Switzerland: Birkhäuser. doi: https://doi.org/10.1007/978-3-030-01431-5

Rudin, W. (1921). *Real and complex analysis* (3th ed. ed.). New York, New York: McGraw-Hill Book Company.

Shashkov, M., & Steinberg, S. (1995). Support-Operator Finite-Difference Algorithms for General Elliptic Problems. *Journal of Computational Physics*, *118*(1), 131-151. doi: https://doi.org/10.1006/jcph.1995.1085

Stolze, C. H. (1978). A history of the divergence theorem. *Historia Mathematica*, *5*(4), 437-442. doi: https://doi.org/10.1016/0315-0860(78)90212-4

Tartar, L. (2007). *An Introduction to Sobolev Spaces and Interpolation Spaces.* Berlin, Heidelberg: Springer. doi: https://doi.org/10.1007/978-3-540-71483-5

Tyrtyshnikov, E. E., & Chan, R. H. (2000). Spectral equivalence and proper clusters for matrices from the boundary element method. *Numerical Methods in Engineering*, *49*(9), 1211-1224. doi: https://doi.org/10.1002/1097-0207(20001130)49:9<1211::AID-NME998>3.0.CO;2-X

Wathen, A. J. (2015, May). Preconditioning. *Acta Numerica*, *24*, 329-376. doi: https://doi.org/10.1017/S0962492915000021

A Spectral equivalence of the LDLT-decomposition

Lemma A.1. *If the LDLT-decomposition of a symmetric positive definite matrix A is given by $A = LDL^T$, then the matrices A and D are spectrally equivalent if and only if L is the identity matrix.*

Proof. By (Tyrtyshnikov & Chan, 2000, Lem. 2.1), two Hermitian positive definite matrices A and B are spectrally equivalent, if there exists positive constants c_1 and c_2 such that

$$c_1 \leq \frac{\vec{x}^T A \vec{x}}{\vec{x}^T B \vec{x}} \leq c_2, \tag{A.1}$$

for any $\vec{x} \in \mathbb{R}^n$.

Let the LU-decomposition of a symmetric matrix A be given by

$$A = LU, \tag{A.2}$$

where L is a lower triangular matrix with ones on the diagonal and U an upper triangular matrix. Let $U = DL^T$, where D is a diagonal matrix (and thus symmetric), then

$$A = LDL^T, \tag{A.3}$$

which is the $LDLT$-decomposition. Then

$$\begin{aligned} \|\vec{x}\|_A^2 &= \vec{x}^T A \vec{x}, && \text{(A.4)} \\ &= \vec{x}^T LDL^T \vec{x}, && \text{(A.5)} \\ &= (L^T \vec{x})^T DL^T \vec{x}, && \text{(A.6)} \\ &= \vec{y}^T D \vec{y}, && \text{(A.7)} \end{aligned}$$

for any $\vec{x}, \vec{y} \in \mathbb{R}^n$. We can find a lower and upper bound for this norm. That is,

$$\begin{aligned} \|\vec{x}\|_A^2 &= \vec{y}^T D \vec{y}, && \text{(A.8)} \\ &\geq \lambda_{\min}(D) \|\vec{y}\|_2^2, && \text{(A.9)} \\ &= \lambda_{\min}(D) \left\|L^T \vec{x}\right\|_2^2, && \text{(A.10)} \end{aligned}$$

and

$$\begin{aligned} \|\vec{x}\|_A^2 &= \vec{y}^T D \vec{y}, && \text{(A.11)} \\ &\leq \lambda_{\max}(D) \|\vec{y}\|_2^2, && \text{(A.12)} \\ &= \lambda_{\max}(D) \left\|L^T \vec{x}\right\|_2^2. && \text{(A.13)} \end{aligned}$$

By combining the spectral equivalence equation A.1 and the lower and upper bound for $\|\vec{x}\|_A^2$, we have that

$$c_1 \leq \frac{\vec{x}^T D \vec{x}}{\vec{x}^T A \vec{x}} \leq c_2, \quad \text{(A.14)}$$

$$c_1 \leq \frac{\|\vec{x}\|_D^2}{\|\vec{x}\|_A^2} \leq c_2, \quad \text{(A.15)}$$

$$c_1 \|\vec{x}\|_A^2 \leq \|\vec{x}\|_D^2 \leq c_2 \|\vec{x}\|_A^2, \quad \text{(A.16)}$$

$$\lambda_{\min}(D) \left\|L^T \vec{x}\right\|_2^2 \leq c_1 \|\vec{x}\|_A^2 \leq \|\vec{x}\|_D^2 \leq c_2 \|\vec{x}\|_A^2 \leq \lambda_{\max}(D) \left\|L^T \vec{x}\right\|_2^2, \quad \text{(A.17)}$$

$$\frac{\lambda_{\min}(D) \left\|L^T \vec{x}\right\|_2^2}{\|\vec{x}\|_A^2} \leq c_1 \leq \frac{\|\vec{x}\|_D^2}{\|\vec{x}\|_A^2} \leq c_2 \leq \frac{\lambda_{\max}(D) \left\|L^T \vec{x}\right\|_2^2}{\|\vec{x}\|_A^2}. \quad \text{(A.18)}$$

Then it is only bounded if

$$\left\|L^T \vec{x}\right\|_2^2 = \|\vec{x}\|_2^2, \quad \text{(A.19)}$$

which is when L^T is an orthogonal matrix. Since L is a lower triangular matrix with ones on the diagonal, this only holds when L is the identity matrix. □

B Python code for functionality tests

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.sparse import csr_matrix, csc_matrix, bmat, lil_matrix

### Defining the grid ########################################
x_start = 0
x_end = 1
hX = 1/50

y_start = 0
y_end = 1
hY = 1/50

### Defining the points in the grid ##########################
x = np.arange(x_start, x_end+hY/2, hX)
y = np.arange(y_start, y_end+hY/2, hY)
N = len(x)
M = len(y)
```

```
x = np.append(-hX/2, x)
y = np.append(-hY/2, y)
for i in range(N):
    x[i] = x[i] + hX/2
    y[i] = y[i] + hY/2

### Functions for u ##########################################
def u_constant(x,y):
    u = np.zeros(((N+1),(M+1)))
    for j in range(M+1):
        for i in range(N+1):
            u[i,j] = 1
    u = u.flatten('F')
    u = u.reshape(-1,1)
    u = np.matrix(u)
    return u

def u_xy(x,y):
    u = np.zeros(((N+1),(M+1)))
    for j in range(M+1):
        for i in range(N+1):
            u[i,j] = x[i]*y[j]
    u = u.flatten('F')
    u = u.reshape(-1,1)
    u = np.matrix(u)
    return u

def u_sinsin(x,y):
    u = np.zeros(((N+1),(M+1)))
    for j in range(M+1):
        for i in range(N+1):
            u[i,j] = np.sin(np.pi * x[i]) * np.sin(np.pi * y[j])
            if abs(u[i,j]) < 10**(-6):
                u[i,j] = 0
    u = u.flatten('F')
    u = u.reshape(-1,1)
    u = np.matrix(u)
    return u

### Creating the gradient operator C_h #########################
def C(N,M):
    Cx = lil_matrix((N*M, (N+1)*(M+1)))
    Cy = lil_matrix((N*M, (N+1)*(M+1)))
    k = 0
    for i in range(N):
        for j in range(M):
            if j == 0 or j == M-1:
                Cx[k,k+i+N+1] = -2/hX
                Cx[k,k+i+N+2] = 2/hX
            else:
                Cx[k,k+i+N+1] = -1/hX
```

```
                Cx[k,k+i+N+2] = 1/hX

            if i == 0 or i == N-1:
                Cy[k,k+i+1] = -2/hY
                Cy[k,k+i+N+2] = 2/hY
            else:
                Cy[k,k+i+1] = -1/hY
                Cy[k,k+i+N+2] = 1/hY
            k+=1
    C = bmat([[Cx],[Cy]])
    C = C.tocsr()
    C = -1*C
    return C

### Creating the divergence operator B_h #########################
def B(N,M):
    Bx = lil_matrix(((N+1)*(M+1), N*M))
    By = lil_matrix(((N+1)*(M+1), N*M))
    k = 0
    for j in range(M+1):
        for i in range(N+1):
            if (j != 0 and j != M) and i == 0:
                Bx[k,k - (N+j)] = 0 # since beta is in our case 0
            elif (j != 0 and j != M) and i == N:
                Bx[k,k - (N+j+1)] = 0 # since beta is in our case 0
            elif(j != 0 and j != M) and (i != 0 or i != N):
                Bx[k,k - (N+j)] = 1/hX
                Bx[k,k-1 - (N+j)] = -1/hX

            if j == 0 and i != 0 and i != N:
                By[k,k-1] = 0 # since beta is in our case 0
            elif j == M and i != 0 and i != N:
                By[k,k-(N+j+1)] = 0 # since beta is in our case 0
            elif (j != 0 and j != M) and (i!= 0 and i != N):
                By[k,k-(j+1)] = 1/hY
                By[k,k-(j+1+N)] = -1/hY
            k += 1
    B = bmat([[Bx, By]])
    B = B.tocsr()
    return B

### Calling matrices B and C ######################################
B = B(N,M)
C = C(N,M)

#Calling for the functions for u ######################################
u_constant = u_constant(x,y)
u_xy = u_xy(x,y)
u_sinsin = u_sinsin(x,y)

### Applying the matrices on u ########################################
```

```
lapl_u_constant = B*C*u_constant
lapl_u_xy = B*C*u_xy
lapl_u_sinsin = B*C*u_sinsin

### Transforming the vectors into a grid as in Section 2 #############
matrix_u_constant = lapl_u_constant.reshape((N+1,M+1))
matrix_u_constant = np.flipud(matrix_u_constant)

matrix_u_xy = lapl_u_xy.reshape((N+1,M+1))
matrix_u_xy = np.flipud(matrix_u_xy)

matrix_u_sinsin = u_sinsin.reshape((N+1, M+1))
matrix_u_sinsin = np.flipud(matrix_u_sinsin)

matrix_u_sinsin_lapl = lapl_u_sinsin.reshape((N+1,M+1))
matrix_u_sinsin_lapl = np.flipud(matrix_u_sinsin_lapl)

### Plotting the results #############################################
# Defining the numbers on the axes
def set_axis_labels(ax, num_ticks = 6):
    tick_labels = np.linspace(0, 1, num_ticks)
    num_data_points = matrix_u_sinsin_lapl.shape[0]
    ticks = np.linspace(0, num_data_points - 1, num_ticks)
    ax.set_xticks(ticks)
    ax.set_yticks(ticks)
    ax.set_xticklabels(np.round(tick_labels, 2))
    ax.set_yticklabels(np.round(np.flip(tick_labels), 2))

# Display the first image
fig1, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 6))

cax1 = ax1.imshow(matrix_u_constant, cmap='viridis',vmin = 0, vmax = 1)
fig1.colorbar(cax1, ax=ax1, orientation='vertical')
ax1.set_title('B*C*U1(x_i, y_j)=B*C*1')
ax1.set_xlabel('x-value')
ax1.set_ylabel('y-value')
set_axis_labels(ax1)

cax2 = ax2.imshow(matrix_u_xy, cmap='viridis',vmin = 0, vmax = 1)
fig1.colorbar(cax2, ax=ax2, orientation='vertical')
ax2.set_title('B*C*U2(x_i, y_j)=B*C*(x_i*y_i)')
ax2.set_xlabel('x-value')
ax2.set_ylabel('y-value')
set_axis_labels(ax2)
fig1.tight_layout()
plt.show()

# Display the second image
fig2, (ax3, ax4) = plt.subplots(1, 2, figsize=(12, 6))

cax3 = ax3.imshow(matrix_u_sinsin, cmap='viridis', vmin = 0, vmax = 1)
```

```
fig2.colorbar(cax3, ax=ax3, orientation='vertical')
ax3.set_title('U3(x_i, y_j) = sin(pi*x_i)sin(pi*y_j)')
ax3.set_xlabel('x-value')
ax3.set_ylabel('y-value')
set_axis_labels(ax3)

cax4 = ax4.imshow(matrix_u_sinsin_lapl, cmap='turbo', vmin = 0, vmax = 20)
fig2.colorbar(cax4, ax=ax4, orientation='vertical')
ax4.set_title('B*C*U3(x_i, y_j)=B*C*sin(pi*x_i)sin(pi*y_j)')
ax4.set_xlabel('x-value')
ax4.set_ylabel('y-value')
set_axis_labels(ax4)
fig2.tight_layout()
plt.show()
```

C Python code for performing *GMRES*

```
import numpy as np
from scipy.sparse.linalg import gmres, inv, spilu, LinearOperator
from scipy.sparse import diags, csr_matrix, csc_matrix, bmat, identity, lil_matrix
import time

### Defining the grid #######################################
x_start = 0
x_end = 1
hX = 1/400

y_start = 0
y_end = 1
hY = 1/400

### Defining the points in the grid #########################
x = np.arange(x_start, x_end+hY/2, hX)
y = np.arange(y_start, y_end+hY/2, hY)
N = len(x)
M = len(y)
x = x[:-1]
y = y[:-1]
for i in range(len(x)):
    x[i] = x[i] + hX/2
    y[i] = y[i] + hY/2

### Functions for u(x,y) =  sin(pi*x)*sin(pi*y) ##############
def u_sinsin(x,y):
    u = np.zeros(((N-1),(M-1)))
    for j in range(M-1):
        for i in range(N-1):
            u[i,j] = np.sin(np.pi * x[i]) * np.sin(np.pi * y[j])
            if abs(u[i,j]) < 10**(-6):
                u[i,j] = 0
    u = u.flatten('F')
```

```
    u = u.reshape(-1,1)
    u = np.matrix(u)
    return u

### Creating the gradient operator C ##########################
def C(N,M):
    Cx = lil_matrix((N*M, (N-1)*(M-1)))
    Cy = lil_matrix((N*M, (N-1)*(M-1)))
    k = 0
    for i in range(N):
        for j in range(M):
            if j == M-1 and i != N-1:
                Cx[k, k-1-i] = -2/hX
            elif j == 0 and i != N-1:
                Cx[k,k-i] = 2/hX
            elif i != N-1:
                Cx[k,k-i-1] = -1/hX
                Cx[k,k-i] = 1/hX

            if i == 0 and j != M-1:
                Cy[k,k-i] = 2/hY
            elif i == N-1 and j != M-1:
                Cy[k,k-i-(N-1)] = -2/hY
            elif j != M-1:
                Cy[k,k-i] = 1/hY
                Cy[k,k-i-(N-1)] = -1/hY
            k+=1
    C = bmat([[Cx],[Cy]])
    C = C.tocsr()
    C = -1*C
    return C

### Creating the divergence operator B ######################
def B(N,M):
    Bx = lil_matrix(((N-1)*(M-1), N*M))
    By = lil_matrix(((N-1)*(M-1), N*M))
    k = 0
    for j in range(M-1):
        for i in range(N-1):
            Bx[k,k+j] = -1/hX
            Bx[k,k+j+1] = 1/hX

            By[k,k+j] = -1/hY
            By[k,k+M+j] = 1/hY
            k += 1
    B = bmat([[Bx, By]])
    B = B.tocsr()
    return B

### Creating the laplace operator F #########################
def f(B,C,u):
```

```
    f = 2*np.pi**2*u
    o = np.zeros((2*N*M,1))
    F = np.vstack((o,f))
    return F

### Creating matrix A #########################################
def A(B, C):
    K = identity(2*N*M, format = 'csr')
    O = csr_matrix(((N-1)*(M-1), (N-1)*(M-1)))
    A = bmat([[K,-1*C],[B,O]])
    A = A.tocsr()
    return A

### Creating the exact inverse of matrix P ###################
def P_inv_exact(B,C):
    S = B*C
    S = S.tocsc()
    S_inv = inv(S)
    K = identity(2*N*M, format = 'csr')
    O = csr_matrix((2*N*M, (N-1)*(M-1)))
    OT = csr_matrix(((N-1)*(M-1), 2*N*M))
    P_inv = bmat([[K, O],[OT, S_inv]])
    return P_inv

### Creating the inverse of matrix P #########################
### Incomplete LU decomposition to approximate the inverse of P ###
def P_inv_linearoperator(B,C):
    S = B*C
    K = identity(2*N*M, format = 'csr')
    O = csr_matrix((2*N*M, (N-1)*(M-1)))
    OT = csr_matrix(((N-1)*(M-1), 2*N*M))
    P = bmat([[K, O],[OT, S]])
    P = P.tocsc()
    P_ilu = spilu(P)
    P_inv = LinearOperator(P.shape, P_ilu.solve)
    return P_inv

### Creating the inverse of matrix S = LDL^T ###
def S_inv(B, C):
    S = B * C
    S = S.tocsc()
    S_ilu = spilu(S)
    U = S_ilu.U
    diagonal_elements = U.diagonal()
    reciprocal_diagonal_elements = 1.0 / diagonal_elements
    D_inv = diags(reciprocal_diagonal_elements)
    return D_inv

### S__inv_approx = D_inv to approximate the inverse of P ###
def P_inv_approx(B, C, S_inv):
    K = identity(2*N*M, format = 'csr')
```

```
    O = csr_matrix((2*N*M, (N-1)*(M-1)))
    OT = csr_matrix(((N-1)*(M-1), 2*N*M))
    P_inv = bmat([[K, O],[OT, S_inv]])
    P_inv = P_inv.tocsc()
    return P_inv

### Calling for matrices B, C, u, F, A and P_inv ##################
B = B(N,M)
C = C(N,M)
u_sinsin = u_sinsin(x,y)
F = f(B,C,u_sinsin)
A = A(B,C)

P_inv1 = P_inv_exact(B,C)

P_inv2 = P_inv_linearoperator(B,C)

S_inv_approx = S_inv(B,C)
P_inv3 = P_inv_approx(B,C), S_inv_approx)

### Checking the eigenvalues of S_h and S = D_hat (only use if hX = hY < 1/20) ##############
##S = B*C
##def D(S):
##    S = S.tocsc()
##    S_ilu = spilu(S)
##    U = S_ilu.U
##    diagonal_elements = U.diagonal()
##    D = diags(diagonal_elements)
##    return D
##D = D(S)
##
##S = S.toarray()
##v_S, w_S = np.linalg.eig(S)
##v_S = np.sort(v_S)
##
##D = D.toarray()
##v_D, w_D = np.linalg.eig(D)
##v_D = np.sort(v_D)
##
##plt.plot(v_S, marker = '.', linestyle ='', color = 'r')
##plt.plot(v_D, marker = '.', linestyle ='', color = 'g')
##plt.xlabel("Eigenvalue index")
##plt.ylabel("Eigenvalue value")
##plt.title("Eigenvalues of S_h and D_hat")
##plt.show()

### Setting up a counter for GMRES ##############################
class gmres_counter(object):
    def __init__(self, disp=True):
        self._disp = disp
        self.niter = 0
```

```
        self.rk = []
    def __call__(self, rk=None, x = None):
        self.niter += 1
        self.rk.append(rk)


### Performing GMRES methods ################################################
u_sinsin_array = np.squeeze(np.asarray(u_sinsin))

print('Start now with GMRES method with step size ', hX)
print('')

### The original system Ax = b ##############################################
print('Ax=b')
start_time1 = time.time()
counter1 = gmres_counter()
gmres_sol, exit_clause = gmres(A,F, maxiter = 10000, callback = counter1, restart = None)
print('The total number of iterations is ', counter1.niter)
print('The last residual norm is ', counter1.rk[-1])
end_time1 = time.time()
elapsed_time1 = end_time1 - start_time1
print('Elapsed time: ', elapsed_time1, ' seconds')

error1 = u_sinsin_array - gmres_sol[2*N*M-1:-1]
measurement_error1 = np.linalg.norm(error1, ord = np.inf)
print('The measurement error for the original system Ax = b is ', measurement_error1)
print('')

### The preconditioned system P-1Ax = P-1b, where P is the exact inverse ###############
print('PAx=Pb with P exact inverse')
start_time2 = time.time()
counter2 = gmres_counter()
gmres_sol_pre_exact, exit_clause_pre_exact = gmres(A,F, maxiter = 10000, M = P_inv1,
                                                callback = counter2, restart = None)
print('The total number of iterations is ', counter2.niter)
print('The last residual norm is ', counter2.rk[-1])
end_time2 = time.time()
elapsed_time2 = end_time2 - start_time2
print('Elapsed time: ', elapsed_time2, ' seconds')
print('')

error2 = u_sinsin_array - gmres_sol_pre_exact[2*N*M-1:-1]
measurement_error2 = np.linalg.norm(error2, ord = np.inf)
print('The measurement error for the preconditioned system P-1Ax = P-1b is ',
        measurement_error2)
print('')

### The preconditioned system P-1Ax = P-1b, where P = ILU #######################
print('PAx=Pb with P incomplete LU')
start_time3 = time.time()
counter3 = gmres_counter()
```

```
gmres_sol_pre_ILU, exit_clause_pre_ILU = gmres(A,F, maxiter = 10000, M = P_inv2,
                                                callback = counter3, restart = None)
print('The total number of iterations is ', counter3.niter)
print('The last residual norm is ', counter3.rk[-1])
end_time3 = time.time()
elapsed_time3 = end_time3 - start_time3
print('Elapsed time: ', elapsed_time3, ' seconds')
print('')

error3 = u_sinsin_array - gmres_sol_pre_ILU[2*N*M-1:-1]
measurement_error3 = np.linalg.norm(error3, ord = np.inf)
print('The measurement error for the preconditioned system P-1Ax = P-1b is ',
        measurement_error3)
print('')

### The preconditioned system P-1Ax = P-1b, where P = D ########################
print('PAx=Pb with P=D')
start_time4 = time.time()
counter4 = gmres_counter()
gmres_sol_pre_D, exit_clause_pre_D = gmres(A,F, maxiter = 10000, M = P_inv3,
                                            callback = counter4, restart = None)
print('The total number of iterations is ', counter4.niter)
print('The last residual norm is ', counter4.rk[-1])
end_time4 = time.time()
elapsed_time4 = end_time4 - start_time4
print('Elapsed time: ', elapsed_time4, ' seconds')
print('')

error4 = u_sinsin_array - gmres_sol_pre_D[2*N*M-1:-1]
measurement_error4 = np.linalg.norm(error4, ord = np.inf)
print('The measurement error for the preconditioned system P-1Ax = P-1b is ',
        measurement_error4)
print('')
```