Delft University of Technology

Master Thesis

GPU acceleration of FEM solver with applications to Geotechnical Engineering

Author:
Jorn Hoofwijk

Student nr:
4396499

Thesis committee:
Prof. dr. ir. C. Vuik (TU Delft)
Dr. ir. S. Brasile (Bentley)
Dr. ir. H.X. Lin (TU Delft)

Additional supervisor:
Dr. ir. M. Parchei-Esfahani (Bentley)

Submitted in partial fulfillment of the requirements for the degree of
Master of Science

in

Applied Mathematics at Delft University of Technology

January 28, 2022

Abstract

In finite element software one has to solve a system of non-linear equations, which is commonly simplified to a sequence of linear system. We research the possibility to solve these systems on a GPU to improve the solve time. We are particularly interested in systems arising from geotechnical models. We compare several combinations of Krylov methods, parallel preconditioners and deflation methods and present a suitable combination. This solver is then compared with existing CPU based solvers in PLAXIS 3D. We show that compared to the current iterative solver, the iteration time can be reduced by 50% up to 85% depending on the problem. While compared to the current direct solver, the memory consumption and initialization time can be reduced significantly.

Contents

1 Introduction

Computers and finite element software have helped engineers to more accurately and quickly investigate structural properties of their designs. To date, a lot of computational power is required to run these simulations quickly. With the advancement of GPU based computing in the past decade, additional 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 [33]) and several research teams have shown the capabilities of using GPU's for certain applications [14]. However, the effectiveness of an iterative method greatly depends on the problem, preconditioner and deflation methods used. Thus, the goal of this 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 and memory usage with the existing solvers in PLAXIS 3D.

Chapter 2 will explain the principles behind a typical finite element discretisation. Chapter 3 will give an overview of what iterative methods are and how one can apply them to efficiently approximate the solution of a matrix problem. Chapter 4 will give a short overview of the current iterative linear solver used in PLAXIS 3D. In Chapter 5 an overview of several parallel preconditioning methods will be given, followed by an overview of some deflation methods in Chapter 6. Some preliminary experimentation of these methods will be shown in Chapter 7. In Chapter 8 we discuss the test problems used to compare the different solvers. In Chapter 9 we provide details on the implementation and look at the influence of different preconditioners and deflation methods on the convergence. Then we show the run time for all combinations of test problem, preconditioner, deflation method and Krylov method in Chapter 10. Finally, in Chapter 11 we present the final conclusions and suggest some future research directions.

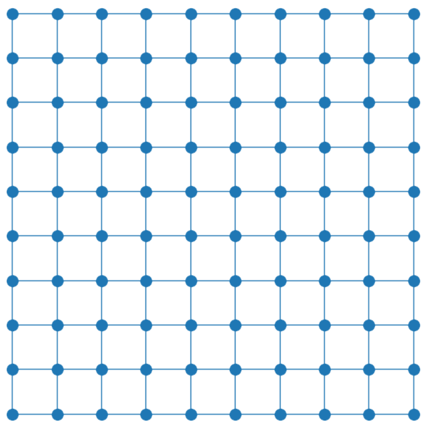
2 Discretization methods

The behaviour of physical systems, such as heat conduction in solids, is modeled using partial differential equations combined with boundary and initial 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 on the grid points. This method is generally very easy to apply on a structured grid (e.g. rectangular), 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 interpret, for example to find the bearing capacity of a bridge.


(a) Finite Difference Grid

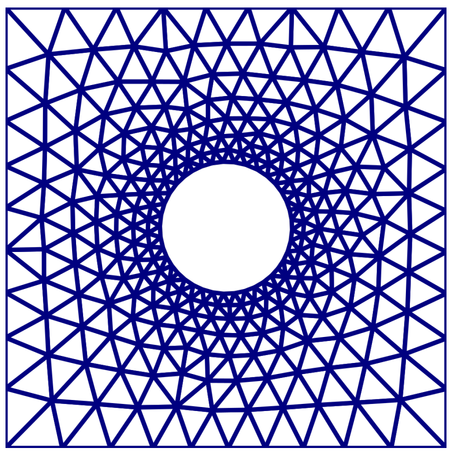

(b) Finite Element Mesh

Figure 1: Discretization options

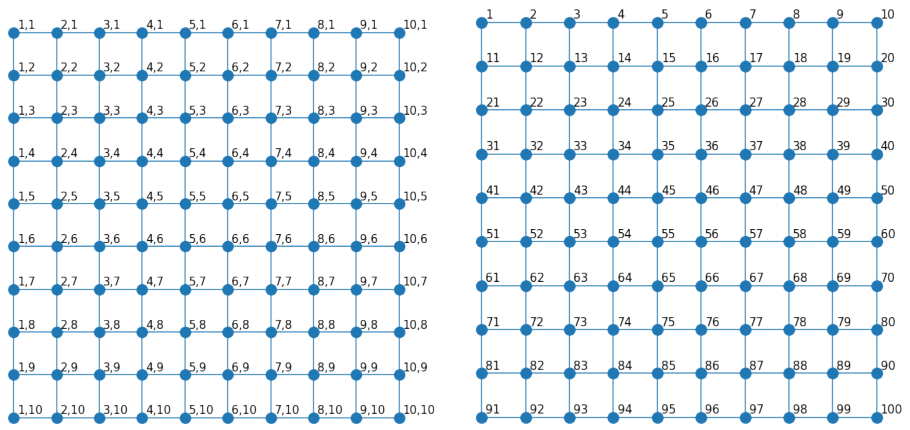
2.1 Finite Difference for Heat equation

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 $g(x,y) = 0$. It means 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 to discretize the domain, we define the solution



(a) Grid numbering (b) Grid renumbering

Figure 2: Discretization options

on a regular grid by $u_{i,j}$, 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 of the solution u (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 from 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, which are usually 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 being 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 ignored. We also see that we get a banded structure, meaning that all non-zero values lie within a certain sized band around the diagonal.

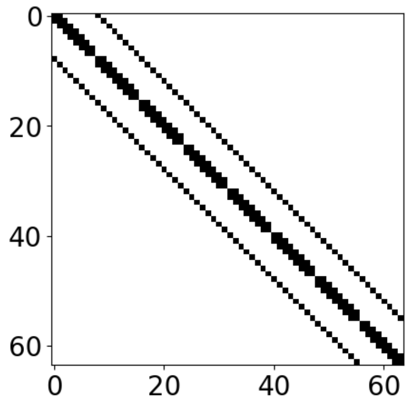


Figure 3: Structure of A

2.2 Finite Element for heat equation

For the finite element method 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, i.e.

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

We can then split this into

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

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 "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 is 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 have three non-zero basis functions. In general form, these equations are given by Equation 2.12.

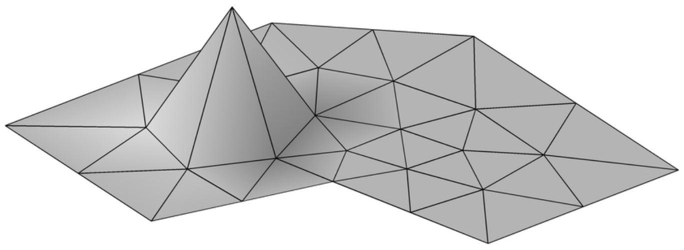


Figure 4: Basis function for linear triangular element, as taken from [6]

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

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 conductivity 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 analytically. 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:

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

where δ_{ij} is the Kronecker delta function. When we compute all element 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

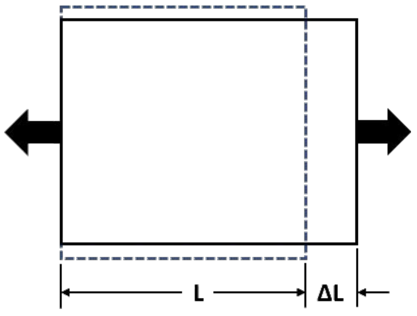
In geotechnical applications one of the main interests is the structural integrity of civil structures and the underlying soil. The steps that have to be taken to construct the system of equations are very similar to the heat equation, but the differential equations are different, leading to some extra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

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

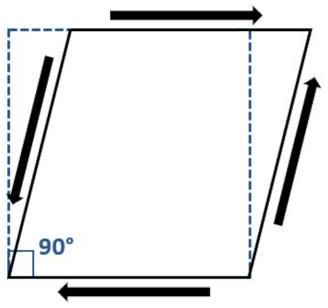
$$\epsilon_y = \frac{\partial v}{\partial y} \tag{2.18}$$

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

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 (engineering) shear strain. The normal strain is a measure of how much the material 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's modulus (indicating the stiffness of a material), ν is the Poisson's ration (indicating the orthogonal expansion of a material upon compression in one direction). σ_x, σ_y are the normal stresses and τ_{xy} is the shear stress. These stresses indicate a (directional) force per unit area, which has to be balanced in all points in order to find an equilibrium 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 [35].

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.

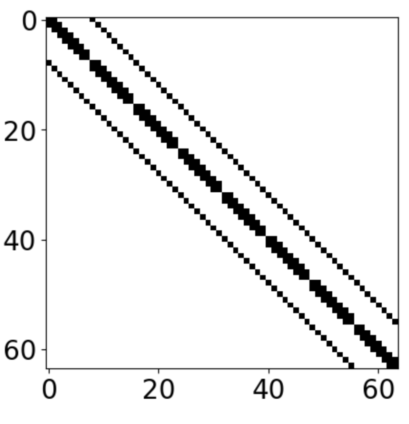
These topics are interesting on their own, yet mostly out of scope for this thesis. These factors do, however, impact the resulting system of equations to be solved. Therefore, they are relevant, especially for choosing the preconditioner. Furthermore, as the linear systems are part of a non-linear iteration,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

3 Iterative solvers

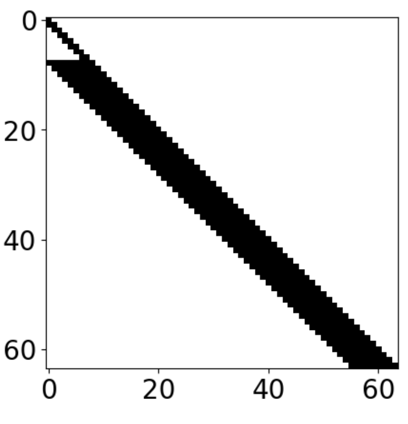
This chapter is mostly based on information provided in the books of Vuik [43] and Saad [36].

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 efficient approach may be to decompose A into a lower and upper triangular matrix ($A = LU$) and solve the systems $LUx = b$. This system can be solved in two steps by a forward and backward substitution. However, this decomposition is usually quite computationally expensive.

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). This means there are many more non-zero entries in the decomposition, which in turn requires a lot of memory and is undesirable. There are methods that somewhat address this issue, for example by reordering [11] or using special direct solvers like Intel MKL PARDISO [21] 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 [36], making them a popular choice for solving large sparse systems. Depending on the type of problem, iterative methods may be faster than direct solvers. To measure the accuracy



(a) Structure of A



(b) Full LU decomposition, structure of L

Figure 6: Sparsity pattern of LU decomposition

of an iterative method, one is interested in the error

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

We desire that this error goes 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 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:

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

Depending on our choice of M and N, we 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, and 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, the smaller the spectral radius, the 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

$$M = \operatorname{diag}(A) \tag{3.7}$$

$$N = \operatorname{diag}(A) - A. \tag{3.8}$$

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

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

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

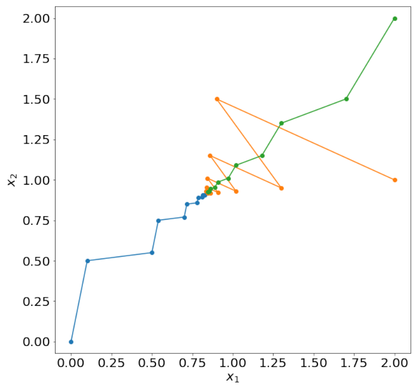


Figure 7: Jacobi iteration with different initial guesses for the system defined by (3.9)

with the norm of the error. We can see that for this problem the error is reduced about a hundred times after 10 iterations.

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

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 approximated solution will approximate the exact solution. The name of these methods stems from the search space that is used, which is the Krylov subspace, defined as:

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

with r the initial residual. This way 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 (*SPD*), i.e. $A = A^T$ and $x^TAx > 0 \quad \forall x \in \mathbb{R}^N \setminus \{0\}$. Then $\|x\|_A := \sqrt{x^TAx}$ is a well-defined norm. The conjugate gradient method iteratively searches along a direction and minimizes the error (in the A-norm) along that direction. This is similar to gradient descent method in that by searching along a given 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. $p_i^TAp_j = 0 \quad i \neq j$, 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.

The reason this is called 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 reused). 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 $j = 0, 1, ...$ (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[36]

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. 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

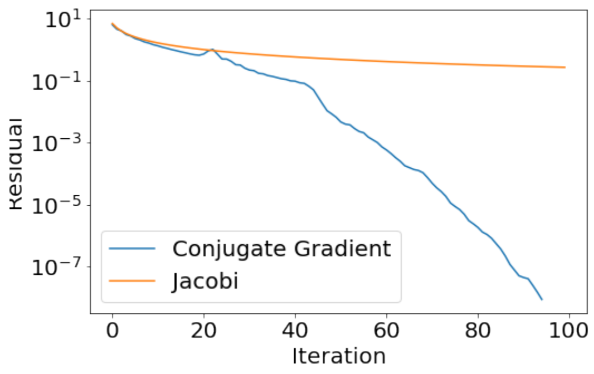


Figure 8: Convergence rate of Conjugate Gradient versus Jacobi method for a 2D Poisson problem on a 30×30 grid.

that the convergence rate of 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), the condition number is given by:

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

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, i.e.

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

This super-linear convergence is a nice property that Krylov methods exhibit.

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 "pre-conditioning". 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 preconditioners, we find the Incomplete LU decomposition, which will be further discussed in Section 5.3.

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:

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

In this form it can easily be 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 meaning 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, i.e. $ICCG(k)$ 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, as illustrated in Figure 9. The figure 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 to 39 (compared to 96 without preconditioning). Allowing one fill-in element per row, hardly increasing the computational load, still reduces the number of iterations further to just 25.

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 non-symmetric matrices, but there

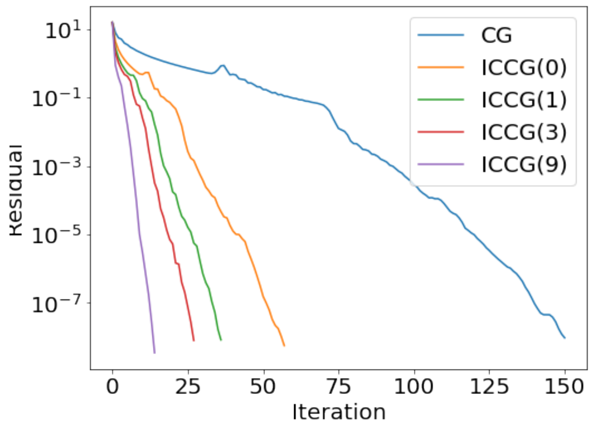


Figure 9: Convergence plot for $ICCG(k)$ with different amounts of fill-in for a 2D Poisson problem on a 30×30 grid.

are several methods that try to mimic CG and keep some of its nice properties. We will look at three methods, Bi-CGSTAB, GMRES and IDR(s) which each 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 Generalized minimal residual method (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 property is also thrown away, although there are ways to still keep (part of) this charasteristic [32].

3.4.2 Biconjugate gradient stabilized method (Bi-CGSTAB)

Bi-CGSTAB takes a different approach than GMRES. It has short recurrence, meaning that only the last two search directions need to be stored. 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. Yet, for a large class of problems, convergence behaviour looks similar to the CG method [43]. However, numerical errors are more significant and may lead to instabilities. Also one should check for ("near") breakdown of the iterations.

3.4.3 Induced Dimension Reduction (IDR(s))

IDR(s) [38] is based on a different idea compared to the previous methods. According to Gijzen and Sonneveld [40]: "IDR(s) is based on the induced dimension reduction theorem, that provides a way to construct subsequent residuals that lie in a sequence of shrinking subspaces." . This method has a parameter (s) that determines the dimension of its subspace. Larger s requires more memory and more work per iteration, but also generally leads to convergence in fewer iterations. Unlike GMRES, this method requires a constant amount of memory (depending on s). Similar to BiCGSTAB there is no proof of optimality, but usually IDR(s) converges in fewer iterations than BiCGSTAB when $s > 1$ [38].

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 about it. 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 [27]. 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).

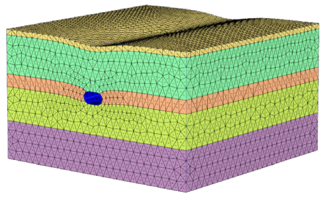


Figure 10: Tunnel through layered soil[27]

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, we want a preconditioner to approximate the inverse of A. A better approximation leads to fewer iterations, but this comes 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 parallelizable 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 [17].

5.2 Block Jacobi

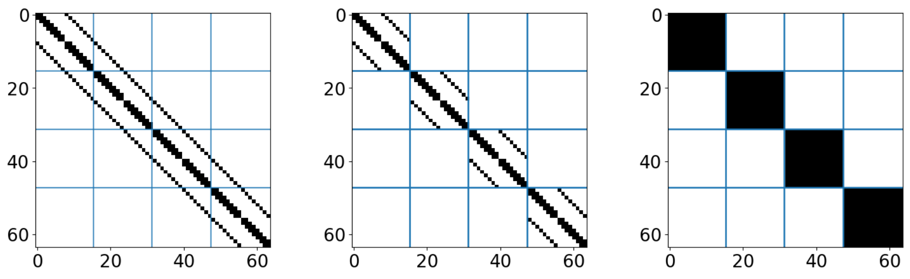
The block Jacobi preconditioner is a generalization of the Jacobi preconditioner. Instead of a preconditioner that takes the inverse of only the diagonal elements, we take the inverse of sub-matrices on the diagonal of A, as illustrated in Figure 11. Formally, we can denote a matrix A by its blocks:

$$A = \begin{bmatrix} D_1 & B_{12} & B_{13} & \\ B_{21} & D_2 & B_{23} & \\ B_{31} & B_{32} & \ddots & \ddots \\ & & \ddots & D_n \end{bmatrix}$$

and by only considering the diagonal blocks, we construct the block Jacobi preconditioner M as [19]:

$$M = \begin{bmatrix} D_1^{-1} & 0 & \cdots & 0 \\ 0 & D_2^{-1} & & 0 \\ \vdots & & \ddots & \vdots \\ 0 & 0 & \cdots & D_n^{-1} \end{bmatrix}.$$

The size of these blocks can be varied to create different variants. Also note that generally taking the inverse of these blocks leads to fill-in, as can be seen in Figure 11(c). Thus, to reduce memory usage and evaluation time it is important that the block size is not too large. As the preconditioner is an explicit matrix, it can be evaluated efficiently in parallel on a GPU.



(a) Structure of A

(b) Structure of diagonal blocks of A

(c) Structure of block Jacobi preconditioner

Figure 11: Sparsity pattern of block Jacobi precondition using 4 blocks.

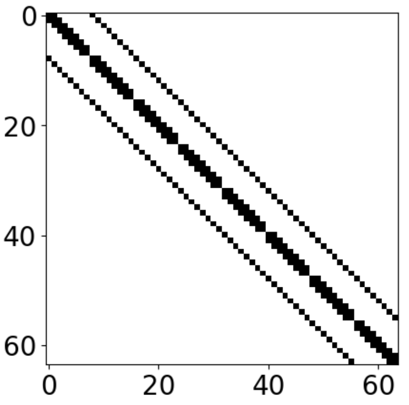
5.3 Incomplete LU (ILU)

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 [37]. Generally this decomposition is expensive to compute and requires a lot of memory due to fill-in, which is illustrated in Figure 12. 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.

5.3.1 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 illustrated in Figure 13. 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


(a) Structure of A

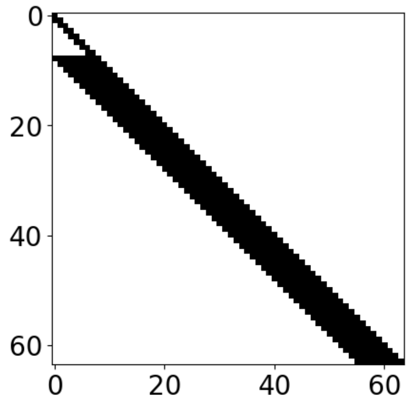
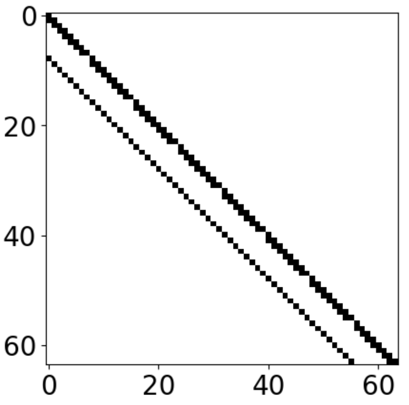

(b) Full LU decomposition, structure of L

Figure 12: Sparsity pattern of LU decomposition

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 also 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.


(a) $ILU(0)$ decomposition

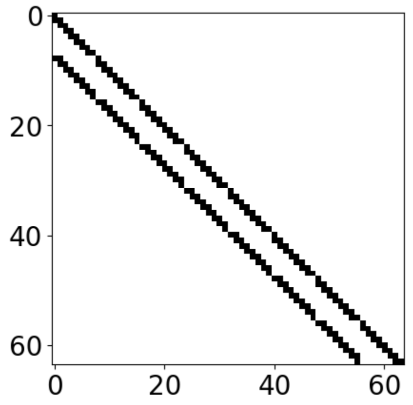
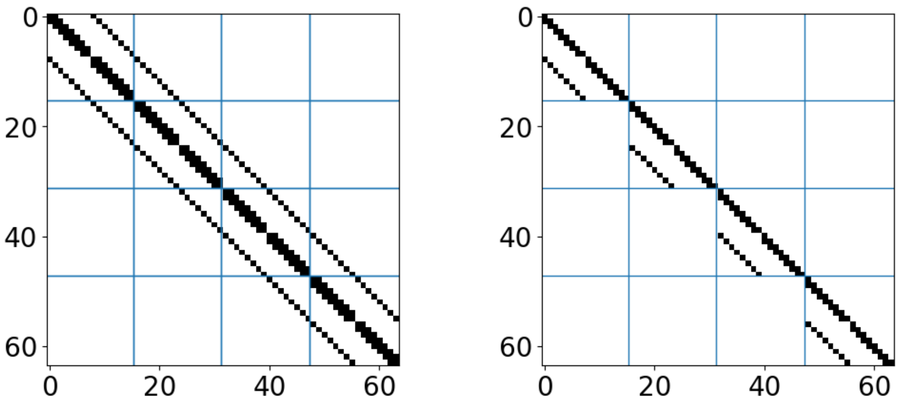

(b) ILU(1) decomposition

Figure 13: Sparsity pattern of L in a ILU decomposition of A

5.3.2 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 [44]. This is illustrated in Figure 14. 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. Generally, however, when the number of blocks increases, so does the number of iterations required [26, 44].



(a) Structure of A

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

Figure 14: Sparsity of Block ILU with a splitting of A into 4 domains

5.3.3 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 row in parallel by assigning a GPU thread to every non-zero in the row [3]. 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.

5.3.4 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 [10, 9]. 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 [10] also proposes a method for iteratively approximating the ILU decomposition in parallel for a given sparsity pattern. Later, Anzt (2018) [4] 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.

ILU Construction We will first look at the parallel construction of an ILU decomposition for a given sparsity pattern [10]. 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

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

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. The *"standard initial guess"*, takes the lower and upper triangular parts of $\hat{A}$. Whereas the *"modified initial guess"* scales 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. If the matrix is not too ill conditioned, the number of iterations needed in the Krylov method is close to the number that the sequential 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

```
Initial guess: $L, U$
For sweep $= 0, 1, ...$ (until convergence)
    Parallel For $(i, j) \in S$
        If $i > j$
            $l_{ij} = \frac{1}{u_{jj}} \left( a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj} \right)$
        Else
            $u_{ij} = \frac{1}{l_{ii}} \left( a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj} \right)$
        End
    End
End
```

Algorithm 2: Parallel ILU Factorization [10]

solving algorithms. One method is to use Jacobi iteration (see Section 3.1) on the triangular matrices [10]. 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 [4]. 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 points that are a non-zero of the residual matrix $R = LU - A$ [4]. 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

```
Initial guess: $L, U$
For sweep $= 0, 1, \ldots$ (until convergence)
    Add $m_L$ and $m_U$ candidate locations to $S_L$ and $S_U$ respectively
    Do one ParILU sweeps
    Remove $m_L$ and $m_U$ elements of smallest magnitude from $S_L$ and $S_U$
    Do one ParILU sweep
End
```

Algorithm 3: ParILUT algorithm [4]

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 [10] for the static sparsity pattern and ParICT [4] 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.

5.4 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 [8, 14]. 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 [5]. Furthermore, the choice of sparsity pattern influences the performance of the preconditioner. Nonetheless, in some cases it can outperform ILU based methods just because of its fast iterations [5, 14].

5.4.1 Construction

The way that the SPAI preconditioner is constructed is by considering every column of M independently via the relation [29]:

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

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.

5.4.2 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. [29]. 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.

5.4.3 Incomplete Sparse Approximate Inverse (ISAI)

Anzt 2018 [5] 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

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

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

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

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 [5] it also leads to better convergence.

5.4.4 Factored Sparse Approximate Inverse (FSAI)

The idea of FSAI [20, 29] is to find an approximate inverse of a factorization of A. so we can use the conjugate gradient method. Say that we have a factorization $A \approx LU$. Then we can use the SPAI or ISAI method to find an 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. No sequential forward and backward substitution is needed. Even though the quality of the preconditioner is then decreased compared to standard ILU, it is now fully parallelized, so that there could still be a significant speed-up [5]. Furthermore, if A is SPD, we can construct a SPD preconditioner $M = CC^T$ so that we can use the CG method. This is an advantage over plain approximate inverse preconditioners.

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 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 also that the deflation method can easily be combined with preconditioning.

6.1 How deflation works

In this thesis we assume A to be SPD for simplicity, following the proof by Jönsthövel 2012 [23]. For general matrices there are more difficult proofs available[15, 45]. 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 needs 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 $P^T x$. Therefore 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. Note that 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 [23]. As we saw earlier, the convergence depends on the 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 used as deflation vectors, the effective condition number then becomes

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

which usually provides 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 $j = 0, 1, ...$ (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 [23]

6.2 Choice of deflation space

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

6.2.1 Using exact eigenvectors

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 one can determine the eigenvalues exactly, there is no need for solving the system numerically. However, it can be used to formally prove statements about the convergence rate from a theoretical standpoint. But more importantly, the other methods can be seen 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 [24]. 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 eigenvectors 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 some computation power by approximating the eigenvectors based on information found by the GMRES algorithm. This is still expensive in practice [39], but a part of the calculation has to be done anyway in order to solve the linear system, therefore it is cheaper than approximating eigenvectors beforehand. After the linear solve is completed, some of the information that is stored during 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 the restarted GMRES method [28, 7, 31].

6.2.4 Subdomain deflation

When the domain is split into subdomains, we can use the indicator function on each domain, i.e.:

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

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. It can be made even more effective if the subdomains are chosen based on physical characteristics of the problem, by for example aligning the subdomains with different materials of the problem [25, 27].

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 [39].

6.2.6 Rigid body modes

Rigid body deflation 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 pretending 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. Rotation along arbitrary angles is a non-linear operation. But for small angles we can use the following (linearized) deflation vectors (which depend on the positions of the nodes) [27]:

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

For mechanical problems, these deflation vectors generally lead to better deflation than levelset deflation. As they only depend on the position of the nodes, they can be computed beforehand. These vectors are an extension of levelset deflation, as levelset deflation corresponds to the translation terms in this method. The strength of rigid body deflation 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 [27].

6.2.7 First-order deflation

In rigid body deflation we used linearized rotational modes to construct the deflation vectors. We find that these rotations consist of two linear modes added together. So we may take these two linear parts independently from each other. For instance, instead of having a deflation vector

$$\text{rotation about x-axis} \quad [0,-z,y,1,0,0]$$

as we have in rigid body deflation, we take three independent deflation vectors:

$$\begin{aligned} &[0, z, 0, 0, 0, 0] \\ &[0, 0, y, 0, 0, 0] \\ \text{if there are rotational DOFs}&[0, 0, 0, 1, 0, 0]. \end{aligned}$$

The space spanned by these vectors contains the rigid body deflation vectors, therefore we expect the performance of this deflation space to be at least as good as rigid body deflation. From a physical point of view, these vectors correspond to shear modes in two directions. Therefore the set of deflation vectors now becomes:

$$\text{translation} \begin{cases} [1, 0, 0] \\ [0, 1, 0] \\ [0, 0, 1] \end{cases}$$

$$\text{shear \& rotation} \begin{cases} [y, 0, 0] \\ [z, 0, 0] \\ [0, x, 0] \\ [0, z, 0] \\ [0, 0, x] \\ [0, 0, y] \end{cases}$$

These vectors are very straightforward to construct, and they are all only dependent on one positional coordinate. There are three extra vectors that seem natural to add to this set of deflation vectors:

$$\text{compression} \begin{cases} [x, 0, 0] \\ [0, y, 0] \\ [0, 0, z] \end{cases}$$

From a physical viewpoint, these three vectors correspond to compression/extension along three directions. Together it forms a space that we shall call the *"first-order delflation"* space, as all vectors depend on the coordinates of the nodes linearly. We do not have any quadratic forms (x^2) or cross-terms (xy).
In total we have 12 deflation vectors for every volume that we deflate. If the model contains plate elements, their nodes will have three extra explicit degrees of freedom for rotation. In such a case, we add three extra deflation vectors corresponding to these DOFs:

$$\text{rotational DOFs} \begin{cases} [0, 0, 0, 1, 0, 0] \\ [0, 0, 0, 0, 1, 0] \\ [0, 0, 0, 0, 0, 1] \end{cases}$$

6.2.8 Sparsity of deflation vectors

The deflation methods that are based on approximations of eigenvectors (Sections 6.2.1-6.2.3) yield dense deflation vectors. As a result, the amount of memory required and the amount of FLOPS required for deflation depends linearly on the number of deflation vectors. In particular, when we use k deflation vectors on a problem of size n, we need $\mathcal{O}(nk)$ memory and FLOPS for the deflation vectors themselves. Additionally we need $\mathcal{O}(k^2)$ to store and evaluate the coarse-grid inverse E^{-1}. Since $n \gg k$ the term $\mathcal{O}(nk)$ is dominant.
For the sparse methods (Sections 6.2.4-6.2.7) we have a different picture. If we use non-overlapping subdomains the vectors are very sparse and since the vectors do not overlap, the storage required for the deflation vectors does not depend on the number of deflation vectors. We only need $\mathcal{O}(n)$ memory and FLOPS for the deflation vectors. In the case of levelset deflation we only need 1 float/DOF. Whereas for first-order deflation we need 4 floats/DOF. On top of this we still need $\mathcal{O}(k^2)$ to store and evaluate the coarse-grid inverse E^{-1}. For a total of $\mathcal{O}(n + k^2)$ memory and FLOPS to store and evaluate the deflation projection.
If, on the other hand, we have overlapping domains, the storage and computational costs will increase. As long as the amount of overlap is sufficiently small, these costs will still be roughly $\mathcal{O}(n + k^2)$.

7 Preliminary experimentation

7.1 Test problems

We show the performance of different preconditioners on a test problem, 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 15. This is noticeable in the convergence plots by the three bends in the lines.

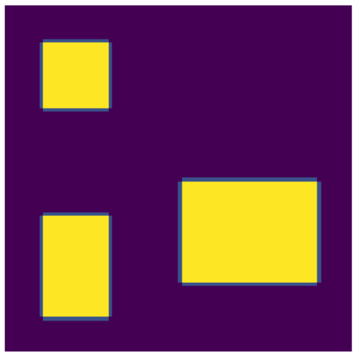


Figure 15: Regions of high conductivity for matrix 2, yellow corresponds to the area with 100× the conductivity of the purple area

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

ParILU has been tested with two different sparsity patterns: A (corresponding to ILU(0)) and A^2 (corresponding to ILU(1)). The test problem is a 30×30 heat equation with high contrast (test problem 2), with symmetric diagonal scaling (as suggested in Section 5.3.4). The convergence using GMRES is shown in Figure 16 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 best performances are achieved with a relatively inaccurate approximation, due to faster iterations, which needs to be researched. The reference ILU decomposition is implemented using the `ilupp` Python package, which is turn is based on `ilu++` [30].

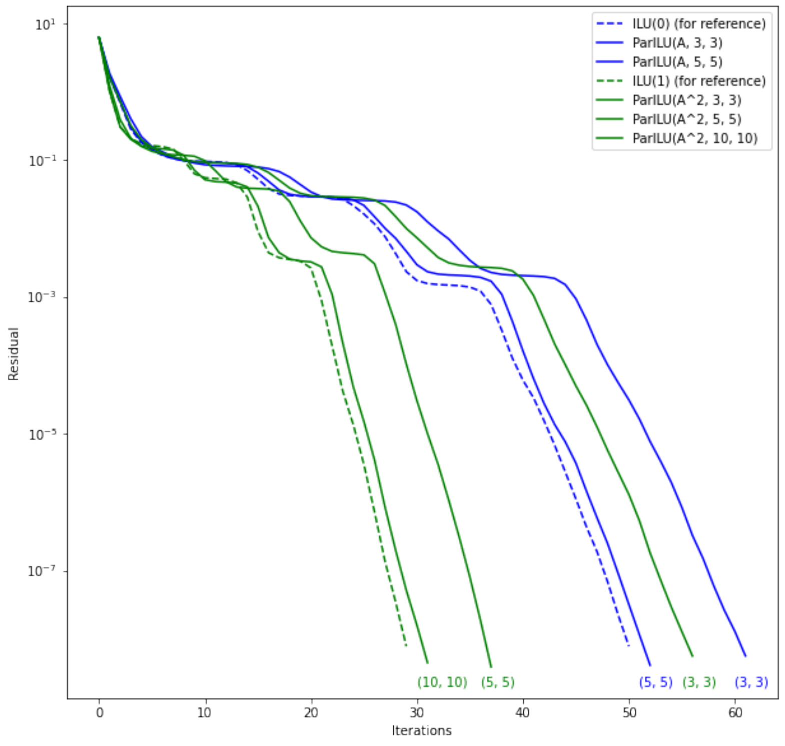


Figure 16: Convergence test for ParILU. The dashed lines represent conventional ILU. The blue lines correspond to ILU(0) and its parallel approximation. The green lines correspond to ILU(1) and its corresponding approximation. The paramaters in ParILU correspond to the sparsity pattern, construction sweeps, Jacobi iterations respectively.

7.3 ParILUT

Using the same test problem we can test ParILUT. Our implementation 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 shown in Figure 17 and a a summary is given in Table 2. We can see that the `scipy.sparse.linalg.spilu` has the most non-zeros while having the worst convergence, which is quite unexpected. 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), due to the (few) extra non-zeros or better sparsity pattern, as explained by Anzt [4].

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 18. 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. As this is the only symmetric SPAI preconditioner in this test, we could thus have been using CG.

7.5 ISAI

Figure 19 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 comparably 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.

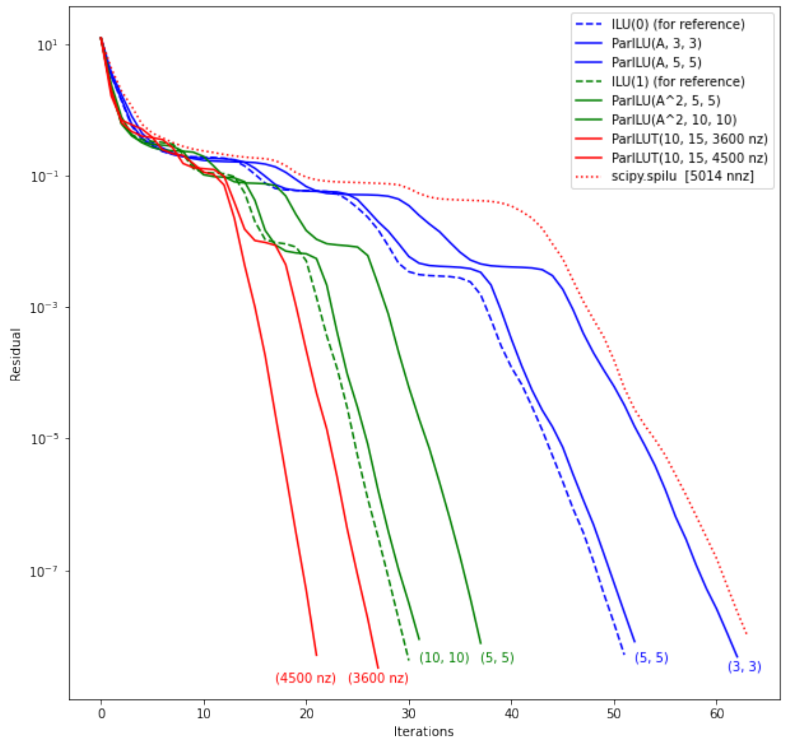


Figure 17: Convergence test for ParILUT. The dashed lines represent conventional ILU. The blue lines correspond to ILU(0) and its parallel approximation. The green lines correspond to ILU(1) and its corresponding approximation. For the parameters in ParILU stand for ParILU(sparsity pattern, construction sweeps, Jacobi iterations) and for ParILUT(construction sweeps, Jacobi iterations, allowed number of non-zeros)

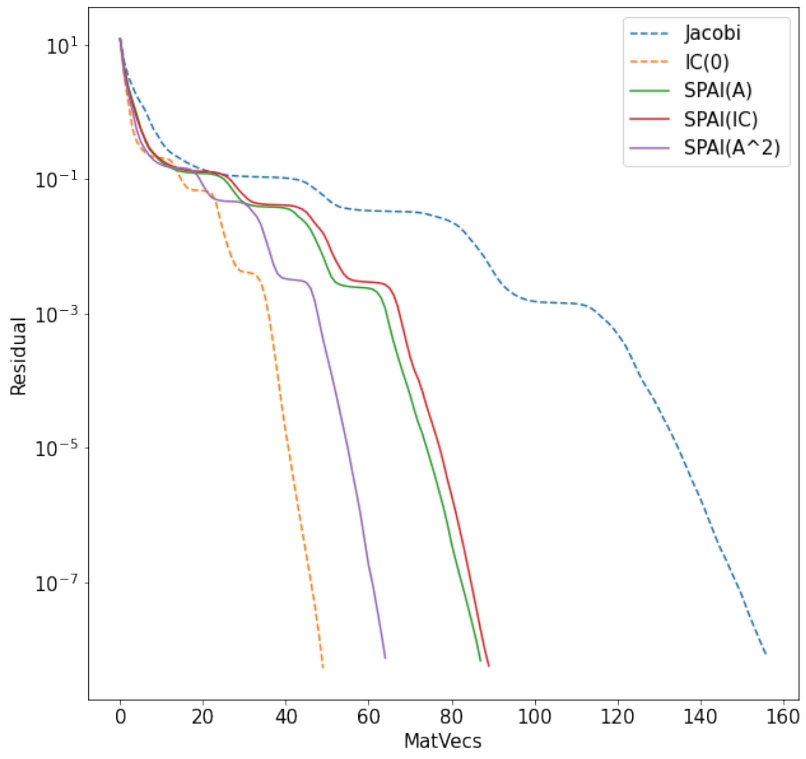


Figure 18: Convergence test for SPAI

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.

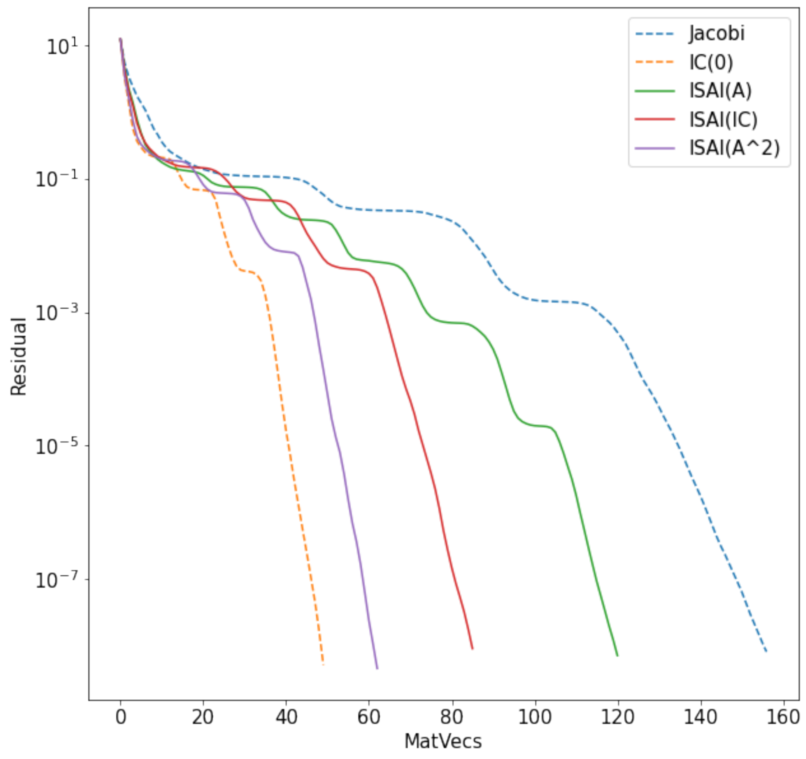


Figure 19: Convergence test for ISAI

7.6 Approximate eigenvector deflation

It has been noted above that for problem 2 there is a high contrast in conductivity, leading 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 20 shows the 4 eigenvectors corresponding to

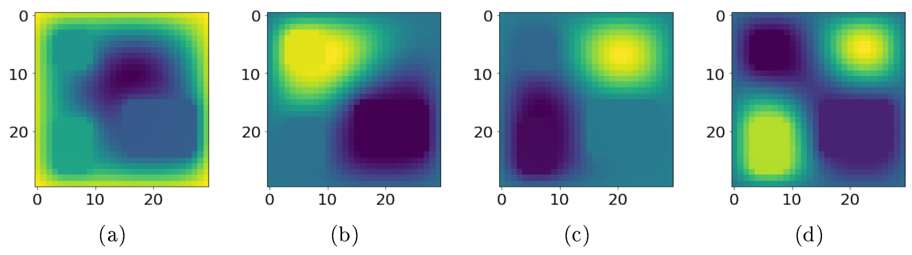
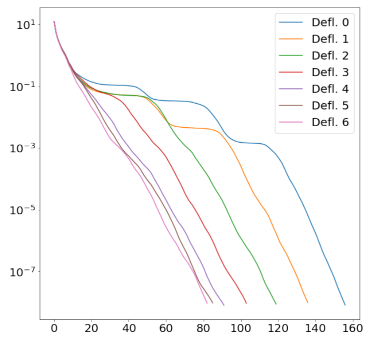


Figure 20: Deflation vectors corresponding to the lowest 4 eigenvectors.

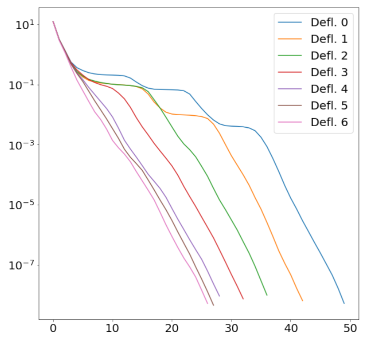
the 4 smallest eigenvalues. The high conductivity regions are easily visible in the eigenvectors with some extra smoothing going on in the neighborhood. Figure 21 shows the convergence plot with varying number of eigenvalues deflated, for both the Jacobi preconditioner and the $IC(0)$ preconditioner. We can see from the convergence that as the number of deflation vectors increases, the bumps in the convergence disappear 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%.

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, as illustrated in Figure 22. 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 23. Interestingly, the convergence for 4 levelset vectors is almost equal to the convergence of the 4 lowest eigenvalues, meaning that with 4 very cheap deflation vectors, we can improve convergence significantly.

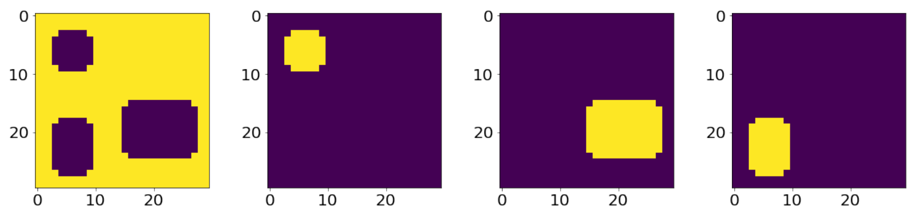


(a) Using Jacobi preconditioning



(b) Using IC(0) preconditioning

Figure 21: Convergence for eigenvector deflation for different amount of deflated eigenvectors.



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

Figure 22: Deflation vectors corresponding to the levelsets.

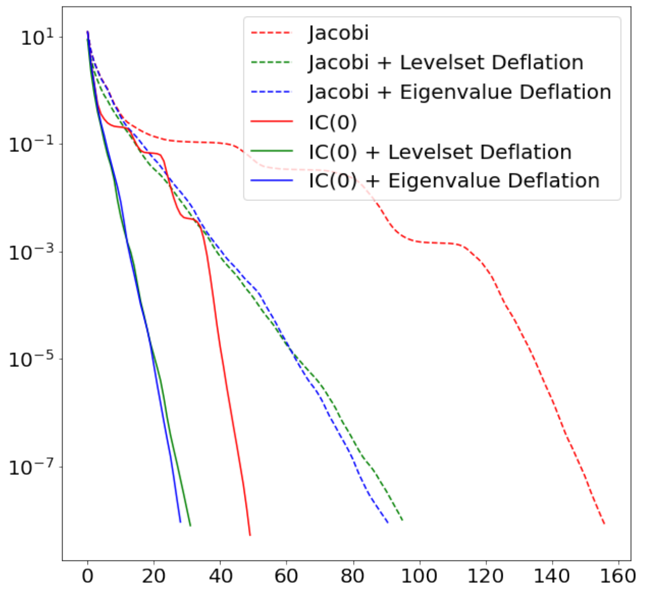


Figure 23: Convergence using levelset deflation versus eigenvalue deflation (both with 4 vectors).

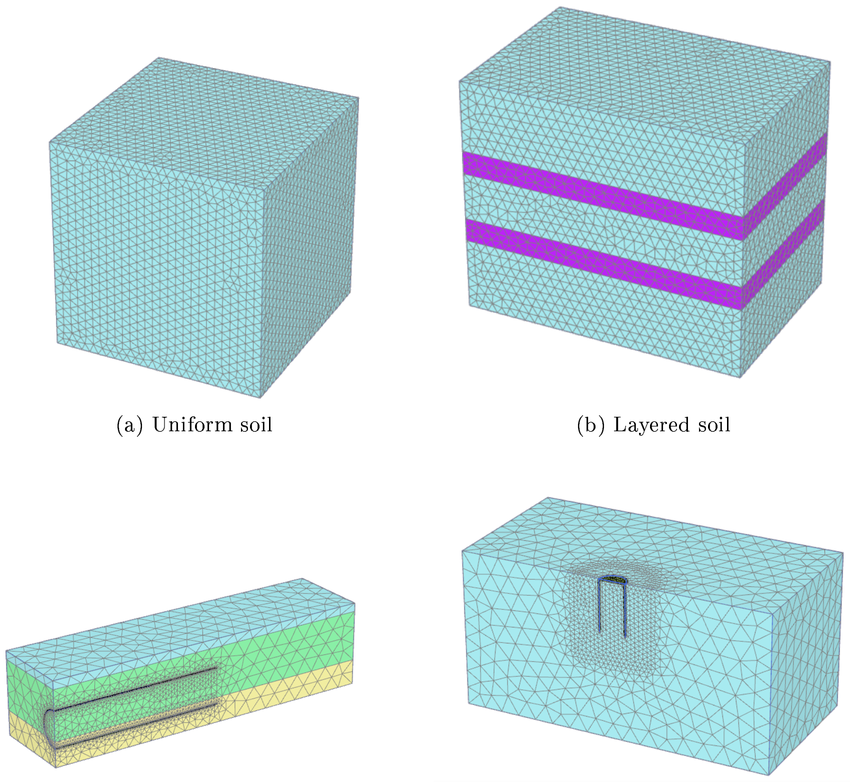
8 Test problems

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

1. A uniform soil cube under gravity loading, 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 [27], model 1. The stiffness of the soil layers differs by several orders of magnitude.
3. Tunnel through a layered soil, as taken from a PLAXIS 3D tutorial [1]. Conceptually similar to Lingen 2014 [27], model 2. The soil layers have an order of magnitude different stiffness, whereas 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 [2]. 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 a highly ill-conditioned problem. In fact, the tutorial suggests to use PARDISO as it solves the problem faster than PICOS.

8.1 Hardware

The hardware specifications of the GPU and CPU are given in Table 4.


(a) Uniform soil (b) Layered soil

(c) Tunnel (d) Suction pile

Figure 24: Test problems

	Desktop
CPU	Intel(R) Core(TM) i7-10700
RAM	16 GB DDR4
CPU cores	8 physical / 16 threads
GPU	NVIDIA GeForce GTX 1660 SUPER
GPU RAM	6 GB DDR6
GPU cores	1408
GPU link	PCIe 3.0 x16

Table 4: Hardware specification

9 Implementation and analysis

In this chapter we will discuss some of the implementation details and intermediate results of the project. The chapter provides insight about the process of developing the code, as well as an understanding of the GPU computing through various examples, especially relevant for anyone aiming to replicate this project or build further on it.

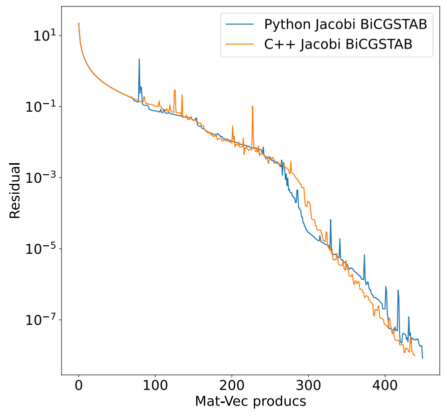
In total three implementations of the same algorithms are made. First in Python, for its ease of use, allowing us to quickly prototype different methods. Using the NumPy library[18] yields reasonable performance, although it is not suitable for actual performance comparisons. Next, a C++ implementation is developed, which improves the performance of the algorithms on the CPU. Lastly, an implementation in CUDA is made for the GPU, to achieve the maximal performance. In all cases we do not implement the linear algebra ourselves. Instead, we rely on libraries (NumPy, SciPy, Eigen, cuSPARSE, cuBLAS) to provide matrix operations for us. Some libraries (Scipy specifically) are also used for their Krylov methods, yet unless explicitly stated, the Krylov methods are custom implementations.

Furthermore, for the GPU algorithms, we run the Krylov method, deflation operations and preconditioner application on the the GPU, while all the setup is performed on the CPU, including reading/writing of data, construction of the preconditioner, generation of deflation vectors and operators. These costs will be reported separately, as they are only needed once per matrix.

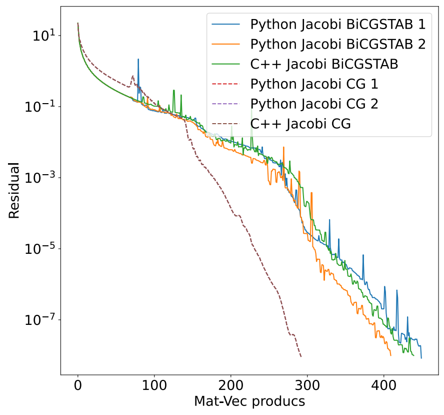
Lastly, usually the CPU is referred to as the *"host"*, whereas the GPU is referred to as the *"device"*.

9.1 Algorithm verification

In order to properly compare different implementations (e.g. host vs device) it is important to check whether the results are the same. That is, the number of required iterations is (almost) equal, and the solution vector is equal up to numerical precision. To our initial surprise, the BiCGSTAB implementation in Python and C++ yields different results, even when the matrix and preconditioner are exactly the same, even for cases where double precision could exactly represent the numbers (numbers such as 4 and 1/4 have exact floating point representations). Using a 2D Poisson problem on a 100×100 grid with Jacobi preconditioning, the convergence is plotted in Figure 25(a). The difference in the number of iterations is surprising given that exactly the same preconditioner is used and the algorithm follows the exact same steps. As it turns out, this difference is caused by the instability of BiCGSTAB. In Figure 25(b) the results are plotted for the CG algorithm as well, where we see the convergence follows the same line and the results are equal up to numerical error. To prove that a small rounding error can lead to different convergence, we also tested the same Python implementation with two slightly different starting vectors: for test 1: $x_0 = [0, 0, 0, ..., 0]$ and in test 2: $x_0 = [10^{-14}, 0, 0, ..., 0]$. Even for such a tiny difference in starting vector, the convergence of BiCGSTAB is significantly different, whereas for CG this difference is not visible. All in all, these differences can be attributed to rounding errors and should be taken into account in any final comparisons, for example by averaging the runtime with several slightly different starting vectors.



(a)



(b)

Figure 25: Convergence plot of Python and C++ implementation. Note that the three CG versions coincide and run on the exact same line.

9.2 Memory transfer speeds

As the host and device have physically separate RAM, any data that is to be used by the device first has to be transferred from the host. Usually, the FEM matrix is constructed on the host, which means that the entire matrix has to be transferred. If this transfer takes too long, any performance benefits of using a GPU are immediately mitigated. Therefore the memory transfer speed from the host to the device is measured by transferring 1 Gigabyte of data, see Table 5. This tests only the bandwidth and not the latency. Note that the units (GB/s) denote giga*bytes* and not gigabits.

Pagelocked memory: Note that behind the scenes, copying memory from the host to the device will first copy it from the host to pagelocked memory and then transfer it to the device. Pagelocked memory is some “special memory” that still resides in the host RAM, the only difference from normal memory being that it cannot be swapped out by the operating system. This two step copy cannot be avoided, but it can be done explicitly if desired, i.e. one could construct the matrix in page-locked memory, but the advantage is minimal and construction of the matrix is out of scope for this research. For completeness, memory copy from host to host, and device to device is also included. This two-step copy also happens when copying back from the device to the host. The

CUDA runtime automatically handles this for us and hides this complication, but for bandwidth tests it is relevant.

From	To	Bandwidth (GB/s)
Host	Host	(1st time) 3.41
Host	Host	(2nd time) 13.20
Host	Device	9.96
Device	Host	9.58
Device	Device	142.44
Host	Pagelocked	13.55
Pagelocked	Device	12.60

Table 5: Memory bandwidth when copying 1 gigabyte of data.

The results from Table 5 need some explanations. First of all, it turns out that copying data from host to host depends on whether it is the first time this data is copied or the second time (by a factor 3.8). This indicates that possibly some kind of caching is happening. However, this does still not explain the difference as the transfer of 1 gigabyte is orders of magnitude larger than the cache size. It should also be noted that swapping any memory to disk was disabled for the entire OS and therefore this also cannot be the cause of the difference.

Next, we can see that transfers from pagelocked memory to the device are slightly faster than from standard memory. However, the difference is minor and in real world use cases the matrix is constructed out of our control. Thus directly copying to the device and letting CUDA handle the pagelocked memory behind the scenes is the fastest and easiest approach. The theoretical maximum bandwidth is 16 GB/s (for PCIe 3.0 x16).

Furthermore, we see that the bandwidth from host to device is close to the bandwidth from host to host. Therefore we should not need to worry too much about the cost of transferring the data. All timing results presented in this chapter shall include the time to transfer the data to the device.

Lastly, we see that device to device copies are about 10 times faster than host to host copies. This indicates that accessing memory on the device is much faster and can provide a performance gain.

9.3 Multi-threaded performance

The Eigen library allows us to easily perform our iterative methods in parallel on the host. We tested the runtime of the algorithm using different number of cores on the desktop. The tests are performed on a 2D Poisson problem with grid size 1024×1024 using a CG algorithm and a Jacobi preconditioner. The total runtime (i.e. wall-clock time) is shown in Figure 26. We see that, initially, more cores means better performance, although when we add more than 4 threads the total runtime remains relatively constant up to 8 threads, indicating that probably at this point the calculations are no longer bound by computational power, but rather by memory bandwidth. When using more than 8 threads, the performance drops again significantly, which can be explained by cache pollution [13]. Coming down to the fact that two hyper-threads run on a single physical core, competing for the same cache space. This in turn leads to more cache misses and thus longer execution times.

From this result we see that using 4 threads provides the best performance and for this reason for future tests we report the runtime using 4 threads.

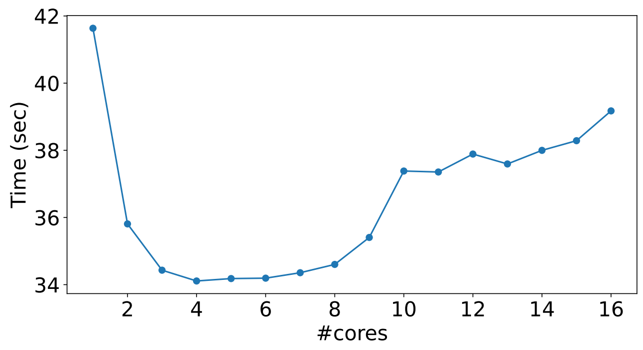


Figure 26: Wall-clock time for solving a 2D Poisson problem using different numbers of threads. Note that the vertical axis does not start at 0.

9.4 GPU performance

The first step for comparing host vs device performance is using the same algorithms on both host and device and comparing the performance and accuracy. We use the same initial conditions, preconditioner and algorithm and all computations are performed in double precision.

9.4.1 Implementation details for CG

For all linear algebra operations we use libraries provided by Nvidia, specifically cuSPARSE and cuBLAS [34] as these libraries are optimized and easy to use. The Krylov method itself, however, is a custom implementation. Another important point is that all sparse matrices are stored in Compressed Row Storage format (CSR) matrix. In the initial implementation, the preconditioner is built on the host (as constructing the preconditioner on the host is easier) and then transferred to the device where it is then used.

In the initial implementation, the results of all vector dot products are returned to the host to compute the α and β variables in the CG algorithm. This is easier to implement but leads to many points where the device is waiting on the communication, and hence suboptimal. In the final implementation, these dot products are stored in the device and α and β are computed on the device, thus saving a lot of communication overhead. Additionally, in the final implementation, the residual has to be sent back to the host in every iteration, since the cuSPARSE function cannot be called from the device and, hence, the host has to determine and communicate to the device when to stop iterations. To improve the performance of this communication, this result is sent back asynchronously, so that while the host determines whether to quit or not, the device continues executing operations. The overhead of the communication of the residual contributes less than 1% of the total runtime for larger matrices, so no further optimization to this aspect seems necessary.

9.4.2 GPU performance

We verified that both implementations require the same number of iterations and give the same result up to numerical error. In all reported times the communication time between the host and device (over PCIe) is included, but the initialization time (loading of libraries) is not. Furthermore, before the tests are executed, we run a small warm-up problem, consisting of solving a (small) 50×50 2D Poisson problem, to ensure that all libraries are fully initialized and neither the host nor the device are in a power saving mode. We compare the performance of different implementations. We also compare the implementation to a Scipy version (`scipy.sparse.linalg.cg`) to verify our host implementation is performant. We use the CG algorithm with Jacobi preconditioning on 2D Poisson problems of different grid sizes, with $x_0 = 0$ as initial condition. The results are shown in Figure 27. We can see that for small matrices, the cus-

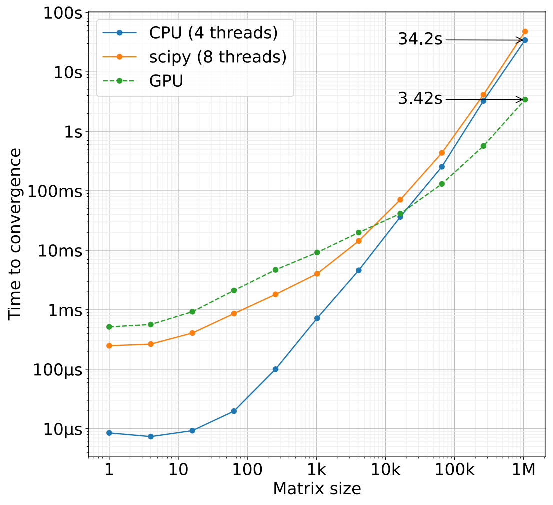


Figure 27: Time to convergence using CG on a 2D Poisson problem of different sizes. Note the log-log scale.

tom host implementation is the fastest. For matrices larger than about 20 000 unknowns, the device shows better performance. Solving even larger problems on the device is about 10 times faster than using the same algorithm on the host. For small systems the communication overhead with the device is large compared to the performance gain. For sufficiently large systems this overhead is negligible.

9.5 Geotechnical system from PLAXIS 3D

Next, we use the same algorithm to solve a finite element system from a geotechnical problem, using a custom version of PLAXIS 3D to export the matrices. Then, we use this exported matrix with our own solvers to compare convergence. We start with the first problem described in Section 8. It is a simple cube of uniform soil under gravity loading. A mesh is generated using the "*very fine*" setting in PLAXIS 3D, shown in Figure 28(a) and the sparsity structure of the matrix is shown in Figure 28(b). Furthermore, the matrix is SPD and some of its properties are listed in Table 6.

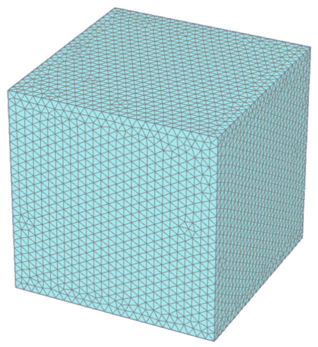
Property	Value
Width, height	232 071
Number of non-zeros	18 401 645
Bandwidth	23941
nnz/row (average)	79.3
nnz/row (max)	308
$\kappa_2(A)$ (condition number)	$\approx 1.1 \cdot 10^4$

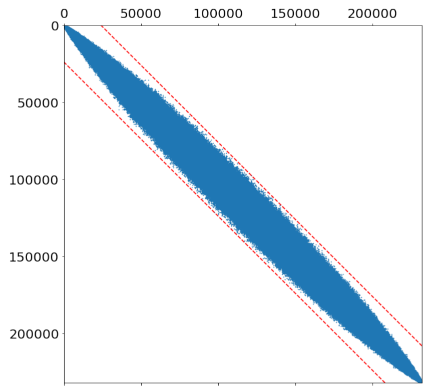
Table 6: Properties of the finite element matrix of the uniform cube soil.

9.5.1 Solving the system

The next step is to solve the system using CG and compare the performance of different preconditioners on a real-world problem. For this matrix, we will look at the following preconditioners:

1. Jacobi preconditioner (Section 5.1)
2. IC(0) preconditioner
3. The FSAI preconditioner (see Section 5.4.4). Specifically, we will use the ISAI preconditioner applied to the IC(0) decomposition.
4. For the general Krylov solvers (BiCGSTAB, IDR(s)) we will also look into the ISAI preconditioner.

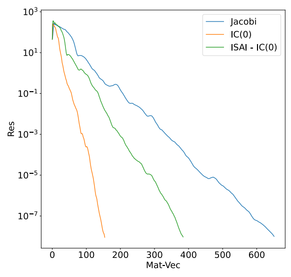

(a) Finely meshed uniform cube



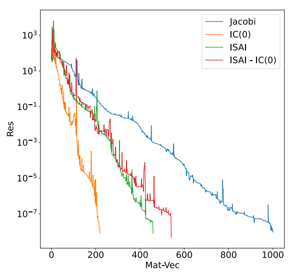
(b) Sparsity structure of corresponding matrix

Figure 28: Cube of uniform soil

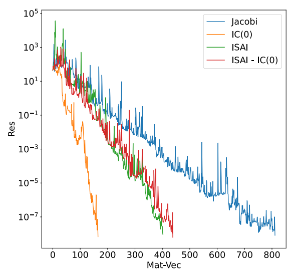
The convergence for the three different methods is shown in Figure 29(a - c). From this, we see that the ISAI preconditioner has a convergence in between the Jacobi and IC(0) preconditioners for all solvers. The big advantage of direct preconditioners over IC(0) is that it can be evaluated in parallel making it suitable for GPU computations. On the downside, it is much more expensive to construct than the Jacobi preconditioner. On the horizontal axis the number of matrix-vector products is shown instead of the number of iterations, to more easily compare the computational cost of different solvers.



(a) using CG



(b) using BiCGSTAB



(c) using IDR(4)

Figure 29: Convergence for the uniform soil cube system using different Krylov Solvers and preconditioners

9.5.2 ParIC

In the previous section we have seen that the IC(0) preconditioner has the best convergence rate among the methods tried so far. We will try to achieve a similar convergence rate using the Parallel IC preconditioner as described in Section 5.3.4. We will try a different number of inner iterations and we will not be using the parallel construction method for the preconditioner, but, instead we will use a normal sequential IC algorithm. This way we have one parameter less we need to vary.

Figure 30 shows the convergence of the CG method using ParIC preconditioner with varying number of inner iterations. From this figure we see that as we increase the number of inner iterations, the convergence improves. This is expected as the ParIC preconditioner should approach the sequential IC precondition when the number of inner iterations is sufficiently large. The improvement becomes less significant every time, until about 10 inner iterations. When we further increase the inner iterations, there is no significant improvement in convergence. For 1 inner iteration, the convergence is similar to Jacobi preconditioning. But as the ParIC preconditioner is much more expensive to evaluate, the time to reach convergence is longer compared to using Jacobi preconditioner.

From a theoretical point we can say the same for any number of iterations. We know that one inner iteration of the ParIC preconditioner is about as expensive as a matrix-vector product with the original matrix. Therefore we can estimate the amount of work required to reach convergence for each of the methods. This estimation is shown in Table 8. From this table we can quickly see that the Jacobi preconditioner requires the least amount of work. From this we conclude that the ParIC preconditioner is not beneficial for this problem.

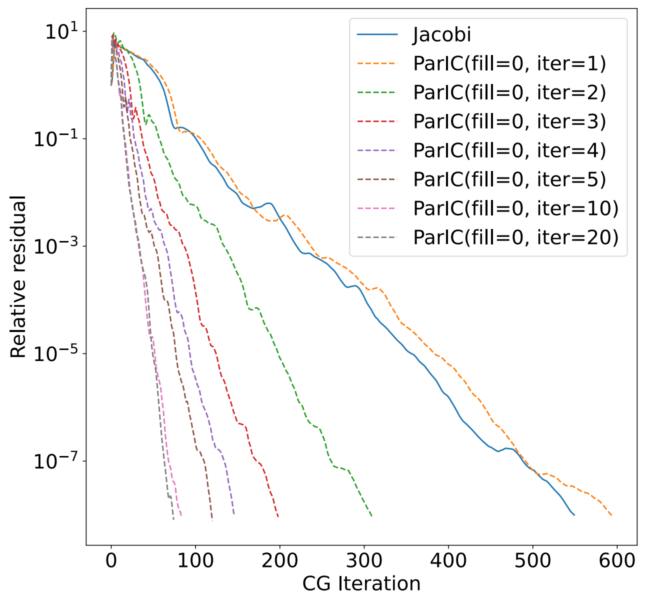


Figure 30: Convergence with ParIC preconditioner with zero fill-in applied to the uniform soil system on a fine grid for a varying number of inner iterations.

Preconditioner	Inner iters	CG iters	Total work (mat-vec equiv)
Jacobi	/	550	550
ParIC(1)	1	595	1190
ParIC(2)	2	310	930
ParIC(3)	3	199	796
ParIC(4)	4	147	735
ParIC(5)	5	121	726
ParIC(10)	10	84	924
ParIC(20)	20	75	1575

Table 8: Estimated work for varying number of inner iteration in ParIC. Note that for every CG iteration, there is 1 mat-vec for CG itself, plus n mat-vec equivalent for the inner iterations of the preconditioner.

Fill-in We are also interested in the results if we do have some amount of fill-in in the ParIC preconditioner. Figure 31 shows the convergence of CG with ParIC preconditioning for a varying number of inner iterations and a varying amount of fill-in. We see that for small numbers of inner iterations, it does not matter how much fill-in is used. The effect of the fill-in only becomes noticeable for high numbers of inner iterations. From this we conclude that the zero-fill ParIC preconditioner is preferable over the the ParIC preconditioners with fill-in.

We can conclude that the ParIC preconditioner requires more work than the Jacobi preconditioner and therefore we shall not investigate this preconditioner any further.

9.5.3 Changing sparsity pattern of ISAI

In Section 7.5 we have seen that we can improve the convergence rate of the ISAI preconditioner by choosing a different sparsity pattern for constructing the preconditioner. In particular, in that section we used the sparsity pattern of A^2. Effectively, this allows the preconditioner to have some amount of fill-in. We would like to try to get similar improvements for this matrix. However, if we use the same strategy, we run into an issue: the amount of fill-in in the sparsity pattern of A^2 is too large to construct the preconditioner. We can however, use a little trick to reduce this problem. By looking at the histogram of the magnitude of the elements in the matrix, relative to the main diagonal (see Figure 32). We see that most matrix elements are very small compared to the diagonal elements. So we can expect that a small fraction of the matrix elements has a large contribution to the result, whereas the majority only has a small influence. We can use this to our advantage. The idea is somewhat similar

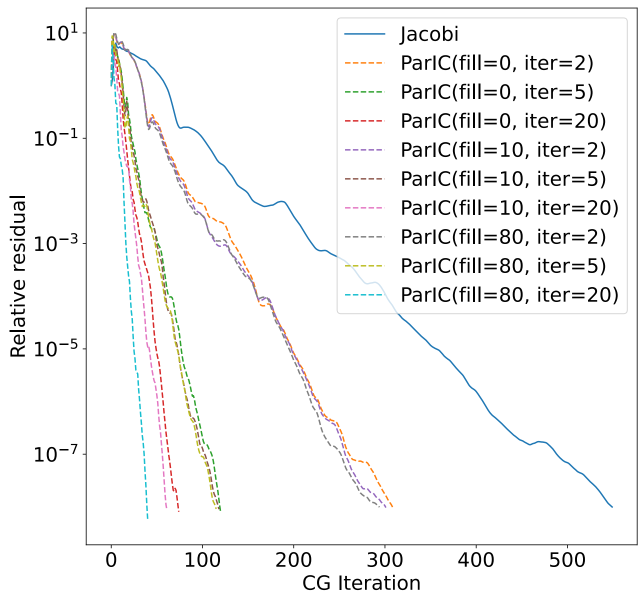


Figure 31: Convergence with ParIC preconditioner with varying fill-in applied to the uniform soil system on a fine grid for a varying number of inner iterations. Note that on average the original matrix has almost 80 elements per row. Thus a zero fill-in incomplete Cholesky decomposition has about 40 elements per row, while a decomposition with fill-in of 80 has on average about 120 non-zeros per row.

to thresholded ILU, stripping away the smallest elements, reducing the memory usage and amount of operations required per application of the preconditioner. At the same time we hope that by stripping away these small elements we do not negatively affect the performance of the preconditioner too much. Specifically,

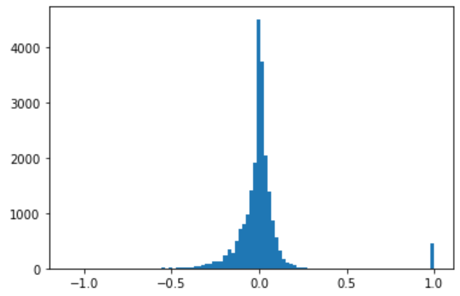


Figure 32: Histogram of relative magnitude of matrix elements, relative to the main diagonal.

we keep the m largest elements per row of A. Let us denote this reduced matrix as B_m. The sparsity pattern of B_m then corresponds to the most significant elements of matrix A. The next step would be to have some fill-in, by taking the sparsity pattern of B_m^2. Then we apply the ISAI precondition on the original matrix, with the sparsity pattern of B_m^2. This is denoted by ISAI(A, B_m^2). The parameter m still has to be determined and we can try with a few small values, such as 5, 11 and 21. The results are shown in Figure 33. We can see that the number of iterations is reduced when we keep more of the sparsity pattern, as this yields a more accurate preconditioner. At the same time, it requires more memory. Furthermore, the setup cost as well as the evaluation cost of the preconditioner will increase. So, in total, there is some optimum, which may be problem dependent and difficult to predict. However, for this particular problem, in terms of runtime, the ISAI(A, B_{11}^2) proved the fastest. More details are shown in Table 9. From this table we also see that the sparsity pattern B_{11}^2 contains just slightly fewer non-zero elements as the original matrix A, yet the convergence is significantly improved. We also note that the convergence using ISAI(A, B_5^2) is very similar to the convergence using ISAI(A, A), yet the memory requirements and evaluation time is much lower.

9.6 Deflation

The next step is to include deflation into the solver. The effect of deflation is best illustrated on a layered soil system (see Figure 24). The finite element matrix corresponding to this problem will have a higher condition number as the stiffness ratio between the two soil types increases, thereby the number of iteration required to reach convergence will increase, as is illustrated in Figure 34.

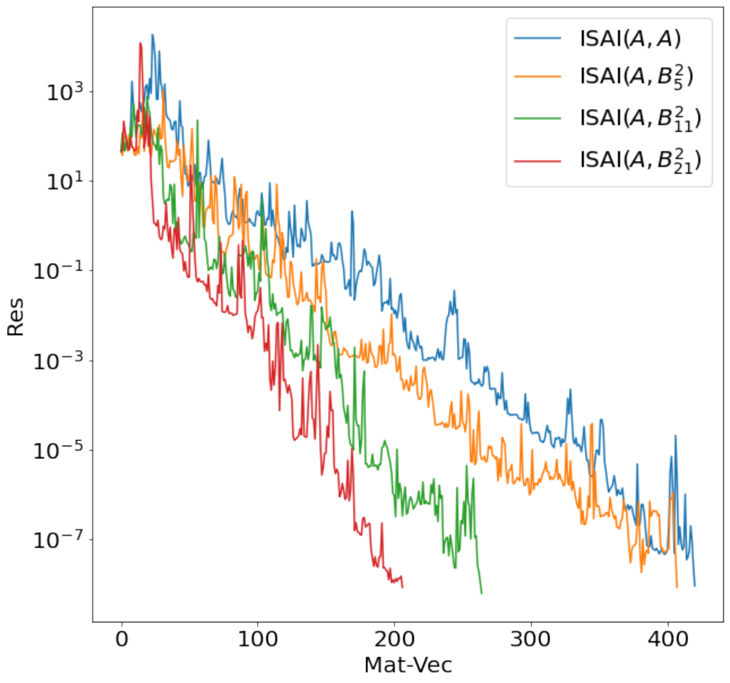


Figure 33: Convergence of ISAI preconditioner with some varied sparsity patterns, using IDR(4).

Preconditioner	non-zeros	# iterations	Iteration time
ISAI(A, A)	18 401 645	421	1.68 s
ISAI(A, B_5^2)	3 931 611	408	1.11 s
ISAI(A, B_{11}^2)	16 508 226	265	1.01 s
ISAI(A, B_{21}^2)	36 709 047	207	1.14 s

Table 9: Some properties of the ISAI preconditioners with different sparsity patterns. As well as some properties of the convergence when using IDR(4) with this preconditioner. The iteration time is measured when running on the GPU and it excludes any setup costs. Iteration is stopped at an absolute tolerance of 10^{-8}.

From this figure we can see that the convergence degrades as the stiffness ratio increases, especially in the initial phase. The number of iterations required to reach convergence is listed in Table 11. We see that for a relative tolerance of 10^{-4} (which is commonly used by the PICOS solver in PLAXIS 3D) the number of iteration required to reach convergence is increased almost by an order of magnitude when going from no contrast in stiffness to a ratio of 10^6.

For this figure and table we use the exact same mesh for each run, which was generated using the "very fine" setting in PLAXIS 3D. This resulted in a system with 224 040 degrees of freedom. Note that we did not limit ourselves to stiffness ratios we would find in nature, but the values are chosen to illustrate the ill-conditioning of the matrix and the effectiveness of deflation. For the solving phase we used Conjugate Gradient with Jacobi preconditioning. A similar effect can be observed when using an IC or ISAI preconditioner, except of course an overall decrease in number of iterations.

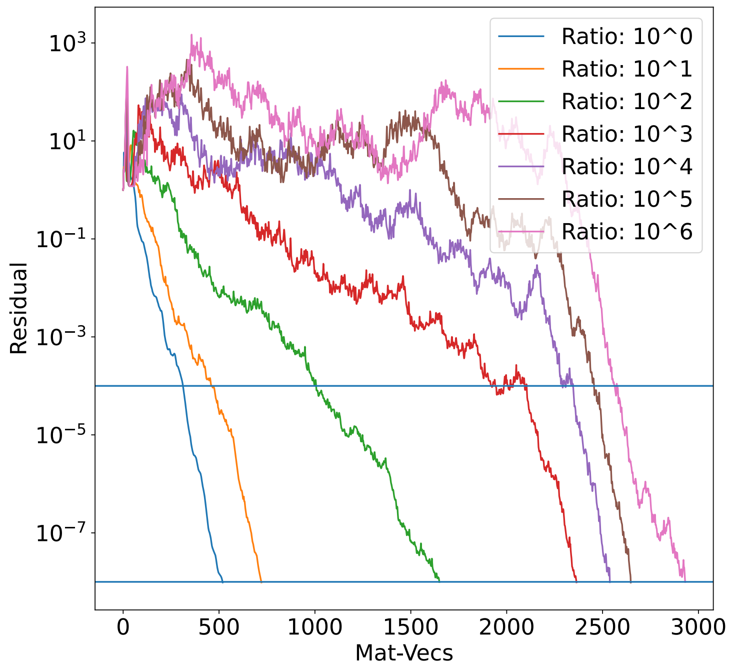


Figure 34: Convergence plot for layered soil system with varying stiffness ratio between the soil types. The horizontal lines indicate two possible relative tolerances at 10^{-4} and 10^{-8}.

9.6.1 Solving

In the previous section we saw how an increased stiffness ratio leads to worse convergence, whereas this section we shall use deflation to reduce this effect.

Stiffness ratio	Iteration required to convergence	
	rel. tol $= 10^{-4}$	rel. tol $= 10^{-8}$
10^0	312	519
10^1	462	721
10^2	1006	1649
10^3	1918	2364
10^4	2292	2538
10^5	2456	2647
10^6	2564	2931

Table 11: Number of iterations required to reach a given relative tolerance for varying stiffness ratios.

We investigate four deflation methods described in Chapter 6:

1. Eigenvector deflation: for small problems, very expensive, only for reference
2. Levelset deflation
3. Rigid body deflation
4. First order deflation

For eigenvector deflation, we compute a given number of eigenvectors and use these as deflation vectors. For the physics based deflation methods, the soil layers will guide the deflation vectors. We achieve this physics based deflation via a few mappings provided by the PLAXIS software. First of all, we have a mapping of which elements are contained in each soil layer. We know which nodes are contained in each of these elements. Lastly, we know the degrees of freedom corresponding to each node. In total we can map which row in the matrix corresponds to every node and to each of the soil layers.

9.6.2 Interface nodes

The nodes which are on the boundary between two soil layers are included in the deflation vectors of both the layer above and below it. Thus, we end up with deflation vectors that slightly overlap on the boundary between two soil layers. It turns out that this is not optimal. By adding these nodes only to the stiffest of the two layers but not to the softer layer, we can get significantly better convergence. Unfortunately, assigning the nodes to the optimal layer requires explicit assigning of these nodes. Vermolen et al. propose a method to automatically set the values of the deflation vectors on the interface nodes based on material properties, such as permeability[41]. We use the Young's modulus to determine the ratio:

$$w_i = \frac{E_i}{E_1 + E_2} \qquad i = 1, 2$$

In principle we can have access to this information, yet in this project it is not used. Instead we determine the ratio based on the values in the matrix (as the matrix values are correlated to the stiffness). We take the sum of the matrix elements corresponding to the interaction with the nodes in the different soil layers.

$$c_i = \sum_{j \in V_i} a_{ij}.$$

c_i is the contribution of a soil layer. V_i represent the degrees of freedom inside a particular layer (excluding any shared nodes). a_{ij} is the value of the matrix element, representing the interaction between the interface node and the nodes inside a particular volume, as illustrated in Figure 35. Then we take the ratio

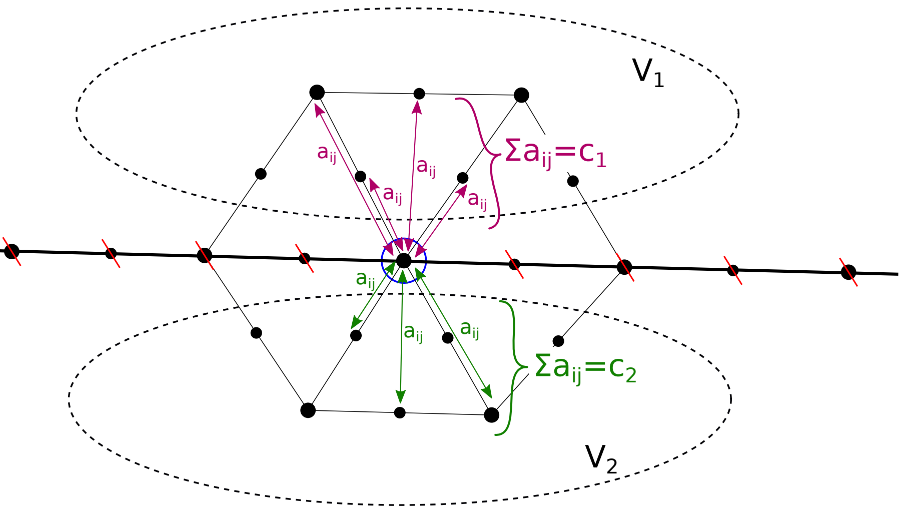


Figure 35: Illustration of method to determine weighting of a node on the interface between two volumes. The values in the matrix represent interactions with other nodes. We take the sum of all interactions with each of the volumes, ignoring the other interface nodes.

of these contributions as the weighting in the deflation vector.

$$w_i = \frac{c_i}{\sum_k c_k}.$$

To apply this weighting to the vector, we simply multiply the value of the deflation vector in a boundary node with its weight. In total we have 4 methods to determine the deflation vectors for levelset and rigid body deflation:

1. Overlapping deflation vectors
2. Only add node to stiffest volume (non-overlapping)
3. Weighted deflation based on material properties [41]
4. Weighted deflation based on matrix elements

The effect on convergence for the different methods is shown in Figure 36. It can be seen that the overlapping deflation vectors perform the worst, while the other three methods have very similar convergence. We continue with the weighted deflation based on matrix elements as it is easiest to extend to problems with general shapes, without needing stiffness information. This will be encountered in Section 9.7. Now we are ready to compare the different deflation methods

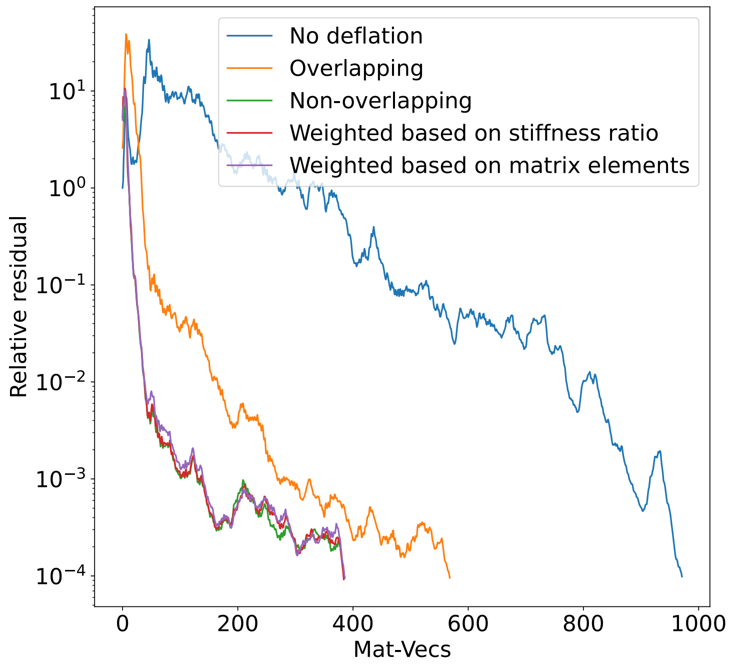


Figure 36: Comparison of the four ways to deal with overlapping deflation vectors on the boundaries between two soil types. Using levelset deflation.

for the layered soil problem, see Figure 37. We see that deflation has a big impact on convergence. Furthermore we see that initially, physics based deflation methods have faster convergence than eigenvector deflation. However, eigenvector deflation has better deflation if we had set the tolerance lower. But it is also the most expensive deflation method and not feasible in practice. Then, we also note that, as expected, rigid body deflation performs better than levelset deflation. However, first order deflation performs the best, as it includes vectors relating to the compression/extension of the soil layers, which is the dominant deformation mode in this case.

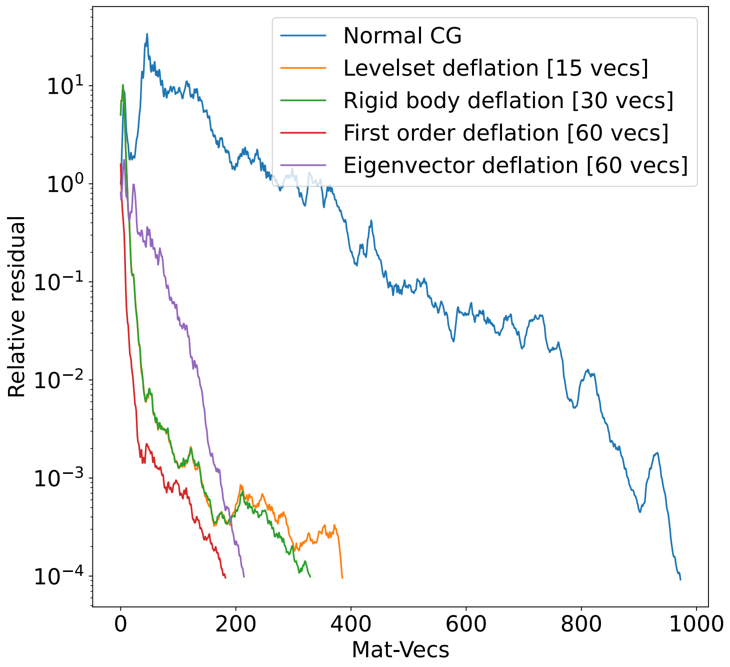


Figure 37: Comparison of eigenvector deflation, levelset deflation, rigid body deflation and linear deflation for the layered soil problem on the coarsest grid with a stiffness ratio of 1000.

9.7 Tunnel problem

We have shown that deflation works very well for a synthetic problem with a high contrast in stiffness. The next step would be to apply the method to a new problem that we could encounter in practice. The model for this tunnel is taken from the PLAXIS 3D tutorials [1] and is illustrated in Figure 24(c). The wall of the tunnel is very stiff and the surrounding soil is softer. The ratio of the stiffness between the soil and the concrete is $3.1 \cdot 10^3$. In addition to this, the model also contains plate elements, which have three extra degrees of freedom corresponding to its rotation. In Figure 38 we see the convergence for this problem. We can see that rigid body deflation reduces the amount of iterations by a factor two. The fastest method is again the first order deflation, reducing the amount of iterations by 83% compared to no deflation at all. More details are shown in Table 12. This experiment shows that first order deflation yields good results for real-world problems as well.

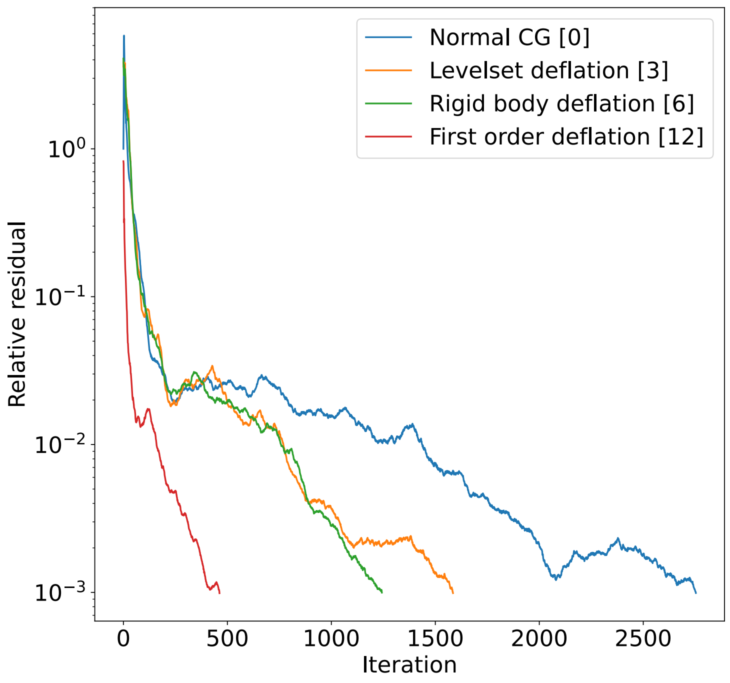


Figure 38: Comparison of deflation methods for the tunnel model, using the CG algorithm with Jacobi preconditioning. The number in the brackets represents the number of deflation vectors per domain. The algorithm uses a relative tolerance of 10^{-3}.

Deflation method	#iterations	CG duration
No deflation	2754	3.09 s
Levelset deflation	1586	1.98 s
Rigid body deflation	1244	1.67 s
First order deflation	463	0.73 s
PICOS	67	1.64 s

Table 12: Number of iteration and run times for CG with Jacobi preconditioning for solving the tunnel problem, comparing different deflation methods.

10 Results

10.1 Systems

We use four test problems to compare the solvers. These test problems are shown in Chapter 8. We use the “very fine” grid setting in PLAXIS 3D to generate the matrices. For the layered soil problem the stiffness ratio between the layers is set to 10^3. Furthermore, the suction pile is modeled as a rigid body in the soil. As a result, the entire suction pile has only 6 degrees of freedom interacting with all adjacent soil element, producing a system matrix having 6 rows with exceptionally many non-zero elements. Hence the large bandwidth and maximum nnz/row.

To solve a model, PLAXIS 3D runs many linear solves using the same matrix with different right-hand sides. However, we only use the last system exported by PLAXIS 3D. Furthermore, the two models from the tutorials (the tunnel and suction pile) have multiple phases. In the tunnel model we use the system from phase 1, while in the suction pile we use the system from phase 4.

Property	Uniform cube	Layered cube	Tunnel	Suction pile
Width, height	232 071	224 039	178 239	334 701
nnz	18 401 645	17 723 971	13 994 057	25 981 930
Bandwidth	23 941	22 020	23 118	285 009
nnz/row (avg)	79.3	79.1	78.5	77.6
nnz/row (max)	308	360	405	11 510
Condition number	$\approx 1.1 \cdot 10^4$	$\approx 8.5 \cdot 10^6$	$\approx 3.1 \cdot 10^8$	*not found*

Table 13: Properties of the finite element matrix of the tunnel model matrix.

10.2 Iteration time

In this Section we look at the run times and number of iterations for various solvers, preconditioners and deflation methods. It is important to note that the presented run time is the time from the start of the first iteration until the end of the last iteration. Any setup times are not included. We test the run time by running each solver three times and taking its average run time. The number of iterations is the same as the algorithms are deterministic. We run the algorithms until we reach a relative tolerance of 10^{-3} or we did not converge after 5000 matrix-vector products. The relative tolerance is the same as in PICOS (by default).

For some preconditioners we do not have a device implementation, therefore we run it on the host, reporting the number of iterations and denoting that we have no time measurement with “-”. For these preconditioners we can already conclude that the number of iterations is so high that they are not competitive, except for the IC(0) preconditioner, which can not efficiently run on the device as it is a highly sequential algorithm. Furthermore, because the ISAI-type preconditioners are not symmetric, they can not be combined with the CG method. Note that we only tried the FSAI and ParIC preconditioners on the tunnel system (Table 16).

The results for different systems are presented in Table 14 (uniform cube system), Table 15 (layered soil system), Table 16 (tunnel system) and Table 17 (suction pile). For each Krylov method, the strategy that yields the fastest run time is given in bold font.

10.2.1 Deflation method

One of the first observations we can make is that, for the first two systems and for any combination of preconditioner and Krylov method, the first order deflation method outperforms the other deflation methods consistently. Furthermore, we note that, for this type of systems, the FSAI and ParIC preconditioners (Table 16) perform worse than Jacobi preconditioning. This is likely due to the high condition number of the matrices. For ParIC a possible solution could be to replace the Jacobi inner iteration with Block Jacobi iteration [10]. For the suction pile problem, we deflated only a single volume, such that deflation has a much smaller effect. There we find that the difference between different deflation methods is quite small and sometimes rigid body deflation yields the fastest convergence, sometimes first order deflation.

10.2.2 Block Jacobi preconditioners

Another interesting observation is that in all cases a Block Jacobi preconditioner yields faster convergence than normal Jacobi iteration. It also appears that the blocksize has only a minor influence on the performance (only a few percent). This is a relevant observation, as it means we do not need to optimize this parameter of the Block Jacobi preconditioner. We can pick a value (say, block size = 20) and expect it to work well for different systems. Furthermore, it seems that when using Block Jacobi preconditioning, the CG method is the fastest, which is to be expected. Moreover, for the general solvers we find that IDR performs better than BiCGSTAB with the (Block) Jacobi preconditioner. However, in some cases, the IDR(4) method is faster, while in some cases the IDR(8) method is faster.

Preconditioner	Deflation	CG	BiCGSTAB	IDR(4)	IDR(8)
Jacobi	None	0.45s [221]	0.48s [231]	0.57s [259]	0.60s [239]
	Rigid body	0.48s [210]	0.53s [235]	0.49s [202]	0.49s [181]
	First order	0.33s [134]	0.34s [141]	0.41s [158]	0.36s [125]
Block Jacobi; size 10	None	0.43s [209]	0.47s [226]	0.56s [249]	0.57s [225]
	Rigid body	0.38s [164]	0.54s [233]	0.43s [174]	0.51s [185]
	First order	**0.32s** [130]	0.34s [139]	0.35s [132]	0.35s [119]
Block Jacobi; size 20	None	0.42s [206]	0.46s [222]	0.53s [235]	0.58s [230]
	Rigid body	0.37s [160]	0.52s [224]	0.45s [181]	0.50s [182]
	First order	0.33s [131]	0.34s [138]	0.33s [124]	0.35s [120]
Block Jacobi; size 30	None	0.42s [203]	0.50s [240]	0.53s [235]	0.54s [213]
	Rigid body	0.37s [160]	0.53s [227]	0.45s [178]	0.49s [176]
	First order	0.33s [131]	0.35s [138]	0.40s [149]	0.40s [137]
ISAI(A, A)	None		0.72s [190]	0.65s [165]	0.74s [176]
	Rigid body		0.55s [136]	0.57s [136]	0.61s [137]
	First order		0.47s [112]	0.50s [116]	0.45s [98]
ISAI(A, B_5^2)	None		0.28s [116]	0.32s [123]	0.31s [110]
	Rigid body		0.24s [92]	0.27s [97]	0.29s [94]
	First order		0.20s [70]	**0.19s** [65]	**0.20s** [62]
ISAI(A, B_{11}^2)	None		0.27s [74]	0.31s [80]	0.37s [92]
	Rigid body		0.26s [66]	0.27s [67]	0.25s [58]
	First order		**0.18s** [45]	0.22s [53]	0.25s [57]
ISAI(A, B_{21}^2)	None		0.32s [59]	0.36s [64]	0.33s [57]
	Rigid body		0.25s [45]	0.27s [47]	0.32s [53]
	First order		0.24s [38]	0.25s [40]	0.23s [36]
IC(0)	None	- [62]	- [69]	- [72]	- [81]
	Rigid body	- [61]	- [67]	- [70]	- [62]
	First order	- [38]	- [46]	- [42]	- [43]

Table 14: Time till convergence (on the GPU) and number of matrix vector products (in brackets) required for the **uniform soil cube model**. Tested with different preconditioners and deflation methods. Calculations run until the relative residual is less than 10^{-3}. The run time includes only iteration times, not any setup costs.

Preconditioner	Deflation	CG	BiCGSTAB	IDR(4)	IDR(8)
Jacobi	None	3.34s [1685]	5.36s [2712]	8.30s [3856]	6.32s [2616]
	Rigid body	0.35s [155]	0.36s [156]	0.48s [195]	0.50s [185]
	First order	0.19s [76]	0.19s [77]	0.15s [56]	0.17s [59]
Block Jacobi; size 10	None	3.21s [1593]	4.35s [2160]	6.20s [2806]	6.57s [2662]
	Rigid body	0.33s [142]	0.33s [142]	0.47s [185]	0.51s [183]
	First order	**0.13s** [50]	0.19s [76]	0.13s [49]	0.16s [55]
Block Jacobi; size 20	None	2.96s [1466]	4.43s [2192]	6.94s [3126]	6.88s [2777]
	Rigid body	0.34s [148]	0.32s [140]	0.44s [173]	0.47s [168]
	First order	**0.13s** [50]	0.20s [76]	0.13s [49]	0.15s [50]
Block Jacobi; size 30	None	2.97s [1469]	4.52s [2228]	8.81s [3946]	5.12s [2060]
	Rigid body	0.35s [150]	0.35s [153]	0.48s [190]	0.46s [164]
	First order	**0.13s** [51]	0.22s [85]	0.14s [51]	0.16s [55]
ISAI(A, A)	None		7.33s [1972]	7.11s [1827]	4.79s [1154]
	Rigid body		0.76s [190]	0.77s [184]	0.77s [174]
	First order		0.18s [41]	0.37s [79]	0.35s [72]
ISAI(A, B_5^2)	None		2.86s [1205]	3.67s [1430]	3.28s [1167]
	Rigid body		0.19s [72]	0.25s [89]	0.31s [96]
	First order		**0.10s** [34]	0.15s [48]	**0.09s** [28]
ISAI(A, B_{11}^2)	None		3.08s [857]	3.03s [804]	2.59s [644]
	Rigid body		0.22s [58]	0.30s [75]	0.35s [82]
	First order		0.12s [28]	**0.11s** [26]	0.13s [28]
ISAI(A, B_{21}^2)	None		3.47s [645]	3.44s [619]	3.06s [527]
	Rigid body		0.32s [58]	0.45s [77]	0.40s [66]
	First order		0.13s [20]	0.14s [21]	0.13s [19]
IC(0)	None	- [393]	- [496]	- [451]	- [424]
	Rigid body	- [83]	- [65]	- [78]	- [91]
	First order	- [21]	- [23]	- [27]	- [29]

Table 15: Time till convergence (on the GPU) and number of matrix vector products (in brackets) required for the **layered soil model**. Tested with different preconditioners and deflation methods. Calculations run until the relative residual is less than 10^{-3}. The run time includes only iteration times, not any setup costs.

Preconditioner	Deflation	CG	BiCGSTAB	IDR(4)	IDR(8)
Jacobi	None	4.26s [2733]	3.18s [2020]	- [>5000]	9.55s [4939]
	Rigid body	2.41s [1293]	2.50s [1371]	4.03s [1960]	5.51s [2446]
	First order	0.91s [432]	1.23s [598]	1.11s [476]	1.24s [494]
Block Jacobi; size 10	None	3.33s [2074]	3.53s [2236]	5.65s [3119]	4.76s [2383]
	Rigid body	1.89s [934]	2.09s [1112]	3.00s [1422]	3.10s [1334]
	First order	0.69s [315]	**0.62s [286]**	0.88s [374]	0.83s [324]
Block Jacobi; size 20	None	3.27s [1989]	2.97s [1858]	5.57s [3066]	6.84s [3386]
	Rigid body	1.75s [906]	2.35s [1218]	3.10s [1463]	2.19s [940]
	First order	0.67s [307]	0.82s [384]	0.72s [303]	0.80s [312]
Block Jacobi; size 30	None	3.16s [1924]	3.14s [1950]	5.38s [2945]	5.40s [2644]
	Rigid body	1.73s [882]	2.05s [1050]	2.06s [961]	2.47s [1044]
	First order	**0.65s [297]**	0.80s [370]	0.72s [302]	0.76s [293]
ISAI(A, A)	None		breakdown	9.56s [3081]	6.87s [2044]
	Rigid body		breakdown	3.14s [921]	2.94s [798]
	First order		breakdown	1.47s [401]	1.69s [433]
ISAI(A, B_5^2)	None		3.84s [2058]	2.83s [1392]	3.13s [1376]
	Rigid body		1.64s [764]	2.04s [869]	1.74s [673]
	First order		0.85s [352]	0.66s [256]	0.81s [289]
ISAI(A, B_{11}^2)	None		2.99s [1104]	2.75s [971]	2.36s [767]
	Rigid body		1.32s [441]	1.33s [422]	1.40s [415]
	First order		0.65s [204]	0.67s [197]	**0.64s [178]**
ISAI(A, B_{21}^2)	None		2.88s [754]	3.83s [947]	2.50s [584]
	Rigid body		1.20s [292]	1.34s [310]	1.26s [276]
	First order		0.74s [168]	**0.50s [110]**	0.65s [137]
FSAI (=ISAI(IC, A))	None	- [4590]	- [>5000]	- [>5000]	- [>5000]
	Rigid body	- [3745]	- [2040]	- [>5000]	- [>5000]
	First order	- [3693]	- [1762]	- [>5000]	- [4890]
IC(0)	None	- [470]	- [405]	- [344]	- [341]
	Rigid body	- [281]	- [312]	- [229]	- [206]
	First order	- [100]	- [192]	- [107]	- [99]
ParIC; 3 iter	None	- [3889]	- [3624]	- [>5000]	- [>5000]
	Rigid body	- [2690]	- [2376]	- [>5000]	- [>5000]
	First order	- [2647]	- [3107]	- [>5000]	- [>5000]

Table 16: Time till convergence (on the GPU) and number of matrix vector products (in brackets) required for the **tunnel model**. Tested with different preconditioners and deflation methods. Calculations run until the relative residual is less than 10^{-3}. The run time includes only iteration times, not any setup costs.

Preconditioner	Deflation	CG	BiCGSTAB	IDR(4)	IDR(8)
Jacobi	None	3.38s [1199]	8.18s [2912]	8.40s [2719]	6.54s [1909]
	Rigid body	3.09s [979]	**5.73s** [1803]	4.20s [1220]	4.53s [1202]
	First order	2.75s [801]	7.14s [2062]	5.00s [1344]	3.91s [963]
Block Jacobi; size 10	None	3.11s [1081]	11.25s [3851]	7.17s [2259]	5.68s [1622]
	Rigid body	2.83s [874]	6.96s [2104]	3.78s [1073]	**3.06s** [792]
	First order	2.52s [720]	6.86s [1905]	5.22s [1368]	3.30s [798]
Block Jacobi; size 20	None	3.11s [1069]	12.35s [4118]	7.21s [2250]	5.68s [1608]
	Rigid body	2.83s [866]	11.06s [3296]	4.23s [1190]	3.60s [924]
	First order	**2.50s** [707]	8.53s [2350]	4.56s [1192]	3.15s [754]
Block Jacobi; size 30	None	3.10s [1062]	10.58s [3521]	7.03s [2179]	5.06s [1418]
	Rigid body	2.80s [856]	8.09s [2407]	4.50s [1243]	3.52s [897]
	First order	2.51s [708]	6.65s [1831]	4.21s [1079]	3.42s [816]
ISAI(A, A)	None		- [$>$5000]	- [$>$5000]	- [$>$5000]
	Rigid body		- [$>$5000]	- [$>$5000]	- [$>$5000]
	First order		- [$>$5000]	- [$>$5000]	- [$>$5000]
ISAI(A, B_5^2)	None		- [$>$5000]	5.42s [1439]	4.27s [1046]
	Rigid body		8.94s [2332]	**3.76s** [913]	4.10s [923]
	First order		- [$>$5000]	4.17s [952]	3.39s [721]
ISAI(A, B_{11}^2)	None		- [$>$5000]	- [$>$5000]	- [$>$5000]
	Rigid body		- [$>$5000]	- [$>$5000]	- [$>$5000]
	First order		- [$>$5000]	- [$>$5000]	- [$>$5000]
ISAI(A, B_{21}^2)	None		- [$>$5000]	- [$>$5000]	- [$>$5000]
	Rigid body		- [$>$5000]	- [$>$5000]	- [$>$5000]
	First order		- [$>$5000]	- [$>$5000]	- [$>$5000]
IC(0)	None	- [385]	- [636]	- [450]	- [420]
	Rigid body	- [329]	- [598]	- [383]	- [269]
	First order	- [274]	- [575]	- [281]	- [265]

Table 17: Time till convergence (on the GPU) and number of matrix vector products (in brackets) required for the **suction pile model**. Tested with different preconditioners and deflation methods. Calculations run until the relative residual is less than 10^{-3}. The run time includes only iteration times, not any setup costs.

10.2.3 ISAI methods

For the tunnel system and layered soil system we find the best performing preconditioner to be an ISAI preconditioner. A minor downside is that it is not straightforward to predict exactly which sparsity pattern will yield the fastest convergence.

Furthermore, a much more critical downside is that for the suction pile model we get no convergence for many of the ISAI based methods (see Table 17). Only the combination ISAI(A, B_5^2) + IDR yields acceptable results. Given that the other ISAI-based preconditioners yield no convergence, we fear that this is a coincidence rather than a general rule and that for general systems this preconditioner is too unreliable to use in practice, especially as there is a good alternative.

10.3 Setup costs

The setup costs were not included in the results as this part of the implementation was not optimized. Mainly because part of the setup code is written in Python. There is a lot of optimization that can be done to make these parts faster, and some parts can even run on the GPU. However, considering these parts are not optimized, the setup times are quite acceptable for large systems (about 200 000 DOFs), i.e.

1. Building the preconditioner: 100 ms up to 5 seconds depending on the preconditioner (C++/Python implementation).
2. Reading the mesh from files and using this to set up deflation vectors and matrices: 1 to 10 seconds depending on the problem (Python implementation)
3. Communication between CPU and GPU: about 50 ms. There is a communication latency of less than 1 ms while the rest comes from data transfer at a rate of about 10 GB/s

In practice we find that the same system is solved many times with a different right-hand side. Therefore, the setup time can be amortized among all solves. We also find that the preconditioners that have fast setup times (such as Jacobi and Block Jacobi) actually perform quite well in the iteration phase, whereas the ISAI preconditioners, which have high setup times, perform less reliably. In setting up the deflation vectors, a major contribution comes from the weighting of nodes on the boundary between two layers, by implementing this part of the code in C++ rather than Python, we expect a large speedup.

10.4 Memory costs

When measuring the memory usage, we are interested in how much each of the components of the solver takes up. Such a breakdown for the tunnel problem is shown in Table 18. We see that a large portion of the total memory usage is consumed by GPU kernels and CUDA libraries, this part is constant and does not depend on the problem size. The next largest component is the system itself. Whereas the other components (CG solver, preconditioner and deflation operators) add an additional 50% of the system size, this part scales linearly with the size of the problem. Based on this, the part that scales with the problem size is about 1.5× the size of the system matrix.

We notice the block Jacobi preconditioner is sparser than we expected. We would expect a block Jacobi precondition with a block size of 20, to have 20 non-zeros per row. In practice we get an average of 4.5 nnz/row for this problem. Apparently the diagonal blocks themselves, as well as their inverses are quite sparse. The memory usage of the preconditioner will change if the equations are to be reordered.

Component	Memory usage
GPU kernels and libraries	361 MB
System matrix	168 MB
Block Jacobi 20 preconditioner	11 MB
First order deflation operators	65 MB
CG Method	12 MB
Total	617 MB

Table 18: Breakdown of memory costs for tunnel problem.

10.5 Comparison

Finally we compare the new solver with the existing solvers, see Table 19. From previous experiments we find that a Block Jacobi preconditioner combined with first order deflation yields good results. Usually, this combination is the best, and when this is not the case, it is only slightly slower than the fastest combination. We also find that varying the block size has a minor impact, so we stick to a block size of 20. As the analyzed models lead to symmetric systems, we use the CG method. We also report the times using IDR(8), to emulate an asymmetric solve. We find that running our parallel solver on the GPU rather than on the CPU yields a speedup of about 7 times.

Compared to the existing solvers, our solver outperforms PICOS in the iteration phase for all problems. Furthermore, we expect that if optimizations are applied to the setup of our solver, the setup can be done in less time than both PICOS and PARDISO. Furthermore we find that in the iteration phase sometimes PARDISO is fastest, sometimes our solver is fastest. Still, due to the high setup costs of PARDISO, our solver may still be faster in cases where the system is only solved a few times.

Solver	Uniform cube	Layered cube	Tunnel	Suction pile
Setup (*)	3.63s	8.2s	6.8s	6.4s
CG (GPU)	0.33s [131]	0.13s [50]	0.67s [307]	2.50s [707]
IDR(8) (GPU)	0.35s [120]	0.15s [50]	0.80s [312]	3.15s [754]
CG (CPU)	2.50s [131]	0.90s [50]	4.69s [307]	20.5s [707]
IDR(8) (CPU)	2.95s [134]	0.93s [50]	6.28s [316]	28.5s [791]
PICOS Setup	2.99s	5.18s	2.30s	7.40s
PICOS Solve	0.68s [15]	1.18s [27]	4.58s [97]	6.66s [100]
PARDISO Setup	16.5s	13.5s	3.25s	12.5s
PARDISO Solve	0.92s	0.81s	0.39s	1.02s

Table 19: Run time and number of matrix-vector products (in brackets) for various solvers. The CG and IDR(8) solver use a Block Jacobi (block size = 20) preconditioner and first order deflation. All solve times are for solving single linear system. (*) Note the setup of our own method is not optimized and partly in Python.

We also compare the memory usage of the different solvers, see Table 20. Note that the reported memory for our solvers is the total, including the memory for GPU kernels and CUDA libraries. We find that for all problems both PICOS and our solver use significantly less memory than PARDISO.

Solver	Uniform cube	Layered cube	Tunnel	Suction pile
CG	657 MB	659 MB	617 MB	783 MB
IDR(8)	703 MB	705 MB	667 MB	875 MB
PICOS	590 MB	565 MB	829 MB	1 225 MB
PARDISO	2 781 MB	2 541 MB	1 276 MB	3 244 MB

Table 20: Memory usage for various solvers. The CG and IDR(8) solver use a Block Jacobi (block size = 20) preconditioner and first order deflation.

11 Conclusion

In total we have implemented several Krylov methods on the GPU and combined them with various parallel preconditioners and deflation methods. We tested the performance of these methods using 4 test problems. For the GPU solvers we tested, we can draw a few conclusions:

1. *Deflation method.* The most significant result is that the first order deflation method (described in Section 6.2.7) leads to the fastest convergence for all problems. The impact of this deflation method is much larger than any of the preconditioners we tested. It is important that the deflation vectors correspond to the different volumes in the model. Furthermore, we suggest that the deflation vectors have a weighted overlap on the interface between two volumes, where the weighting depends on the material properties (see Section 9.6.2).

2. *Preconditioner.* Of the tested preconditioners we find that it is best to use the block Jacobi preconditioner. The block size does not seem to matter much, so we suggest to use a block size of 20. The ISAI type preconditioners are competitive in terms of speed for 3 of our test problems. However, for the suction pile model the ISAI type preconditioners lead to a method that does not converge within 5000 iteration. For this reason we suggest not to use ISAI preconditioners.

3. *Krylov method.* All our test problems lead to symmetric system matrices. As expected, for these matrices we find it is always beneficial to use the Conjugate Gradient method. The results also indicate that the IDR(8) method is the preferred method out of the general Krylov solvers.

11.1 Comparison to existing solvers

We find that for the problems we tested the proposed GPU solver consistently outperforms the PICOS in terms of runtime. From the results we can conclude that the iteration phase of the new solver is about 2 to 3 times faster than PICOS. Furthermore, in cases where the same system only needs to be solved a few times (e.g. five times), the new method can outperform PARDISO in terms of total run time, due to the lower setup costs.

Furthermore, as both the new method and PICOS use significantly less memory than PARDISO, they can both be used in large problems where PARDISO runs out of memory.

11.2 Future research

This research proposes an iterative method for solving geotechnical systems on a GPU. We show that we can outperform the current iterative solver in PLAXIS 3D. There are still some topics where we recommend further research:

1. Our system matrices are all symmetric, whereas in practice one may encounter non-symmetric systems too. We found that the IDR(8) method with block Jacobi preconditioning and first order deflation yields good results for symmetric systems. We expect that the same method will provide an efficient algorithm for solving non-symmetric systems too, but we suggest to verify this experimentally.

2. Our research indicates that first order deflation leads to significantly faster convergence compared to rigid body deflation. As the PICOS solver uses rigid body deflation, we suggest to experiment with using first order deflation in PICOS as well to potentially improve the rate of convergence.

3. We have seen that for the suction pile model deflation had a smaller effect compared to the other test problems. We suggest two research directions that could improve the convergence in this model:

   (a) The reason could be that the suction pile model has only a single large volume that is used for deflation. The PICOS solver uses a domain decomposition technique where large volumes of a single material are split into smaller chunks. A similar approach can be combined with our solver. By splitting large volumes into smaller parts, more deflation vectors can be generated, which could potentially lead to faster convergence.

   (b) When the same system is solved multiple times, deflation vectors can be automatically generated based on information from previous solves [12, 28, 7]. This could make deflation more effective in some problems.

4. When using the ParIC preconditioner we found that the convergence rate was worse than with a Jacobi preconditioner. The authors of the ParIC method also noticed similar results when matrices are very ill-conditioned and they suggest to use a block Jacobi method for the triangular solve step in such cases [9]. It could be interesting to try this approach and compare the results.

5. It can be interesting to combine the proposed method with matrix reordering schemes [11]. This can potentially improve the convergence rate at the expense of more setup costs.

6. A more advanced method using multiple coupled GPUs can be developed to solve the systems even faster, or allow for more memory usage and thus larger problems to be solved. Also mixed GPU & CPU solvers could be developed.

List of software

Other than these libraries that are used in the code, I of course used build tools, plotting tools and tools to organize the code, such as: gcc, nvcc, CMake, make, Python, Jupyter notebooks, matplotlib, git.

Package name	Platform	Description
cuBLAS [34]	GPU	Dense linear algebra library
cuSPARSE [34]	GPU	Dense linear algebra library
PyBind11 [22]	C++/Python	Library allowing to import C++ functions into python
Eigen (3.4) [16]	C++	Linear algebra library for C++
SciPy [42]	Python	Sparse linear algebra for Python, also includes Krylov methods
Numpy [18]	Python	Dense linear algebra library for Python

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, k is the level of fill-in
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

Abbreviation	**Long version**
IDR(s)	Induced Dimension Reduction (s being the dimension of the subspace)
FGMRES	Flexible GMRES
PCG	Preconditioned CG
DPCG	Deflated Preconditioned CG
Computer related terms	
GPU	Graphics Processing Unit
CPU	Central Processing Unit
RAM	Random Access Memory
GB (or GB/s)	GigaByte (or GigaByte per second). In this thesis we consistently report bytes to measure memory capacities and bandwidth speeds, never bits.
Gbit (or Gbit/s)	Gigabit, we do not report bit counts in this thesis, only bytes.
Wall Time / Wall clock time	Time that elapses in the real world, from start to finish. Name comes from the time that a *clock on the wall* would measure. All reported times are wall-clock times.
CPU Time	Amount of CPU time is used, when using 8 threads this would mean that 8 CPU seconds are used in every real world second. On the other hand, if a thread is idle, no CPU time is used while real-world time passes. This time is not used throughout this thesis.
PCIe	Peripheral Component Interconnect Express, is a communication bus in computers, usually used to link between the CPU and GPU.

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

Symbol	Meaning
S	a **set**, in Algorithm 2 on page 24 it denotes the set of 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
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)
$\alpha, \beta, \ldots$	Other Greek letters usually refer to a real number
$n, m, \ldots$	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

Symbol	Meaning
A^T	Transpose of A
A^{-1}	Matrix inverse of A
$\mathcal{I}$ or $\mathcal{J}$	a set of indices
$A\left(\mathcal{I},\mathcal{J}\right)$	the matrix A restricted to keep only the rows $\mathcal{I}$ and columns $\mathcal{J}$
$x\left(\mathcal{I}\right)$	the vector x restricted to keep only the elements with index in $\mathcal{I}$
$a_{i,j}$ or $\left(A\right)_{i,j}$	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
$\left\Vert x\right\Vert_2$	2-norm of a vector, defined as: $\left\Vert x\right\Vert_2 = \sqrt{\sum_{i=0}^{N} x_i^2}$
$\left\Vert A\right\Vert_2$	2-norm of a matrix, which turns out to be the maximum row sum. Not similar to the 2-norm of a vector.
$\left\Vert A\right\Vert_F$	Frobenius norm of a matrix: $\left\Vert A\right\Vert_F = \sqrt{\sum_{i=0}^{N}\sum_{j=0}^{N} a_{i,j}^2}$. Similar to a 2-norm of a vector.
∇	Nabla, N-dimensional analog of the differential operator, defined as $\nabla = \left(\frac{\partial}{\partial x}, \frac{\partial}{\partial y}\right)^T$ (its dimension is implied by the context)
Δ	Laplace operator, N-dimensional analog of the second derivative, defined as $\Delta = \nabla\cdot\nabla = \left(\frac{\partial^2}{\partial x^2}, \frac{\partial^2}{\partial y^2}\right)^T$

References

[1] PLAXIS 3D - Tutorial 06: Phased excavation of a shield tunnel. `https://communities.bentley.com/products/geotech-analysis/w/plaxis-soilvision-wiki/45577/plaxis-3d-tutorial-06-phased-excavation-of-a-shield-tunnel`. [accessed 29-11-2021].

[2] PLAXIS 3D - Tutorial 03: 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].

[3] 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.

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

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

[6] 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.

[7] Kevin Burrage and Jocelyne Erhel. On the performance of various adaptive preconditioned gmres strategies. *Numerical linear algebra with applications*, 5(2):101–121, 1998.

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

[9] 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.

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

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

[12] G Diaz Cortes. *POD-based deflation method for reservoir simulation*. PhD thesis, Delft University of Technology, 2019. `https://doi.org/10.4233/uuid:0458fd29-920b-43cb-8c2b-e04be8db0dc7`.

[13] Eigen. Eigen and multi-threading. `https://eigen.tuxfamily.org/dox/TopicMultiThreading.html`. [accessed 29-09-2021].

[14] 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.

[15] 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.

[16] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. `http://eigen.tuxfamily.org`, 2010.

[17] Rohit Gupta, Dimitar Lukarski, Martin B van Gijzen, and Cornelis Vuik. Evaluation of the deflated preconditioned CG method to solve bubbly and porous media flow problems on gpu and cpu. *International Journal for Numerical Methods in Fluids*, 80(11):666–683, 2016.

[18] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[19] Markus Hegland and Paul E Saylor. Block jacobi preconditioning of the conjugate gradient method on a vector processor. *International journal of computer mathematics*, 44(1-4):71–89, 1992.

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

[21] 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].

[22] Wenzel Jakob, Jason Rhinelander, and Dean Moldovan. pybind11 – seamless operability between c++11 and python, 2017. `https://github.com/pybind/pybind11`.

[23] 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.

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

[25] 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`.

[26] Ruipeng Li and Yousef Saad. Gpu-accelerated preconditioned iterative linear solvers. *The Journal of Supercomputing*, 63(2):443–466, 2013.

[27] 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.

[28] D Loghin, D Ruiz, and A Touhami. Adaptive preconditioners for nonlinear systems of equations. *Journal of Computational and Applied Mathematics*, 189(1-2):362–374, 2006.

[29] 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.

[30] 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.

[31] Ronald B Morgan. A restarted GMRES method augmented with eigenvectors. *SIAM Journal on Matrix Analysis and Applications*, 16(4):1154–1171, 1995.

[32] Kentaro Moriya and Takashi Nodera. The deflated-gmres (m, k) method with switching the restart frequency dynamically. *Numerical linear algebra with applications*, 7(7-8):569–584, 2000.

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

[34] NVIDIA. CUDA, release: 11.4, 2021.

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

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

[37] Blume L. Simon C. *Mathematics for Economists.* Norton, 1994.

[38] Peter Sonneveld and Martin B Van Gijzen. IDR(s): A family of simple and fast algorithms for solving large nonsymmetric systems of linear equations. *SIAM Journal on Scientific Computing*, 31(2):1035–1062, 2009.

[39] 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.

[40] Martin B Van Gijzen and Peter Sonneveld. Algorithm 913: An elegant idr (s) variant that efficiently exploits biorthogonality properties. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1–19, 2011.

[41] Fred Vermolen, Kees Vuik, and Guus Segal. Deflation in preconditioned conjugate gradient methods for finite element problems. In *Conjugate Gradient Algorithms and Finite Element Methods*, pages 103–129. Springer, 2004.

[42] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

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

[44] Mingliang Wang, Hector Klie, Manish Parashar, and Hari Sudan. Solving sparse linear systems on nvidia tesla gpus. In *International Conference on Computational Science*, pages 864–873. Springer, 2009.

[45] 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.