Delft University of Technology

Literature Review

GPU acceleration of FEM solver with applications to Geotechnical Engineering

Author:
Jorn Hoofwijk

University supervisor:
Prof. dr. ir. C. Vuik

Company supervisors:
Dr. ir. S. Brasile
Dr. ir. M. Parchei-Esfahani

A literature review submitted in partial fulfillment of the requirements for the degree of Master of Science

in

Applied Mathematics

August 19, 2021

TU Delft PLAXIS Bentley Advancing Infrastructure

Contents

1 Introduction

Computers and finite element software have helped engineers to more accurately and quickly investigate structural properties of their designs. Still a lot of computational power is required to run these simulations quickly. With the advancement of GPU based computing in the past decade, a lot of computational power has become accessible to the public. However, classical iterative solvers may not always be able to make use of this resource. So, in order to harness the power of the GPU, newer algorithms have been developed and evaluated. It has to be noted that there are FEM/CFD software packages that already support GPU computations (such as Ansys Fluent [19]) and several research teams have shown the capabilities of using GPU's for certain applications [9]. However, the effectiveness of an iterative method greatly depends on the problem, preconditioner and deflation methods used. Thus, the goal of my research is to find a combination of preconditioner, deflation vectors and Krylov method which is most suitable for the type of problems for which Plaxis 3D is generally used. This algorithm will be compared, in terms of speed, accuracy and memory usage with the existing solvers in Plaxis 3D.

Section 2 will explain the principles of finite elements. Section 3 gives an overview of what iterative methods are and how one can apply them to efficiently approximate the solution of a matrix problem. Section 4 gives a short overview of the current iterative linear solver used in Plaxis 3D. In Section 5 an overview of several parallel preconditioning methods is given, followed by an overview of some deflation methods in Section 6. Some preliminary experimentation of these methods is shown in Section 7. Finally in Section 8 the approach to find the best GPU based iterative method is discussed.

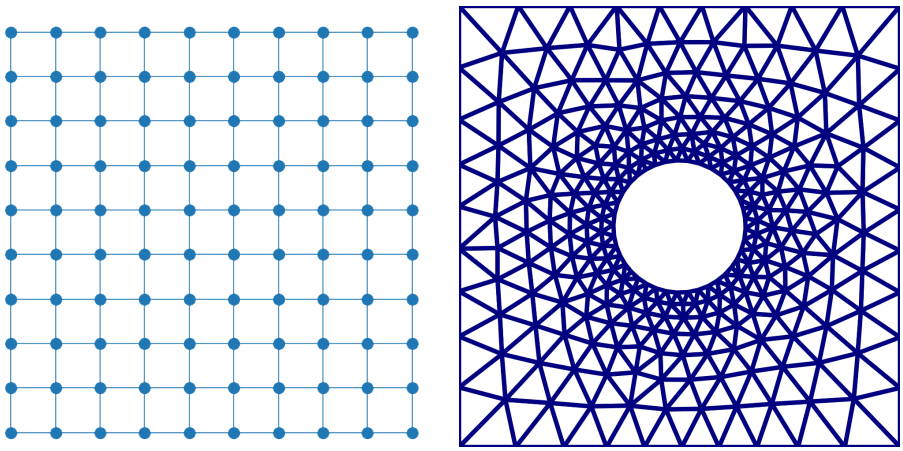
2 Discretization methods

The behaviour of physical systems, such as heat conduction in solids, are modeled using partial differential equations combined with boundary conditions. Usually, it is impossible to analytically find a solution to such problems. So, numerical methods are used to approximate the solution. There are a few options to do this. One of the easiest methods is by using the Finite Difference Method. In this method, we take a regular grid (as illustrated in Figure 1a) and approximate the solution in the grid points. This method is generally very easy to apply on a regular square grid, but can become very difficult for more complex shapes. For more general shapes, usually the Finite Element Method is used. Using the Finite Element Method, we split the domain into many small elements, such as triangles (see Figure 1b) and try to to come up with a solution such that the PDE is valid on each element. This method is more versatile and can be used to model irregularities in the domain shape. It also allows local refining or coarsening of the grid according to the required accuracy.

Both methods construct a set of equations that approximately model the physical behaviour of the system. These equations are linearized if needed and assembled into a system of equations, usually written in matrix form:

$$Ax = b.$$

This system of equations is then solved for the unknown vector x. The solution is then converted into something an engineer can interpreted, for example to find the maximal load of a bridge.



(a) Finite Difference Grid (b) Finite Element Mesh

Figure 1: Discretization options

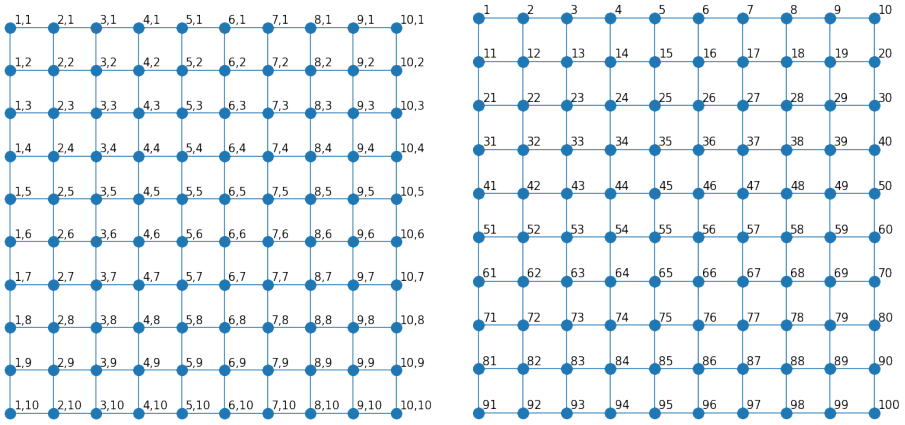
2.1 Finite Difference for Heat equation

First, we will look at the finite difference method applied to the steady state heat diffusion equation. The heat diffusion equation, for a material with unit heat conductivity, is given by:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f \quad \text{for interior points} \tag{2.1}$$

$$u(x, y) = g(x, y) \quad \text{on the boundary} \tag{2.2}$$

Where u is the temperature and f is the heat source term. Generally f and g can be any function, but for simplicity, we take $f(x, y) = 1$ and $g(x, y) = 0$. Meaning that we assume the temperature at the boundary is 0 units (can be degrees Celsius), while throughout the plate, heat is being generated at a rate of 1 units/second. The first step is do discretize the domain, we represent the



(a) Grid numbering (b) Grid renumbering

Figure 2: Discretization options

domain by a regular grid of points $u_{i,j}$, where i is the column index and j is the row index (see Figure 2a). Then, using a Taylor expansion we can approximate the second derivative (with respect to x) at each interior point using:

$$\frac{\partial^2 u_{i,j}}{\partial x^2} \approx \frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{\Delta x^2}. \tag{2.3}$$

We may do the same for the y-direction, to get:

$$\frac{\partial^2 u_{i,j}}{\partial y^2} \approx \frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{\Delta y^2}. \tag{2.4}$$

This system has two indices for the grid points, one in the x and one in the y direction, to prepare for putting it in matrix form, we will renumber the nodes,

starting at the top-left and line by line work to the bottom-right to number all nodes (see Figure 2b). As an example the finite difference equation for node 33 then becomes (assuming $\Delta x = \Delta y$) :

$$\frac{1}{\Delta x^2}\left[u_{23} + u_{32} - 4u_{33} + u_{34} + u_{43}\right] = f_{33} \tag{2.5}$$

The resulting equations are written in matrix form. We also need to take the boundary conditions into account, usually they are removed from the coefficient matrix (A) and their influence on surrounding elements is taken into account in the right-hand side vector (b). The resulting coefficient matrix has a structure as illustrated in Figure 3, the non-zero elements are colored black. We can see that most of the matrix elements are zero (white). Such a matrix is called a *sparse matrix*. A lot of memory and computational cost can be saved by only storing non-zero values and remembering that all values that are not stored are zero and therefore can be ignore. We also see that we get a banded structure, meaning that all values lie within a certain sized band around the diagonal.

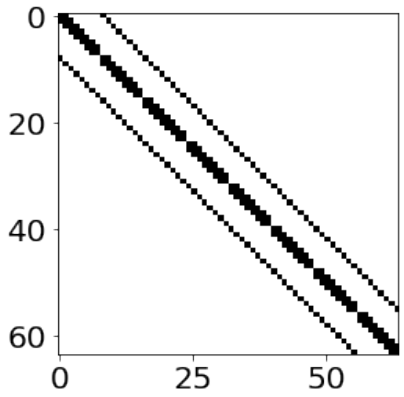


Figure 3: Structure of A

2.2 Finite Element for heat equation

For finite elements the approach is a little different. We again start from the differential equation. This time we will use a mathematical notation using the differential operator (∇):

$$-\nabla^2 u = \nabla \cdot \nabla u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f. \tag{2.6}$$

Which can be rewritten into:

$$\nabla^2 u + f = 0 \tag{2.7}$$

Interestingly, as $\nabla^2 u + f$ is zero in the interior, we can multiply it with another function (ϕ), which is zero on the boundary, integrate it and the product will still be zero.

$$\iint_\Omega \left(\nabla^2 u + f(x,y)\right) \phi(x,y)\,\mathrm{d}x\,\mathrm{d}y = \iint_\Omega 0 \cdot \phi(x,y)\,\mathrm{d}x\,\mathrm{d}y = 0 \qquad \text{for any function } \phi : (\phi|_\Gamma = 0). \tag{2.8}$$

We can then split this into

$$\iint_\Omega \phi \nabla^2 u\,\mathrm{d}x\,\mathrm{d}y + \iint_\Omega f\phi\,\mathrm{d}x\,\mathrm{d}y = 0$$
$$-\iint_\Omega \phi \nabla^2 u\,\mathrm{d}x\,\mathrm{d}y = \iint_\Omega f\phi\,\mathrm{d}x\,\mathrm{d}y \tag{2.9}$$

And using Gauss divergence theorem, combined with $\phi|_\Gamma = 0$, we can rewrite this into:

$$\iint_\Omega \phi f\,\mathrm{d}x\,\mathrm{d}y = -\oint_{\partial\Omega} \phi \frac{\partial u}{\partial n}\,\mathrm{d}\Gamma + \iint_\Omega \nabla\phi \cdot \nabla u\,\mathrm{d}x\,\mathrm{d}y \tag{2.10}$$
$$= \iint_\Omega \nabla\phi \cdot \nabla u\,\mathrm{d}x\,\mathrm{d}y \tag{2.11}$$

This is called the weak formulation of the problem, and it holds for any admissible function ϕ, we can even chose multiple functions ϕ_i and the equality will hold for all of them. These ϕ_i are called the test functions. In the Galerkin approach, these test functions also form the basis functions for constructing the solution, u, and they determine how we will solve the problem. The simplest example would be to use linear triangular elements. Linear referring to the idea that the basis function will be linear within each triangle. The exact basis functions for a single triangle are illustrated in Figure 4. The function is 1 at one node and 0 at all other nodes. Within each element, we then have three non-zero basis functions. In general form, these equations are given by Equation 2.12.

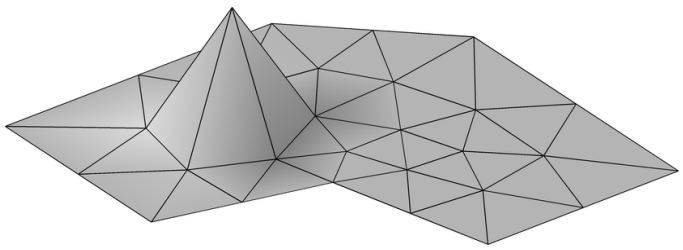


Figure 4: Basis function for linear triangular element, as taken from [4]

$$\begin{aligned} \phi_1(x,y) &= a_1 + b_1 x + c_1 y \\ \phi_2(x,y) &= a_2 + b_2 x + c_2 y \\ \phi_3(x,y) &= a_3 + b_3 x + c_3 y \end{aligned} \tag{2.12}$$

As these functions are linear, the first derivatives are constants:

$$\begin{aligned}\frac{\partial \phi_1}{\partial x} &= b_1 \\ \frac{\partial \phi_1}{\partial y} &= c_1.\end{aligned} \tag{2.13}$$

And as such, integrating the left-hand side of Equation 2.10 for the three basis functions inside this element becomes very easy, we usually write it in matrix form, this matrix is called the element stiffness matrix (note this system is different for every element):

$$\begin{bmatrix} S_{11} & S_{12} & S_{13} \\ S_{21} & S_{22} & S_{23} \\ S_{31} & S_{32} & S_{33} \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \\ f_3 \end{bmatrix}. \tag{2.14}$$

Where S_{ij} is given by

$$\begin{aligned} S_{ij} &= \iint_\Omega \nabla \phi_i \cdot \nabla \phi_j \, \mathrm{d}\Omega \\ &= \iint_\Omega \begin{bmatrix} b_i \\ c_i \end{bmatrix} \cdot \begin{bmatrix} b_j \\ c_j \end{bmatrix} \mathrm{d}\Omega \\ &= (b_i b_j + c_i c_j) \iint_\Omega 1 \, \mathrm{d}\Omega. \end{aligned} \tag{2.15}$$

Where $\iint_\Omega 1 \, \mathrm{d}\Omega$ is the area of the triangular element. The right-hand side of Equation 2.10 is more difficult to integrate. But we can use numerical integration, such as Newton-Cotes to approximate the right-hand side to use in Equation 2.14. With linear elements this would yield:

$$f_i = \iint_\Omega \phi_i f \, \mathrm{d}x \, \mathrm{d}y \overset{\text{Newton-Cotes}}{\approx} \sum_{j=1}^{3} \phi_i(x_j) f(x_j) = \sum_{j=1}^{3} \delta_{ij} f(x_j) = \frac{1}{3} f(x_i). \tag{2.16}$$

Where δ_{ij} is the Kronecker delta function. When we compute all element stiffness matrices and right-hand sides, we can assemble them into one big system of equations, which we write in a familiar form, $Ax = b$.

2.3 Finite Element for solids

In geotechnical applications one of the main interests is the structural integrity of civil structures and the underlying soil. The steps that have to be taken to construct the system of equations are very similar to the heat equation, but the differential equations are different, leading to some extra considerations. For solids, we have displacements in the x and y-direction, leading to strains, given

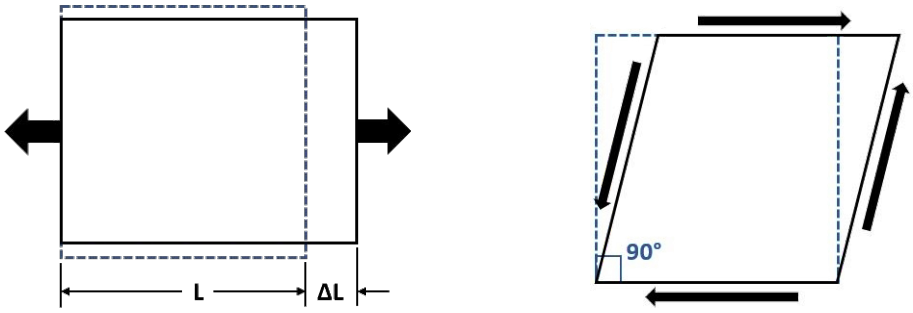
by:

$$\epsilon_x = \frac{\partial u}{\partial x} \tag{2.17}$$

$$\epsilon_y = \frac{\partial v}{\partial y} \tag{2.18}$$

$$\gamma_{xy} = \frac{\partial v}{\partial x} + \frac{\partial u}{\partial y} \tag{2.19}$$

Where u, v are the displacement in the x and y-direction, respectively. ϵ_x, ϵ_y are the normal strains, and γ_{xy} is the (engineer) shear strain. The normal strain is a measure of how much the substance gets compressed in a direction, whereas the shear strain indicates how much it is sheared (see Figure 5). Given these



(a) Normal strain (b) Shear strain

Figure 5: Strains

strains, and the material properties, we can derive the stresses. If we assume a linear relation, we can write in matrix notation:

$$\begin{bmatrix} \sigma_x \\ \sigma_y \\ \tau_{xy} \end{bmatrix} = \frac{E}{1-\nu^2} \begin{bmatrix} 1 & \nu & 0 \\ \nu & 1 & 0 \\ 0 & 0 & \frac{1}{2}(1-\nu) \end{bmatrix} \begin{bmatrix} \epsilon_x \\ \epsilon_y \\ \gamma_{xy} \end{bmatrix}. \tag{2.20}$$

Where E is the Young modulus (indicating the stiffness of a material), ν is the Poisson ration (indicating the orthogonal expansion of a material upon compression in one direction), σ_x, σ_y is the normal stress, and τ_{xy} the shear stress. These stresses indicate a (directional) force per unit area, which has to be balanced in all points in order to find a steady state solution.

In the end, the equations in structural analysis are a bit more complicated than the diffusion equation, yet can be approached similarly using the finite element method. There are a few mayor differences one has to take into account:

1. There are multiple unknowns per node, so that on top of choosing in what order to number the nodes. One also can choose whether to start numbering the unknowns first by element, then by dimension (e.g. $[u_1, v_1, u_2, v_2, ...]$) or first by dimension, then by element (e.g. $[u_1, u_2, ..., u_N, v_1, v_2, ...]$)

2. Equation 2.20 is a bit over-simplified. It is presented as if the material properties are determined by a few linear parameters. Whereas in reality these relations are non-linear, meaning that we first have to linearize, and then solve the system, linearize again, etc... In Plaxis this non-linear iteration is done using a Quasi-Newton method for fast convergence. As a result, the system to be solved does not contain the displacements themselves, but a derivative thereof.

3. For the structural engineering we get some extra compatibility equations that have to be satisfied, see [21].

4. For the most part Plaxis 3D uses quadratic tetrahedral elements. But there are also, plate elements, beam elements, interface elements, etc...

5. It is possible to state the system of equations as function of strains, or stresses rather than displacements. These three methods yield different systems. In Plaxis 3D the systems are displacement based.

6. The boundary conditions become somewhat different. E.g. on some boundaries we may have constraints on the normal displacement, but not on the orthogonal displacement.

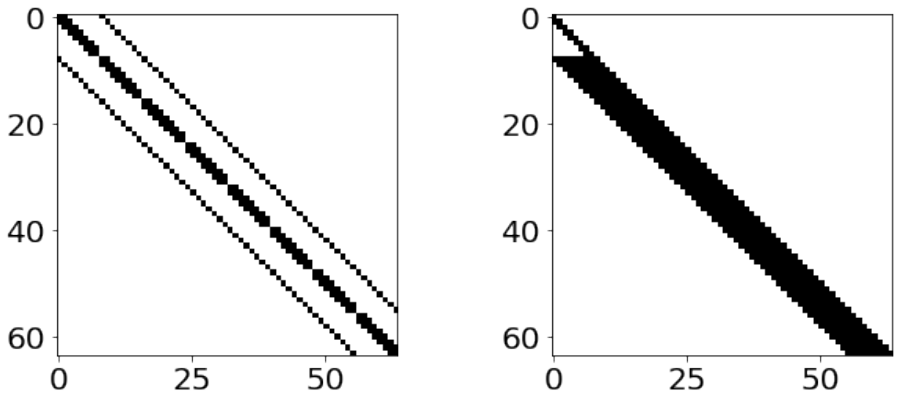
These topics are interesting on their own, yet mostly out of scope for this report. These factors do, however, impact the resulting system of equations to be solved. Therefore they are relevant, especially for the choice of preconditioner. Furthermore, as the linear systems are part of a non-linear iteration, they form a collection of related linear systems. This relation can potentially be exploited to improve performance.

3 Iterative solvers

This chapter is mostly based on information provided in the books of Vuik [24] and Saad [22].

Consider the system of equations $Ax = b$. This system can be solved directly by inverting A explicitly, i.e. $x = A^{-1}b$. For big matrices however, doing this in practice is very slow and memory consuming. A slightly more efficiently approach may decompose A into a lower and upper triangular matrix such that $A = LU$ and solve the systems $LUx = b$. This system can be solved in two steps by a forward and backward substitution. However, this is usually still quite slow.

Another issue that arises with this method is that using the Finite Element Methods, one usually generates very sparse matrices, where every row of A has only a few non-zero entries. Sparse matrices like these can be very efficiently stored in memory by only considering the non-zero elements. However, when inverting the matrix explicitly, or by LU-decomposition, this sparsity is lost (see Figure 6), meaning there are many more non-zero entries, which requires a lot of memory and is undesirable. There are methods that somewhat address this issue, for example by reordering [8] or using special direct solvers like Intel MKL PARDISO [12] but generally direct methods cost a lot of memory. To counter the memory limitations and speed up the process, iterative methods have been developed. These methods do not solve the system of equations exactly, but instead start with a guess for the solution and iteratively improve the solution (hence their name). Generally, these methods use less memory than direct methods **[ref]** and can often be made faster than direct solvers, depending on the method used and the desired accuracy **[ref]**, making them a popular choice for solving large sparse systems. To measure the accuracy of an



(a) Structure of A (b) Full LU decomposition, structure of L

Figure 6: Sparsity pattern of LU decomposition

iterative method, one is interested in the error

$$e_k = x - x_k \tag{3.1}$$

We desire that this error will go to zero. However, as the exact solution, x, is not known, the error can not be computed exactly. A more popular measure of accuracy is the residual, which is defined by:

$$\begin{aligned} r_k &:= b - Ax_k \\ &= Ax - Ax_k = Ae_k. \end{aligned} \tag{3.2}$$

It has the nice property that, when the error goes to zero, so does the residual. And the residual is *much* easier to compute than the error itself.

3.1 Basic iterative methods / Fixed-Point iteration

The Basic Iterative Method (BIM) is one of the simplest iterative methods. Generally, these methods are not very efficient, but they are quite easily understood. First we will look at the method, then we will give a numerical example. BIMs are based on a splitting of the matrix A into two components

$$A = M - N. \tag{3.3}$$

One can now rewrite

$$Ax = b \iff (M - N)\, x = b \iff Mx = Nx + b \tag{3.4}$$

Which can then be converted into an iterative scheme:

$$\begin{aligned} x_{k+1} &= M^{-1}\left(Nx_k + b\right) \\ &= M^{-1}\left(\left(M - A\right) x_k + b\right) \\ &= M^{-1}\left(Mx_k + b - Ax_k\right) \\ &= x_k + M^{-1}\left(b - Ax_k\right) \\ &= x_k + M^{-1} r_k. \end{aligned} \tag{3.5}$$

Depending on your choice of M and N, you will get different methods. Note that in this iterative scheme the computation Ax_k is used, which is relatively cheap when A is sparse. Furthermore, $M^{-1}r_k$ has to be solved, in order to get an efficient method, this should be an "easy" operation. This is the case when M is diagonal or triangular. Furthermore, the error at every iteration can be stated in recurring form:

$$e_{k+1} = \left(I - M^{-1}A\right) e_k. \tag{3.6}$$

From this recurring relation we can derive that the error e_k will go to zero if and only if the absolute value of the eigenvalues of the iteration matrix $I - M^{-1}A$ are strictly less than 1. In other words: the spectral radius of $I - M^{-1}A$ should

be strictly less than 1 in order for x_k to converge to x. Furthermore, when the spectral radius is smaller, fewer iterations are needed to obtain a sufficiently accurate solution. Depending on the choice of M, the method may or may not converge, and if it does, the speed of convergence is heavily influenced by the choice of M. The method is very simple and memory efficient, yet generally converges quite slowly which makes it not a popular choice of iterative method in practice.

Example 1 One of the simplest BIMs is constructed by the splitting

$$M = \text{diag}(A) \tag{3.7}$$
$$N = \text{diag}(A) - A. \tag{3.8}$$

The resulting BIM is called *Jacobi iteration*. Let us look at the example with

$$A = \begin{bmatrix} 5 & -4 \\ -1 & 2 \end{bmatrix}, b = \begin{bmatrix} 0.5 \\ 1 \end{bmatrix}, \text{so } M = \begin{bmatrix} 5 & 0 \\ 0 & 2 \end{bmatrix}, N = \begin{bmatrix} 0 & 4 \\ 1 & 0 \end{bmatrix}. \tag{3.9}$$

Which has $x = \left(\frac{5}{6}, \frac{11}{12}\right)^T$ as the exact solution. The eigenvalues of $I - M^{-1}A$ are $\pm\sqrt{0.4} = \pm 0.6324$. Which means that this BIM should converge and the error is reduced with about 37% each iteration. Figure 7 shows the the first 10 Jacobi iterations for the starting vectors

$$\begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 2 \\ 1 \end{bmatrix}, \begin{bmatrix} 2 \\ 2 \end{bmatrix}.$$

We can see that for all starting vectors this method converges to the exact solution. Furthermore, Table 1 shows the values of the first 10 iterations along with the norm of the error, we can see that for this problem the error is reduced about a hundred times after 10 iteration.

3.2 Krylov Methods

A more popular class of iterative methods are the Krylov based methods. The general idea of Krylov methods is to iteratively construct a search space, which is a subspace $\mathbb{R}^N$, and find the best solution within this search space. Then, as the search space gets bigger, the approximate solution will approximate the exact solution. The name of these methods stems from the search space that is used, which is the Krylov subspace, this space is defined as:

$$\mathcal{K}_k(A, r) = \text{span}\left\{r, Ar, A^2r, \ldots, A^{k-1}r\right\} \tag{3.10}$$

with r the initial residual. So that we can define a growing sequence of subspaces.

3.2.1 Conjugate Gradient

For the conjugate gradient method we will need a matrix A that is symmetric and positive definite, i.e. $x^TAx > 0 \, \forall \, x \in \mathbb{R}^N \setminus \{0\}$. Then $\|x\|_A := \sqrt{x^TAx}$ is

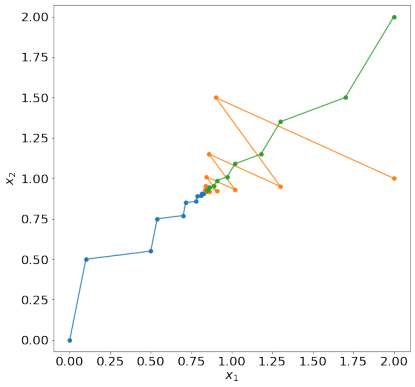


Figure 7: Jacobi iteration with different initial guesses for the system defined by (3.9)

Table 1: First 10 Jacobi iterations for the system defined by (3.9) with starting vector $x_0 = (0, 0)^T$

Iteration	x_0	x_1	Error
0	0.000	0.000	1.239
1	0.100	0.500	0.843
2	0.500	0.550	0.496
3	0.540	0.750	0.337
4	0.700	0.770	0.198
5	0.716	0.850	0.135
6	0.780	0.858	0.079
7	0.786	0.890	0.054
8	0.812	0.893	0.032
9	0.815	0.906	0.022
10	0.825	0.907	0.013

a well-defined norm. The conjugate gradient method iteratively searches along a search direction and minimizes the error (in the A-norm) along that search direction. This is similar to gradient descent method in that by searching along a given search direction the problem is only one-dimensional and thereby easier to solve. One problem with gradient descent however, is that it may repeat a search direction. So, if we make sure that all search directions (p_i) are orthogonal to one another (with respect to the inner product induced by A, i.e. $p_i A p_j = 0 \quad i \neq j$, this is called conjugate) we will never need to search in the same direction twice. In fact, if we minimize the error along a sequence of conjugate search directions, we will actually get the approximate solution that minimizes the error over the entire search space. This is the key of the conjugate gradient method.

Now the reason that this is a Krylov method, is that the first search direction is chosen to be the initial residual r_0, and every subsequent search direction is chosen such that the search directions are conjugate and form a basis for the Krylov space $\mathcal{K}_k(A, r_0)$.

The full conjugate gradient method is given by Algorithm 1. Note that the residual can also be computed as $r_j = b - Ax_j$ but the formulation in the given algorithm is equivalent up to rounding errors and saves a matrix vector product (as Ap_j can be re-used). The p_j represents the search direction and α is the contribution of that search direction. β is used to keep the search directions conjugated. In exact arithmetic, CG will converge to the exact solution in

Initial guess: x_0
$r_0 = b - Ax_0$
$p_0 = r_0$
For $j = 0, 1, \ldots$ (until convergence)

$\alpha_j = \frac{r_j^T r_j}{p_j^T A p_j}$
$x_{j+1} = x_j + \alpha_j p_j$
$r_{j+1} = r_j - \alpha_j A p_j$
$\beta_j = \frac{r_{j+1}^T r_{j+1}}{r_j^T r_j}$
$p_{j+1} = r_{j+1} + \beta_j p_j$

End

Algorithm 1: Conjugate Gradient Method as taken from Saad 2003[22]

N iterations. However, due to round-off errors this does not happen exactly. Furthermore, for large systems this property is not useful for it takes too many iterations. Usually, the iteration is stopped when the residual is sufficiently small.

3.2.2 Convergence rate

To compare the computational work to be done with the achieved accuracy, usually one looks at the convergence rate of the iterative methods. This is

the amount that the error (or residual) decreases in each iteration. A better convergence rate results in fewer iterations and therefore a lower solve time. Figure 8 shows the norm of the residual for the Conjugate Gradient method and the Jacobi method for a 2D Poisson problem on a 30×30 regular grid, this problem corresponds to a heat equation. It is immediately clear that the CG method has much faster convergence than the Jacobi method, which is to be expected. Another interesting property we can see is that the CG method initially converges only slightly faster than the Jacobi method, but after about 40 iterations, the convergence becomes much faster. This property is called super linear convergence and it occurs when the error corresponding to the lowest eigenvalue is eliminated. It can be shown that the convergence rate of

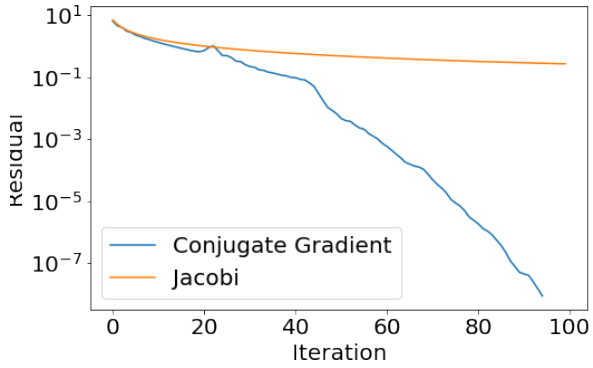


Figure 8: Convergence rate of Conjugate Gradient versus Jacobi method for a 2D Poisson problem on a 30×30 grid.

the CG method depends on the condition number of the matrix. This condition number in turn depends on the eigenvalues of the matrix A, specifically, when the eigenvalues are sorted from smallest (λ_1) to largest (λ_N):

$$\kappa_2 (A) = \frac{\lambda_N}{\lambda_1}. \tag{3.11}$$

And the convergence rate can be written as

$$\frac{\sqrt{\kappa_2 (A)} + 1}{\sqrt{\kappa_2 (A)} - 1}. \tag{3.12}$$

This is more or less the convergence rate as seen between iteration 10 to 40. After that, the error corresponding to the smallest eigenvalue is eliminated and the convergence now depends on the effective condition number, which, at this stage, is determined by the second smallest eigenvalue

$$\kappa_{\text{eff}}(A) = \frac{\lambda_N}{\lambda_2}. \tag{3.13}$$

This property, that the convergence rate improves after some number of iterations is called super-linear convergence, which is a nice property that Krylov methods have.

3.3 Preconditioning

As said before, the convergence rate of the CG method depends on the condition number. So, to speed up the convergence, we can transform an ill-conditioned system $Ax = b$ into another system, with the same solution, but with a lower condition number, such that we get faster convergence. This idea is called preconditioning. Preconditioning is an essential step in getting a well performing CG method. Mathematically, the transformation can be written as:

$$MAx = Mb \tag{3.14}$$

Where M is the (left) preconditioner. The solution x is still the same, but the convergence rate now depends on $\kappa_2(MA)$, which is ideally much smaller than $\kappa_2(A)$. We want M to be a good approximation of A^{-1}, as this generally yields fewer iterations. Unfortunately, when M approximates A^{-1} well, it will usually also be expensive to compute and thereby resulting in very few, but slow iterations. Thus, one has to find an optimum between having a fast preconditioner, and having a preconditioner that approximates A^{-1} well. There are many different types of preconditioners, among the best known we find the Incomplete LU decomposition, which will be further discussed in Section 5.2.

3.3.1 Incomplete Cholesky decomposition

The Incomplete Cholesky is very comparable to a full Cholesky decomposition, it splits the matrix A into a lower and an upper triangular part C such that:

$$A \approx CC^T. \tag{3.15}$$

Which is easily inverted. But, as said before, such a splitting may yield a lot of fill-in. However, usually the magnitude of the fill-in values quickly decreases the farther away the fill-in is from the sparsity pattern of A. Therefore small fill-in values can be omitted while still keeping a fairly good approximation: One can specify the amount of fill-in that may occur, zero fill-in means that C has the same sparsity pattern as the lower triangular part of A. Then the amount of fill-in can be specified as the amount of non-zero elements the method can add to every row. This leads to a class of methods indicated by the amount of fill-in, $ICCG(k)$ for Incomplete Cholesky Conjugate Gradient with k fill-in elements per row. Usually, more fill-in leads to longer setup times, more memory usage and slower iterations, but the total number of iterations is decreased, this is illustrated in Figure 9, which shows the convergence for $ICCG(k)$ for a 2D Poisson problem on a 30×30 grid. For this case, the $IC(0)$ preconditioner reduces the number of iterations 39 (compared to 96 without preconditioning). Allowing one fill-in element per row, hardly increasing the computational load, still reduces the number of iterations to just 25.

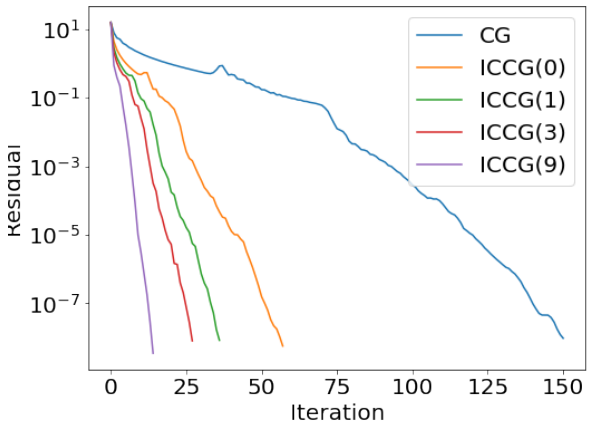


Figure 9: Convergence plot for $ICCG(k)$ with different amounts of fill-in for a 2D Poisson problem on a 30×30 grid.

3.4 General Krylov methods

So far we have only looked at CG, which requires the matrix to be SPD. There are alternative methods that can deal with general matrices A which have similar convergence properties as CG. Unfortunately, as the matrix is no longer SPD, it does no longer define a norm and therefore we cannot simply minimize $\|x - x_k\|_A$, which means that some nice properties of CG are lost. As it turns out, there is no optimal Krylov method for a-symmetric matrices, but there are several methods that try to mimic CG and keep some of its nice properties. We will look at two methods, Bi-CGSTAB and GMRES which both have their advantages and disadvantages. They also need to be combined with a preconditioner in order to get good convergence. Since they can solve general systems, the preconditioner no longer needs to be SPD and so more choices of preconditioners become available. We will look at specific preconditioners relevant to this research in Section 5.

3.4.1 GMRES

GMRES minimizes the norm of residual in the search space. In every GMRES iteration, the new search direction will be orthogonalized to all previous search directions. As a result, we can prove that GMRES finds the optimal solution in the Krylov subspace. The big disadvantage is that this requires us to store all previous search directions costing a lot of memory and orthogonalization will cost a lot of computing power. Every iteration, GMRES will slow down a bit. If the preconditioner is very good, very few iterations are needed and this method is feasible. If many iterations are needed, one may throw away all previous search directions and start over, which is called restarted GMRES. A major

downside is that any super-linear convergence is also thrown away, although there are ways to still keep this good convergence **[ref, ref, ref]**.

3.4.2 Bi-CGSTAB

Bi-CGSTAB takes a different approach than GMRES. It has short recurrence, meaning you do not need to store all search directions. Therefore the iteration is faster and will not slow down. The trade-off here is that there is no proof of optimality nor a proof of convergence. In most practical applications, it will converge **[ref]** although it requires more iterations than GMRES.

4 Current solvers in Plaxis

At the moment, Plaxis 3D provides the user the choice between three linear solvers:

1. PARDISO, this is a parallel direct solver library developed by Intel
2. Classic solver, this is a single core iterative solver that uses an incomplete decomposition as preconditioner
3. PICOS, this is a multi-core iterative solver which is the successor of the classic iterative solver.

As PARDISO is a third party direct solver, we will not go into the details. The classic solver is quite straightforward, so we do not need to go into more detail. But the last solver, PICOS, is a bit more complex, we shall give a short overview of the methods it uses.

4.1 PICOS

PICOS achieves parallelism through domain decomposition [16]. It splits the domain into a number of subdomains equal to the number of computer cores. On each subdomain, it uses an incomplete decomposition as preconditioner. The values of the boundary of each subdomain are communicated to the neighboring subdomains in order to eventually reach a global solution. There are several ways to do this, in PICOS this is done via the restricted additive Schwarz method. The details of this method are not too important for this thesis, but the effect is that it takes more iterations to solve the problem when it is split into subdomains, but as you can solve it on multiple computer cores in parallel, the total time for finding the solution is reduced.

PICOS uses a second preconditioner, on the global level. This coarse grid preconditioner is mathematically equivalent to deflation. In particular, PICOS uses rigid body modes (see Section 6.2.6) where the considered rigid bodies correspond to the subdomains. This aims at reducing the cost of splitting the domain into subdomains. It has to be noted that the subdomains are chosen in a particular way, such that the subdomains correspond as much as possible with regions of similar material. This is especially effective when the model has a layered soil with vastly different stiffness (such as Figure 10).

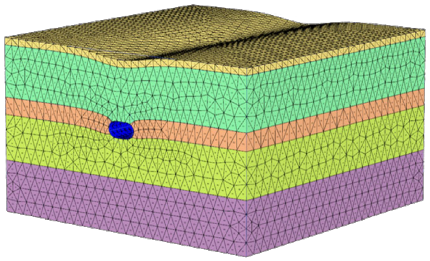


Figure 10: Tunnel through layered soil[16]

5 Parallel preconditioners

As said before, preconditioners are essential to get good convergence rates for Krylov methods. They aim to transform an ill-conditioned system into a new system with a lower condition number, thereby improving the convergence rate. Generally, you want a preconditioner to approximate the inverse of A. A better preconditioner leads to fewer iterations at the cost of more work per iteration, so there is a trade-off to be made. Furthermore, as we are interested in GPU computing, we will specifically look into highly parallel preconditioners.

5.1 Jacobi / diagonal scaling

One of the simplest preconditioners is the Jacobi preconditioner, also called diagonal scaling. As the latter name suggests, it scales the system by the value on the main diagonal, thereby scaling the diagonal back to 1. It is mostly beneficial when the diagonal values vary significantly. This means that it is interesting for applications in which material properties may vary significantly.

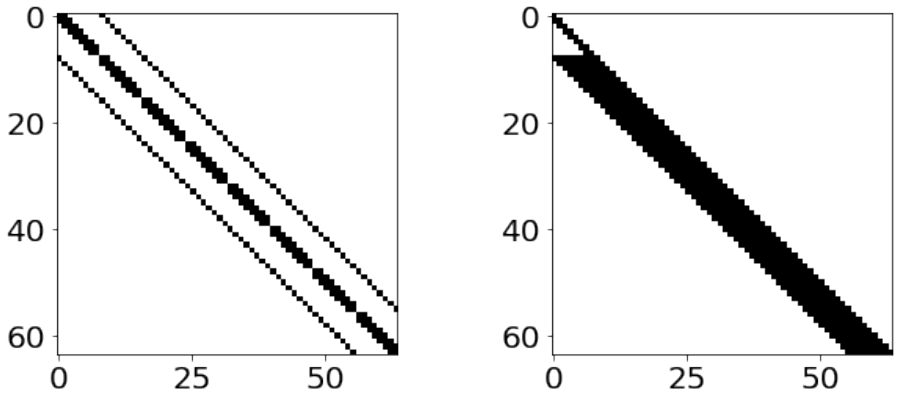
The Jacobi preconditioner is memory efficient, as it only needs to store one diagonal vector and it is easily parallelizable, as each row can be considered independent of every other row, which makes it a suitable preconditioner for usage on a GPU. Every individual iteration is very fast, but many iterations will be needed, as it is a very simple preconditioner. In some cases, these fast iterations may outweigh the cost of doing many iterations **[ref]**.

5.2 Incomplete LU (ILU)

A more advanced preconditioner is the Incomplete LU decomposition, or ILU for short. It is comparable to the Incomplete Cholesky Decomposition except that it works for non-symmetric matrices as well. It is based on the idea that, if we allow pivoting, every non-singular matrix has an LU decomposition **[ref]**. Generally this decomposition is expensive to compute and requires a lot of memory due to fill-in, which is illustrated in Figure **11**. The idea for the Incomplete LU decomposition is to approximate this LU decomposition by dropping small values in every row to limit the amount of fill-in. There is a choice to be made as to how many and which values should be kept. The more values are keeps, the better the approximation becomes, at the expense of more memory and computational load. This method is very popular in CPU based solvers, yet due to the forward and backward substitution steps it is not easy to parallelize, we will look at some methods that make more efficient use of the parallelism available in GPUs.

5.2.1 ILU(n) and ILUT

For ILU there are two main ways to select which values are allowed to fill-in. ILU(n) is based solely on the sparsity pattern of the matrix, and allows fill-in of the locations corresponding to the second (or third, etc) level neighbors, as is



(a) Structure of A (b) Full LU decomposition, structure of L

Figure 11: Sparsity pattern of LU decomposition

illustrated in Figure 12. This has the advantage that the sparsity pattern can be known in advance, especially when the grid is very regular.

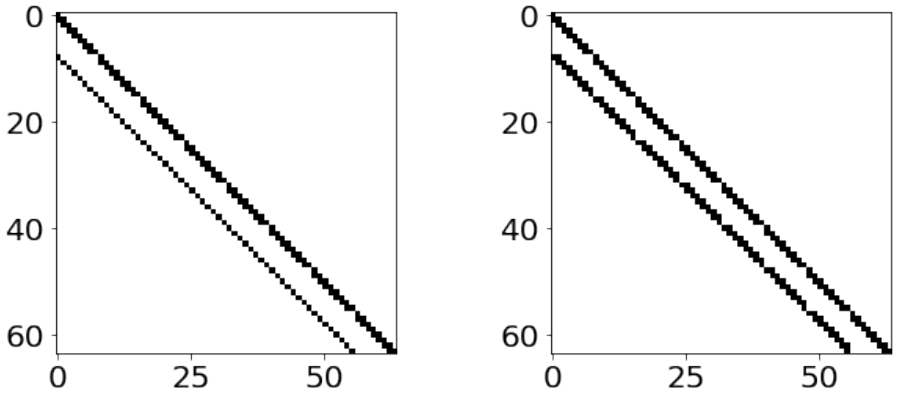
The second method is Thresholded ILU (ILUT), where the fill-in values are kept if they are greater than a specific threshold. Generally this leads to better preconditioners for the same amount of fill-in, as the most significant values are kept. On the downside, it is more difficult to compute and choosing the right threshold is difficult. It is also possible to specify the amount of fill-in and keep the largest n values in magnitude per row, leading to a more predictable memory usage at the expense of some extra computational effort when building the preconditioner.

5.2.2 Block-ILU

Block ILU works by splitting the domain into separate smaller regions and apply ILU to every subdomain discarding any non-zeros outside of the main block diagonal. This is illustrated in Figure 13. One ends up with a block structure in the preconditioner where every block can be considered independently by a thread, without any information about other blocks and as such can be considered in parallel leading to fast iterations. However, this parallelism comes at a cost. The main drawback of this method is that as you have more threads (and thus more blocks) the preconditioner will become less effective **[ref]**. Since the number of threads on a GPU is very large (order of 1024) this may be a significant drawback of using this method **[ref]**.

5.2.3 Fine-grained parallel ILU

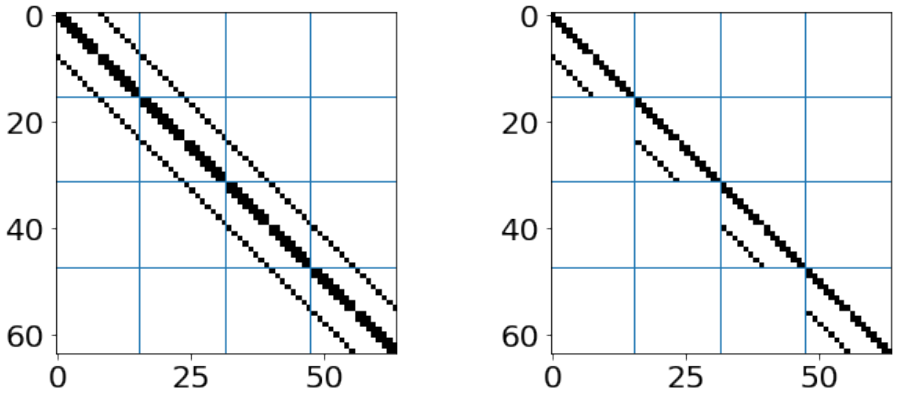
Normally we get parallelism by assigning different rows of the matrix to different threads/cores. But if we have a lot of non-zeros per row, we can process a single



(a) $ILU(0)$ decomposition

(b) ILU(1) decomposition

Figure 12: Sparsity pattern of L in a ILU decomposition of A



(a) Structure of A

(b) Block ILU(0) decomposition, structure of L

Figure 13: Sparsity of Block ILU with a splitting of A into 4 domains

row in parallel by assigning a GPU thread to every non-zero in the row [1]. This is only possible when communication costs between threads are extremely low, such as on a GPU. A big advantage of this method is that it has the same convergence properties as the original ILU decomposition. On the downside, it is very difficult to implement efficiently, and more importantly, it is only beneficial when every row of the preconditioner has a lot of non-zero elements, otherwise, only a small fraction of the available computation power will actually be used.

5.2.4 Iterative ILUT

The iterative ILU method is designed specifically for highly parallel hardware. Instead of applying the preconditioner via forward and backward substitution, the preconditioner can be solved using Jacobi iteration [7, 6]. For many problems just a few Jacobi iterations are needed to get an effective preconditioner, although the optimal number of iterations is hard to determine beforehand.

Chow 2018 [7] also proposes a method for iteratively approximating the ILU decomposition in parallel for a given sparsity pattern. Later, Anzt 2018 [2] proposes a method called ParILUT for iteratively updating the sparsity pattern to further improve the preconditioner while keeping the same number of nonzero elements. Both methods were shown to yield good preconditioners in few iterations.

Construction ILU We will first look at the parallel construction of an ILU decomposition for a given sparsity pattern [7]. The method is based on the property in the conventional ILU, that for the sparsity pattern S of L and U we have:

$$(LU)_{ij} = a_{ij} \quad (i,j) \in S.$$

The elements of L and U can thus be written out explicitly as

$$l_{ij} = \frac{1}{u_{jj}} \left(a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj} \right)$$
$$u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj}.$$

This is also what is used to construct the conventional ILU decomposition. Although these equations are non-linear, Chow proposes to use a fixed-point iteration using this property. To improve the convergence properties, first the matrix is scaled by symmetric diagonal scaling $\hat{A} = DAD$, where D is the diagonal matrix such that the scaled matrix has unit diagonal. We start from some initial guess with the desired sparsity pattern and then update this via the procedure described in Algorithm 2. Although this name is not proposed by Chow, we shall refer to it as ParILU. For starting the iterations, Chow proposes two initial guesses, one they call the *standard initial guess*, which is to take the

```
Initial guess: L, U
For sweep = 0, 1, ... (until convergence)
    Parallel For (i, j) ∈ S
        If i > j
            l_ij = (1/u_jj) (a_ij − Σ_{k=1}^{j−1} l_ik u_kj)
        Else
            u_ij = (1/l_ii) (a_ij − Σ_{k=1}^{i−1} l_ik u_kj)
        End
    End
End
```

Algorithm 2: Parallel ILU Factorization [7]

lower and upper triangular parts of $\hat{A}$. Then the *modified initial guess* is to take the same, but now scale the rows of L and the columns of U such that the product LU has a unit diagonal. If we are solving a sequence of linear problems where the sparsity pattern does not change, we can take the LU decomposition of the previous linear problem as initial guess, which can significantly increase performance.

The number of sweeps needed to get a good preconditioner is relatively low, in most cases somewhere between 1 to 5, even though it is not yet converged, the number of iterations needed in the Krylov method is almost equal to the number that the standard ILU preconditioner would need.

Preconditioner application To apply this preconditioner in parallel, we will need to replace the forward-/backward substitution steps by a parallel triangular solving algorithms. One method is to use Jacobi iteration (see Section 3.1) on the triangular matrices [7]. Let D_L and D_U be the diagonal of L and U respectively, then the Jacobi iteration is given by:

$$y_{k+1} = \left(I - D_U^{-1}U\right) y_k + D_U^{-1} b$$

and when we are satisfied with the upper triangular solution we will use

$$x_{k+1} = \left(I - D_L^{-1}L\right) x_k + D_L^{-1} y$$

to complete the preconditioning step. As the iteration matrix $(I - D_U^{-1}U)$ has zero diagonal, convergence is guaranteed. Unfortunately, the solution may diverge before converging. If the factors are (close to being) diagonally dominant, only few iterations (order of 1 to 6) are needed to improve the method over forward/backward substitution. Note that although the number of Krylov iterations is usually increased, the total time is reduced due to the improved parallelism.

ParILUT construction ParILUT is an extension of ParILU where the pattern is dynamically updated [2]. The method tries to find a lower and upper triangular sparse matrix that approximates A. Formally this method aims to minimize

$$\|LU - A\|_F$$

using a predetermined number of non-zero elements in each of the factors. The idea is to alternate one sweep of ParILU with an update of the sparsity pattern. The exact procedure is given in Algorithm 3. The candidate locations are the

```
Initial guess: L, U
For sweep = 0, 1, ... (until convergence)

    Add m_L and m_U candidate locations to S_L and S_U respectively.
    Do one ParILU sweeps
    Remove the m_L and m_U elements of smallest magnitude from S_L and S_U
    Do one ParILU sweep

End
```

Algorithm 3: ParILUT algorithm [2]

points that are a non-zero of the residual matrix $R = LU - A$, one can choose to add only the candidate locations corresponding to the biggest residual elements in magnitude, however, Anzt considers it best to add all candidate points. For the restriction step, the smallest elements of L and U are removed until the desired number of non-zeros is left (always keeping the diagonal). This is hard to perform in parallel, thus, Anzt suggests to divide the matrix in blocks of rows and for each block compute a local threshold which would remove the desired number of non-zeros from that block of rows. Then, compute a global threshold as the median of all local thresholds. This results in the total number of non-zeros to fluctuate a bit, but overall it should not diverge.

Once the preconditioner is constructed, Jacobi iteration is again used to apply the preconditioner in the Krylov method.

SPD variants There are also two variants for SPD matrices, we shall call them ParIC [7] for the static sparsity pattern and ParICT [2] for the method with updates of the pattern. These methods are very similar to ParILU and ParILUT respectively, except that it only needs to compute one Cholesky factor, saving half the time and memory.

5.3 Sparse Approximate Inverse Preconditioners (SPAI)

Instead of decomposing the matrix into a lower and upper part, one may also approximate the inverse of the matrix explicitly with another sparse matrix [5, 9]. This method is known as Sparse Approximate Inverse Preconditioning

(SAIP or SPAI). Usually, the idea is to construct a preconditioner M with a predetermined sparsity pattern that minimizes the error

$$\|I - AM\|_F .$$

This preconditioner can be constructed and applied in parallel. Due to its parallel performance, the preconditioner can have very fast iterations, but generally the convergence rate is quite slow [3]. Furthermore, the choice of sparsity pattern heavily influences the performance of the preconditioner **[source?]**. Regardless, in some cases it can outperform ILU based methods just because of its fast iterations [3, 9].

5.3.1 Construction

The way that the SPAI preconditioner is constructed is by considering every column of M independently via the relation [17]:

$$\|I - AM\|_F^2 = \sum_{k=1}^{N} \left\|(I - AM)\, e_k^T\right\|_2^2 = \sum_{k=1}^{N} \|Am_k - e_k\|_2^2 .$$

Where the m_k form the columns of M and e_k are the columns of I. We can choose each m_k independently of the others, making this a parallel method. Furthermore, to save computational costs, we can use the predetermined sparsity pattern of m_k to reduce a big minimization problem into a much smaller one. We denote the prescribed sparsity pattern of m_k by $\mathcal{J}_k$, the set of the non-zero indices. Then we only need to consider the columns of A corresponding to $\mathcal{J}_k$. Furthermore, as A is also sparse, there are only a few rows that have a non-zero element in the considered rows, we denote these rows with index $\mathcal{I}_k$, formally:

$$\mathcal{I}_k = \left\{ i \in \{1, \ldots, N\} : \sum_{j \in \mathcal{J}_k} |a_{ij}| \neq 0 \right\}$$

Now we can drop all rows and columns that would be guaranteed to lead to a zero element in the product Am_k (due to the sparsity of m_k and A):

$$\begin{aligned} \hat{A}_k &= A\left(\mathcal{I}_k, \mathcal{J}_k\right) \\ \hat{m}_k &= m_k\left(\mathcal{J}_k\right) \\ \hat{e}_k &= e_k\left(\mathcal{I}_k\right) \\ \|Am_k - e_k\|_2^2 &= \left\|\hat{A}_k \hat{m}_k - \hat{e}_k\right\|_2^2 . \end{aligned}$$

Restricting the system this way does not change the result, but the new least squares problem is much smaller than the original problem and can be solved easily using QR decomposition for example. After solving the least squares problem, we can use m_k to assemble the matrix M explicitly. The construction here leads to a right preconditioner, but in a similar way we could also have constructed a left preconditioner. It must be noted that generally M will not be symmetric, even when A is SPD, thus conjugate gradient can not be used with this preconditioner.

5.3.2 Sparsity pattern

The choice of the sparsity pattern greatly influences the performance of SPAI. According to Lukash 2012 common choices are the main diagonal, similar to Jacobi preconditioning, the sparsity pattern of A, and A^2, A^3, etc. [17]. Generally, more elements leads to better approximations, at the cost of computation power and memory. It is also possible to update the sparsity pattern of the approximate inverse dynamically, adding candidate non-zeroes and dropping small elements. However, they also found that as the construction of the preconditioner is very expensive, the reduced number of iterations does not outweigh the cost of iteratively updating the sparsity pattern, unless the original SPAI did not lead to convergence.

5.3.3 FSAI (Factored Sparse Approximate Inverse)

The idea of FSAI [11, 17] is to find an approximate inverse of a factorization of A. The advantage being that for a SPD matrix A, we get a SPD sparse preconditioner $M = CC^T$ so we can use the conjugate gradient method. In contrast to normal factorization, this preconditioner can be applied directly and fully in parallel, on the downside, the Krylov method needs more iterations to converge.

5.3.4 ISAI (Incomplete Sparse Approximate Inverse)

Anzt 2018 [3] proposes ISAI, which is a new method comparable to SPAI that approximates an inverse of A on a given sparsity pattern for M. The main difference however, is that instead of solving the least squares problem

$$\min_{m_k} \|A\left(\mathcal{I}_k, \mathcal{J}_k\right) m_k\left(\mathcal{J}_k\right) - e_k\left(\mathcal{I}_k\right)\|_2^2$$

it tries to minimize the least squares problem restricted to $\mathcal{J}_k$, where $\mathcal{J}_k \subset \mathcal{I}_k$:

$$\min_{m_k} \|A\left(\mathcal{J}_k, \mathcal{J}_k\right) m_k\left(\mathcal{J}_k\right) - e_k\left(\mathcal{J}_k\right)\|_2^2 .$$

As the sub-matrix $\tilde{A}_k = A\left(\mathcal{J}_k, \mathcal{J}_k\right)$ is square, this can be solved exactly if $\tilde{A}_k$ is non-singular. This ISAI preconditioner is cheaper to compute, and according to [3] it also leads to better convergence. This method can be applied as is on SPD matrices, but as we will see in Section 7, this is not beneficial, instead it is better to combine ISAI with ILU, similar to FSAI.

ILU + ISAI In the combination of ILU with ISAI, the first step is to compute the ILU decomposition of A. Next, the triangular matrices are approximated by a sparse approximate inverse, i.e. $M_L \approx L^{-1}, M_U \approx U^{-1}$ where M_L has the same sparsity pattern as L (or optionally $L^2, L^3, ...$). Then, these approximate inverses may be used as preconditioners directly. Even though the quality of the preconditioner is then decreased compared to standard ILU, it is now fully parallelized, so that there may still be a significant speed-up [3].

6 Deflation methods

Deflation is related to preconditioning in the sense that we try to transform a linear system $Ax = b$ into another system that is easier to solve. But in contrast to conventional preconditioning, the system is now split into two independent systems using projections onto a subspace and its complement. The idea is then to solve the two sub-systems independently. In Section 6.1 we will look at how this works conceptually and mathematically. Then, in Section 6.2, we will discuss some different choices for projections and their properties. Note that the deflation method can easily be combined with preconditioning.

6.1 How does it work

In this report we will assume A to be SPD for simplicity, following the proof by Jönsthövel 2012 [13], for general matrices there are more difficult proofs available[10, 25]. In deflation we chose some subspace $S \subset \mathbb{R}^n$ to deflate and let the columns of $V \in \mathbb{R}^{n \times k}$ be a basis of S. We will split the solution x into two parts, one part in the subspace S and one part in its complement S^c:

$$x = \left(I - P^T\right) x + P^T x$$

where P is a projection matrix, defined by

$$P = I - AV \left(V^T AV\right)^{-1} V^T.$$

When V has rank k, the product $E = V^T AV$ is SPD and thus invertible. Usually we take k to be small, so that E and E^{-1} can be computed explicitly or via QR-decomposition. Using this, we can compute one part of the solution explicitly:

$$\left(I - P^T\right) x = VE^{-1}V^T Ax = VE^{-1}V^T b.$$

The other part of the solution $P^T x$ still has to be computed. Note that

$$PA = AP^T$$

We solve the projected problem using our preferred Krylov method (CG)

$$PA\hat{x} = Pb. \tag{6.1}$$

We do have to note that PA is singular and thus the solution is not unique. However, the projected solution $P^T\hat{x}$ *is* unique and equal to $P^T x$. So that the total solution becomes:

$$x = VE^{-1}V^T b + P^T\hat{x}. \tag{6.2}$$

Where $VE^{-1}V^T b$ is calculated explicitly, while $P^T\hat{x}$ is found by solving $PA\hat{x} = Pb$ using a Krylov method. Now the big advantage comes from the fact that the subspace S is no longer part of the problem $PA\hat{x} = Pb$, thus effectively the subspace S is "hidden" from the Krylov method [13]. As we saw earlier,

the convergence depends condition number, and thus on the eigenvalues of the matrix A:

$$\kappa(A) = \frac{\lambda_N}{\lambda_1}$$

When the eigenvectors corresponding to the lowest few eigenvalues are deflated, the effective condition number then becomes

$$\kappa_{\text{eff}}(A) = \frac{\lambda_N}{\lambda_k},$$

which is usually a big improvement.

The full DPCG is given in Algorithm 4, it is also possible to apply classical PCG to System (6.1) and use Equation (6.2) to find the full solution.

Initial guess: x_0
$r_0 = b - Ax_0$
$\hat{r}_0 = Pr_0$
$y_0 = M^{-1}\hat{r}_0$
$p_0 = y_0$
For $j = 0, 1, \ldots$ (until convergence)

$\hat{w}_j = PAp_j$
$\alpha_j = \frac{\hat{r}_j^T y_j}{\hat{w}_j^T p_j}$
$\hat{x}_{j+1} = \hat{x}_j + \alpha_j p_j$
$\hat{r}_{j+1} = \hat{r}_j - \alpha_j \hat{w}_j$
$y_{j+1} = M^{-1}\hat{r}_{j+1}$
$\beta_j = \frac{\hat{r}_{j+1}^T y_{j+1}}{\hat{r}_j^T y_j}$
$p_{j+1} = y_{j+1} + \beta_j p_j$

End
$x = ZE^{-1}Z^T b + P^T \hat{u}_{j+1}$

Algorithm 4: Deflated Preconditioned Conjugate Gradient Method as taken from [13]

6.2 Choice of deflation space

The deflation method allows a lot of liberty in the choice of the deflation subspace. We usually look at this subspace by defining its basis, the columns of V.

6.2.1 Using exact eigenvalues

From a theoretical point of view it would be ideal to deflate the eigenvectors corresponding to "bad" eigenvalues, which are almost always the lowest eigenvalues. In practice however it is difficult to compute this. If you can determine the

eigenvalues exactly, you probably do not need a computer to solve the system. However, it can be used to formally prove statements about the convergence rate from a theoretical point. And more importantly, you can view other methods as perturbations of using the exact eigenvalues.

6.2.2 Using approximate eigenvalues

This method comes closest to the theoretically ideal deflation. The eigenvectors of the system are approximated via some iterative method, such as the Lanczos algorithm. It turns out that the approximation does not have to be very precise to be effective [14]. Unfortunately, finding approximate eigenvectors is very expensive, so it is often faster to *not* deflate the eigenvectors this way. However, when one needs to solve the same system many times, it may be beneficial to approximate these eigenvalues once and use them many times to speed up convergence.

6.2.3 Reusing eigenvectors from repeated/restarted GMRES

Instead of approximating the eigenvectors beforehand, we can save a bit of computation power by approximating the eigenvectors based on information found by the GMRES algorithm. This is still expensive in practice [23], but a part of the calculation has to be done anyway in order to solve the linear system, it is cheaper than approximating eigenvalues beforehand. After the linear solve is completed, some of the information that is stored in the GMRES iterations can be condensed into an approximate eigenvector. Reusing this eigenvector for the next system to be solved to speed up any subsequent solves. To a lesser extend, this method can also be applied to restarted GMRES.

6.2.4 Subdomain deflation

When the domain is split into subdomains, we can consider the indicator function on each domain

$$I_{D_i}(v) = \begin{cases} 1 & v \in D_i \\ 0 & \text{otherwise} \end{cases}.$$

We can turn this into a vector which has elements 1 if the element on that position is in that domain. Then all these vectors together can form the basis for the deflation space. The advantage of this is that it is very easy to construct these deflation vectors and all vectors are sparse. Unfortunately, this deflation space is not always effective as it need not correlate with the underlying physics, although the domains can be chosen based on physical properties [15, 16].

6.2.5 Levelset deflation

This deflation method is based on the underlying physics. It is very similar to subdomain deflation in that we use a indicator function to construct the vectors, but we do not need to split the domain into actual subdomains. Instead, we

group connected vertices together based on physical properties, such as stiffness, or permeability. We may even split these regions further using classical domain splitting techniques. Then we define the deflation vectors as the indicator vectors on these groups of vertices. Note again that these vectors are sparse. This deflation method is computationally quite efficient and can sometimes be very effective [23].

6.2.6 Rigid body modes

This is similar to, and slightly more advanced, than levelset deflation but applied to mechanical problems specifically. Again we split the domain into levelsets based on stiffness. We will then proceed to pretend that each group is a rigid body, i.e. the group as a whole can move and rotate in all directions, but it cannot bend or stretch. As a result, we get the following deflation vectors (which depend on the positions of the nodes) [16]:

$$\begin{aligned}
&\text{translation along x-axis}[1,0,0,0,0,0]\\
&\text{translation along y-axis}[0,1,0,0,0,0]\\
&\text{translation along z-axis}[0,0,1,0,0,0]\\
&\quad\text{rotation about x-axis}[0,-z,y,1,0,0]\\
&\quad\text{rotation about y-axis}[z,0,-x,0,1,0]\\
&\quad\text{rotation about z-axis}[-y,x,0,0,0,1]
\end{aligned}$$

For mechanical problems, these deflation vectors generally lead to better deflation than the levelset deflation, and can be computed beforehand. When only considering the translations, the similarity to levelset deflation is very apparent. The strength of this method comes from the idea that the smallest eigenvectors of a system can be approximated by a linear combination of these rigid body modes, even though its not perfect it can perform very well in practice [16].

7 Preliminary experimentation

7.1 Test problems

We try different preconditioners on a test problem. The test problems are based on the 2D finite difference heat problem with Dirichlet boundary conditions.

1. Matrix 1 is a simple 2D Poisson matrix using a 5-point stencil
2. Matrix 2 is a finite difference heat problem using a 5-point stencil, for this matrix, the problem has 3 regions where the conductivity is 100 times higher than in other areas, as illustrated in Figure 14. This is noticeable in the convergence plots by the three bends in the lines

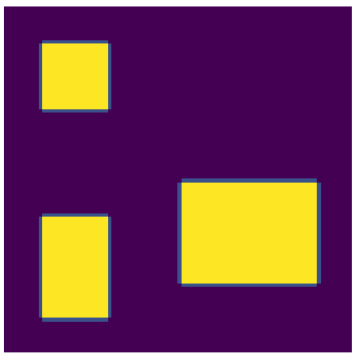


Figure 14: Regions of conductivity for matrix 2, yellow corresponds to the area with 100× the conductivity of the purple area

Both test problems are SPD, yet, as not all preconditioners are symmetric, we will use full GMRES in all cases for easy comparison of the effect of the different preconditioners. We consider the method converged when the norm of the residual is less than 10^{-8}.

7.2 ParILU

I tested ParILU with two different sparsity pattern: A (corresponding to ILU(0)) and A^2 (corresponding to ILU(1)). The test problem is 30×30 heat equation with high contrast (test problem 2), with symmetric diagonal scaling (as suggested in Section 5.2.4). The convergence using GMRES is shown in Figure 15 for different number of construction sweeps and Jacobi iterations. We can see that as we increase the amount of sweeps/Jacobi iterations the convergence of ParILU approximates that of ILU. However, possibly the fastest times are

achieved with a relatively inaccurate approximation, due to faster iterations, this is to be researched. The reference ILU decomposition is implemented using the `ilupp` python package, which is turn is based on `ilu++` [18].

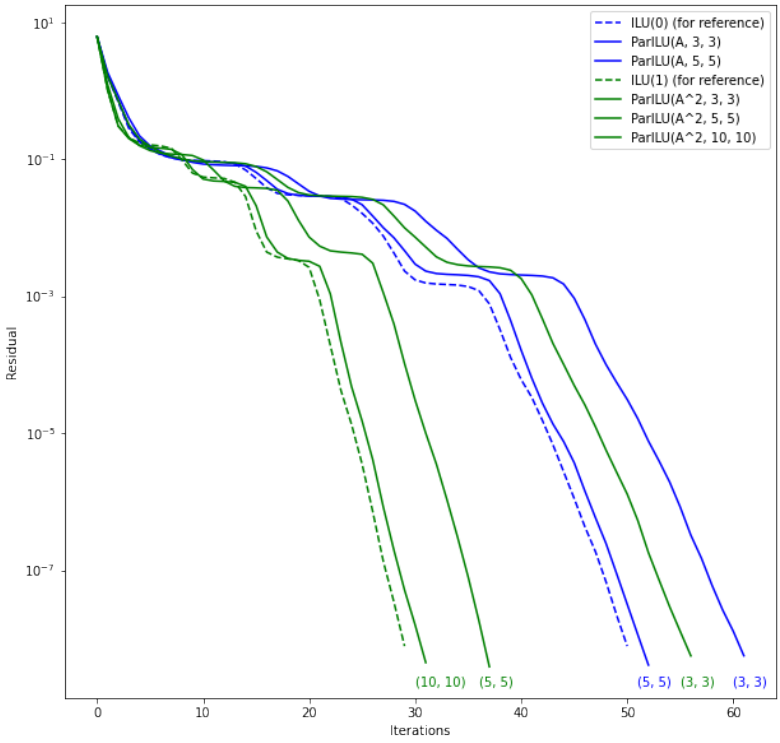


Figure 15: Convergence test for ParILU. The dashed lines represent conventional ILU. The blue lines correspond to ILU(0) and its parallel approximation. The green lines correspond to ILU(1) and its corresponding approximation. For the paramaters in ParILU stand for ParILU(sparsity pattern, construction sweeps, Jacobi iterations)

7.3 ParILUT

Using the same test problem we can test ParILUT. In our implementation, it was built such that the desired number of non-zeros per factor can be chosen. As we are using a 5-point stencil. The matrix has (for most rows) 5 non-zeros per row.

The ILU(0) decomposition then has 3 non-zeros per row for both factors, as the diagonal is both in the lower and upper factor. The convergence for ParILUT is with some parameters is shown in Figure 16. A overview is also given in Table 2. From this we can see that the `scipy.sparse.linalg.spilu` has the most non-zeros while having the worst convergence. This is quite unexpected and I do not understand why this happens. Furthermore, we see that for almost same number of non-zeros, ParILUT(3600 nz) has slightly better convergence than ILU(1) and ParILU(A^2), this is possibly due to the (few) extra non-zeros or better sparsity pattern, as **[Anzt 2018]** explained.

Method	nnz	nnz/row	GMRES iterations
ILU(0)	2640	2.93	52
ParILU(A, 3, 3)	2640	2.93	63
ParILU(A, 5, 5)	2640	2.93	53
ILU(1)	3566	3.96	31
ParILU(A^2, 5, 5)	3481	3.87	38
ParILU(A^2, 10, 10)	3481	3.87	32
ParILUT(10, 15, 3600nz)	3600	4.00	28
ParILUT(10, 15, 4500nz)	4500	5.00	22
`scipy.sparse.linalg.spilu`	5014	5.57	64

Table 2: Overview of convergence and number of non-zeros for different ILU implementations.

7.4 SPAI

We test the same problem (on a 30×30 grid) with SPAI using different sparsity patterns. The convergence is shown in Figure 17. The SPAI preconditioner yields a worse convergence rate than IC for the same number of non-zeros, but again, due to its fast iteration, it may still be a good choice. SPAI(IC) stands for SPAI applied to the $IC(0)$ decomposition, this is the only symmetric SPAI preconditioner in this test and could thus have been used with CG.

7.5 ISAI

Figure 18 shows the convergence for the ISAI preconditioner using several sparsity patterns. Again ISAI(IC) stands for ISAI applied to the $IC(0)$ decomposition. Table 3 shows the number of iterations needed for both SPAI and ISAI preconditioners. It has to be noted that SPAI(A) is better than ISAI(A), for the other sparsity patterns and approximate inverses of IC(0), both methods perform very comparable in terms of convergence. It also has to be noted that the ISAI preconditioner is faster to construct than the SPAI preconditioner for the same sparsity pattern.

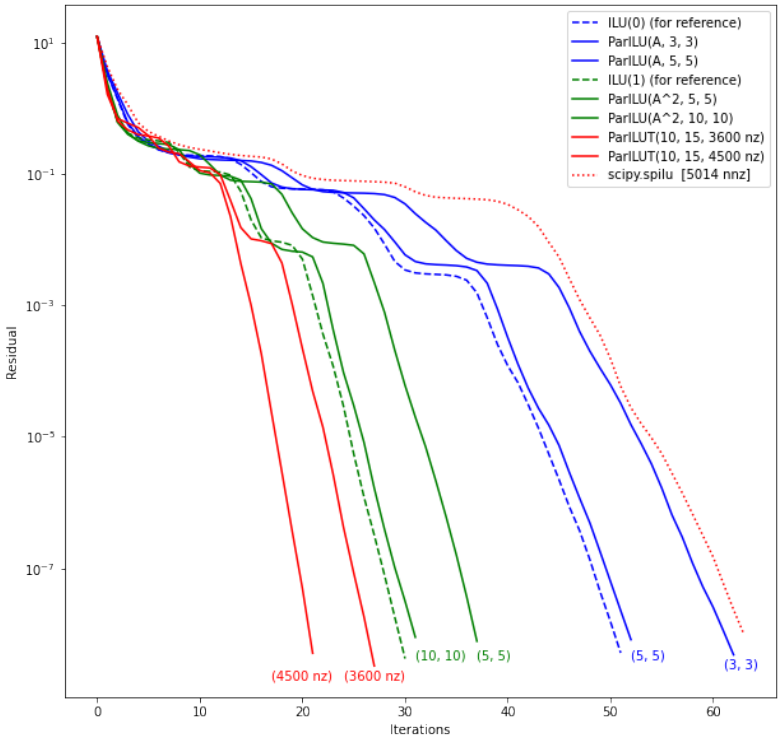


Figure 16: Convergence test for ParILUT. The dashed lines represent conventional ILU. The blue lines correspond to ILU(0) and its parallel approximation. The green lines correspond to ILU(1) and its corresponding approximation. For the parameters in ParILU stand for ParILU(sparsity pattern, construction sweeps, Jacobi iterations) and for ParILUT(construction sweeps, Jacobi iterations, allowed number of non-zeros)

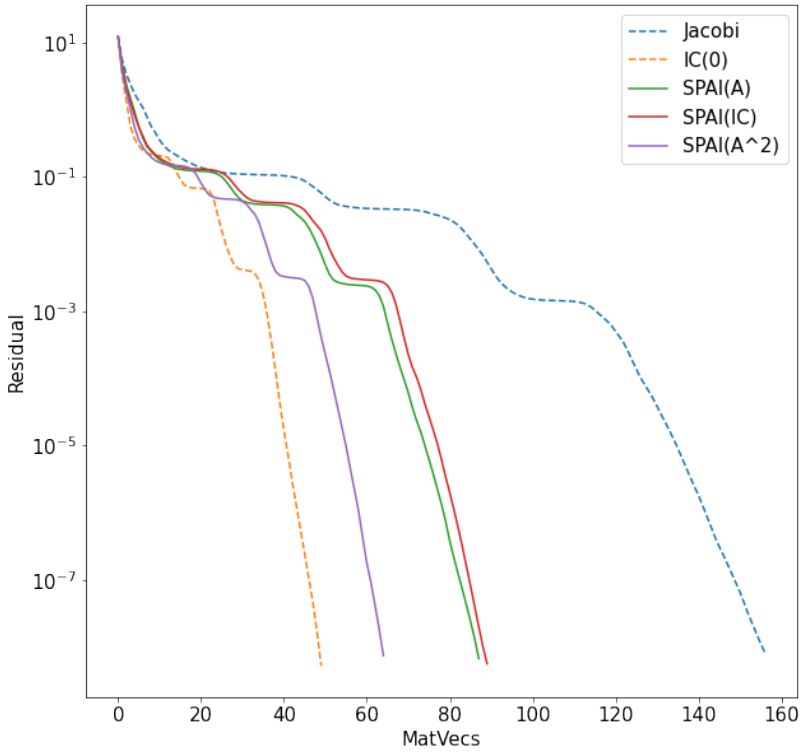


Figure 17: Convergence test for SPAI

Method	GMRES iterations
Jacobi	157
IC(0)	50
SPAI(A)	88
SPAI(A^2)	65
SPAI(IC)	90
ISAI(A)	121
ISAI(A^2)	63
ISAI(IC)	86

Table 3: Overview of convergence for SPAI and ISAI preconditioners.

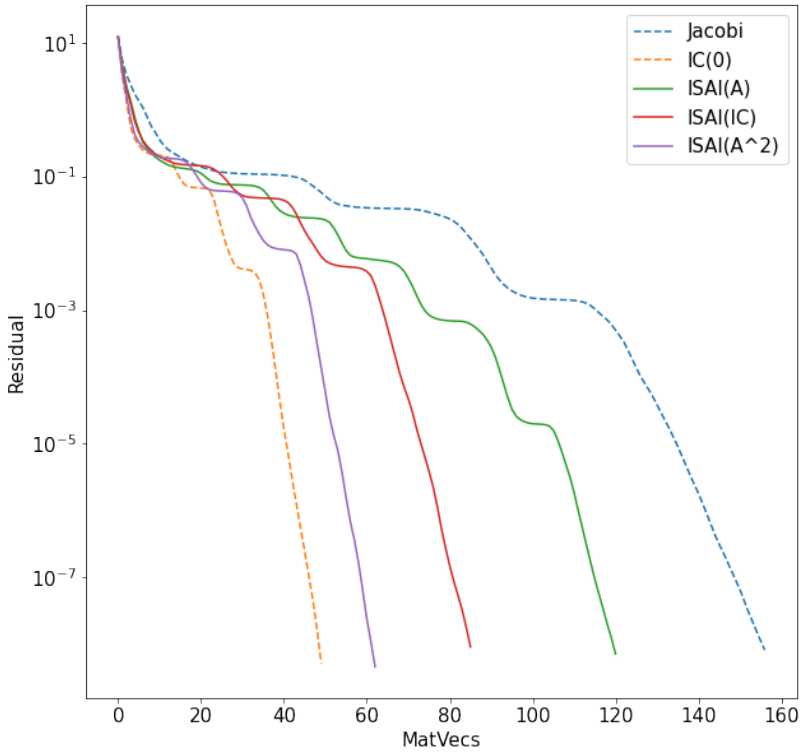


Figure 18: Convergence test for ISAI

7.6 approximate eigenvalue deflation

It has been noted above that for problem 2 there is a high contrast in conductivity, this leads to slow convergence. In particular, in every convergence plot we see three bumps, corresponding to the 3 slowly converging eigenvalues. On a small grid (30×30) the matrix is relatively small (900×900) and we can compute the eigenvalues numerically. Figure 19 shows the 4 eigenvectors corresponding to the 4 smallest eigenvalues. As can be seen, the large conductivity regions are easily visible in the eigenvectors with some extra smoothing going on in the neighborhood. Figure 20 shows the convergence plot with varying number of eigenvalues deflated. For both the Jacobi preconditioner and the $IC(0)$ preconditioner. We can see from the convergence that as the number of deflation vectors increases, the bumps in the convergence are removed one by one. Furthermore, we see that the first 4 eigenvectors have a significant impact on the convergence, whereas the next few eigenvectors barely improve convergence. All in all, using 4 eigenvectors reduces the amount of iterations by over 40%.

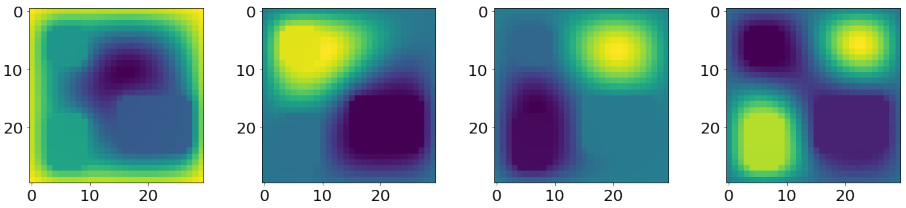
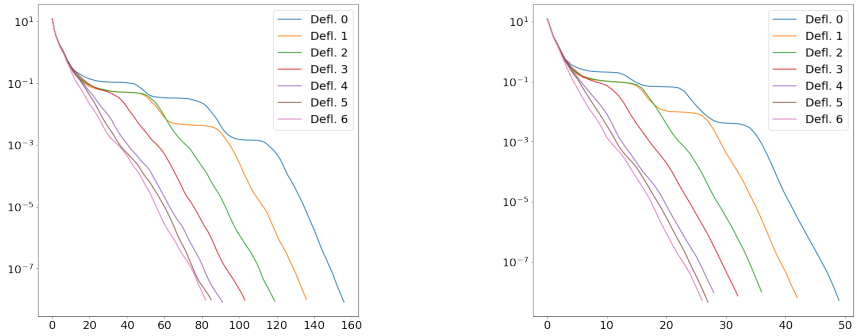


Figure 19: Deflation vectors corresponding to the lowest 4 eigenvectors.

7.7 Levelset deflation

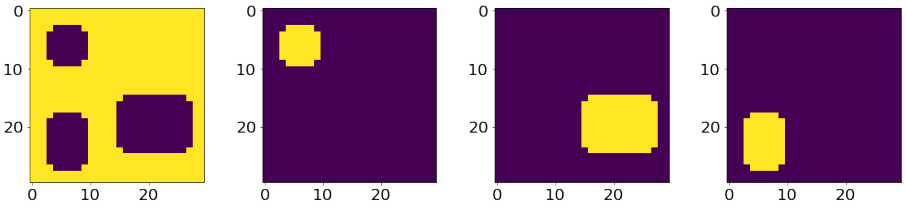
Instead of using eigenvectors, we can opt for the levelset vectors. These vectors are based on the physical properties of the problem. In the case of our test problem, we could split it into 4 vectors based on conductivity. These vectors are illustrated in Figure 21. They do not represent the eigenvectors, but a linear combination of these may approximate an eigenvector good enough. The convergence using these deflation vectors is shown in Figure 22. Interestingly, the convergence for 4 levelset vectors is almost equal to the convergence of the 4 lowest eigenvalues, this is very nice as this means that with 4 very cheap deflation vectors, we can improve convergence significantly, as well as with 4 very expensive deflation vectors.



(a) Using Jacobi preconditioning

(b) Using IC(0) preconditioning

Figure 20: Convergence for eigenvector deflation for different amount of deflated eigenvectors.



(a) Background (b) Region 1 (c) Region 2 (d) Region 3

Figure 21: Deflation vectors corresponding to the levelsets.

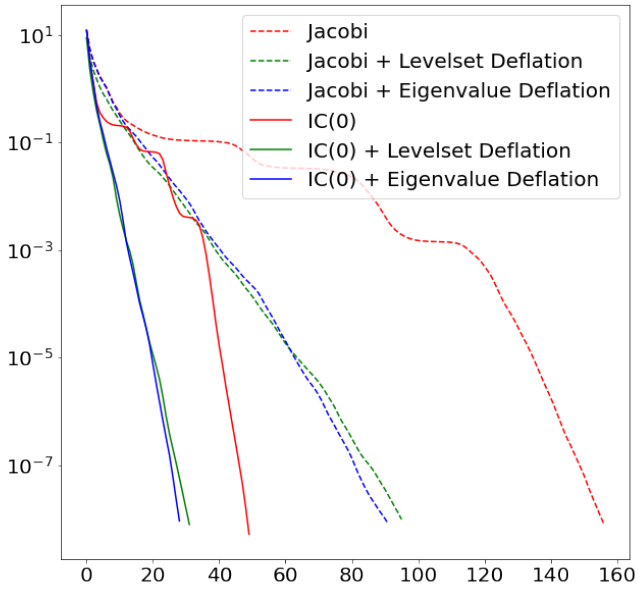


Figure 22: Convergence using levelset deflation versus eigenvalue deflation (both with 4 vectors).

8 Method

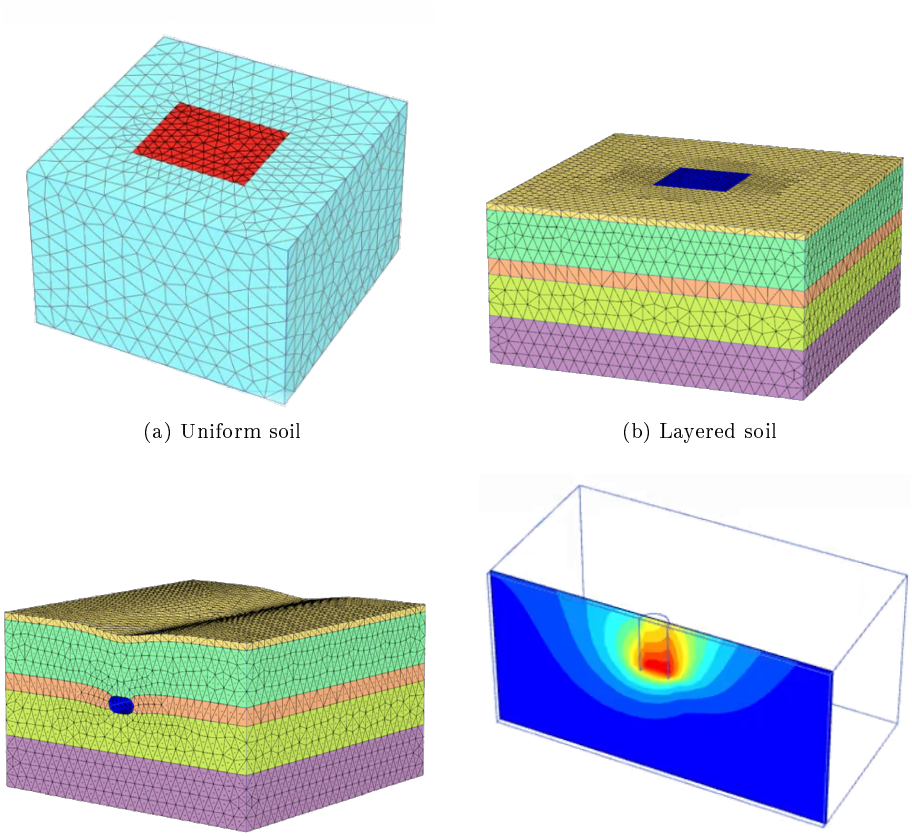
In this report we investigate different combinations of preconditioners, Krylov methods and deflation vectors. As the performance of each combination depends on the problem it is applied to, we will want to apply it to a set of test problems that are representative for the cases that Plaxis users will encounter in practice. The performance, in terms of time and memory, of the different components of the solver will then be analyzed, for different preconditioners and sets of deflation vectors. This will be compared to the PICOS solver as well as the PARDISO solver that are currently available in Plaxis 3D. The test models we will use will be a stiff structure embedded in a softer soil, as this leads to high contrast in stiffness and thus ill-conditioned matrices.

1. A uniform soil with a load in the middle. As to verify that the GPU algorithm performs well for very simple test cases.

2. A layered soil with different stiffness per layer, similar to Lingen 2014 [16], model 1. The different soil layers have an order of magnitude different stiffness.

3. Tunnel through layered soil, similar to Lingen 2014 [16], model 2. The different soil layers have an order of magnitude different stiffness, and the concrete lining inside the tunnel has a stiffness several orders of magnitude larger than the soil.

4. Loading of suction pile in clay, as taken from a Plaxis 3D tutorial [20]. The suction pile is a steel cylinder closed at the top and is used to anchor large structures to the seafloor. The cylinder is very stiff compared to the surrounding soil, leading to such an ill-conditioned problem that the tutorial suggest to use PARDISO as it solves the problem faster than PICOS.



(a) Uniform soil

(b) Layered soil

(c) Tunnel

(d) Suction pile

Figure 23: Two test problems

Nomenclature

List of abbreviations

Abbreviation	Long version
SPD	Symmetric Positive Definite (matrix)
SPSD	Symmetric Positive Semi-Definite (matrix)
FEM	Finite Element Method
PARDISO	PARallel Direct Solver
PICOS	Plaxis Iterative COncurrent Solver
BIM	Basic Iterative Method
QR-decomposition	Decompose $A = QR$ where Q is orthogonal and R upper triangular
nz/nnz	non-zeros / number of non-zeros
Preconditioners	
LU	Lower - Upper triangular decomposition. $A = LU$
ILU(k)	Incomplete LU decomposition, k is denotes allowed fill-in, $A \approx LU$
IC(k)	Incomplete Cholesky decomposition, $A \approx CC^T$
SPAI	SParse Approximate Inverse
ISAI	Incomplete Sparse Approximate Inverse
FSAI	Factored Sparse Approximate Inverse
ILUT(k, τ)	Thresholded ILU, k is level of fill-in, τ is the cutoff threshold
ParILU	Parallel ILU
ParILUT	Parallel Thresholded ILU
ICCG(k)	Incomplete Cholesky (k) Conjugate Gradient
AMG	Algebraic Multi-Grid
Krylov method	collection term for CG, BiCGSTAB, GMRES, IDR etc
CG	Conjugate Gradient
GMRES	Generalized Minimal RESidual
BiCGSTAB	Bi-Conjugate Gradient STABelized
IDR(s)	Induced Dimension Reduction (s being the dimension of the subspace)

Abbreviation	Long version
FGMRES	Flexible GMRES
PCG	Preconditioned CG
DPCG	Deflated Preconditioned CG
GPU	Graphics Processing Unit
CPU	Central Processing Unit

Common symbols

Some symbols remain the same throughout the thesis. Here is a list of some mathematical symbols with a well defined meaning, as well as unofficial conventions that this thesis adheres to.

Symbol	Meaning
$\mathbb{R}$	Real numbers, i.e. decimal numbers 5.53, $-\frac{1}{12}$
$\mathbb{N}$	Integers, whole numbers, 7, -3
i, j, k	iteration index or coordinates in a matrix
S	a **set**, in Algorithm 2 it denotes the non-zero coordinates
$p \in S$	"in", to denote that p is an element of a set S.
$S_1 \subset S_2$	subset, all elements of $S_1$are also in S_2, S_2 may have more elements
A	system matrix
M	a preconditioner
P	a (deflation) projection
V	Matrix whose columns span the deflation subspace, generally *not* square
I	Identity matrix, size is implied by the context
U	upper triangular matrix
L	lower triangular matrix
$\lambda_1 ... \lambda_N$	eigenvalues of a matrix, sorted such that λ_1 is the smallest and λ_N the largest
ϵ / eps / epsilon	tolerance, usually we declare convergence when the norm of the residual is smaller than ϵ.
τ	threshold / drop tolerance for elements in a preconditioner

Symbol	Meaning
δ_{ij}	Kronecker delta: $\delta_{ij} = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases}$
Ω	domain, usually domain of integration in FEM
Γ / $\partial\Omega$	boundary of a domain (Ω)
α, β	Other Greek letters usually refer to a real number
n, m	Latin letters usually refer to an integer, m, n usually refer to the size of a matrix
x	solution of a system $Ax = b$
b	right-hand side of a system $Ax = b$
e_k	elements in the *standard basis* of a space. e_k has all zeros except the k-th element which is 1. e.g. $e_2 = [0, 1, 0, ...]$. The length of the vector is implied by the context.
$\kappa_2(A)$	Condition number of a matrix, $\kappa_2(A) = \lambda_N / \lambda_1$
$\kappa_{\text{eff}}(A)$	Effective condition number of a matrix, which actually determines the convergence rate
A^T	Transpose of A
A^{-1}	Matrix inverse of A
$\mathcal{I}$ or $\mathcal{J}$	a set of indices
$A(\mathcal{I}, \mathcal{J})$	the matrix A restricted to keep only the rows $\mathcal{I}$ and columns $\mathcal{J}$
$x(\mathcal{I})$	the vector x restricted to keep only the elements with index in $\mathcal{I}$
$a_{i,j}$ or $(A)_{i,j}$	the element on row i and column j of matrix A.
x_k	approximate solution in an iterative method after k iterations
x_i	i'th element of a vector x. A bit conflicting with previous definition, but usually clear from context.
r_k	residual after k steps: $r_k = b - Ax_k$
p_k	(in Krylov methods) search direction in k-th iteration
$\|x\|_2$	2-norm of a vector, defined as: $\|x\|_2 = \sqrt{\sum_{i=0}^{N} x_i^2}$
$\|A\|_2$	2-norm of a matrix, which turns out to be the maximum row sum. Not similar to the 2-norm of a vector.
$\|A\|_F$	Frobenius norm of a matrix: $\|A\|_F = \sqrt{\sum_{i=0}^{N} \sum_{j=0}^{N} a_{i,j}^2}$. Similar to a 2-norm of a vector.

Symbol	Meaning
∇	Nabla, differential operator, defined as $\nabla = \left(\frac{\partial}{\partial x}, \frac{\partial}{\partial y}\right)^T$ (can be 3 dimensional as well based on context)
Δ	Laplace operator, higher dimensional second derivative, defined as $\Delta = \nabla^2 = \nabla \cdot \nabla = \left(\frac{\partial^2}{\partial x^2}, \frac{\partial^2}{\partial y^2}\right)^T$

References

[1] José I Aliaga, Ernesto Dufrechou, Pablo Ezzatti, and Enrique S Quintana-Ortí. An efficient gpu version of the preconditioned gmres method. *The Journal of Supercomputing*, 75(3):1455–1469, 2019.

[2] Hartwig Anzt, Edmond Chow, and Jack Dongarra. Parilut—a new parallel threshold ilu factorization. *SIAM Journal on Scientific Computing*, 40(4):C503–C519, 2018.

[3] Hartwig Anzt, Thomas K Huckle, Jürgen Bräckle, and Jack Dongarra. Incomplete sparse approximate inverses for parallel preconditioning. *Parallel Computing*, 71:1–22, 2018.

[4] Douglas Arnold, Richard Falk, and Ragnar Winther. Finite element exterior calculus: From hodge theory to numerical stability. *Bulletin of the American Mathematical Society*, 47, 06 2009.

[5] Edmond Chow. A priori sparsity patterns for parallel sparse approximate inverse preconditioners. *SIAM Journal on Scientific Computing*, 21(5):1804–1822, 2000.

[6] Edmond Chow, Hartwig Anzt, Jennifer Scott, and Jack Dongarra. Using jacobi iterations and blocking for solving sparse triangular systems in incomplete factorization preconditioning. *Journal of Parallel and Distributed Computing*, 119:219–230, 2018.

[7] Edmond Chow and Aftab Patel. Fine-grained parallel incomplete lu factorization. *SIAM journal on Scientific Computing*, 37(2):C169–C193, 2015.

[8] Elizabeth Cuthill and James McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 1969 24th national conference*, pages 157–172, 1969.

[9] Jiaquan Gao, Kesong Wu, Yushun Wang, Panpan Qi, and Guixia He. Gpu-accelerated preconditioned gmres method for two-dimensional maxwell's equations. *International Journal of Computer Mathematics*, 94(10):2122–2144, 2017.

[10] André Gaul, Martin H Gutknecht, Jorg Liesen, and Reinhard Nabben. A framework for deflated and augmented krylov subspace methods. *SIAM Journal on Matrix Analysis and Applications*, 34(2):495–518, 2013.

[11] Thomas Huckle. Factorized sparse approximate inverses for preconditioning. *The Journal of Supercomputing*, 25(2):109–117, 2003.

[12] Intel. onemkl pardiso - parallel direct sparse solver interface. `https://software.intel.com/content/www/us/en/develop/documentation/onemkl-developer-reference-fortran/top/`

sparse-solver-routines/onemkl-pardiso-parallel-direct-sparse-solver-interface.html, 2021. [Accessed 24-06-2021].

[13] TB Jönsthövel, MB Van Gijzen, S MacLachlan, C Vuik, and A Scarpas. Comparison of the deflated preconditioned conjugate gradient method and algebraic multigrid for composite materials. *Computational Mechanics*, 50(3):321–333, 2012.

[14] Karsten Kahl and Hannah Rittich. The deflated conjugate gradient method: Convergence, perturbation and accuracy. *Linear Algebra and its Applications*, 515:111–129, 2017.

[15] K. B. Kaliszka, C. Vuik, and M. B. van Gijzen. Developing a parallel solver mechanical problems. Master's thesis, Delft University of Technology, 2010. http://resolver.tudelft.nl/uuid:cea77d32-d6df-443c-9cee-ca89e21733ac.

[16] FJ Lingen, PG Bonnier, RBJ Brinkgreve, MB Van Gijzen, and C Vuik. A parallel linear solver exploiting the physical properties of the underlying mechanical problem. *Computational Geosciences*, 18(6):913–926, 2014.

[17] Mykola Lukash, Karl Rupp, and Siegfried Selberherr. Sparse approximate inverse preconditioners for iterative solvers on gpus. In *Proceedings of the 2012 Symposium on High Performance Computing*, page 13. Society for Computer Simulation San Diego, CA, USA, 2012.

[18] Jan Mayer. Ilu++: A new software package for solving sparse linear systems with iterative methods. In *PAMM: Proceedings in Applied Mathematics and Mechanics*, volume 7, pages 2020123–2020124. Wiley Online Library, 2007.

[19] NVidia. Gpu-accelerated ansys fluent. https://www.nvidia.com/en-us/data-center/gpu-accelerated-applications/ansys-fluent/. [accessed 17-08-2021].

[20] PLAXIS. Plaxis 3d - tutorial manual - loading of a suction pile. https://communities.bentley.com/products/geotech-analysis/w/plaxis-soilvision-wiki/45575/plaxis-3d-tutorial-03-loading-of-a-suction-pile, 2018. [accessed 24-06-2021].

[21] J. N. Reddy. *Introduction to the Finite Element Method, Third Edition*. McGraw-Hill Education, New York, 3rd edition. edition, 2006.

[22] Y. Saad. *Iterative methods for sparse linear systems*. Society for Industrial and Applied Mathematics, 2 edition, 2003.

[23] Joost H van der Linden, Tom B Jönsthövel, Alexander A Lukyanov, and Cornelis Vuik. The parallel subdomain-levelset deflation method in reservoir simulation. *Journal of Computational Physics*, 304:340–358, 2016.

[24] C. Vuik and D.J.P. Lahaye. *Scientific Computing*. Delft University of Technology, 2019.

[25] M.C. Yeung, J.M. Tang, and C. Vuik. On the convergence of gmres with invariant-subspace deflation. `http://resolver.tudelft.nl/uuid:f21da1b4-d4ed-4e46-a604-e1a9bdef70de`, 2010.