Master of Science Thesis

The application of Algebraic Multigrid-based linear solvers for performance enhancement of geomechanical simulators

Arnoud Glasbeek

Faculty of Electrical Engineering, Mathematics and Computer Science

Delft University of Technology

The application of Algebraic Multigrid-based linear solvers for performance enhancement of geomechanical simulators

Master of Science Thesis

submitted in partial fulfillment of the
requirements for the degree of

Master of Science

in

Applied Mathematics

by

Arnoud Glasbeek

This research was performed in:

Numerical Analysis Group
Delft Institute of Applied Mathematics (DIAM)
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Supported by:

Abingdon Technology Center
Schlumberger Oilfield UK Ltd. (SLB)

The work in this thesis was supported by Schlumberger Oilfield UK Ltd.

Delft University of Technology
Department of
Delft Institute of Applied Mathematics (DIAM)

The undersigned hereby certify that they have read and recommend to the Faculty of Electrical Engineering, Mathematics and Computer Science for acceptance a thesis entitled **"The application of Algebraic Multigrid-based linear solvers for performance enhancement of geomechanical simulators"** by **Arnoud Glasbeek** in partial fulfillment of the requirements for the degree of **Master of Science**.

Dated: October 3, 2024

Chairman: _____
prof.dr.ir. C. Vuik

Advisor: _____
prof.dr.ir. H.X. Lin

Committee Members: _____
dr. ir. T.B.Jönsthövel

Abstract

Geomechanical simulations can give essential insights into subsurface processes, but typically require solving large, ill-conditioned linear systems. An important method for solving these linear systems is the Conjugate Gradient method, but applying this method to ill-conditioned matrices can result in slow convergence. To improve the convergence of the Conjugate Gradient method, the iterative solver is preconditioned using the Algebraic Multigrid method. In Algebraic Multigrid methods, a hierarchy of matrices of different sizes is derived. When applying these methods as a preconditioner to the Conjugate Gradient method, on each level of the multigrid hierarchy fast convergence is observed in particular components of the residual. This leads to much fewer iterations being required in the Conjugate Gradient method, at the cost of the iterations being computationally more expensive. These Algebraic Multigrid methods do require a more problem-specific setup configuration than more simple preconditioners like the Jacobi preconditioner. In this research, the Conjugate Gradient method preconditioned with various Algebraic Multigrid methods is studied and compared with the Jacobi preconditioned Conjugate Gradient method. For this, the Conjugate Gradient method, preconditioned with both the Jacobi and Algebraic Multigrid-based methods, is applied to linear problems derived from geomechanical simulations. Using Algebraic Multigrid preconditioners can reduce the number of iterations required for convergence of the Conjugate Gradient method by a factor of 80. While a single iteration with an Algebraic Multigrid preconditioner is more time-expensive than an iteration with a Jacobi preconditioner, significant reductions, of up to five times, are observed in the runtimes of the linear solver. This comes at the cost of a higher peak memory requirement in the application of the linear solver. The vast reduction of the runtime of the linear solvers makes the studied geomechanical simulations significantly faster. This makes the Algebraic Multigrid preconditioners a valuable addition to these simulations.

Acknowledgments

The research for this thesis project was conducted during a six-month internship at the SLB Abingdon Technology Centre (AbTC) in the United Kingdom. I am thankful to have had the opportunity to conduct this research within the company and to gain valuable insights into what it is like to work in this company.

From the side of Delft University of Technology, this project was supervised by Prof. Kees Vuik. I am grateful for his help with the setup of this project and for his valuable insights into the topics studied in this project. For the proposal of this project, I would also like to thank Dr. Tom Jönsthövel, who formed the first contact with SLB and supported me during the project.

During the internship at SLB I was directly supervised by Dr. Giovanni Isotton. I would like to thank him for his great explanations of various topics and his enthusiastic support of my research. I have gotten further support from Dr. Andrew Pearce, Dr. Clairet Guerra and the rest of the Geomechanics Team at AbTC and would also like to express my appreciation for their support.

I could not have made it where I am without the support of my family. I would like to thank my father Eduard Glasbeek, who always takes care of me in difficult situations, and my brother Lennard Glasbeek and my sister Femke Glasbeek, who are always there to make me laugh and help me through difficult times. I also want to thank my grandparents, Jos and Ity van der Horst, who always help and encourage me to keep going.

Lastly, I would like to express my gratitude to my mother Iteke Glasbeek, who unfortunately is no longer with us. I am forever grateful for everything she taught me and hope I am making her proud.

Arnoud Glasbeek
Gouda, The Netherlands
October 3, 2024

Contents

List of Figures

List of Tables

Table of Acronyms

Acronym	Description	First Occurrence
AMG	Algebraic Multigrid	Chapter 1
BiCGSTAB	Biconjugate Gradient Stabilised	Section 3.2
BIM	Basic Iterative Method	Section 3.1
CCS	Carbon Capture and Storage	Chapter 1
CG	Conjugate Gradient	Section 3.2
CO_2	Carbon Dioxide	Chapter 1
FEM	Finite Element Method	Chapter 1
GMRES	Generalized Minimum Residual Method	Section 3.2
GS	Gauss-Seidel	Section 3.1
ILU	Incomplete LU	Section 5.4.2
ILUT	Incomplete LU with Threshold(s)	Section 5.4.2
nnz	Number of nonzeros	Chapter 1
PDE	Partial Differential Equation	Chapter 1
SOR	Successive Over Relaxation	Section 3.1
SPD	Symmetric Positive Definite	Section 2.3
SSOR	Symmmetric Successive Over Relaxation	Section 3.1

Table of Symbols

Symbol	Description
Bold letters ($\mathbf{x}, \mathbf{y}$, etc.)	Generally used for vectors, $\mathbf{x} \in \mathbb{R}^n$ denotes vector of size n.
$\mathbf{\Phi}(\cdot)$	Vector function, $\mathbf{\Phi} : \mathbb{R}^n \to \mathbb{R}^m$ for some m, n.
$\mathbf{0}_n$	Vector of size n with elements all 0
$\mathbf{1}_n$	Vector of size n with elements all 1
$\mathbf{e}_j$	Standard basis vector, consisting of all but one 0's and one 1 in position j
$(\mathbf{x}, \mathbf{y})$	Inner product of vectors, if not specified Euclidean inner product $(\mathbf{x}, \mathbf{y}) = \mathbf{x}^T\mathbf{y}$ is used
$\|\mathbf{x}\|$	Norm of vector , if not specified Euclidean norm $\sqrt{(\mathbf{x}, \mathbf{x})}$ is used
Capital letters (A, B, etc.)	Generally used for matrices, $A \in \mathbb{R}^{m \times n}$ denotes matrix of size m by n.
$O_{n,m}$	Zero matrix of sizes n by m
$\|A\|$	Norm of matrix, if not specified Euclidean norm $\sup_{x \neq 0} \frac{\|Ax\|_2}{\|x\|_2}$ is used
$\lambda_i(A), \lambda_{\max}(A), \lambda_{\min}(A)$	i'th, maximum and minimum eigenvalue of matrix A
$\sigma(A)$	Spectrum of A, set of all eigenvalues
$\kappa(A)$	Condition number of A, given by $\kappa(A) = \|A\|\|A^{-1}\|$
$\rho(A)$	Spectral radius of A, given by $\rho(A) = \max_i\{\|\lambda_i(A)\|\}$
Brackets with subscript of small case, E.g. $[\mathbf{x}]_i, [A]_{i,j}$	Used to denote specific elements of vectors and matrices. $[A]_{i,j}$ denotes the element of matrix A in row i and column j
Brackets with subscript of two letters with :, E.g. $[\mathbf{x}]_{i:j}, [A]_{i:j,k:l}$	Used to denote blocks of elements of vectors and matrices. $[A]_{i:j,k:l}$ denotes the block of elements of matrix A from row i to row j and from column k to column l
Brackets with subscript of capital letter E.g. $[\mathbf{x}]_S, [A]_{S,T}$	Used to denote blocks of elements of vectors and matrices. $[A]_{S,T}$, with $S, T \subset \{1, \dots n\}$, denotes the block of elements of matrix A in rows i for $i \in S$ and columns j for $j \in T$

1 Introduction

Geomechanical simulations form an important tool in the modelling of subsurface deformations. Modelling these deformations is of essential importance in a wide range of applications, for example, to ensure the integrity of wellbores **(Allawi and Al-Jawad** [1]), to predict subsidence caused by the removal of fluids or gases **(Fredrich et al.** [12]) and to predict the effects of CO_2 Capture and Storage (CCS) **(Li et al.** [20]) **(Metz et al.** [21]).

To model these subsurface deformations, changes over time in displacement, stress and strain in the area of interest are computed. For this, the area of interest is discretised using numerical methods such as the Finite Element Method (FEM) **(Zienkiewicz and Taylor** [39]). Through these discretisations, accurate estimates of changes in displacement, stress and strain can be obtained, but this can be computationally expensive for large models. In these simulations, typically the most computationally expensive part is solving large linear problems. There are a wide range of methods to solve these problems, but there are large differences in performance on large models between these methods. This means that the right method has to be found to solve these linear problems.

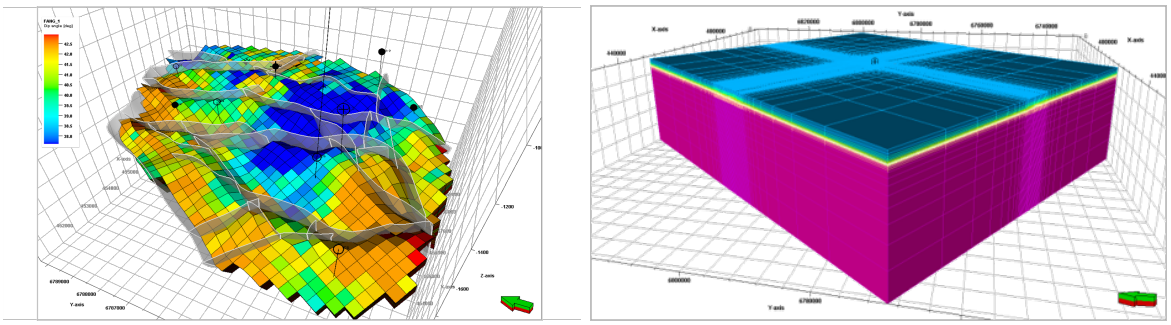
The Conjugate Gradient (CG) method is one of the most effective methods for solving large linear problems **(Saad** [26]). The performance of this method highly depends on the condition number of the observed matrix. To further improve the performance of this method, preconditioning methods can be applied to the linear system. However, finding the best preconditioner for a linear system can be complicated. Simple preconditioners, such as the Jacobi preconditioner **(Saad** [26]), can be used, but these typically do not give as much improvement as more complicated methods such as deflation preconditioners **(Jönsthövel** [16]) and Algebraic Multigrid (AMG) preconditioners **(Stüben** [31]). These more complicated methods have the downside of requiring a proper setup configuration for the approached application.

Especially AMG-based preconditioners can give significant improvements if a good problem-specific setup is used. For this, different AMG preconditioners are approached. For these AMG preconditioners, a good setup for the linear systems obtained from geomechanical simulations is tried to be obtained.

The observed linear systems are obtained through simulations in the Visage finite-element geomechanics simulator **(SLB** [30]). This simulator performs numerical calculations of rock stresses, strains, displacements and failures in geomechanical processes. This can be used in a wide range of modelling applications, such as for modelling the integrity of wells, subsidence, and the effects of CO_2 Capture and Storage (CCS) **(SLB** [28]).

In the simulations studied, the Visage simulator is coupled with the Intersect high-resolution reservoir simulator **(SLB** [29]). The Intersect simulator models the flow of fluids and gases in a reservoir. In these coupled simulations, both the flow of fluids and

gasses and the effect of this flow on the stress and strain in the surrounding rock are modelled, with both simulators being used for their own application. An example of a discretised reservoir domain as encountered in simulations within Intersect is shown in Figure 1.1a. The corresponding geomechanical FEM discretisation as used in the simulations in Visage is shown in Figure 1.1b. This reservoir domain is embedded in the geomechanical domain.



(a) (b)

Figure 1.1: Example of (a) reservoir model observed in the Intersect simulator and (b) geomechanical model observed in the Visage simulator

In these simulations, the geomechanical models observed in Visage are typically much larger than the models observed in Intersect. This leads to the simulation in Visage being computationally much more expensive. For the geomechanical simulation in Visage, the most expensive part is solving linear problems. This means that to decrease the runtime of the coupled simulations, an effective linear solver has to be used.

In this research, the AMG-based preconditioners are applied to the linear systems encountered in the geomechanical simulations in the Visage simulator. The performance of three AMG-based preconditioners is compared with each other and with previously used preconditioners. There, significant improvements are seen when using the AMG-based preconditioners for both the required number of iterations in the CG method and the runtime of the linear solvers. For the number of iterations, improvements as high as a factor of 80 are seen by applying the AMG-based preconditioners. Although single iterations are longer with the AMG-based methods, the runtimes are improved by up to a factor five through the application of the AMG methods. This improvement in runtime comes at the cost of a higher peak memory requirement.

Here, first, an overview of the workflow for geomechanical simulations performed in the Visage simulator is given in Chapter 2. Then, iterative linear solver methods are discussed in Chapter 3, with a focus on the two most important methods for this application, the CG and AMG methods. After this, in Chapter 4, it is discussed how the experiments on the basis of which the linear solver methods are compared are carried out. The results of these experiments and the comparison of the methods are presented in Chapter 5. Lastly, the conclusions drawn from this research together with recommendations on the use of linear solvers in these simulations are given in Chapter 6.

2 Visage Framework

The geomechanical simulations considered here are obtained using the Visage finite-element geomechanics simulator. The workflow of the geomechanical simulations performed in the Visage simulator can be simplified into three stages. In Figure 2.1 an overview is given of this workflow. From the input, the finite element problem is formulated. This finite element problem is a nonlinear problem, which is solved iteratively using the Newton-Raphson method. Applying the Newton-Raphson method requires solving a linear problem, which is done with one of several linear solvers. When the Newton-Raphson method converges to a suitable solution, the simulator advances to the next load step. When the last load step is reached, the results of the simulation are returned.

In a geomechanical simulation in Visage, the domain that is observed is typically of the form of a cuboid. The reservoir observed in the flow simulation in Intersect is located at the barycenter of this cuboid. The domain around this reservoir is extended in every direction, with the so-called overburden, underburden and sideburden. These are mandatory to properly set the boundary conditions of the mechanical problem. For the domain in these simulations, a boundary condition is prescribed on all six sides of the cuboid. Using an FEM discretisation, the domain is discretised using hexahedral elements with eight nodes.

This chapter discusses the three stages of the Visage workflow. First, a description

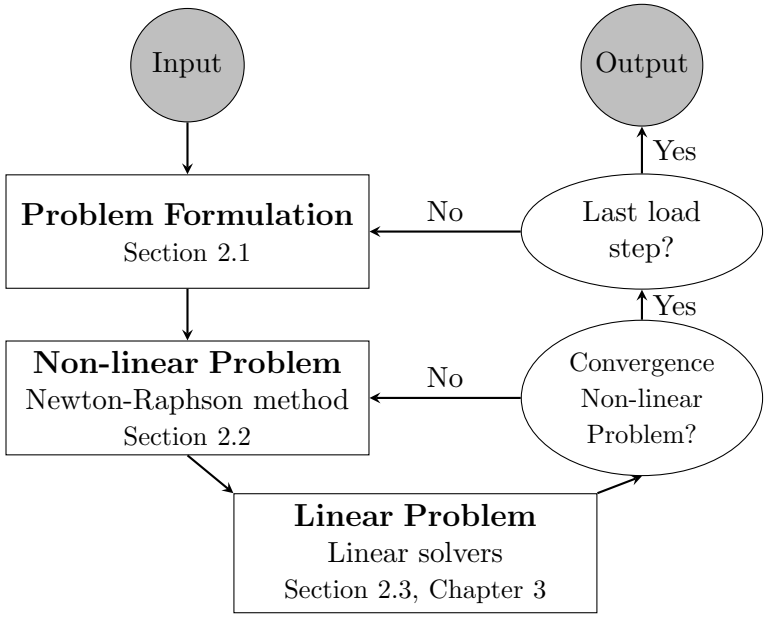


Figure 2.1: Overview of workflow of geomechanical simulations in Visage

of how the nonlinear problem is formulated is given in Section 2.1. After this, it is described how this nonlinear problem is solved using the Newton-Raphson method in Section 2.2. Finally, an overview of linear solvers is given in Section 2.3, which is expanded upon in Chapter 3.

2.1 Problem Formulation

Modelling the deformation behaviour of a volume of rock or soil is done through the stress, strain and displacements in that volume. To model the stresses in a volume, a global stress equilibrium equation needs to be solved. This states that the external forces on the volume of rock or soil are equal to the internal forces. As the stress in the rock describes the internal forces, this can be used to determine the stresses within the rock or soil, which is then used to describe the strain and displacement.

2.1.1 PDE problem

To formulate the equations that model the equilibrium equation, first the concepts of stress, strain and displacement are described. For this, let Ω describe the volume in which the equilibrium equation is to be solved and let $\mathbf{p} = (x_p, y_p, z_p)$ describe a point in this domain. In this point, the displacement $\mathbf{d}$, stress $\boldsymbol{\sigma}$ and strain $\boldsymbol{\epsilon}$ are given by the vectors

$$\mathbf{d} = \begin{pmatrix} d_x \\ d_y \\ d_z \end{pmatrix} \qquad \boldsymbol{\sigma} = \begin{pmatrix} \sigma_{xx} \\ \sigma_{yy} \\ \sigma_{zz} \\ \tau_{xy} \\ \tau_{yz} \\ \tau_{zx} \end{pmatrix}. \qquad \boldsymbol{\epsilon} = \begin{pmatrix} \epsilon_{xx} \\ \epsilon_{yy} \\ \epsilon_{zz} \\ \gamma_{xy} \\ \gamma_{yz} \\ \gamma_{zx} \end{pmatrix} \tag{2.1}$$

There, the displacement $\mathbf{d}$ describes the movement or shift of the point $\mathbf{p}$ and consists of three components, one for each direction. The stress $\boldsymbol{\sigma}$, which describes the force in a unit area around $\mathbf{p}$ and the strain $\boldsymbol{\epsilon}$, which describes the change in shape due to stress, consist of six components. Both these vectors consist of three normal components, of form σ_{xx} for stress and ϵ_{xx} for strain, and three shear components, of form τ_{xy} for stress and γ_{xy} for strain. Figure 2.2 illustrates the different components of the stress vector on a small cube. There, it is seen that there are actually nine components for stress and strain. However, due to symmetries, only the six components of stress and strain given in Equation 2.1 need to be considered.

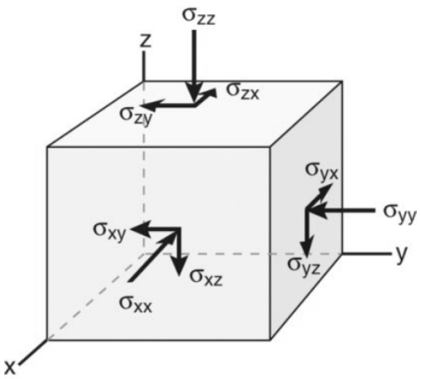


Figure 2.2: Stress components in a small cube. This example uses σ_{xy} instead of τ_{xy} for shear stress components **(Fossen, p. 77 [10])**

Along with the three vectors for displacement, stress and strain, several forces can also be described at $\mathbf{p}$. These include external forces, denoted by $\boldsymbol{\beta}$,

and traction forces, denoted by $\mathbf{t}$. These forces have three components each, one in every direction, similar to the displacement.

Using the described notation, the global stress equilibrium equation can be described. Timoshenko and Goodier [36] describes this stress equilibrium as the following Partial Differential Equation (PDE),

$$\begin{cases} \frac{\partial \sigma_{xx}}{\partial x} + \frac{\partial \tau_{xy}}{\partial y} + \frac{\partial \tau_{zx}}{\partial z} & = \beta_x \\ \frac{\partial \tau_{xy}}{\partial x} + \frac{\partial \sigma_{yy}}{\partial y} + \frac{\partial \tau_{yz}}{\partial z} & = \beta_y \\ \frac{\partial \tau_{zx}}{\partial x} + \frac{\partial \tau_{yz}}{\partial y} + \frac{\partial \sigma_{zz}}{\partial z} & = \beta_z \end{cases} \qquad \text{on } \Omega. \tag{2.2}$$

This PDE can be rewritten to

$$S^T \boldsymbol{\sigma} = \boldsymbol{\beta}, \tag{2.3}$$

with

$$S^T = \begin{bmatrix} \frac{\partial}{\partial x} & 0 & 0 & \frac{\partial}{\partial y} & 0 & \frac{\partial}{\partial z} \\ 0 & \frac{\partial}{\partial y} & 0 & \frac{\partial}{\partial x} & \frac{\partial}{\partial z} & 0 \\ 0 & 0 & \frac{\partial}{\partial z} & 0 & \frac{\partial}{\partial y} & \frac{\partial}{\partial x} \end{bmatrix}. \tag{2.4}$$

Together with this PDE, two possible boundary conditions are described. Typically, the boundary Γ is divided into two parts, a displacement boundary Γ_d, where a displacement is induced and a traction boundary Γ_σ, where a strain is induced. This gives the boundary condition

$$\begin{cases} \mathbf{d} & = \bar{\mathbf{d}}, \mathbf{t} = \mathbf{0} \qquad \text{on } \Gamma_d \\ \mathbf{t} & = \bar{\mathbf{t}} = \bar{\boldsymbol{\sigma}} \cdot \mathbf{n} \qquad \text{on } \Gamma_\sigma, \end{cases} \tag{2.5}$$

where $\bar{\mathbf{d}}, \bar{\mathbf{t}}$ and $\bar{\boldsymbol{\sigma}}$ are given displacements, traction forces and strains and $\mathbf{n}$ is a normal vector orthogonal to the boundary.

By solving the equilibrium equation, the stresses are obtained, which can then be related to the strain in $\mathbf{p}$. This is done using the relations obtained from Hooke's law, which states

$$\boldsymbol{\sigma} = D\boldsymbol{\epsilon}. \tag{2.6}$$

Here, D describes the stress-strain relationship. A simple example for this matrix is the isotopic stress-strain relationship described by Timoshenko and Goodier [36], as

$$D = \frac{E}{1-\nu^2} \begin{bmatrix} 1 & \nu & \nu & 0 & 0 & 0 \\ \nu & 1 & \nu & 0 & 0 & 0 \\ \nu & \nu & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1-\nu}{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1-\nu}{2} & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{1-\nu}{2} \end{bmatrix}. \tag{2.7}$$

There, E describes the Young's modulus and ν the Poisson's ratio. The values of these depend on the properties of the material. In many problems D is a more complicated matrix, which can depend on the stress or strain itself. This makes the stress-strain relationship nonlinear.

The strain in this point is then used to determine the displacement in the point. Zienkiewicz and Taylor [39] **(p.22)** describes the relations between these as

$$\boldsymbol{\epsilon} = S\mathbf{d}. \tag{2.8}$$

There, S is the transpose of the operator described in Equation 2.4. With this, the global stress equilibrium equation can be described in terms of the displacement as

$$S^T DS\mathbf{d} = \boldsymbol{\beta}. \tag{2.9}$$

As this PDE depends on the matrix D, which is dependent on the description of the problem, nothing can be said directly about this problem. However, for the problems observed in this research, it can be assumed that the matrix D is such that the PDE is elliptic.

2.1.2 FEM problem

Solving the described continuous problems is usually not possible. This means that a way has to be found to describe the continuous problem as a discrete problem. This is done through the Finite Element Method (FEM), which reduces the continuous problem to a problem in a finite number of elements.

To transform a continuous problem into a discrete problem using FEM, the area of interest is divided into triangular or quadrilateral elements for two-dimensional problems and into triangular pyramid or cuboid elements for three-dimensional problems. Figure 2.3 shows a simple example of a two-dimensional domain discretised in triangles using the FEM method. This simple example is not representative of the problems observed in the geomechanical simulations considered here. Figure 1.1b gives a better, but more complicated, representation of an FEM discretisation of a large domain that is observed in the simulations considered here. The domains observed in the simulations are large, three-dimensional domains, that are discretised in hexahedral elements with eight nodes.

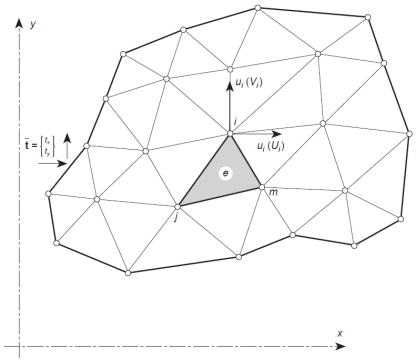


Figure 2.3: Example of FEM discretisation of a two-dimensional region using triangular elements **(Zienkiewicz and Taylor, p. 20 [39])**.

The elements in an FEM discretisation are represented by a finite number of interconnected nodal points. In the elements, the continuous functions are described by a basis of standard functions. The coefficients of these functions form a discrete problem.

For problems like those studied here, the finite number of unknowns are the displacements at the nodal points. The displacement within an element is described through the displacements in the nodal points. The displacement functions then describe the strain within the elements in terms of the nodal displacement. These

strains then describe the stress in the element **(Zienkiewicz and Taylor, p. 18** [39]).

In the FEM model, the displacement at a point $\mathbf{p}$ is approximated by the nodes of the element e in which $\mathbf{p}$ is located. The displacements in the nodes of an element are described by $\mathbf{u}^e \in \mathbb{R}^{3N_o^e}$, where N_o^e is the number of nodes in the element e. This vector is of the form $\mathbf{u}^e = \begin{pmatrix} \mathbf{u}_1^e \\ \vdots \\ \mathbf{u}_{N_o^e}^e \end{pmatrix}$, where $\mathbf{u}_i^e$ is the displacement in a node i. From the nodal displacement the displacement in $\mathbf{p}$ can be obtained as

$$\mathbf{d} = \sum_{i=1}^{N_o^e} F_i^e \mathbf{u}_i^e, \tag{2.10}$$

where $F_i^e \in \mathbb{R}^{3\times 3}$ represents the function that maps the coordinates of node i to the coordinates of the point in which the displacement is obtained. The position functions together form a shape function $F^e \in \mathbb{R}^{3\times 3N_o^e}$ for this element, where

$$F^e = \begin{bmatrix} F_1^e & F_2^e & \cdots & F_{N_o^e}^e \end{bmatrix}. \tag{2.11}$$

This leads to the displacement in $\mathbf{p}$ being described as

$$\mathbf{d} = F^e \mathbf{u}^e, \tag{2.12}$$

Now, the strain at $\mathbf{p}$ can be described by the displacement at the point, which means that the strain can be described in terms of the nodal displacements. For this, Equation 2.8 and Equation 2.12 are combined, to obtain

$$\boldsymbol{\epsilon} = B^e \mathbf{u}, \tag{2.13}$$

where $B^e = SF^e$. Using Equation 2.6, the stress can then be obtained at this point.

Similarly to the global equilibrium, an equilibrium in a single element can be obtained. Then it is obtained that the internal stresses are equal to the combination of the distributed body forces $\boldsymbol{\beta}$ in the element, the traction forces $\bar{\mathbf{t}}$ on the boundary of the element, and the nodal forces $\mathbf{q}^e \in \mathbb{R}^{3N_o^e}$ in the element. Zienkiewicz and Taylor [39] **(p.23)** describes the equilibrium in a single element as

$$K^e \mathbf{u}^e + \mathbf{f}^e = \mathbf{q}^e. \tag{2.14}$$

Here, K is described through the relations between stress, strain and displacement, as

$$K_e = \int_{V^e} (B^e)^T D B^e \mathrm{d}V^e, \tag{2.15}$$

where V^e describes the volume of the element. $\mathbf{f}^e$ is described using the distributed body forces and external loading on the boundary $\bar{\mathbf{t}}$, as

$$\mathbf{f}^e = -\int_{V^e} (F^e)^T \boldsymbol{\beta} \mathrm{d}V^e - \int_{\Gamma^e} (F^e)^T \bar{\mathbf{t}} \mathrm{d}\Gamma^e, \tag{2.16}$$

where Γ^e describes the boundary of the element.

This equation in a single element can be used to obtain an equation for the entire domain. This is simply done by extending the concepts described in a single element to the entire domain. The vector of nodal displacements is then given by $\mathbf{u} \in \mathbb{R}^{N_o}$, where N_o is the number of nodes in the domain. Then the displacement in a point is given as

$$\mathbf{d} = F\mathbf{u}, \tag{2.17}$$

where $F \in \mathbb{R}^{3\times 3N_o}$ is a global shape function, consisting of similar submatrices as the shape functions of a single element. The global shape function is described through the submatrices of the position function of the elements. For a point in element e, this global shape function has submatrices $F_i = [F^e]_i$ for the nodes in element e and $F_i = O_{3,3}$ for the nodes not in element e. With this global shape function, a global version of the matrix B^e is obtained that describes the strain at a certain point as $B = SF$.

Using this global shape function, the equation in one element in Equation 2.14 can be extended to a global equation as

$$K\mathbf{u} + \mathbf{f} = \boldsymbol{\rho}. \tag{2.18}$$

There, $\boldsymbol{\rho}$ describes external concentrated forces at the different nodes and K and $\mathbf{f}$ are global versions of K^e and $\mathbf{f}^e$, given by

$$\begin{aligned} K &= \int_V B^T D B \mathrm{d}V^e \\ \mathbf{f} &= -\int_V F^T \boldsymbol{\beta} \mathrm{d}V - \int_{\Gamma^e} F^T \bar{\mathbf{t}} \mathrm{d}\Gamma. \end{aligned} \tag{2.19}$$

Here, V represents the entire domain in which the equilibrium equations are solved, with Γ representing its boundary. As K generally depends on $\mathbf{u}$, this problem can be expected to be nonlinear.

2.2 Nonlinear Problem

The FEM discretisation of the global stress equilibrium equation is expected to be a nonlinear problem. This means that an effective method for solving nonlinear problems has to be employed to solve this equation. To apply a nonlinear solver method, the problem is rewritten in the form

$$\boldsymbol{\Phi}(\mathbf{u}) = \boldsymbol{\eta}. \tag{2.20}$$

There, $\boldsymbol{\Phi} : \mathbb{R}^n \to \mathbb{R}^n$ a nonlinear vector function of $\mathbf{u}$, given by $\boldsymbol{\Phi}(\mathbf{u}) = K\mathbf{u}$ and $\boldsymbol{\eta} \in \mathbb{R}^n$ is a vector, given by $\boldsymbol{\eta} = \boldsymbol{\rho} - \mathbf{f}$. This can be transformed to

$$\boldsymbol{\Psi}(\mathbf{u}) = \boldsymbol{\Phi}(\mathbf{u}) - \boldsymbol{\eta} = \mathbf{0}, \tag{2.21}$$

which means that the problem that is to be solved is finding a zero of a nonlinear function. Finding a nonzero of a nonlinear function can in general only be done using

an iterative method. A wide range of iterative methods for finding zeros of nonlinear problems is available, as described by Zienkiewicz and Taylor [40].

To solve the nonlinear problem observed here, the Newton-Raphson method is used. This is the iterative method with the fastest convergence of all iterative methods for solving nonlinear problems. The method uses the Jacobian of the function $\mathbf{\Psi}(\mathbf{u})$ to determine how $\mathbf{u}$ should be changed to converge to a solution.

For this, let $\mathbf{u}_i$ be an approximate solution to Equation 2.21. Then it is desired to find $\mathbf{u}_{i+1} = \mathbf{u}_i + \Delta\mathbf{u}_i$, so that $\mathbf{u}_{i+1}$ is a better approximate solution and $\Delta\mathbf{u}_i$ is a small change to $\mathbf{u}_i$.

This is done through the Taylor expansion of $\mathbf{\Psi}(\mathbf{u})$ around $\mathbf{u}_i$, which gives

$$\mathbf{\Psi}(\mathbf{u}) = \mathbf{T}(\mathbf{u}; \mathbf{u}_{i+1}) = \mathbf{\Psi}(\mathbf{u}_i) + \left(\frac{\partial \mathbf{\Psi}(\mathbf{u}_i)}{\partial \mathbf{u}}\right)(\mathbf{u} - \mathbf{u}_i) + \mathcal{O}((\mathbf{u} - \mathbf{u}_i)^2). \tag{2.22}$$

Evaluating this in $\mathbf{u}_{i+1}$ gives

$$\begin{aligned} \mathbf{\Psi}(\mathbf{u}_{i+1}) &= \mathbf{\Psi}(\mathbf{u}_i) + \left(\frac{\partial \mathbf{\Psi}(\mathbf{u}_i)}{\partial \mathbf{u}}\right)(\mathbf{u} - \mathbf{u}_i) + \mathcal{O}((\mathbf{u} - \mathbf{u}_i)^2) \\ \mathbf{\Psi}(\mathbf{u}_{i+1}) &= \mathbf{\Psi}(\mathbf{u}_i) + \left(\frac{\partial \mathbf{\Psi}(\mathbf{u}_i)}{\partial \mathbf{u}}\right)(\Delta\mathbf{u}_i) + \mathcal{O}(\Delta\mathbf{u}_i^2). \end{aligned} \tag{2.23}$$

Since the update $\Delta\mathbf{u}_i$ can be assumed to be small, the higher order terms $\mathcal{O}(\Delta\mathbf{u}_i^2)$ are small and thus

$$\mathbf{\Psi}(\mathbf{u}_{i+1}) = \mathbf{\Psi}(\mathbf{u}_i) + \left(\frac{\partial \mathbf{\Psi}(\mathbf{u}_i)}{\partial \mathbf{u}}\right)\Delta\mathbf{u}_i \approx 0. \tag{2.24}$$

Now let $K_T = -\frac{\partial \mathbf{\Psi}(\mathbf{u}_i)}{\partial \mathbf{u}}$ and $\mathbf{\Psi}_i = \mathbf{\Psi}(\mathbf{u}_i)$, then the linear problem

$$K_T \Delta\mathbf{u}_i = \mathbf{\Psi}_i \tag{2.25}$$

is obtained. By solving this linear problem, the update of $\mathbf{u}_i$ is found and thus $\mathbf{u}_{i+1} = \mathbf{u}_i + \Delta\mathbf{u}_i$ can be calculated. The linear problem is solved using a linear solver method, as described in Section 2.3 and Chapter 3.

In Algorithm 2.1 an overview is given of the algorithm used to find a suitable approximation to a nonlinear problem using the Newton-Raphson method. The stopping condition is typically based on tolerance of the residual or on a maximum number of iterations.

2.3 Linear Problem

The most time-consuming part in solving the nonlinear problem with the Newton-Raphson method is obtaining the corrections to the approximate solutions $\mathbf{u}_i$. To obtain this correction, the linear problem in Equation 2.25 has to be solved. To solve such a linear problem, a wide range of methods exists, of which a selection is discussed here. The methods discussed here are the specific methods that can be used in simulations using the Visage simulator.

Algorithm 2.1: Newton-Raphson method

Data: Vector function $\mathbf{\Psi}(\cdot)$, Jacobian of vector function $\frac{\partial \mathbf{\Psi}(\cdot)}{\partial \mathbf{u}}$, Initial approximate solution $\mathbf{u}_0$
Result: Approximate solution $\mathbf{u}$
Initialize $i = 0$
while *Stopping condition not satisfied* **do**
  Compute $K_T = -\frac{\partial \mathbf{\Psi}(\mathbf{u}_i)}{\partial \mathbf{u}}$ and $\mathbf{\Psi}_i = \mathbf{\Psi}(\mathbf{u}_i)$
  Solve $K_T \Delta \mathbf{u}_i = \mathbf{\Psi}_i$ using a linear solver method
  Update $\mathbf{u}_{i+1} = \mathbf{u}_i = \Delta \mathbf{u}_i$
  Set $i = i + 1$
end

To simplify the notation for describing the linear solvers, the linear problem is described as

$$A\mathbf{x} = \mathbf{b}, \tag{2.26}$$

with $A \in \mathbb{R}^{n \times n}$ a matrix of coefficients, $\mathbf{x} \in \mathbb{R}^n$ a vector of unknowns and $\mathbf{b} \in \mathbb{R}^n$ right-hand side vector. This problem directly corresponds to the linear problem that is solved in each iteration of the nonlinear solver given in Equation 2.21. Then A corresponds to K_T, $\mathbf{x}$ to $\Delta \mathbf{u}_i$ and $\mathbf{b}$ to $\mathbf{\Psi}_i$.

The methods that can be used to solve linear problems can be divided into two groups. The first of these are direct methods, which are described in Section 2.3.2. These methods directly find an exact solution to a linear system with a nonsingular matrix. The second type of method instead iteratively finds an approximation to the exact solution. These iterative methods do in general not find exact solutions to linear problems, but are much more suitable for large linear problems that are typically encountered in simulations. A selection of iterative linear solver methods is described in Chapter 3.

2.3.1 Properties of linear problem

Before discussing methods to solve the linear problems encountered in the simulations, it is important to ensure that these methods are applicable. For this, some of the properties of the observed problems are discussed.

Firstly, the observed PDEs are elliptic. From this it is obtained that, if the FEM discretisation is performed well, the matrices in the considered linear problems are Symmetric Positive Definite (SPD) **(Süli [34])**. The matrix being SPD is helpful in the application of linear solver methods, particularly in for the iterative methods discussed in Chapter 3. The matrix being SPD can be confirmed by observing the matrix itself. There it is seen that the matrix A is a symmetric matrix. Furthermore, the assumption can be made that the diagonal elements of this matrix are all positive, $[A]_{i,i} > 0$ for all i. In addition, it can be assumed that for every row i, $[A]_{i,i} > \sum_{j=1}^{N} [A]_{i,j}$, which means that A is diagonally dominant. Together, these properties confirm that the coefficient matrix A is SPD **(Horn and Johnson, p. 438 [14])**.

An important property for all linear solvers that directly follows from the matrices being SPD is that they are nonsingular. This means that a unique solution to this linear system is guaranteed to exist.

The linear problems observed in these types of simulations are typically large and sparse. Because the problems are large, the linear solvers used should be methods that scale well with the size of the linear problems. Furthermore, the methods have the added requirement of preserving the sparsity of the problems, as the loss of sparsity can give significant increases in memory requirements. Both of these properties make direct solver methods less suitable than iterative methods for the problems observed in these simulations.

Lastly, the observed matrices can be assumed to be ill-conditioned. In large problems, there can be large irregularities in the FEM discretisation, with large differences between the sizes of elements. This leads to ill-conditioned matrices, which make the application of iterative methods more complicated. Typically, a more ill-conditioned linear system requires more iterations than a well-conditioned system. To improve the conditioning of the linear system, preconditioning methods must be applied, as covered in Chapter 3.

2.3.2 Direct Linear Solver Methods

Direct methods are the most simple methods for solving linear problems, but are typically not suitable for solving large linear systems observed in the considered geomechanical simulations. Two of methods of this type, matrix inversion and triangular substitution through Choleski decomposition, are discussed here. These are the two methods that can be employed for small problems in the Visage simulator. More direct methods exist for solving linear problems and are covered by Vuik, C. and Lahaye, D.J.P. [38].

The most simple method used to directly solve a linear system is matrix inversion. For matrix inversion, a inverse matrix A^{-1} of A is found such that

$$A^{-1}A = AA^{-1} = I. \tag{2.27}$$

Then this inverse matrix is used to obtain the solution to $A\mathbf{x} = \mathbf{b}$ as $\mathbf{x} = A^{-1}\mathbf{b}$. For a nonsingular matrix, this inverse matrix is guaranteed to exist. The calculation of this matrix can be done using several methods, such as Gaussian elimination and eigendecomposition of the matrix. However, this method is computationally very expensive and the inverse matrix cannot be guaranteed to be sparse when A is sparse. Together, this means that matrix inversion can only be applied to very small linear systems.

Triangular substitution through Choleski decomposition is computationally more efficient than matrix inversion. To apply triangular substitution to a matrix, the matrix is decomposed into an upper and lower triangular matrix, after which triangular substitution is applied **(Vuik, C. and Lahaye, D.J.P., p. 54 [38])**. For the Choleski decomposition, a lower triangular matrix C is computed such that $A = CC^T$. The matrix C is obtained entry-wise from the matrix A. For this, the columns of C, starting with the first column, are obtained by first computing the diagonal entry in

the column as

$$[C]_{k,k} = \sqrt{[A]_{k,k} - \sum_{j=1}^{k-1} C_{k,j}^2}. \tag{2.28}$$

This is used to compute the entries below the diagonal, starting with the first entry below the diagonal, as

$$[C]_{i,k} = \frac{1}{[C]_{k,k}} \left([A]_{i,k} - \sum_{j=1}^{k-1} C_{i,j} C_{k,j} \right). \tag{2.29}$$

Although this method gives a significant improvement over matrix inversion, it is still computationally too expensive to be applied to the large systems that are typically encountered in geomechanical simulations. Also, like matrix inversion, this method can lead to a loss of sparsity for sparse systems. This means that to find a solution to the encountered problems iterative methods must be used.

Iterative Linear Solver Methods

3

The direct methods discussed in Section 2.3.2 can be used to solve the linear problem described in Equation 2.25, but are computationally far too expensive to be applied to large problems. To solve the large problems observed in geomechanical simulations, iterative methods must be used.

This chapter discusses several iterative methods that can be used to solve linear systems. The methods discussed first, in Section 3.1, are methods that make use of simple iterations and are referred to as Basic Iterative Methods (BIM). In most cases, these methods are not used to solve the approached linear systems but instead are used to improve the performance of other methods. One such method, the Conjugate Gradient (CG) method, is discussed in Section 3.2. This is commonly the best method for solving linear systems with SPD matrices. The performance of this method can be improved by using a preconditioning method, for which the BIMs can be used. Although BIMs are very useful for this, more advanced methods can be used as a better preconditioner for the CG method. One such method is derived from Algebraic Multigrid methods, on which the main focus of this project is put. In Section 3.3, an extensive overview of this method of preconditioning is given.

3.1 Basic Iterative Methods

Basic iterative methods typically make use of iterations of the form

$$\mathbf{x}_{k+1} = G\mathbf{x}_k + \mathbf{c}, \tag{3.1}$$

where $\mathbf{x}_k$ should converge to the exact solution $\mathbf{x}$ to Equation 2.26. The matrix $G \in \mathbb{R}^{n\times n}$ is most commonly obtained by splitting the original matrix A into $A = M - N$, where M and N are such that every nonzero element in A is represented in the corresponding position in either one of the matrices. Using this splitting, G is given by

$$G = M^{-1}N = I - M^{-1}A. \tag{3.2}$$

The exact way in which A is split in M and N differs between several iterative methods. In most cases, these are described in terms of the matrices obtained from the splitting

$$A = D - E - F. \tag{3.3}$$

There, $D \in \mathbb{R}^{n\times n}$ is a diagonal matrix, containing the diagonal elements of A, $E \in \mathbb{R}^{n\times n}$ is a lower triangular matrix containing the elements of A located below the diagonal, multiplied by -1 and $F \in \mathbb{R}^{n\times n}$ is an upper triangular matrix containing the elements of A above the diagonal, multiplied by -1. The other element of the basic iteration,

the vector $\mathbf{c} \in \mathbb{R}^n$, is obtained from the right-hand side vector of the original system, as

$$\mathbf{c} = M^{-1}\mathbf{b}.$$

The first iteration in Equation 3.1 is performed with an initial vector $\mathbf{x}_0$. It is desirable for this initial guess to be as close as possible to the exact solution, as this means less iterations being required to converge to the solution. However, rarely enough is known about the exact solution to choose an initial vector close enough to the exact solution. In most cases where nothing is known about the solution, this initial vector is chosen randomly.

With $G = I - M^{-1}A$, Equation 3.1 can be seen as solving the system

$$M^{-1}A\mathbf{x} = M^{-1}\mathbf{b}, \tag{3.4}$$

which is exactly the original system multiplied to the left with M^{-1}. This is called a preconditioned system, where M is called a preconditioner. As the convergence of the basic iteration is in many cases very slow, this is the most prevalent use of the basic iterative methods.

3.1.1 Examples of Basic Iterative Methods

Using the splitting given by Equation 3.3 several iterative methods can be described. A very simple iteration is the Jacobi iteration, which uses $M = D$, giving $G = I - D^{-1}A$ and $\mathbf{c} = D^{-1}\mathbf{b}$. As D is a diagonal matrix, inversion of this matrix and multiplication by this matrix are not complicated in terms of computations. Using this, the iteration for the Jacobi method is obtained as

$$\mathbf{x}_{k+1} = D^{-1}(E + F)\mathbf{x}_k + D^{-1}\mathbf{b}. \tag{3.5}$$

The Jacobi iteration can be altered into a damped or relaxed variant, which uses a weighted average of the current iterant, $\mathbf{x}_k$ and the next iterant obtained from applying one step of the Jacobi iteration. This gives

$$\mathbf{x}_{k+1} = (1 - \omega)\mathbf{x}_k + \omega(D^{-1}(E + F)\mathbf{x}_k + D^{-1}\mathbf{b}). \tag{3.6}$$

This corresponds to an iteration of the form of Equation 3.1 with $G = I - \omega D^{-1}A$.

Another basic iteration method derived in a similar way as the Jacobi iteration is the Gauss-Seidel iteration. Like the Jacobi iteration, this uses an iteration of the form of Equation 3.1, but for this method $G = I - (D - E)^{-1}A$ is used. This gives the iteration

$$\mathbf{x}_{k+1} = (D - E)^{-1}F\mathbf{x}_k + (D - E)^{-1}\mathbf{b}. \tag{3.7}$$

Obtaining the iteration matrix for the Jacobi method was not complicated as there only the easily computed inverse of a diagonal matrix is required. It is more difficult to obtain the iteration matrix for the Gauss-Seidel method, as this requires the inverse of a lower triangular matrix $D - E$. In general, it is not possible to compute this inverse in a efficient way. Therefore, the next iterant is intead obtained by solving the system

$$(D - E)\mathbf{x}_{k+1} = F\mathbf{x}_k + \mathbf{b}.$$

As $D - E$ is lower triangular, this system can be easily solved in an efficient way, by making use of forward substitution **(Vuik, C. and Lahaye, D.J.P., p. 49 [38])**.

In the same way as this Gauss-Seidel method using forward substitution is described, a backward Gauss-Seidel method can be defined. For this each iteration step is done by solving

$$(D - F)\mathbf{x}_{k+1} = E\mathbf{x}_k + \mathbf{b}.$$

Now $D - F$ is upper triangular, which means that this system has to be solved using backward substitution. This corresponds to an iteration of the form of Equation 3.1, with $G = I-(D-F)^{-1}A$. Both Gauss-Seidel methods can be combined in a Symmetric Gauss-Seidel method, which uses an iteration that consists of a forward Gauss-Seidel iteration followed by a backward Gauss-Seidel iteration.

By introducing overrelaxation, the Gauss-Seidel method can be generalised. This is done by introducing a parameter ω, with which a different splitting of A of the form

$$\omega A = (D - \omega E) - (\omega F + (1 - \omega)D). \tag{3.8}$$

is described. Using this splitting the Successive OverRelexation (SOR) method is defined as

$$(D - \omega E)\mathbf{x}_{k+1} = (\omega F + (1 - \omega)D)\mathbf{x}_k + \omega\mathbf{b}. \tag{3.9}$$

In the same way as for the Gauss-Seidel iteration, this system can be solved by forward substitution. This iteration corresponds to an iteration as described by Equation 3.1 with

$$\begin{aligned} G_\omega &= (D - \omega E)^{-1}(\omega F + (1 - \omega)D) \\ \mathbf{c}_\omega &= \omega(D - \omega E)^{-1}\mathbf{b}. \end{aligned}$$

Similar to the Gauss-Seidel method, this method can be extended to a symmetric iteration, referred to as Symmetric SOR. This makes use of two iteration steps,

$$\begin{aligned} (D - \omega E)\mathbf{x}_{k+\frac{1}{2}} &= (\omega F + (1 - \omega)D)\mathbf{x}_k + \omega\mathbf{b} \\ (D - \omega F)\mathbf{x}_{k+1} &= (\omega E + (1 - \omega)D)\mathbf{x}_{k+\frac{1}{2}} + \omega\mathbf{b}. \end{aligned}$$

3.1.2 Preconditioning using BIMs

While the basic iteration methods described here can be applied directly to find an approximation of the solution to a linear system, this is typically not the best way to use them. The methods are only guaranteed to converge if the spectral radius is such that $\rho(G) < 1$, and even then the convergence is in many cases slow **(Saad, p. 115 [26])**.

Although basic iterative methods are often insufficient for solving large linear systems, they can be used to transform a linear system so that other more advanced methods can be applied more effectively. Using a preconditioning matrix M, the system is transformed, by multiplying it to the left, to

$$M^{-1}A\mathbf{x} = M^{-1}\mathbf{b}. \tag{3.10}$$

This transformation applied to linear systems with the goal of decreasing the condition number while making sure the solution to the system is the same is called preconditioning. This specific way of preconditioning, by multiplication with a preconditioning matrix to the left, can be referred to as left preconditioning, with similar methods of right and split preconditioning being defined through a multiplication to the right and on both sides respectively, given by

$$AM^{-1}\mathbf{u} = \mathbf{b}, \qquad \mathbf{x} = M^{-1}\mathbf{u}, \tag{3.11}$$

$$M_L^{-1}AM_R^{-1}\mathbf{u} = M_L^{-1}\mathbf{b}, \qquad \mathbf{x} = M_R^{-1}\mathbf{u}. \tag{3.12}$$

These two variants, however, are not used as commonly as the left preconditioning variant, and, unless specified, 'preconditioning' refers to the left variant.

As will be discussed in Section 3.2, the efficiency for many methods highly depends on the condition number of the coefficient matrix, given, for a nonsingular matrix by

$$\kappa(A) = \|A^{-1}\|\|A\|. \tag{3.13}$$

The norm used here in general is the Euclidian or L^2-norm, which gives the condition number in the 2-norm. This condition number in 2-norm can be described as

$$\kappa(A) = \frac{|\lambda_{\max}(A)|}{|\lambda_{\min}(A)|}. \tag{3.14}$$

Here, $\lambda_{\max}(A)$ and $\lambda_{\min}(A)$ describe the, in absolute value, largest and smallest eigenvalues of A, respectively. It can be easily seen that the condition number for any matrix satisfies $\kappa(A) \geq 1$. In general, the performance of iterative methods for solving a linear system is best when the condition number is close to 1 and decreases as the condition number increases. This means that, when applying a preconditioning method to a linear system, it is desired to use a preconditioning matrix M^{-1} such that $\kappa(M^{-1}A) \approx 1$.

To obtain a condition number close to 1, it is obvious that M should approximate A, as $M^{-1} = A^{-1}$ would lead to a condition number of 1. However, for effective application of a preconditioning matrix, there is the additional requirement of it being simple enough to obtain a solution to a linear system with the preconditioning matrix. This means that it should be simple to a system of the form

$$M\mathbf{x} = \mathbf{u}. \tag{3.15}$$

The matrices obtained for the BIMs all approximate the matrix A and it is simple to find a solution to linear problems using these matrices, making them great candidates for preconditioning methods. For the four methods described in Section 3.1.1, this gives preconditioning matrices

$$M_{\text{JAC}} = D, \tag{3.16}$$

$$M_{\text{GS}} = D - E, \tag{3.17}$$

$$M_{\text{SOR}} = \frac{1}{\omega}(D - \omega E), \tag{3.18}$$

$$M_{\text{SSOR}} = \frac{1}{\omega(2-\omega)}(D - \omega E)D^{-1}(D - \omega F). \tag{3.19}$$

3.1.3 Hybrid and ℓ_1- preconditioning matrices

Using the preconditioner obtained from the iterative methods described previously, especially Gauss-Seidel, two sets of new preconditioner matrices can be described. These new preconditioners are described by Baker et al. [3] **(p.5)** and are referred to as hybrid and ℓ_1- preconditioner matrices. These preconditioning methods are typically applied to iterative methods which make use of parallel computing methods.

For the hybrid preconditioner, a non-overlapping partition of the unknowns is considered. For this let $\Omega = \{1, \dots n\}$ be the set of indices of the unknowns, which is partitioned as

$$\begin{aligned} \Omega &= \bigcup_{k=1}^{N_p} \Omega_k, \\ \Omega_i \cap \Omega_j &= \varnothing, \qquad i \neq j. \end{aligned} \tag{3.20}$$

with each Ω_k representing subset of the unknowns. When using multiple processors, this partition can often be done such that each Ω_k represents the unknowns in the processor k and N_p is the total number of processors **(Baker et al., p. 10 [3])**. With this partition, the coefficient matrix A can be divided in blocks, along its rows and columns, resulting in N_p^2 blocks. These blocks, denoted by A_{kl}, are such that block A_{kl} has rows with indices in Ω_k and columns with indices in Ω_l. With this, we consider the systems obtained from the diagonal blocks of A, given by

$$A_{kk}\mathbf{x}_k = \mathbf{b}_k.$$

From this system, the Gauss-Seidel preconditioner of this matrix can be obtained as $M_k = D_k - E_k$, where D_k and E_k are the diagonal and lower triangular parts of A_{kk} respectively. The hybrid preconditioner matrix M_H is obtained from these block preconditioner matrices, as a block-diagonal matrix, with the block preconditioner matrices on the diagonal, given by

$$M_H = \begin{bmatrix} M_1 & O & \cdots & O \\ O & M_2 & \cdots & O \\ \vdots & \vdots & \ddots & \vdots \\ O & O & \cdots & M_{N_p} \end{bmatrix}. \tag{3.21}$$

Although the application of these hybrid preconditioners generally provides a good improvement in the convergence of many iterative methods, they do not guarantee convergence **(Baker et al., p. 13 [3])**. To fix this problem, ℓ_1-preconditioners are proposed, which attempt to fix the lack of guaranteed convergence of hybrid preconditioners by adding an appropriate diagonal matrix. For this let D^{ℓ_1} be the matrix with entries

$$[D^{\ell_1}]_{i,i} = \sum_{j=1}^{n} \big|[A]_{i,j}\big|, \tag{3.22}$$

so for the blocks A_{kk}, $D_k^{\ell_1}$ has entries

$$\big[D_k^{\ell_1}\big]_{i,i} = \sum_{j=1}^{n} \big|[A_{kk}]_{i,j}\big|. \tag{3.23}$$

Then the ℓ_1-preconditioner matrix M_{ℓ_1} is described as the block-diagonal matrix

$$M_{\ell_1} = M_H + D^{\ell_1} = \begin{bmatrix} M_1 + D_1^{\ell_1} & O & \cdots & O \\ O & M_2 + D_2^{\ell_1} & \cdots & O \\ \vdots & \vdots & \ddots & \vdots \\ O & O & \cdots & M_{N_p} + D_{N_p}^{\ell_1} \end{bmatrix}. \tag{3.24}$$

Transforming a linear system using the ℓ_1-preconditioner and applying an iterative method to this system does guarantee convergence **(Baker et al., p. 13 [3])**.

3.2 The Conjugate Gradient Method

As discussed before, convergence of methods based on basic iterations can be slow for large problems. The main downside of these methods is that these methods require a matrix-vector multiplication at every iteration step, which can be problematic in terms of computations when the system is of large size and many iterations are required.

To avoid these problems, it is required to define a method that can be expected to require fewer iterations than basic iterative methods. For this, techniques based on projections on a Krylov subspace obtained using the initial residual can be very helpful. These Krylov subspaces can be used to find the best approximation of the solution of a linear system that is of the form $p(A)\mathbf{b}$, where $p(A)$ approximates A^{-1}. Here $p(A)$ is a polynomial of A, of which the degree is the same as the degree of the Krylov subspace observed.

While a wide range of this type of method exists, the most effective method for linear systems with SPD matrices is the Conjugate Gradient method. This method is specifically developed for the application to this class of systems. Other methods that make use of similar principles, such as Generalised Minimal Residual (GMRES), Generalised Conjugate Residual (GCR) or BiConjugate Gradient Stabilised (BiCGSTAB), can also be used to solve the considered type of linear system, but are not covered further here. Saad [26] gives an extensive overview of these and other methods, along with an explanation of the Conjugate Gradient method.

The Conjugate Gradient method makes use of projections on the Krylov subspace $\mathcal{K}_m(\mathbf{r}_0, A)$ given by

$$\mathcal{K}_m(\mathbf{r}_0, A) = \text{Span}\{\mathbf{r}_0, A\mathbf{r}_0, A^2\mathbf{r}_0, \ldots, A^{m-1}\mathbf{r}_0\}, \tag{3.25}$$

where $\mathbf{r}_0$ is the initial residual, given by $\mathbf{r}_0 = \mathbf{b} - A\mathbf{x}_0$. The approximate solution obtained at each iteration of the Conjugate Gradient method is such that $\mathbf{x}_m \in \mathbf{x}_0 + \mathcal{K}_m$, the residual of which is orthogonal to the Krylov subspace $\mathcal{K}_m$.

The CG method is developed specifically for SPD linear systems. For solving linear systems of this type, CG is in many cases the best performing technique **(Saad, p. 196 [26])**. As discusses in Section 2.3, the linear problems encountered in the discussed geomechanical simulations can be expected to be SPD.

3.2.1 Conjugate Gradient Algorithm

The algorithm for the CG method is derived from the Lanczos method, a procedure for producing an orthogonal span of a Krylov subspace, like the one given in Equation 3.25. This procedure works particularly well for SPD matrices. As described by **(Saad, p. 196 [26])**, the Lanczos algorithm finds an approximate solution $\mathbf{x}_m$ as

$$\begin{aligned} \mathbf{x}_m &= \mathbf{x}_0 + V_m \mathbf{y}_m \\ \mathbf{y}_m &= T_m^{-1} \left(\beta \mathbf{e}_1\right). \end{aligned} \tag{3.26}$$

The matrix $V_m \in \mathbb{R}^{n \times m}$ is obtained as a matrix whose columns are the so-called Lanczos vectors $\mathbf{v}_j$, which are orthogonal vectors spanning the Krylov subspace $\mathcal{K}_m\left(\mathbf{r}_0, A\right)$. The matrix $T_m \in \mathbb{R}^{m \times m}$ is a tridiagonal matrix that contains the orthogonalisation coefficients obtained from the Lanczos algorithm. Lastly, the scalar β is obtained from the initial residual $\beta = \|\mathbf{r}_0\|_2$ and $\mathbf{e}_1 \in \mathbb{R}^m$ is the first unit vector of length m. A major advantage of this method is that the residual in an iteration step can be computed using only what is obtained at this step. For this, $\mathbf{r}_m$ is obtained as

$$\mathbf{r}_m = \beta_{m+1} \mathbf{e}_m^T \mathbf{y}_m \mathbf{v}_{m+1}. \tag{3.27}$$

From this it can be seen that the residuals at each step, $\{\mathbf{r}_1, \ldots \mathbf{r}_m\}$, are orthogonal to each other, as the Lanczos vectors are orthogonal to each other as well.

The final set of vectors, the auxiliary vectors $\mathbf{p}_i$, required to define the algorithm for the CG method is obtained from the columns of the matrix $P_m = V_m U_m^{-1}$, where U_m is obtained from the LU-decomposition of T_m. Now it is undesirable to actually compute this matrix P_m and fortunately this is not required. Instead, $\mathbf{p}_{j+1}$ can be obtained directly from the previous iterate as

$$\mathbf{p}_{j+1} = \mathbf{r}_{j+1} + \beta_j \mathbf{p}_j, \tag{3.28}$$

with $\beta_j = \frac{(\mathbf{r}_{j+1}, \mathbf{r}_{j+1})}{(\mathbf{r}_j, \mathbf{r}_j)}$. These auxiliary vectors can be used to describe the iteration of the approximate solution and residual

$$\begin{aligned} \mathbf{x}_{j+1} &= \mathbf{x}_j + \alpha_j \mathbf{p}_j \\ \mathbf{r}_{j+1} &= \mathbf{r}_j - \alpha_j A \mathbf{p}_j, \end{aligned}$$

with $\alpha_j = \frac{(\mathbf{r}_j, \mathbf{r}_j)}{(A\mathbf{p}_j, \mathbf{p}_j)}$.

With what is described above, an algorithm can be described for the CG method, which is provided in Algorithm 3.1. The stopping condition used is generally in the form of a relative tolerance on the residual, so the condition is satisfied if the residual is such that

$$\|\mathbf{r}_j\| < \epsilon \|\mathbf{r}_0\|, \tag{3.29}$$

for some chosen $\epsilon \in \mathbb{R}^+$. Alternatively, a maximum amount of iterations or absolute tolerance can be used either on its own or in combination with a relative tolerance condition. However, these stopping conditions cannot be assumed to give any information about the convergence of the method and are usually only used to limit cases where convergence is much slower than expected.

Algorithm 3.1: Conjugate Gradient Method **(Saad, p. 199 [26])**

Data: Coefficient Matrix A, Right-hand side Vector $\mathbf{b}$, Initial Approximate Solution $\mathbf{x}_0$
Result: Approximate Solution $\mathbf{x}$, Residual $\mathbf{r}$
Initialize $\mathbf{r}_0 = \mathbf{b} - A\mathbf{x}_0$
$\mathbf{p}_0 = \mathbf{r}_0$
$j = 0$
while *stopping condition not satisfied* **do**
  $\mathbf{q}_j = A\mathbf{p}_j$
  $\alpha_j = \frac{(\mathbf{r}_j, \mathbf{r}_j)}{(\mathbf{q}_j, \mathbf{p}_j)}$
  $\mathbf{x}_{j+1} = \mathbf{x}_j + \alpha_j \mathbf{p}_j$
  $\mathbf{r}_{j+1} = \mathbf{r}_j - \alpha_j \mathbf{q}_j$
  $\beta_j = \frac{(\mathbf{r}_{j+1}, \mathbf{r}_{j+1})}{(\mathbf{r}_j, \mathbf{r}_j)}$
  $\mathbf{p}_{j+1} = \mathbf{r}_{j+1} + \beta_j \mathbf{p}_j$
  $j \mathrel{+}= 1$
end
$\mathbf{x} = \mathbf{x}_j$
$\mathbf{r} = \mathbf{r}_j$

3.2.2 Preconditioned Conjugate Gradient Method

Saad [26] **(p.215)** describes an upper bound for the error at the m-th step of the CG algorithm by

$$\|\mathbf{x}_* - \mathbf{x}_m\|_A \leq 2 \left(\frac{\sqrt{\kappa(A)} - 1}{\sqrt{\kappa(A)} + 1} \right)^m \|\mathbf{x}_* - \mathbf{x}_0\|_A. \tag{3.30}$$

Here, it is seen that the rate of convergence, given by the fraction $\frac{\sqrt{\kappa(A)}-1}{\sqrt{\kappa(A)}+1}$, is close to 0 when the condition number $\kappa(A)$ is close to 1 and becomes close to 1 when $\kappa(A)$ is large. Since this upper bound for the error decreases faster when the rate of convergence is small, it is desirable to apply CG to a system with a condition number close to 1. As discussed in Section 3.1.2, preconditioning can be used to transform a linear system into a new system, for which the condition number of the observed matrix is closer to 1. Since CG is commonly applied to systems that are ill-conditioned, it is often advantageous to apply a preconditioning method before applying the CG iteration.

A major concern when applying preconditioning while solving a linear system using the CG method is the fact that, to ensure convergence of the CG method, it has to be applied to an SPD matrix. This means that preconditioning to the left and right can in general not be applied without problems, as $M^{-1}A$ and AM^{-1} are only guaranteed to be SPD if A and M^{-1} commute.

A possible way of solving the problem of the preconditioning matrix not being an SPD matrix is by using a split preconditioned system of the form

$$L^{-1}AL^{-T}\mathbf{u} = L^{-1}\mathbf{b}, \qquad x = L^{-T}\mathbf{u}. \tag{3.31}$$

There, L is obtained from the Cholesky factor decomposition of M. This ensures that

$L^{-1}AL^{-T}$ is SPD and thus guarantees convergence when applying the CG method to it. However, this method of preconditioning does still have the disadvantage of requiring to compute the Cholesky decomposition of the matrix M, which can be computationally costly.

As simply applying the CG method to a preconditioned system in most cases is not expected to work well, it is desired to develop an adapted method for the CG method that makes use of preconditioning. The main problem that arises when applying CG to a regular left preconditioned system is the fact that $M^{-1}A$ is not necessarily self-adjoint with the Euclidian inner product. To solve this, $M-$inner products can be used instead of the standard Euclidean inner product.

For an SPD matrix M, the $M-$inner product is defined as $(\mathbf{x}, \mathbf{y})_M = \mathbf{x}^T M\mathbf{y}$. This inner product has the property

$$(\mathbf{x}, \mathbf{y})_M = (M\mathbf{x}, \mathbf{y}) = (\mathbf{x}, M\mathbf{y}). \tag{3.32}$$

Using this property and the fact that A is self-adjoint with the standard Euclidean inner product, it is obtained that

$$(M^{-1}A\mathbf{x}, \mathbf{y})_M = (A\mathbf{x}, \mathbf{y}) = (\mathbf{x}, A\mathbf{y}) = (\mathbf{x}, M^{-1}A\mathbf{y})_M. \tag{3.33}$$

This means that $M^{-1}A$ is self-adjoint with the $M-$inner product, which can be used to develop an adapted CG method that can be applied to the preconditioned system.

By replacing the inner products used in the original CG method with $M-$inner products and introducing a new residual $\mathbf{z}_j = M^{-1}\mathbf{r}_j$, an alternative variant of the CG method can be described. This variant is equivalent to applying CG to a preconditioned system $M^{-1}A\mathbf{x} = M^{-1}\mathbf{b}$. This new variant works in the same way as the method described by Algorithm 3.1, but with the scalars α_j and β_j now being obtained from

$$\alpha_j = \frac{(\mathbf{z}_j, \mathbf{z}_j)_M}{(M^{-1}A\mathbf{p}_j, \mathbf{p}_j)_M}$$
$$\beta_j = \frac{(\mathbf{z}_{j+1}, \mathbf{z}_{j+1})_M}{(\mathbf{z}_j, \mathbf{z}_j)_M}$$

The computation of the $M-$inner product is in general less efficient than the computation of the regular Euclidean inner product. This makes it desirable to transform the M-inner products in the calculation of α_j and β_j into Euclidean inner products. For this it is obtained that

$$(\mathbf{z}_j, \mathbf{z}_j)_M = (M\mathbf{z}_j, \mathbf{z}_j) = (\mathbf{r}_j, \mathbf{z}_j),$$
$$\left(M^{-1}A\mathbf{p}_j, \mathbf{p}_j\right)_M = \left(MM^{-1}A\mathbf{p}_j, \mathbf{p}_j\right) = (A\mathbf{p}_j, \mathbf{p}_j).$$

With these, α_j and β_j can instead be computed using only regular Euclidian inner products as

$$\alpha_j = \frac{(\mathbf{r}_j, \mathbf{z}_j)}{(A\mathbf{p}_j, \mathbf{p}_j)}$$
$$\beta_j = \frac{(\mathbf{r}_{j+1}, \mathbf{z}_{j+1})}{(\mathbf{r}_j, \mathbf{z}_j))}.$$

Additionally, the update of the vector $\mathbf{p}_j$ is based on the residual of the preconditioned system, given by

$$\mathbf{p}_{j+1} = \mathbf{z}_{j+1} + \beta_j \mathbf{p}_j. \tag{3.34}$$

Lastly, the computation of the vector $\mathbf{z}_j = M^{-1}\mathbf{r}_j$ is required in each step of the algorithm. Algorithm 3.2 provides a complete overview of the newly obtained algorithm for this Preconditioned CG (PCG) method.

Algorithm 3.2: Preconditioned Conjugate Gradient Method **(Saad, p. 277 [26])**

Data: Coefficient Matrix A, Right-hand side Vector $\mathbf{b}$, Initial Approximate Solution $\mathbf{x}_0$
Result: Approximate Solution $\mathbf{x}$, Residual $\mathbf{r}$
Initialize $\mathbf{r}_0 = \mathbf{b} - A\mathbf{x}_0$
$\mathbf{p}_0 = \mathbf{r}_0$
$j = 0$
while *stopping condition not satisfied* **do**
  $\mathbf{q}_j = A\mathbf{p}_j$
  $\alpha_j = \frac{(\mathbf{r}_j, \mathbf{z}_j)}{(A\mathbf{p}_j, \mathbf{p}_j)}$
  $\mathbf{x}_{j+1} = \mathbf{x}_j + \alpha_j \mathbf{p}_j$
  $\mathbf{r}_{j+1} = \mathbf{r}_j - \alpha_j \mathbf{q}_j$
  $\mathbf{z}_j = M^{-1}\mathbf{r}_j$
  $\beta_j = \frac{(\mathbf{r}_{j+1}, \mathbf{z}_{j+1})}{(\mathbf{r}_j, \mathbf{z}_j))}$
  $\mathbf{p}_{j+1} = \mathbf{z}_{j+1} + \beta_j \mathbf{p}_j$
  $j \mathrel{+}= 1$
end
$\mathbf{x} = \mathbf{x}_j$
$\mathbf{r} = \mathbf{r}_j$

3.3 Algebraic Multigrid

Iterative methods such as the CG method can theoretically perform well for any system to which they can be applied. In practice, however, applying these methods to large linear systems can cause problems as these systems commonly have large condition numbers, resulting in slow convergence. Furthermore, applying basic operations to linear systems becomes computationally more expensive as the size of the linear system increases. This together means that larger systems require both more operations and the computations of those operations are more expensive, which makes the Krylov subspace-based methods less suitable for large systems.

For problems too large to be efficiently solved using a Krylov iteration, multigrid methods can be used to improve the performance of the Krylov iteration. The goal of a multigrid method is to transform a linear system into a smaller system, to which a linear solver method is applied. The solution obtained from this is then transformed back to the larger linear system. This is especially useful on systems with a high level of sparsity, as the decrease in size from applying a multigrid method to the system can be large.

Multigrid methods can be divided into two distinct groups. The first type of multigrid methods is referred to as Geometric Multigrid and requires extensive knowledge about the grid used to describe the linear system that is observed. To apply a geometric multigrid method, linear systems of different sizes are obtained from different discretisations of the observed problem. If different levels of discretisation are available, a geometric method can be very effective. However, without the different levels of discretisations, applying a geometric multigrid method requires defining the different levels in the setup of the solver, which can be expected to not be a viable option. For the problems observed here, it cannot be expected that one has extensive knowledge of the grid structure of the variables. This means that geometric multigrid is not expected to give useful results and will therefore not be considered further. Briggs et al. [6] does provide a more extensive overview of this topic.

The second type of multigrid methods is the Algebraic Multigrid (AMG) methods, which, like Geometric Multigrid, makes use of a transformation to a system on a smaller grid, but does not require knowledge about the actual underlying grids. AMG instead derives smaller systems using only the matrix that describes the linear system that is desired to be solved. This makes AMG much more suitable for application to large, sparse linear systems.

The framework in which AMG is approached here is the classical framework, as discussed by Ruge and Stüben [25]. Other approaches to AMG methods can be used, such as aggregation-based methods, but they are not covered here. The main difference between the aggregation-based and classical AMG methods is the way in which coarsening and prolongation are defined **(Stüben [33])**. Vanek et al. [37] gives an complete overview of these aggregation-based AMG methods.

When discussing AMG methods, the structure that defines the method is still commonly referred to as 'grids' or 'levels of grid'. These terms are used because they give an intuitive approach to the theoretical method. It is important to note that even though these terms are used, no actual grid system is required to be built. Instead, only

the matrices that define the linear systems are practically used to define the method used.

Here, several components required to define an AMG method are discussed. First, to give a general overview of how an AMG method can be applied, a simple way of applying a multigrid cycle, which only uses two levels, is discussed. Secondly, different ways to expand on this simple two-grid cycle with different cycling techniques are discussed. After this, the components of the multigrid cycle are discussed, those being coarsening methods, smoothing methods and techniques for prolongation and restriction. Lastly, it is discussed how this AMG method can be combined with the CG method, in the form of a preconditioner to the CG method, to make use of the advantages of both methods.

3.3.1 AMG Algorithm

3.3.1.1 Two-grid Cycle

The most simple way to apply a multigrid method is to use a two-grid cycle. A two-grid cycle uses a transformation between a fine grid, denoted by Ω_h and a coarse grid, indicated by Ω_H. Within the application of an AMG method, these grids are not actually defined, but for theoretical explanation, they are still considered. In practice, two grid levels are not always sufficient to solve the problem at hand, as the obtained coarse grid is, in most cases, still too large to effectively solve a linear system. However, most multigrid methods are described in a similar way, making the two-grid cycle very useful for a simple overview of the framework of a multigrid method.

To describe a two-grid cycle, two linear systems are required. The first system, on the fine grid, is the original linear system, as described in Equation 2.26 and denoted by h, which is desired to be solved. The second linear system, on the coarse grid, is given by

$$A_H \mathbf{x}^H = \mathbf{b}^H \tag{3.35}$$

This linear system is defined within the application of the multigrid method. This second linear system is smaller than the fine system. As the coarse linear system is of smaller size, it can be expected that it will be much less complicated to obtain an approximation of the solution for this linear system. This means that it is desirable to only solve linear systems on the coarse grid and use the result obtained from this to find an approximate solution on the fine grid.

Both linear systems are defined within a set-up phase, applied before applying the actual solver. In this set-up phase the structure of the coarse grid is defined along with a restriction operator

$$I_h^H : \Omega_h \to \Omega_H \tag{3.36}$$

and a prolongation operator

$$I_H^h : \Omega_H \to \Omega_h. \tag{3.37}$$

These operators perform a transformation between the two different grids. How they are obtained is described in Section 3.3.4.

Applying an AMG cycle like a two-grid cycle is done iteratively, with in each iteration a new iterant on the fine grid level $\mathbf{x}_{j+1}^h$ being obtained from the $\mathbf{x}_j^h$ the

previous grid level. This next iterant is obtained with a minimal amount of operations on the fine grid level. Instead, most computational work is done on the coarse grid level. To transform the iterant to the coarse grid level, a chosen amount μ of pre-smoothing operations is first applied, resulting in a smoothed iterant $\mathbf{u}_j^h$. How these smoothing operations are performed is discussed in depth in Section 3.3.2.

From this smoothed iterant, a residual on the fine grid, $\mathbf{r}_j^h = \mathbf{f}^h - A_h\mathbf{u}_j^h$ is obtained. Using the restriction operator, this residual is transformed to the coarse grid, as

$$\mathbf{r}_j^H = I_h^H \mathbf{r}_j^h. \tag{3.38}$$

This coarse residual is then used to obtain a correction $\boldsymbol{\delta}_j^H$ from solving

$$A_H \boldsymbol{\delta}_j^H = \mathbf{r}_j^H, \tag{3.39}$$

where A_H is the transformed coefficient matrix obtained through the Galerkin projection given by

$$A_H = I_h^H A_h I_H^h. \tag{3.40}$$

The system in Equation 3.39 is solved using some linear solver method. This system is desired to be of small enough size for a direct method to be applied effectively. In most cases, this is not possible with a hierarchy of only two levels, which means that either more levels must be used, or an iterative solver method has to be used to obtain the correction.

The correction $\boldsymbol{\delta}^H$ is used to correct the previously obtained iterant $\mathbf{u}_j^h$, by

$$\tilde{\mathbf{u}}_j^h = \mathbf{u}_j^h + I_H^h \boldsymbol{\delta}^H. \tag{3.41}$$

From this, a new iterant $\mathbf{x}_{j+1}^h$ is obtained by applying ν post-smoothing steps to $\tilde{\mathbf{u}}_j^h$.

Algorithm 3.3: Two-grid cycle AMG **(Saad, p. 442 [26])**

Data: AMG structure: {Fine Coefficient Matrix A_h, Coarse Coefficient Matrix A_H, Restriction Operation Matrix I_h^H, Prolongation Operation Matrix I_H^h, Smoothing Operators $\mathcal{S}, \mathcal{T}$}, right-hand side $\mathbf{b}^h$, Initial Approximate Solution $\mathbf{x}_0$

Result: Approximate Solution $\mathbf{x}$, Residual $\mathbf{r}$

while *stopping condition not satisfied* **do**

- $\mathbf{u}_j^h = \mathcal{S}^\mu(A_h, \mathbf{x}_j^h, \mathbf{b}^h)$
- $\mathbf{r}_j^H = I_h^H \mathbf{r}_j^h$
- Solve $A_H \boldsymbol{\delta}_j^H = \mathbf{r}_j^H$ for $\boldsymbol{\delta}_j^H$ using a chosen linear solver technique.
- $\tilde{\mathbf{u}}_j^h = \mathbf{u}_j^h + I_H^h \boldsymbol{\delta}^H$
- $\mathbf{x}_{j+1}^h = \mathcal{T}^\upsilon(A_h, \tilde{\mathbf{u}}_j^h, \mathbf{f}^h)$
- $j \mathrel{+}= 1$

end

$\mathbf{x} = \mathbf{x}_j$

$\mathbf{r} = \mathbf{r}_j$

As can be seen, the only linear system that must be solved is in Equation 3.39. This linear system is on the coarse grid and can be expected to be computationally much less costly than solving the original system on the fine grid.

Algorithm 3.3 gives an overview of the algorithm for this two-grid cycle. This algorithm consists of the iteration described above, which is repeated until a chosen stopping condition is satisfied. This stopping condition is typically based on the residual in an iteration j being relatively small compared to the residual of the initial solution, as previously described in Equation 3.29.

3.3.1.2 Cycling techniques

The two-grid cycle in practice is rarely useful, as in most cases where the application of AMG is useful, the first coarsened level is still too large to solve directly. This means that instead of the two linear systems, the original and the coarse system in Equation 3.35, used in the two-grid cycle, a series of linear systems is used. These are given by

$$A_k \mathbf{x}^k = \mathbf{b}^k, \tag{3.42}$$

where $k = 0, \dots K$ is the grid level of the linear system. There, $A_0\mathbf{x}^0 = \mathbf{b}^0$ corresponds to the original linear system as described by Equation 2.26. As there are now more than two grid levels, the two-grid cycle does not anymore suffice, and it is necessary to describe more advanced techniques for AMG cycles. Generally, these techniques are based on the same principles as described for two-grid cycles but are applied to AMG structures of more than two levels.

The most simple, and in many cases most useful, way of cycling through a bigger multigrid structure is using a V-cycle. This cycle is described in a way very similar to the two-grid cycle, but instead of solving after one coarsening operation, coarsening operations are repeated until the most coarse grid level of the multigrid structure is reached.

Figure 3.1a gives an overview of what a V-cycle looks like for a multigrid structure of four levels. At each step of the cycle, an operation is applied depending on the level of the step and the level of the next step. The highest level of the multigrid structure is the most coarse level and thus the smallest linear system. This is the only level on which the linear system is solved and a correction for the approximate solution is obtained. On the more fine levels, if the next step of the cycle is on a more coarse level, first pre-smoothing is applied μ times. After this, the restriction operator I_{k+1}^k is used to obtain a system at a higher level. On the other hand, if the next step of the cycle is on a finer level, prolongation, given by I_k^{k+1}, is applied first. After that, υ post-smoothing operations are applied. The operators for these operations are all obtained in the same way on each level, but the exact operators are not the same, as the linear systems on the different levels are not the same.

This V-cycle can be extended upon further by, instead of directly returning to the finest level after finding a correction to the approximate solution on the coarsest level, first returning to the coarsest level to improve the correction to the approximate solution. Cycles of this type are referred to as W cycles and examples of type of cycle are shown in Figure 3.1b. The number of times the cycle returns to the coarser level

before returning to a finer level is called the cycle index and is denoted by γ. This index is given such that the cycle returns to a finer grid level on the γ'th time of arriving at a certain grid level from lower levels. This means that performing a W-cycle with $\gamma = 1$ is the same as performing a V-cycle.

The advantage of a W-cycle is that a larger part of each iteration is performed on the coarse grid levels, on which computations are less costly. This can lead to fewer full iterations being required, at the trade-off of each single iteration becoming slightly more costly. However, it does not guarantee that fewer iterations are required and therefore can just lead to iterations being more costly without any advantages **(Saad [26])**.

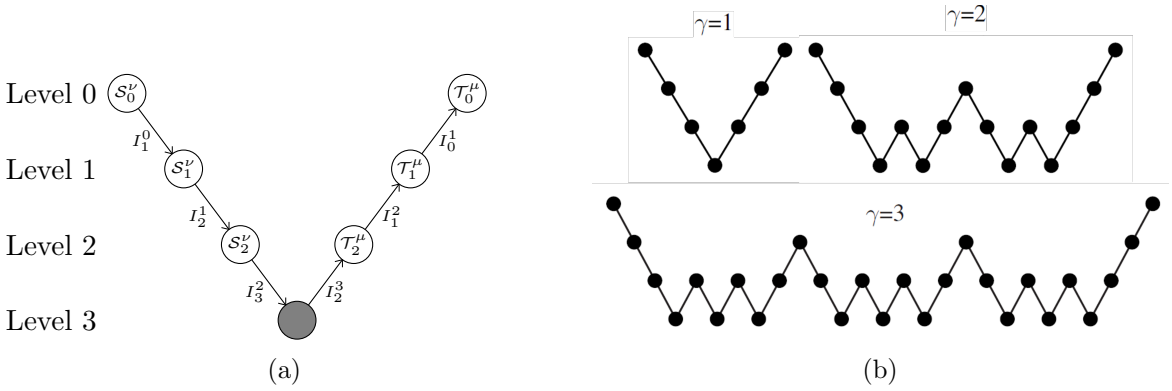


Figure 3.1: (a) Example of V-cycle with four levels. I_{k+1}^{k} and I_{k}^{k+1} denote the restriction and prolongation operators respectively used to map between levels. $\mathcal{S}_k^{\mu}$ and $\mathcal{T}_k^{\upsilon}$ denote the smoothing operations. On the most coarse level, marked grey, the system is solved and the coarse grid correction $\boldsymbol{\delta}_j$ is obtained. (b) Three Examples of W-cycles of four levels with different values of cycle index γ **(Saad, p. 445 [26])**. $\gamma = 1$ gives the same as the regular V-cycle.

3.3.2 Smoothing

The first aspect that must be defined to describe the multigrid cycle discussed above is the smoothing operation. This operation is described by Stüben et al. [32] **(p.16)** as

$$\mathcal{S}(A_h, \mathbf{x}^h, \mathbf{b}^h) = S_h\mathbf{x}^h + (I_h - S_h)A_h^{-1}\mathbf{b}^h. \tag{3.43}$$

This gives a transformation of the error $\mathbf{e} = \mathbf{x}_*^h - \mathbf{x}^h$, with $\mathbf{x}_*^h$ the exact solution to Equation 2.26, of

$$\bar{\mathbf{e}}^h = S_h\mathbf{e}^h, \tag{3.44}$$

where $\bar{\mathbf{e}}^h$ is the error obtained from the $\mathbf{x}^h$ after the application of the smoother, $\bar{\mathbf{e}}^h = \mathbf{x}_*^h - \mathcal{S}^{\mu}(A_h, \mathbf{x}^h, \mathbf{f}^h)$.

In practice, it is not viable to make use of the inverse of A_h within the smoother, and the matrix S_h is commonly of the form $S_h = I_h - M_h^{-1}A_h$. This results in a smoothing operation of the form

$$\mathcal{S}(A_h, \mathbf{x}^h, \mathbf{f}^h) = (I_h - M_h^{-1}A_h)\mathbf{x}^h + M_h^{-1}\mathbf{f}^h. \tag{3.45}$$

As can be seen, this is similar to the iteration described by Equation 3.1. This suggests that the BIM's discussed in Section 3.1 can be useful for the description of the smoothing operation.

These methods are commonly not useful for finding an accurate approximation of the solution to large linear systems because of their slow convergence of the error. However, these methods can give fast convergence in the first few iterations in which the method is applied. As the application of a multigrid correction can be very costly, it is desirable to apply the multigrid process only if it is absolutely necessary. This is the case if the error is algebraically smooth, which is defined as an error $\mathbf{e}^h$ such that $\|S_h\mathbf{e}^h\| \approx \|\mathbf{e}\|$. If this is the case, then the simple iteration of repetitively applying $\mathcal{S}$ will not perform well and the correction obtained from the application of an AMG method can give a significant improvement.

Now, a smoother which causes errors to be algebraically smooth after several iterations is not enough for it to be a well performing smoother. The smoother also has to be guaranteed to be efficient in reducing the error as long as $\|\mathbf{e}\|_2$ is large compared to $\|\mathbf{e}\|_1$ **(Clees, p. 42 [8])**. More specifically, a smoother S_h has to satisfy the smoothing property for any SPD matrix, which is the case if Equation 3.46 is satisfied for some $\sigma > 0$, independent of $\mathbf{e}$.

$$\|S_h\mathbf{e}\|_1^2 \leq \|\mathbf{e}\|_1^2 - \sigma\|\mathbf{e}\|_2^2 \tag{3.46}$$

With this smoothing property, it can be proven that several different types of matrices can be used as effective smoothers. Stüben et al. [32] provides the proof that several different preconditioning matrices, as described in Section 3.1.2, satisfy this smoothing property. First, it is shown in Theorem 3.1 **(Stüben et al., p. 29 [32])** that the Gauss-Seidel preconditioning matrix satisfies the smoothing property with $\sigma = \frac{1}{(1+\gamma_-)(1+\gamma_+)}$, where

$$\gamma_- = \max_i \left\{ \frac{1}{[\mathbf{w}]_i [A]_{i,i}} \sum_{j<i} [\mathbf{w}]_j \, |[A]_{i,j}| \right\}$$

$$\gamma_+ = \max_i \left\{ \frac{1}{[\mathbf{w}]_i [A]_{i,i}} \sum_{j>i} [\mathbf{w}]_j \, |[A]_{i,j}| \right\}$$

for any vector $\mathbf{w} > \mathbf{0}$. Then, it is shown in Theorem 3.2 **(Stüben et al., p. 30 [32])** that the relaxed Jacobi preconditioning matrix satisfies the smoothing property for $\sigma = \omega(2 - \omega\eta)$, where η is such that $\eta \geq \rho(D^{-1}A)$. The hybrid and ℓ_1-matrices discussed in Section 3.1.3 are also commonly used as smoothers in AMG methods. Baker et al. [3] discusses how these matrices are used as smoothing matrices and that they satisfy this smoothing property.

3.3.3 Coarsening

Before the AMG algorithm can be applied, it is required to establish an AMG structure. This requires, on each level of the AMG structure, a set of variables that describe the problem on that level. These sets of variables are denoted by $\Omega_0, \ldots \Omega_K$, where K is the total number of grid levels used. Ω_0 corresponds to the finest grid level and includes

all variables of the original system. This original set of variables is used to describe the sets of variables at higher levels **(Stüben et al., p. 16 [32])**.

To derive a system on a more coarse level, say $k+1$, the set of variables on grid level k is divided into two subsets. These subsets are such that the first, C_k, represents the variables of Ω_k that will be used at the coarser level $k+1$. This means that the variables on the coarser level are obtained as $\Omega_{k+1} = C_k$. The second subset, F_k, represents the variables only used on level k and not on level $k+1$. These two subsets can be referred to as the coarse variables (C_k) and the fine variables (F_k) of the level k **(Falgout [9])**. C_k and F_k are such that

$$C_k \cup F_k = \Omega_k$$
$$C_k \cap F_k = \emptyset.$$

These two subsets will later, in Section 3.3.4, be used to describe the prolongation and restriction operators, which are used to map between different grid levels. As the variables on each grid level are obtained from a subset of the variables on more fine level, it is obtained that

$$\Omega_K \subset \Omega_{K-1} \subset \cdots \subset \Omega_1 \subset \Omega_0.$$

The number of grid levels is usually determined by the coarsening process. One method of determining the number of grid levels is by deriving higher grid levels as long as the unknowns can be split into sets C_k and F_k in a way that satisfies the constraints described by the coarsening method.

Only once this split in coarse and fine variables can not be made without conflicting with the constraints of the coarsening method, no new levels are derived anymore. Then, the last derived grid level is used as the coarsest level. Alternatively, it is possible to not derive more coarse levels if the last derived level is of such a small size that a linear solver can be applied effectively. In practice, this is in most cases the best method to determine the required amount of grid levels, as this way it is made sure that not more levels are created than necessary.

To give an accurate description of the coarsening methods, it is necessary to first introduce some concepts concerning what is by Stüben et al. [32] **(p.64)** referred to as connections of variables. For this, Definition 3.1 introduces the concept of connected variables. Definition 3.2 extends upon Definition 3.1 by introducing a special kind of connectivity, as described by Falgout [9]. In this definition of strong connectivity, any positive connection between variables is considered weak. This specific way of defining a strong connection is sometimes referred to as *strong negative connectivity* **(Stüben et al., p. 64 [32])**, but since it is sufficient here to consider positive connections as weak, this distinction is not made here.

Definition 3.1. A variable of index i is **connected** to another variable of index j if $[A]_{ij} \neq 0$, with its **set of couplings** represented by

$$N_i = \{j \in \Omega : [A]_{ij} \neq 0\}$$

Definition 3.2. A variable of index i is **strongly connected** (or **strongly negative connected**) to another variable of index j if, for some chosen threshold $0 < \theta_{ecg} < 1$,

$$-[A]_{ij} \geq \theta_{ecg} \max_{k \neq i, [A]_{ik} < 0} [A]_{ik}$$

With this, the **set of strong connections** of a variable i is given by

$$S_i = \{j \in N_i : i \text{ strongly coupled to } j\}$$

and similarly, the **set of strong transpose connections**

$$S_i^T = \{j \in \Omega : i \in S_j\}.$$

Theoretically, there are no requirements for the way in which the variables are split between the sets C_k and F_k and thus any desired method can be used. However, it is advantageous for the convergence of the AMG method to make use of a separation that is as uniform as possible. Moreover, the performance of the AMG method is improved if any variable in F_k is surrounded by variables in C_k. This means that every variable in F_k is connected to only one, or a minimum number, of variables in C_k.

Here, two different methods of applying coarsening are discussed, Standard and Aggressive Coarsening. After this, the concept of complexities is discussed, which can give useful information on how many elements an AMG structure has.

3.3.3.1 Standard Coarsening

Standard Coarsening is based on direct couplings between variables in F_k and C_k. When using standard coarsening, all variables that are chosen in F_k must be strongly connected to a minimum number of variables in C_k **(Stüben et al., p. 64 [32])**.

Standard coarsening, as well as other coarsening methods described later, can be described as a simple iteration. For this, initially, all variables are added to a set of unclassified variables U. From this unclassified set, at each iteration, one variable is chosen and added to the set C and all the variables to which it is strongly connected are added to the set F. This is repeated until no variables are left in the unclassified set, after which C is used as the set C_k and F is used as the set F_k.

This simple iteration cannot be used on its own, as this can result in a random distribution of fine and coarse variables where it is desired to have a uniform distribution of coarse variables. To ensure that there is order in the way in which variables are chosen to be added to C, Stüben et al. [32] **(p.65)** introduces the concept of measure of importance υ_i, as

$$\upsilon_i = |S_i^T \cup U| + 2|S_i^T \cap F|. \tag{3.47}$$

This means that the choice of the next unknown that is added to C is based on how many of its strong connections are with unclassified variables and how many of its connections are with variables that are already classified in F. There, more importance is given to the strong connections with the variables in F_k. With this measure of importance, the iteration described above is used, with the coarse variables chosen based on the measure of importance.

If this is applied to a matrix that is symmetric, any variable that has any connections to other variables is classified as either coarse or fine. If there are variables that do not have any connections to other variables, they are classified as fine. A variable can only have no connections to other variables if the only nonzero element in the corresponding row of the matrix is on the diagonal. If this is the case, the solution in this variable can be easily obtained directly. These types of variables do not have to be considered in the AMG structure.

Figure 3.2 gives an example of what this method of coarsening would look like on a five-by-five grid. Note here that despite an actual grid being shown in this example, in practice such a grid is never used when deriving an AMG structure. The graph structure shown here corresponds to a matrix of size 25, with each unknown corresponding to a node in the graph and each edge between nodes corresponding to a strong connection between two unknowns. To describe this grid, let $x_{i,j}$ be the node in the i'th row starting from the bottom and j'th column starting from the left, so $x_{1,1}$ is the bottom left node.

In the first figure, Figure 3.2a, none of the unknowns have been processed yet and in all nodes the measure of importance is presented. In the first step of the iteration that divides the unknowns into C_k and F_k, the unknown with the highest measure of importance, or one of the unknowns with the highest measure of importance if there is no unique unknown with the highest measure of importance, is chosen to be in C_k. This is shown in Figure 3.2b, where $x_{2,2}$ is chosen to be added to C_k. All unknowns to which it is strongly connected are then added to F_k. After this, the measures of importance are updated according to Equation 3.47 as shown in Figure 3.2c. This is repeated until every unknown is either added to C_k or to F_k as shown by Figure 3.2d-j.

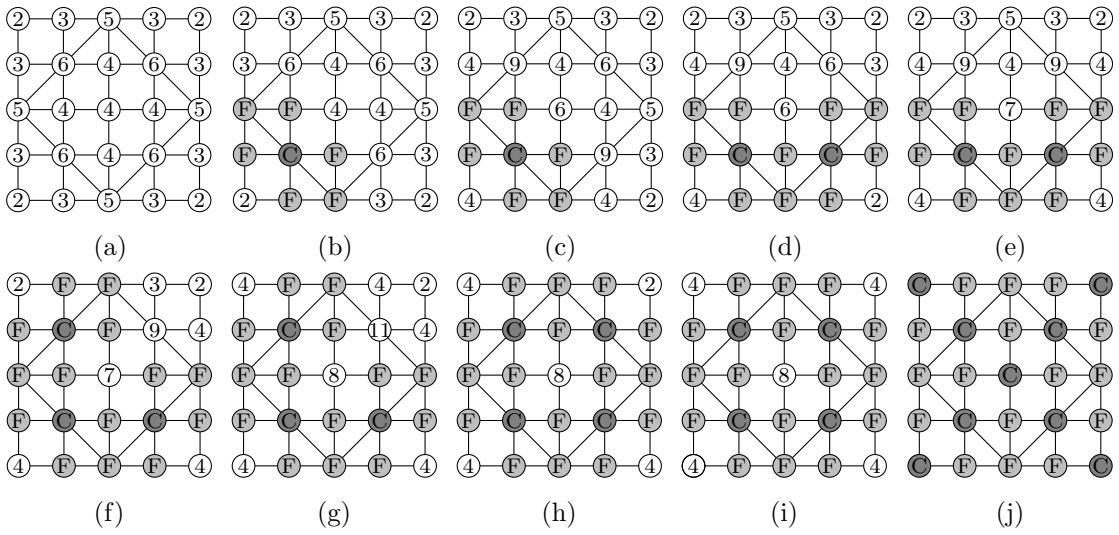


Figure 3.2: Standard coarsening on an example of a 5x5 grid. At each step, the variable i with the largest value of v_i is chosen to be added to C_k, with all variables strongly connected to i being added to F_k ((b),(d),(f),(h) and (j)). After adding the chosen variables to the subsets, v_i is updated for all variables that are not chosen to be added to one of the sets ((c),(e),(g) and (i)). (j) Here the last step is shown as one step, while this is separate iterations in practice.

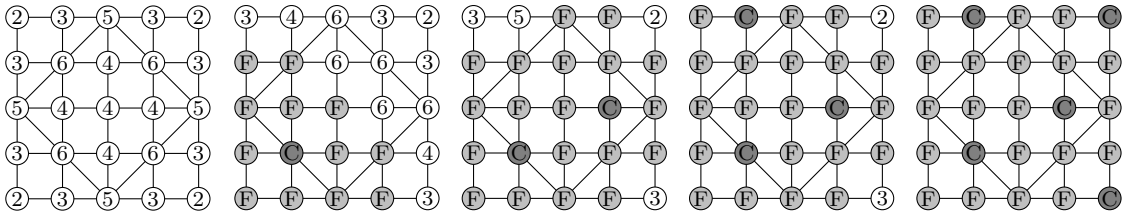
3.3.3.2 Aggressive Coarsening

Aggressive Coarsening extends on the idea of strong connections between variables in C_k and F_k, but where standard coarsening uses direct connections, aggressive coarsening instead uses paths of strong connections. For this, Ruge and Stüben [25] introduces the concept of long-range strong connections according to Definition 3.3.

Definition 3.3. A variable i_0 is **strongly connected along a path of length ℓ** to i_ℓ if there exists a sequence of variables $i_0, i_1, \dots i_\ell$ such that $i_k \in S_{i_{k-1}}$ for $k = 1, 2, \dots \ell$. In extension, i_0 is **strongly connected with respect to** (p, ℓ) to i_ℓ if there are at least p paths of maximum length ℓ along which i_0 is strongly connected with i_ℓ.

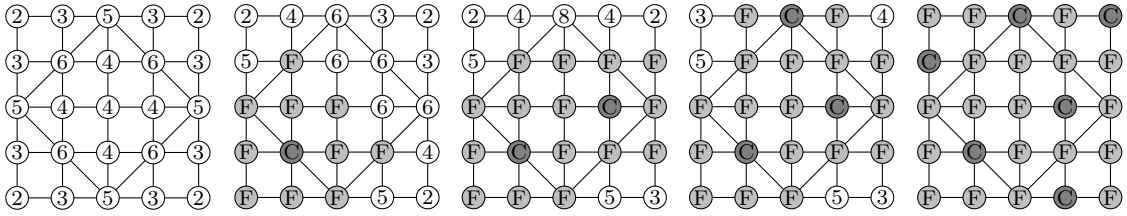
Aggressive coarsening is done exactly the same way as standard coarsening, but instead of the previously used strong connections, these strong connections with respect to (p, ℓ) are used. This means that aggressive coarsening can be seen as a generalisation of standard coarsening, as the strong connections used in the method for standard coarsening described above are the same as strong connections with respect to $(1, 1)$.

Aggressive coarsening is applied with the values specifically chosen for p and ℓ. Practically, only small values of p and ℓ are useful, since large values would lead to too many variables being classified as fine **(Stüben et al., p. 67 [32])**. The two aggressive coarsening methods that turn out to be the most useful are known as A1-coarsening and A2-coarsening. A1-coarsening makes use of connections with respect to $(p = 1, \ell = 2)$, similarly, A2-coarsening uses connections with respect to $(p = 2, \ell = 2)$. Furthermore, aggressive coarsening is commonly only advantageous at the finest grid level. On coarser grid levels, standard coarsening can be expected to be required.



(a) (b) (c) (d) (e)

Figure 3.3: Example of A1-coarsening on the same example as used in Figure 3.2. For less extensive representation, both the addition of unknowns to C_k and F_k and the update of the measure of importance are done at the same time here.



(a) (b) (c) (d) (e)

Figure 3.4: Example of A2-coarsening on the same example as used in Figure 3.2.

Figure 3.3 shows how A1-coarsening is applied to the example of 25 unknowns used in Figure 3.2. As the grid is initially the same as in Figure 3.2a, the same node, $x_{2,2}$, is added to C_k. However, with this aggressive coarsening method, every node connected by a path of length two or less to $x_{2,2}$ is added to F_k in Figure 3.3b. This means that every unknown that either is strongly connected to $x_{2,2}$ or is strongly connected to a variable that is strongly connected to $x_{2,2}$ is added to F_k. Then in Figure 3.3c, the same is applied and now $x_{3,4}$ is added to C_k and all unknowns connected by a path of less than 2 are added to F_k. In Figure 3.3d-e, the same method is repeated until all unknowns are added to C_k or F_k.

In a similar way, Figure 3.4 gives an overview of the application of A2-coarsening. Again, the process of choosing unknowns for the coarser grid level is started by adding $x_{2,2}$ to C_k. However, now only the unknowns that have at least two paths to $x_{2,2}$ of a length of at most two are added to F_k. As shown in Figure 3.4b, this leads to a difference for two unknowns compared to Figure 3.3b, $x_{4,1}$ and $x_{1,4}$. These two unknowns are only connected to $x_{2,2}$ by one path of length two and thus were added to F_k for the and therefore were added to F_k when using A1-coarsening, but are not when using A2-coarsening. In Figure 3.4c-e, the process is repeated again until all unknowns are added to either C_k or F_k.

3.3.3.3 AMG complexity

With different coarsening methods, multilevel grid structures are obtained in different ways. With these different grid structures, it is desirable to compare the memory requirements of the different methods. For this, the concept of multigrid complexities is introduced. These complexities describe the ratio between the size of the original problem and the AMG problem obtained by coarsening. The advantage of these complexities is that they can give a simple estimate of the size of the AMG structure. This can be used to estimate the total required memory for the different matrices of the AMG structure. Furthermore, these complexities can be useful in comparing AMG structures obtained in various methods.

Here, two different AMG complexities will be used. The first type of complexity involves the total number of variables in the AMG structure and is called grid complexity. This describes the ratio between the total number of variables in all levels and the number of variables in the original linear problem. With N_k denoting the number of variables of Ω_k, the grid complexity is obtained as

$$\omega_g = \frac{\sum_{k=0}^{K} N_k}{N_0}. \tag{3.48}$$

In a similar way, the second type of complexity, operator complexity, is described. This describes the ratio between the total number of non-zero entries in the matrices obtained on all levels of the multi-grid structure and the amount of nonzero entries in the original matrix. To describe the operator complexity, let nnz_k denote the number of nonzero elements of the matrix A_k obtained on the grid level Ω_k. Then the operator complexity is obtained as

$$\omega_a = \frac{\sum_{k=0}^{K} \text{nnz}_k}{\text{nnz}_0}. \tag{3.49}$$

Theoretically, any value such that $\omega > 1$ is possible for these complexities. In practice, the range of these complexities is commonly more limited. For example, a coarsening technique that allows a newly created grid to be larger than half the size of the previous level is generally not very useful. This means that the grid complexity can always be expected to be less than 2. Furthermore, it is practically not possible for the grid complexity to be too close to 1, as this would mean the size of the coarse levels is very small compared to the original problem. This means that in most cases the grid complexity is at least greater than 1.1 **(Cleary et al. [7])**. Similarly, the operator complexity can be expected to be such that $1.2 < \omega_a < 4$.

3.3.4 Prolongation and Restriction

The coarsening methods described above determine which variables are to be used at the coarse level, but do not give a method to describe linear systems at the coarser grid levels. Only the linear system at the most fine level is actually defined and this must be used to define the linear systems at the newly derived grid levels. When defining these linear systems on higher grid levels, it is important that the newly defined system is similar to the larger linear system from which it is obtained.

As described in Section 3.3.1.1, the best way to obtain a linear system on a more coarse grid is to use the Galerkin projection. To derive a linear system on the grid level $k+1$ from a linear system on grid level k, it is required to describe the operator mapping between these levels. This done using the restriction operator

$$I_k^{k+1} : \Omega_k \to \Omega_{k+1} \tag{3.50}$$

and the prolongation operator

$$I_{k+1}^{k} : \Omega_{k+1} \to \Omega_k, \tag{3.51}$$

which are such that $\left(I_k^{k+1}\right)^T = I_{k+1}^k$. Using these operators, the matrix and right-hand side vector on the higher grid level are defined as

$$\begin{aligned} A_{k+1} &= I_k^{k+1} A_k I_{k+1}^k \\ \mathbf{b}^{k+1} &= I_k^{k+1} \mathbf{b}^k. \end{aligned} \tag{3.52}$$

This means that to describe the linear systems on higher grid levels, only these prolongation and restriction operators are required. These operators can be defined in several ways. Here, only the most classically used method for obtaining the prolongation and restriction operator, called unknown-based prolongation, is described **(Saad [26])**. Other methods of defining this operator can alternatively be used, such as prolongation operators obtained using bootstrap cycles, as described by Brandt et al. [5].

To describe these operators, the notion of smooth errors discussed in Section 3.3.2 is used. There it was derived that errors within a multigrid cycle are smooth and thus

$$\|S_k \mathbf{e}^k\|_{A_k} \approx \|\mathbf{e}^k\|_{A_k} \tag{3.53}$$

and smoothers satisfy the smoothing property. As discussed by Saad [26] **(p.456)**, this leads to the error being such that $(A_k \mathbf{e}^k, \mathbf{e}^k) \approx 0$ and thus $A_k \mathbf{e}^k \approx 0$. With this, the

individual components of the error can be approached and it is obtained that for a unknown i on level k,

$$[A_k]_{ii}[\mathbf{e}^k]_i \approx -\sum_{j \in N_i} [A_k]_{ij}[\mathbf{e}^k]_j. \tag{3.54}$$

This description of the error in unknown i is used to determine the restriction operator, by rewriting this equation such that on the right-hand side there are only unknowns which are chosen for the higher grid level Ω_{k+1}. For this, the sum on the right-hand side can be split in grid points chosen for Ω_{k+1} and grid points that are not. To achieve this, let

$$C_i = N_i \cup C_k \tag{3.55}$$
$$F_i = S_i \cup F_k \tag{3.56}$$
$$W_i = (N_i / S_i) \cup F_k, \tag{3.57}$$

so the first set, C_i, are all unknowns connected to i that are chosen for the higher grid level, both weakly and strongly connected. The next set, F_i, are the unknowns that are strongly connected, but not chosen for the higher grid level. The last set, W_i, are all weakly connected unknowns that are chosen for the higher grid level. Then the sum in Equation 3.54 can be split into these three groups as

$$[A_k]_{ii}[\mathbf{e}^k]_i \approx -\sum_{j \in C_i} [A_k]_{ij}[\mathbf{e}^k]_j - \sum_{j \in F_i} [A_k]_{ij}[\mathbf{e}^k]_j - \sum_{j \in W_i} [A_k]_{ij}[\mathbf{e}^k]_j. \tag{3.58}$$

As it is required to only have unknowns that are in C_i on the right-hand side, these last two sums have to be eliminated. As the last sum, over the weak connections is assumed to be small, it is common to add it to the diagonal term, giving

$$\left([A_k]_{ii} + \sum_{j \in W_i} [A_k]_{ij}\right)[\mathbf{e}_k]_i \approx -\sum_{j \in C_i} [A_k]_{ij}[\mathbf{e}^k]_j - \sum_{j \in F_i} [A_k]_{ij}[\mathbf{e}^k]_j - [\mathbf{e}^k]_j. \tag{3.59}$$

After this, it is necessary to eliminated the terms in F_i. To achieve this, let j be a variable in F_i, then for this variable, the same expression as for variable i can be obtained. According to Saad [26] **(p.459)**, this expression for variable j can be approximated by

$$[A_k]_{jj}[\mathbf{e}^k]_j \approx -\sum_{l \in C_i} [A_k]_{jl}[\mathbf{e}^k]_l. \tag{3.60}$$

Substituting this back in Equation 3.58 gives the desired result of an expression for $[\mathbf{e}^k]_i$ using only variables in C_i. This is given by

$$[\mathbf{e}^k]_i = \sum_{j \in C_i} w_{ij}[\mathbf{e}^k]_j, \tag{3.61}$$

with

$$w_{ij} = -\frac{[A_k]_{ij} + \sum_{l \in F_i^s} \frac{[A_k]_{il}[A_k]_{lj}}{\sum_{p \in C_i}[A_k]_{lp}}}{[A_k]_{ii} + \sum_{l \in F_i^w}[A_k]_{il}}.$$

This expression is then used to obtain the prolongation operator as

$$\left[I_{k+1}^{k}\right]_{ij} = \begin{cases} 1 & i \in V_C, i = j \\ 0 & i \in V_C, i \neq j \\ w_{ij} & i \notin V_C, j \in V_C \\ 0 & i \notin V_C, j \notin V_C \end{cases}. \tag{3.62}$$

The restriction operator is then directly obtained as $I_k^{k+1} = \left(I_{k+1}^k\right)^T$.

3.3.5 AMG as Preconditioner

While the AMG method can be used by applying several multigrid cycles, as explained in Section 3.3.1, it is often more useful in combination with a Krylov method. This combines the robustness of the Krylov methods with the efficiency of reducing all components of the error of the AMG method. This is commonly referred to as using the AMG method as a preconditioner or using the Krylov method to accelerate the AMG method **(Stüben et al., p. 74 [32])**. Here, the former description is used.

Applying an AMG method as a preconditioner is done in a way very similar to how it was done using other preconditioning methods. In a preconditioned Krylov method, in each iteration a preconditioning operation of the form $\mathbf{z}_j = M^{-1}\mathbf{r}_j$ is performed, where M^{-1} gives a preconditioning operation. For the more simple methods discussed above, this operation is a single matrix. When using an AMG method, instead, an iteration of the AMG cycle is applied to $\mathbf{r}_j$ **(Oosterlee and Washio, p. 89 [22])**. From this a corrected residual is obtained, which is then used in the iteration of the Krylov method.

Although the practical application of an AMG method is typically performed as described here, the use of AMG as a preconditioner suggests that an iteration of the AMG method can be written as a single matrix. This is indeed the case, though it is inconvenient to describe a full multigrid cycle as a single matrix. The two grid cycle, as given in Algorithm 3.3, can be described by a single matrix **(Saad, p. 442 [26])**, as

$$M^{-1} = T_h^{\upsilon}\left(I - I_H^h A_H^{-1} I_h^H A_h\right) S_h^{\mu}. \tag{3.63}$$

There, S_h and T_h are the smoothing operators and I_H^h and I_h^H the prolongation and restriction operators, as they were used above in Algorithm 3.3. A preconditioning matrix for an AMG method with more levels can be described using this matrix, by repeatedly replacing the matrix at the coarser level, A_H.

Using a multigrid hierarchy with levels $\{0, \dots K\}$, with the 0 being the most fine level that corresponds to the original linear system and K being the most coarse level, this leads to a preconditioning matrix of the form

$$M^{-1} = T_0^{\upsilon_0}\left(I - \left(I_1^0 T_1^{\upsilon_1} I_2^1 \cdots T_{K-1}^{\upsilon_{K-1}} I_K^{K-1}\right) A_K^{-1} \left(I_{K-1}^K S_{K-1}^{\mu_{K-1}} I_{K-2}^{K-1} \cdots S_1^{\mu_1} I_0^1\right) A_h\right) S_0^{\mu_0}. \tag{3.64}$$

When applying an AMG method here, this will be done as a preconditioner. The Krylov method will in most cases, apart from a small investigation into other Krylov methods, be the CG method.

4 Solver Techniques

In the previous chapter an overview of iterative linear solver methods has been given. Using the Visage simulator discussed in Chapter 2, these iterative methods can be applied to linear problems obtained from geomechanical simulations. Through experiments using various linear solver methods on several linear systems, a comparison between the methods can be made. However, before discussing the results of these experiments, it must be established what solvers are used and how these solvers are applied.

This chapter gives an overview of how experiments, of which the results are presented in Chapter 5, are performed. First, the different linear solver methods that were approached in these experiments are discussed in Section 4.1, After this, an overview of parallel computing, which is made use of in the experiments, is given in Section 4.2. Next, a description of the environment in which the experiments are performed is given in Section 4.3. Lastly, an overview of how the methods are compared based on the results obtained in the experiments is given in Section 4.4.

4.1 Approached Linear Solver Methods

As discussed in Section 2.3, the simulations of geomechanical processes in Visage can be performed with both direct and iterative linear solver methods. However, direct methods are typically only suitable for small problems and the focus here is on large problems. This means that when comparing methods, only iterative methods have to be considered.

The iterative solver used in the Visage simulator is the PCG method using different types of preconditioners. This CG method is performed in the way described in Section 3.2. Before this research, two preconditioning methods were used to improve the CG method, a Jacobi preconditioner and a deflation-based preconditioner. The Jacobi preconditioner is the preconditioner based on the diagonal of the original matrix, as described in Section 3.1.2. This Jacobi preconditioner gives a decrease in the condition number of the matrix in the observed linear system, but this improvement is not always big enough.

A bigger decrease of the condition number can be obtained by removing certain extreme eigenvalues from the spectrum of the matrix. This is done using deflation, which is a sophisticated method for preconditioning linear systems which are solved using a Krylov method. The theory behind deflation is not covered here, but is extensively described by Jönsthövel et al. [17] and Jönsthövel [16].

The stopping criterion used in this application of the CG method is based on decreasing the relative residual, as described by Equation 3.29. Typically, the chosen value for the tolerance ϵ is 10^{-8}.

Although good results are obtained using these methods, the vast majority of runtime during simulations is spent on the linear solver. This means that there is potential for improvement in the linear solver. Therefore, different preconditioning methods were proposed, all based on AMG methods. The various methods consist of different implementations of the AMG method, provided through three software distributions.

All these linear solver methods make use of the CG method preconditioned with an AMG method. Because of differences in implementation between the software distributions, there will be differences between the methods, especially in the AMG preconditioner. All these linear solvers make use of the CG method, using different preconditioning methods. To distinguish between different methods, the methods will in most cases be referred to by the software distribution from which they are obtained. For example, the "SAMG method" or "SAMG" refers to the CG method preconditioned using the AMG method obtained from the SAMG software distribution.

4.1.1 SAMG

The first software library that is approached here is the SAMG (Algebraic Multigrid Methods for Systems) library. SAMG is a library of subroutines that are used to efficiently compute solutions to large linear systems using multigrid methods **(Fraunhofer SCAI [11])**. The library is developed by the Fraunhofer SCAI.

The specific method from SAMG that is used here is SAMGp Release 2023.3 (January 26, 2024).The SAMGp library is a version of the SAMG library that can make use of parallel computing.

4.1.2 hypre

The second software library observed is the hypre library **(Lawrence Livermore National Security [19])**. This is an open-source library with a wide range of parallel preconditioners that is developed by the Lawrance Livermore National Laboratory.

The preconditioner in hypre that is used here is the BoomerAMG preconditioner. This method uses an Algebraic Multigrid method, as described above. When referring to the use of "hypre", "the hypre method", or "the hypre library", this always refers to using the CG method with the BoomerAMG preconditioner, as provided by the hypre library.

A major advantage of the hypre library is the potential use of an efficient parallel implementation and the possibility to use GPU acceleration. Although GPU acceleration is not investigated here, it can potentially be used to gain further improvements in the future.

The specific hypre release that has been used in these experiments is the hypre Release 2.31.0 (February 14, 2024).

4.1.3 PETSc

The last software library that is used is the PETSc (Portable, Extensible Toolkit for Scientific Computation) library **(Balay et al.** [4]**)**. Like hypre, PETSc has a wider range of applications than just the use of AMG preconditioners. Also, like hypre, PETSc is available from an open source distribution. The PETSc release used here is PETSc version 3.21.1 (May 27, 2024). The PETSc library is developed by UChicago Argonne, LLC and the PETSc Development Team.

The preconditioner described in PETSc that is investigated here is the PCGAMG preconditioner. Like the preconditioners considered for the other two libraries, this is a preconditioner based on the AMG methods discussed above. Again, "PETSc" or "the PETSc method" typically refers to the CG method preconditioned with PCGAMG as it is implemented in the PETSc library.

Although PETSc has a wider range of possible applications than the other two investigated libraries, the description of the AMG preconditioner is not as extensive as that for the other two libraries. Because of this, some difficulties were encountered in the implementation of PETSc. This means that the results obtained with the preconditioner provided through PETSc are not as extensive as those obtained for the other methods.

While PETSc did not turn out to be as useful as initially hoped for using AMG-based preconditioners, one of the other methods provided through PETSc proved to be very useful. The SLEPc library, which is an extension of PETSc, turned out to be very useful for obtaining estimates of the eigenvalues of the observed matrices, which in turn were used to determine the condition numbers of the matrices. The version of SLEPc that is used here is SLEPc version 3.21 (April 27, 2024) **(Roman et al.** [24]**)**.

4.2 Parallel Computing

For methods that are used to solve large problems, such as the linear solvers here, great advantages can be obtained by using parallel computing methods **(Rauber and Rünger** [23]**)**. The methods that are covered here all have the possibility to make use of parallel implementations and to improve the performance of the methods, an investigation is conducted in how parallel programming is used best for these methods.

Parallel computing has two major advantages. The first is that part of the computational work can be divided over multiple machines. By dividing the work, it can be performed at the same time, reducing the total runtime required to solve the problem. Secondly, by dividing the work over multiple machines, the memory of all machines is used. This leads to more memory being available in total. For large problems, this can be essential, as typically the memory of a single machine is not enough to solve these problems.

Parallel computing is here, for the most part, done by distributing the work over multiple MPI processes. To do this effectively, the problem needs to be divided over MPI processes in a correct manner. This is done by making use of a good domain partition, which is potentially improved by reordering certain parts of the matrices. Section 4.2.1 discusses how this domain partition is performed and how it can be

improved.

If a well-working parallel method has been established, metrics to determine how well this method works need to be established. For this, the concepts of strong and weak scalability are introduced in Section 4.2.2.

4.2.1 Domain Partitioning

When using multiple processes, the problem is divided over the different processes, and each process solves a certain part of the problem. In linear problems like the ones observed here, the unknowns are assigned to one of the processes, where the linear problems are solved for all unknowns on a single process. In problems derived from a discretised domain, unknowns are commonly divided in a way that corresponds to a partition of the discretised domain **(Saad, p. 469 [26])**.

Interactions between different processes are undesirable, as these interactions require work from multiple processes, which slows down the program. To determine how many interactions between different processes are required, the number of unknowns which have an interaction with unknowns allocated to a different process is considered. These unknowns are termed halo unknowns and the set of them is termed the halo. For a good implementation of a parallel method, the size of the halo has to be kept as small as possible.

In Figure 4.1a an example of a domain that is partitioned over multiple processes is seen, together with the corresponding discretisation in Figure 4.1b. In the discretisation, some of the elements have an interaction with elements allocated to a different processes, visible by interactions crossing the dashed lines. These elements are called halo elements and in a good partition this halo is as small as possible. In Figure 4.1c the structure of the matrix that corresponds to this discretisation with this partition is shown. The blocks show interactions between different processes, with blocks on the diagonal signifying interactions on the same process and blocks not on the diagonal signifying interactions between different processes.

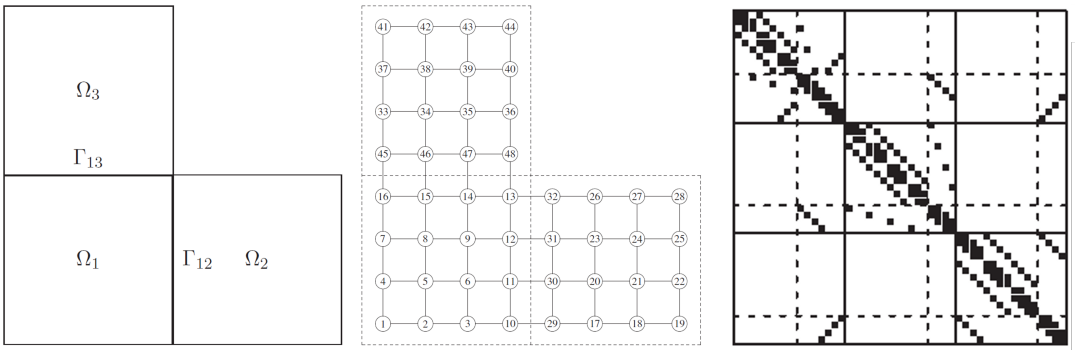


Figure 4.1: Example of domain partition in three parts. (a) A partition of a two-dimensional domain and (b) the corresponding discretisation and (c) matrix are shown. **(Saad, p. 477 [26])**.

In many cases it is difficult to find the best distribution of unknowns over the different processes and a remapping is required. By moving some of the elements from one process to another, the halo can be significantly reduced. In large problems, this redistribution can be very complicated. To use optimal distributions of the large matrices observed here, the k-way partitioning algorithm provided through ParMETIS **(Karypis and Schloegel** [18]**)** is used to redistribute the matrices across multiple processes. This partitioning method aims to reduce the number of elements in the halo.

4.2.2 Scalability of methods

The performance of parallel computing for the methods observed here will be evaluated using the concept of parallel scalability. For this evaluation two types of scalability are considered, strong and weak scalability. To make good use of these concepts, an overview is given of what strong and weak scalability are and what it means for these two types of scalability to be ideal.

To describe the concepts of scalability, it is first required to describe the speed-up and efficiency of a parallel method. The speed-up of a parallel method is defined by running the parallel method that is observed and a corresponding purely sequential method. With $T^*(n)$ being the runtime of a certain sequential program for problem of size n and $T^p(n)$ being a corresponding parallel program with p processors, the speed-up of a method is described, by Rauber and Rünger [23] **(p.162)** as

$$S_p(n) = \frac{T^*(n)}{T_p(n)}. \tag{4.1}$$

It can be assumed that $S_p(n) \leq p$ and in theory this always holds. In practice, the runtime of a sequential program can be unexpectedly high, leading to possible superlinear speed-up.

Using this notion of speed-up, the efficiency of a parallel method is described as

$$E_p(n) = \frac{S_p(n)}{p} = \frac{T^*(n)}{p \cdot T_p(n)}. \tag{4.2}$$

From the fact that $S_p(n) \leq p$ if no superlinear speed occurs, it is obtained that $E_p(n) \leq 1$.

The potential speed-up of parallel programs is restricted by the fact that it is never possible to parallelise the full program. For every program, the runtime can be divided into a sequential part, s, and a part that can be performed in parallel, f. This leads to Amdahl's law **(Amdahl** [2]**)**, which describes the speed-up of a method as

$$S_p(n) = \frac{1}{s + \frac{f}{p}}. \tag{4.3}$$

This means that potential speed-up is restricted by the sequential part.

Now, the strong scalability of a method is studied by observing the speed-up of a method after a certain increase of processes used. For a method that has good strong scalability, using p processes should lead to a speed-up close to p. This means that

according to Amdahl's law, the sequential part of this program should then be small. The strong scalability of a method is ideal if the speed-up is ideal, so $S_p(n) = p$, which means that the sequential part is negligible.

In many cases, the speed-up of a method stagnates after a certain number of processes at a fixed problem size. This stagnation would lead to the conclusion that the scalability of the method is poor if no other problem sizes are considered. This leads to Gustafson's **(Gustafson [13])** law, which states that a scaled speed-up is obtained, when scaling a problem accordingly, that is described by

$$S_p(n) = p + (1 - p)s. \tag{4.4}$$

This still means that the speed-up is poor if s is large. With Gustafson's law, the scalability of a method is dependent on a scaled problem size and thus gives a better insight in the performance of a method for larger numbers of processes. This is referred to as the weak scalability of a method.

The weak scalability of a method is studied by observing corresponding problems of different sizes. For this, suppose that there are problems of sizes n_1 and $n_2 = kn_1$. To these problems a program is applied that is assumed to scale linearly with the size of the problem. This program is applied to the problem of size n_1 with p_1 processes and to the problem of size n_2 with $p_2 = kp_1$ processes. Then if the method scales well with the number of processes, according to Gustafson's law, a speed-up of k should be observed. As the problem size also is increased k times and the program is assumed to scale linearly with the size of the problem, the runtimes that would be observed for these two problems would stay the same.

4.3 Description of Test Environment

In Chapter 5 a wide range of numerical results is presented discussing the performance of the solver methods discussed in Section 4.1. The experiments on larger models, from which the results used to draw conclusions, are all performed on a state-of-the-art High Performance Computing cluster. This cluster consists of 27 HPC CPU nodes connected via infiniband HDR200. Each CPU has 32 cores with 1.8-2.4 Ghz AMD EPYC 7352 Processors. Every node has a maximum available memory of one terabyte and 500 GB of SSD memory available. The cluster runs on the Linux Red Hat ES 8.4 operating system.

During the experiments, at most four CPUs are used at a time. This gives access to a maximum of 128 cores, but the experiments described in Chapter 5 do not require the use of all 128 cores. The highest number of cores used in the shown results is 64. In general, tests with 16 or less cores were performed on one CPU. For experiments using 32 cores, two CPUs were used. For the largest experiments, using 64 cores, four CPUs were used.

Some of the initial experiments on a smaller model, referred to as the GEE model in Chapter 5, were performed on a local machine. This machine used has 32 GB of installed RAM and uses a 2.40 GHz 12th Gen Intel(R) Core(TM) i7-12800H processor. The tests were performed on a virtual Linux machine that used up to 16 GB of RAM.

4.4 Overview of Comparison

Before the numerical results of the various linear solvers are presented, it is important to form a precise basis on which these methods will be compared. To make an accurate comparison between methods, it has to be ensured that the observed results actually represent solving the same linear problem. For this end, the stopping criterion the solvers use is defined and the basis upon which these methods are compared is described. Lastly, a general description of the way in which the numerical results are presented in Chapter 5.

4.4.1 Stopping Criterion

Apart from being the same criterion between different methods, the stopping criterion should also not be too lenient. The solution obtained when reaching the stopping criterion actually has to be a good approximation of the solution. Furthermore, a too lenient stopping criterion might be satisfied too quickly by certain methods that converge quickly at the start. Other methods that do not converge as quickly at the start then appear to perform worse but might actually perform better for a more strict stopping criterion.

On the other hand, the stopping criterion should not be too strict. A too strict stopping criterion might cause well-working methods to be disregarded. A certain method can work well but might not reach a certain criterion if that criterion is set to high. Another reason to not use a too strict stopping criterion is to reduce the runtime of the experiments. An unnecessarily strict stopping criterion makes the time required for each experiment longer, which means that fewer experiments can be done.

In addition to the comparison of solvers, the choice of stopping criterion is also important in actual simulations. For such purposes, the stopping criterion must be sufficiently strict so that the obtained approximate solution accurately approximate the exact solution. On the other hand, it is desired to keep the runtime of the linear solver as low as possible, meaning that the stopping criterion must not be too strict, as extra, unnecessary iterations have to be avoided.

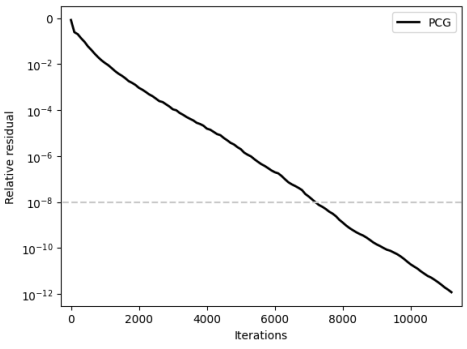


Figure 4.2: Convergence of residual using PCG with the Jacobi preconditioner on example model, showing that the stopping criterion of 10^{-8} for the relative residual is sufficient.

For the experiments carried out here, the stopping criterion used is based on the relative residual, as described by Equation 3.29. In almost all experiments, the value of the tolerance parameter used is $\epsilon = 10^{-8}$. This has proven for all solvers to be a value to which convergence can be obtained, but convergence is not obtained too quickly. In practical simulations, a more lenient value for the tolerance parameter ($\epsilon > 10^{-8}$) could be viable, but since the focus of the experiments carried out here is on the linear solver methods, this more strict value was

chosen. For more strict values of the tolerance parameter ($\epsilon < 10^{-8}$) convergence of the linear solvers can also be obtained, but it is not required for an accurate comparison between methods. In practical use, it can be better to use a more strict value for the tolerance parameter if convergence can be guaranteed. For the comparison of different linear solver methods, this is not necessary and the less strict value of $\epsilon = 10^{-8}$ is sufficient.

Figure 4.2 shows an example of the convergence of the relative error with the Jacobi-preconditioned CG method, on one of the test models, GUL-01M-ST, which is discussed in Chapter 5. In this figure, the number of the iteration j is shown against the relative residual, $\frac{\|\mathbf{r}_j\|}{\|\mathbf{r}_0\|}$. Here, it can be seen that for this method, convergence is quicker in the first few iterations, up to a relative residual of 10^{-2}. After that, until a relative residual of 10^{-12}, the relative residual decreases consistently. The chosen relative residual of 10^{-8}, here represented by a dashed line, is therefore strict enough to be used for the comparison of different solver methods.

4.4.2 Choices in Parallel Computing

When using parallel computing methods in this research, in most experiments this is done using configurations of multiple MPI processes. Only a small investigation is conducted into using hybrid configurations of MPI processes and OpenMP threads in Section 5.7.4.

For general usage of parallel computing using multiple MPI processes in the Visage simulator, a certain number of processes are prescribed, based on the size of the problem. This configuration of MPI processes is referred to as the default configuration for the particular problem size. This default configuration is the configuration up to which a method is expected to scale well. The number of MPI processes N_{MPI} is obtained from the amount of nodes in the FEM discretisation of a problem by taking a power of 2 close to the outcome of the formula

$$\tilde{N}_{\text{MPI}} = \frac{32 \cdot N_o}{10^7}. \tag{4.5}$$

Or alternatively, based on the number of unknowns, N, in the linear problem

$$\tilde{N}_{\text{MPI}} = \frac{32 \cdot N}{3 \cdot 10^7}. \tag{4.6}$$

This roughly leads to the default configurations given by Table 4.1. Although this table does not fully follow the formula provided, it gives a good overview of how the number of MPI processes is selected. For the numerical results, this is the number of MPI processes that is used, unless it is specified that another number of processes is used or no parallelisation is used at all.

4.4.3 Basis of Comparison

To compare the PCG methods with different preconditioners, they are applied to linear problems obtained from geomechanical simulations in the Visage simulator. The

simulations observed for this are as described in Chapter 2, but only a single step of the nonlinear solver is used and only two load steps, an initialisation and an actual load step, are observed. In practical applications, typically more load steps and more steps in the nonlinear solver are applied, but the linear systems in the additional steps of the nonlinear solver and the additional load steps give linear problems similar to the one problem studied here. This means that the comparison of the preconditioning methods is valid for actual simulations.

To compare the approached solver methods two main aspects are considered. The first aspect is how fast the solver methods converge to a solution of desirable accuracy. The second aspect is the peak memory requirement in the application of the solver methods.

In many cases, the rate of convergence is compared in terms of the number of iterations required to find an accurate approximation of the solution. For a comparison between the Jacobi- and deflation-based preconditioners, this is a valid comparison, but this does not provide much information in a comparison with AMG-based preconditioners. An iteration of a method that uses an AMG-based preconditioner is in general much more costly than an iteration of a method that uses a Jacobi preconditioner. On the other hand, less iterations can be expected to be required using the AMG preconditioner.

For a valid comparison, instead of the number of iterations, the main comparison is based on the computational work required by each of the linear solvers. To gain insight into the amount of computational work required by these methods, the wall clock times required to apply these methods are observed. The wall clock times are obtained from the wall clock time of the machine on which the experiment is performed.

The computation times of these methods can be approached in different ways. In most cases, the full time required from the start to the end of the solver call is approached. This includes both the setup stage of the solver and the actual iterative solver. Because, especially for the AMG based methods, a major part of the total runtime can be spent on the setup of the method, it can be more fair to instead only compare the time required by the iterative solver. If multiple simulation steps are performed with similar linear problems, it is potentially possible to reuse the set-up of a method. This puts more value in the speed of the iterative solver, which might lead

No. of nodes (N_o)		Size linear problem (N)		N_{MPI}
Min	Max	Min	Max	
	500,000		1,500,000	1
500,000	750,000	1,500,000	2,250,000	2
750,000	1,750,000	2,250,000	5,250,000	4
1,750,000	3,500,000	5,250,000	10,500,000	8
3,500,000	7,500,000	10,500,000	22,500,000	16
7,500,000	15,000,000	22,500,000	45,000,000	32
15,000,000	30,000,000	45,000,000	90,000,000	64
30,000,000		90,000,000		≥ 128

Table 4.1: Default configurations of MPI processes used in simulations

to an advantage for the AMG-based preconditioners.

The computation time is also used to measure how well the different methods perform if they are performed in parallel. For this, both the strong scalability and weak scalability are investigated. To investigate the strong scalability of the methods, they will be applied to problems with repeatedly doubling numbers of MPI processes. The computation times required with increasing numbers of MPI processes will be compared to each other to determine if the methods scale well strongly. If a method has good strong scalability, it can be expected that doubling the number of MPI processes will half the computation time.

To investigate the weak scalability of the methods, the linear solvers are applied to the same problem of increasing sizes with increasing amounts of MPI processes. For this, if the problem size doubles, then the number of MPI processes is also doubled. The studied linear solver methods can be assumed to scale linearly with the size of the problem. If the weak scalability of a method is good, this leads to the same computation time being observed when both the problem size and number of MPI processes are doubled. If the weak scalability of a method is not as good, this will result in the computation times being higher for the larger problem. In that case, the extra processes are not used efficiently.

While the computation times of methods are important, it is not the only aspect that needs to be compared between the different methods. Another aspect that heavily influences the viability of the linear solvers is their memory requirement. The memory requirements are investigated by measuring the peak requirements in terms of RAM of the CPU during the application of the different methods. While the primary goal is improving the runtime of the linear solvers, it is essential that their memory requirements do not exceed the maximum available memory.

The last requirement to make a good comparison of the methods is a clear explanation of the notation used in the following results. Some of the symbols used repeatedly in the results are presented in Table 4.2. In general, t and T are used for various computation times, with t being used for the time of one iteration of the linear solver method and T being used for different total computation times within the application of the solver methods. m and M are used for memory requirements, with m being used for memory requirements expressed in MB and M for memory requirements expressed in GB.

The last item presented in this table, the total peak memory requirement M_{tot}, usually consists of the sum of several memory requirements. The first aspect within the total memory requirement is the peak memory requirement of the simulator M_{sim}. For the methods that do not use an AMG-based preconditioner, this is the only component of the total memory requirement. For the methods that do use an AMG-based preconditioner, there is an additional requirement of the memory required in the application of the solver, M_{AMG}. Lastly, if a method is used in which a reordering of the unknowns to improve the domain partition is applied, this requires a copy of the original coefficient matrix, which gives a memory requirement for the reordering of the matrix M_{reor}. This gives for the total peak memory requirement,

$$M_{\text{tot}} = M_{\text{sim}} + M_{\text{AMG}} + M_{\text{reor}}. \tag{4.7}$$

Symbol	Description
T_{sup}	Computation time (in seconds) in setup of a solver method
t_{itr}	Computation time (in seconds) of one iteration of a solver method
n_{itrs}	Number of iterations
T_{sol}	Total time of iterative solver, $T_{\text{sol}} = t_{\text{itr}} \cdot n_{\text{itrs}}$
T_{tot}	Total time spend on solver method, $T_{\text{tot}} = T_{\text{sup}} + T_{\text{sol}}$
m	Peak memory requirement of m in MB
M	Peak memory requirement of M in GB
M_{AMG}	Peak memory requirement of AMG-based solver in GB
M_{tot}	Total peak memory requirement in GB

Table 4.2: Overview of symbols used in results in Chapter 5

5 Numerical Results

Using the guidelines discussed in Chapter 4, experiments can be performed using the linear solvers discussed. Linear problems as they are observed in simulations using the Visage simulator, as discussed in Chapter 2, are approached using the different methods of preconditioning the CG method. The results of the experiments performed are presented here, based on which a comparison between the methods is made.

First, an overview of the linear problems considered in these experiments is given in Section 5.1. Next, in Section 5.2, a spectral analysis of one of the smaller problems is performed, to show that the preconditioning methods can be expected to perform well. After this, the results of the application of the different preconditioning methods are presented, with the preconditioners originally used in the Visage simulator in Section 5.3, the SAMG method in Section 5.4, the hypre method in Section 5.5 and the PETSc method in Section 5.6. These methods are then compared in Section 5.7. Lastly, an overview of the improvement in coupled simulations of the Visage simulator is given in Section 5.8.

5.1 Observed Problems

To compare the various linear solver methods, a selection of linear problems are approached. All of these problems are described using the formulation given in Chapter 2. Here an overview of the specific models observed during the experiments is given. First, an overview of the details of the observed problems is given. After this, more details on one of the smaller models are provided.

5.1.1 Overview of problems

In total, seven models were approached, all consisting of two types of linear problem. These approached models are obtained from real cases. The two linear problems correspond to two different types of load step. The linear problem marked with the code "00" corresponds to an initialisation step. The linear problem marked by "ST" is the model used in actual simulation load steps. The difference between these two load steps is in the definition of the boundary conditions of the problem, with the simulation steps having more strict boundary conditions and thus more constraints on elements close to the boundary.

In simulations, multiple time steps on these ST-models are performed using stiffness matrices with the same nonzero structure. As the models on which multiple time steps are performed dominate the total runtime of the simulations, the focus has to be on the performance of the solvers on these models. In general, problems with fewer constraints can be expected to require fewer iterations and thus be easier to solve.

For the comparison between solvers, it is not expected that there are large differences between the two types of models. A solver that performs well on the 00-models is expected to perform well on the ST-models as well.

The seven models were used for different purposes. The first and smallest model, here by the code of "GEE", was used in experiments using the linear solvers to find the right configurations and test implementations of the solvers. Four of the models, with the code "GUL" and their number of nodes in millions, are the most deeply investigated models and are used in most of the results shown in this chapter. The last two models, with codes "YRK" and "GRO", are used to confirm that the results seen on the GUL-models are valid across different models.

Table 5.1 gives an overview of the problems observed in this project and the sizes of these problems. The model size can be approached in two ways, the first of which is by looking at the FEM discretisation. Here, this is done by the number of nodes, given by N_o, from which the elements are built, and the number of constraints, given by N_c, on these nodes. The other way to approach the size of the models is by looking at the sizes of the linear systems obtained from these models, which is most important here. For these linear systems, three main aspects are presented, the size of the linear system, N, the number of nonzero entries of the matrices, nnz, and the average amount of nonzero entries per row, obtained as $\frac{\text{nnz}}{N}$. In the used discretisations, each element is expected to interact with itself and 80 other unknowns, with the exception of elements interacting with the boundaries. This means that most of the rows will have 81 nonzero entries. Some of the rows, however, have less than 81 nonzero entries because of them corresponding to elements interacting with the boundary. This leads to average amounts of nonzero entries in each row slightly below 81. Lastly, the size of the linear

Model	N_{MPI}	N_o	N_c	N	nnz	nnz per row
GEE-00	1	13,671	441	40,572	3,006,568	72.88
GEE-ST			2,841	37,968	2,767,188	74.10
GUL-01M-00	4	1,052,870	6,230	3,060,775	242,557,867	79.25
GUL-01M-ST			6,902	3,008,569	237,477,647	78.93
GUL-05M-00	16	5,189,990	30,710	15,102,037	1,208,335,105	80.01
GUL-05M-ST			31,382	14,986,663	1,197,035,813	79.87
GUL-10M-00	32	10,170,420	60,180	29,605,005	2,374,009,819	80.19
GUL-10M-ST			60,852	29,443,571	2,358,175,787	80.09
GUL-20M-00	64	20,392,554	120,666	59,378,241	4,769,273,149	80.32
GUL-20M-ST			121,338	59,149,691	4,746,831,353	80.25
YRK-00	4	1,147,770	14,715	3,417,354	271,128,118	79.34
YRK-ST			15,023	3,379,736	267,476,018	79.14
GRO-00	32	11,294,304	117,649	33,711,848	2,698,785,430	80.05
GRO-ST			247,609	33,580,236	2,685,914,088	79.98

Table 5.1: Overview of problems with the default MPI configuration (N_{MPI}), number of nodes(N_0) and constraintsN_c), number of unknowns (N), number of non-zero's (nnz) and average amount of number of non-zero's per row.

system relates to the numbers of nodes and constraints by

$$N = 3 \cdot N_o - N_c. \tag{5.1}$$

When parallel computing is used in the application of linear solvers to these problems, unless otherwise stated, the number of processes is as described in Section 4.4.2. For these problems, this gives the numbers of MPI processes as given in the column N_{MPI}.

5.1.2 Analysis of Problems

For the application of the linear solver, it is important to know more about the exact properties of the observed matrices. First, an overview of the sparsity structures of some of the problems is given. After this, it is numerically confirmed that the observed problems can be assumed to be SPD and that the proposed methods can be applied to the observed problems.

5.1.2.1 Matrix structure

To gain more insight in how the linear problems exactly look like, the nonzero structures of some of the matrices are observed. As these matrices are obtained from an FEM discretisation, a structure is expected that appears as bands of diagonals. For these particular problems, around 81 nonzero entries per row of the matrix are expected. This suggests a structure that resembles 81 diagonals. Because the problems are unstructured, it can not expected to actually observe these 81 diagonals, and instead small jumps can be expected. For the bands of diagonals, the expectation is that the nonzero elements are expected to roughly be located in three bands, each of which consists of three bands itself, as the problems are three-dimensional. Because of the unstructured nature of the observed problems, small jumps can also be expected in these bands.

Figure 5.1 confirms the expectations for the matrix structure. There, first the full structure of the matrix is shown, and then the structures of a series of submatrices of decreasing size located around the centre of the matrix are shown. The nonzero entries of the matrix are represented by black dots. The expected bands of diagonals can clearly be seen here. In Figure 5.1b, it can be seen that the matrix consists mainly of three bands of diagonals. Then in Figure 5.1d, it appears that the middle band again consists of three bands of diagonals, which, as shown in Figure 5.1f, consist of nine diagonals each. This corresponds to the expected 81 diagonals of nonzero elements. Because of the large size of the matrix, the expected jumps in diagonals can not be seen here, as these are typically small.

An irregularity in the structure of the matrix is observed around $[A]_{850{,}000;850{,}000}$. There a block of around $200,000$ entries is seen that does not follow the diagonal structure seen in the rest of the matrix. The reason for this irregularity is difficult to determine but possibly has to do with a mesh refinement in the corresponding reservoir grid. The irregular portion consists only of few elements compared to the diagonals. When using a domain decomposition in the application of parallel methods, this irregularity can lead to difficulties. As the unknowns corresponding to this irregular

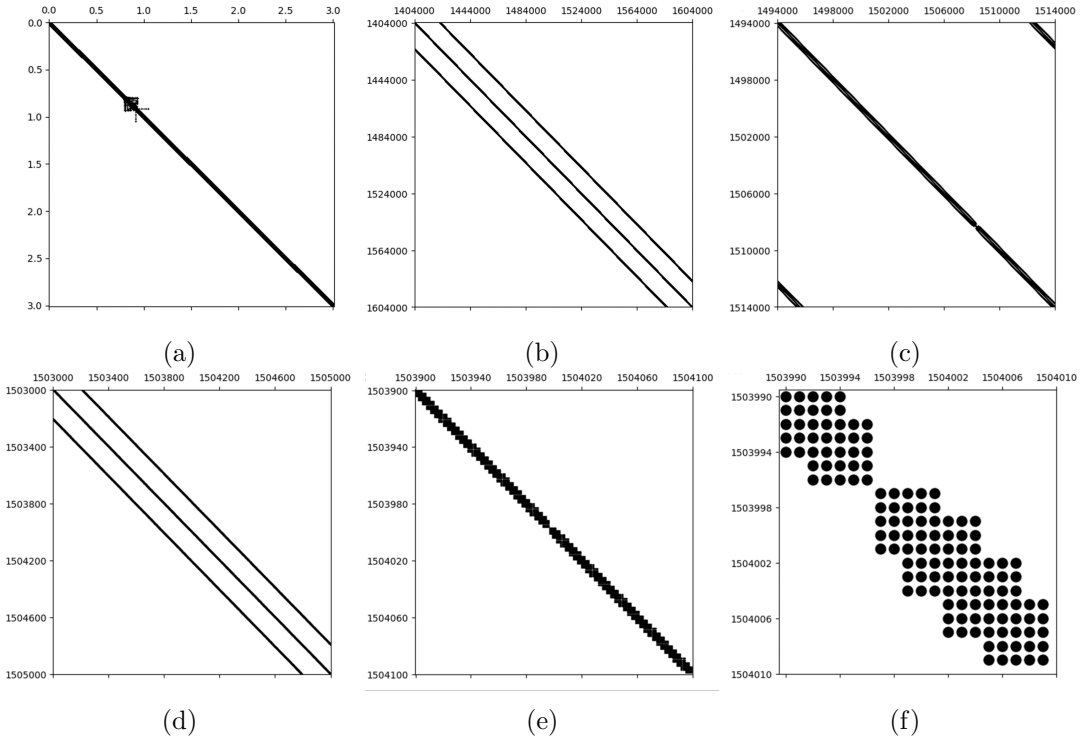


Figure 5.1: Detailed overview of matrix observed in GUL-01M-ST problem, zoomed around the element $[A]_{\frac{1}{2}N,\frac{1}{2}N}$. (a) First, the original matrix structure is shown. (b-f) Then, the zoomed-in blocks in the centre of the matrix are shown, each time of the form $[A]_{(\frac{1}{2}N-N_B):(\frac{1}{2}N+N_B),(\frac{1}{2}N-N_B):(\frac{1}{2}N+N_B)}$, where N_B describes the number of rows and columns of these blocks. The value of N_B starts at (b) 100,000 for the first zoomed in matrix and (c-e) decreases by a factor 10 for each subsequent zoomed-in image of the matrix. (f) In the last image, $N_B = 10$

block are connected to unknowns in a way different from the unknowns corresponding to other parts, in some cases more unknowns corresponding to this block can end up in the halo in a domain decomposition. This would result in the halo being larger than the halo of a matrix without the irregular portion.

5.1.2.2 SPD properties

The use of the CG method requires that the problems be SPD, which is expected for all problems observed here, as discussed in Section 2.3. However, in practice, the problems might not end up being SPD because of numerical issues.

For the observed problems, it can easily be confirmed that they can be assumed to be SPD. The main difficulty of these problems lies in the symmetry of the problems. Although the matrices should in theory be symmetric, there can be numerical asymmetries. It must be confirmed that any numerical asymmetries are caused by round-off errors.

This is done for the two smallest problems. First, it is observed that for the smallest problem, the GEE problems, no asymmetries exist. For any element of the matrices

used in the GEE problems, $[A]_{i,j} = [A]_{j,i}$ is observed.

For the smallest problem that is actually used in the investigation of the AMG methods, GUL-01M-ST, some asymmetries are observed. To confirm that these asymmetries are small enough to not be problematic for the application of the CG method, the relative maximum difference between two elements in positions of symmetry is observed. Then it is seen that

$$\frac{|[A]_{i,j} - [A]_{j,i}|}{|[A]_{i,j}|} = 5.3937 \cdot 10^{-14}, \tag{5.2}$$

which is small enough to assume the difference is due to round-off errors. As the asymmetries are due to round-off errors, this matrix can be assumed to be symmetric for the purpose of an iteration method like CG, which uses a tolerance large enough for this to not be a problem. For the other, larger matrices, similar results can be expected to be seen.

As the assumption can be made that the matrices are symmetric, it can easily be obtained that the matrices are SPD if they are strictly diagonally dominant and all diagonal entries are positive. The matrices observed here are all strictly diagonally dominant and have positive diagonal entries. This, together with the assumption they can be viewed as symmetric for the purpose of an iterative method like CG, the assumption that they are SPD for this purpose can also be made.

5.2 Spectral Analysis

To show that the preconditioning of the matrix using the different methods discussed here actually gives an advantage to the CG method, the condition numbers of the preconditioned systems are approached. As covered in Section 3.2, the number of iterations required for a preconditioned CG method to converge is directly influenced by the condition number. To obtain the condition numbers of the preconditioned matrices, their extreme eigenvalues must be computed.

The computations of the eigenvalues are performed on the GEE model, the smallest of the models approached. The condition numbers obtained for this model do not directly tell anything for the larger models observed in actual simulations, but the reduction of the condition number can be expected to be similar.

The eigenvalues obtained here were obtained with a relative tolerance of 10^{-3}. For an estimated pair of eigenvalue and eigenvector $(\tilde{\lambda}, \tilde{\mathbf{u}})$, where the eigenvector is such that $\|\mathbf{u}\| = 1$, the residual is computed as $\tilde{\mathbf{p}} = A\tilde{\mathbf{u}} - \tilde{\lambda}\tilde{\mathbf{u}}$. This residual is such that $\frac{\|\tilde{\mathbf{p}}\|_2}{\|\tilde{\mathbf{u}}\|_2} \leq 10^{-3}|\tilde{\lambda}|$. Now, since the observed matrices are SPD, it holds that $|\lambda - \tilde{\lambda}| \leq \|\tilde{\mathbf{p}}\|_2$ **(Saad, p. 61 [27])**. This means that $\frac{|\lambda - \tilde{\lambda}|}{|\tilde{\lambda}|} \leq 10^{-3}$, which means that the estimates obtained for the eigenvalues are accurate enough estimates to tell something valuable about the condition numbers.

To gain a good understanding of how much the condition number is reduced by applying AMG preconditioners, AMG preconditioners of different levels are observed. This is done both with and without a Jacobi smoother. For the matrices without

smoothing, the matrices that are studied are of the form $M^{-1}A$, with

$$M^{-1} = I - \left(I_1^0 I_2^1 \cdots I_k^{k-1}\right) A_k^{-1} \left(I_{k-1}^k I_{k-2}^{k-1} \cdots I_0^1\right). \tag{5.3}$$

By using different values of k, the effect of AMG methods of different levels is observed. Starting with $k = 0$, the eigenvalues of the original matrix A are obtained, as then $M^{-1} = I$. k is increased upto K, the total number of levels in the AMG hierarchy, with $k = K$ representing the application of a AMG method without smoothing.

In a similar way, for the application of AMG with Jacobi smoothing, the studied matrices are of the form $M^{-1}A$, with

$$M^{-1} = I - \left(I_1^0 I_2^1 \cdots I_k^{k-1}\right) A_k^{-1} \left(S_k I_{k-1}^k S_{k-1} I_{k-2}^{k-1} S_{k-2} \cdots I_0^1 S_0\right). \tag{5.4}$$

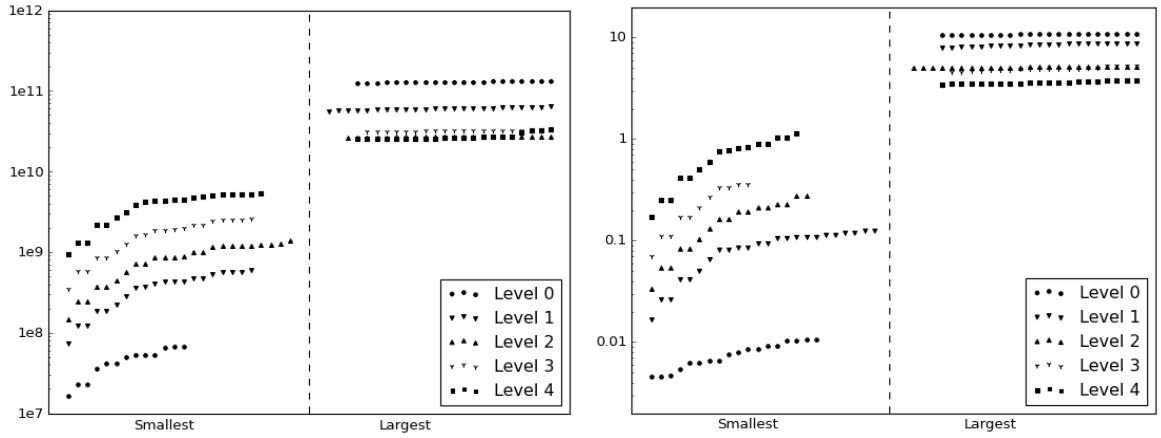
The smoothing matrix S_0 used is a damped Jacobi smoother matrix with $\omega = 0.4$. For $k = 0$, the is the original matrix multiplied by the damped Jacobi preconditioner is observed. The matrices on higher levels are obtained in the same way as for the unsmoothed matrices. The matrices on higher levels of the multigrid hierarchy and the prolongation and restriction operators are obtained using the SAMG method, discussed in Section 5.4.

In Table 5.2 condition numbers and eigenvalues for the different observed matrices are shown. Unfortunately, it should be noted that the values shown are not the correct values for the observed matrices. The values shown are of matrices of the form A_k and $S_k A_k$. The method used to obtain the eigenvalues was not performed correctly, which was discovered when there was no possibility to recompute the correct values. The same applies to the values in Figure 5.2, where the largest and smallest eigenvalues of the matrices are shown. Only the results for the original matrix and the matrix preconditioned with the Jacobi method (both matrices with $k = 0$) are the correct values.

The results obtained are similar to what is expected for the results that would be observed if the eigenvalues of the correct matrices were obtained. The condition number is expected to decrease with the application of the preconditioner, which is true in the results seen here. Furthermore, the decrease seen by the application of the AMG preconditioner is much more severe than that of the Jacobi preconditioner. This also further improves by using smoothing and when more AMG levels are used. Both aspects are seen here and would be expected to be seen if the correct values were

			No smoothing			ω-Jacobi smoothing		
P	N	nnz	κ	$\lambda_{\max}$	$\lambda_{\min}$	κ	$\lambda_{\max}$	$\lambda_{\min}$
0	37968	2767188	8041.6	$1.33 \cdot 10^{11}$	$1.66 \cdot 10^{7}$	2413.5	10.99	$4.55 \cdot 10^{-3}$
1	8673	585901	865.4	$6.40 \cdot 10^{10}$	$7.39 \cdot 10^{7}$	529.7	8.84	$1.67 \cdot 10^{-2}$
2	4095	453088	184.0	$2.73 \cdot 10^{10}$	$1.48 \cdot 10^{8}$	150.8	5.12	$3.39 \cdot 10^{-2}$
3	1773	185793	94.2	$3.30 \cdot 10^{10}$	$3.50 \cdot 10^{8}$	74.9	5.32	$7.10 \cdot 10^{-2}$
4	647	102257	35.4	$3.28 \cdot 10^{10}$	$9.29 \cdot 10^{8}$	22.1	3.81	$1.72 \cdot 10^{-1}$

Table 5.2: Condition numbers and extreme eigenvalues of matrices used in AMG on GEE without smoothing and with smoothing using a damped Jacobi smoother.



(a) (b)

Figure 5.2: Overview of largest and smallest eigenvalues of matrices used on different levels of AMG, performed (a) without smoothing and with (b) Jacobi smoothing

obtained. Lastly, the numbers of iterations required for the CG method to converge that were observed in the experiments discussed in this chapter roughly correspond to the condition numbers observed here.

5.3 Original preconditioners

For the first results of the linear solvers on the problems described in Section 5.1, the solvers that were originally available within the Visage simulator are used. Originally, two methods were available, a PCG method using a Jacobi preconditioner and a CG method using a deflation-based preconditioner, as described in Section 4.1.

To start, a comparison is made between these two preconditioners to see which of the two is best to use as a benchmark for AMG-based methods. There, it is observed that the most simple preconditioner of the two, the Jacobi preconditioner, is used best as a benchmark. For this preconditioner, a further investigation of the strong scalability is performed.

5.3.1 Jacobi and Deflation preconditioner

The first way to compare the two preconditioning methods used originally in the Visage simulator is by comparing the convergence of the residual against the required iterations. The convergence of the deflation PCG method is expected to require fewer iterations than the Jacobi PCG method.

Figure 5.3 shows convergence of the norm of the residual of these two methods applied on the linear problem of the GUL-01M-00 model, in a similar way to how it was done for Figure 4.2. There it is seen that the PCG method using deflation converges slightly quicker than the method preconditioned using the Jacobi preconditioner. However, the difference in convergence for this problem is very small, only a few hundred

iterations, while in total several thousand iterations are required.

In certain problems, the increase of the norm of the residual after applying the restart of the CG method is much greater than what is observed here. In these types of problem, the decrease in the required number of iterations by using the CG method that uses a deflation preconditioner is much more significant. For the particular problems that are observed here, this is not the case, making the deflation preconditioner not as effective as it can be.

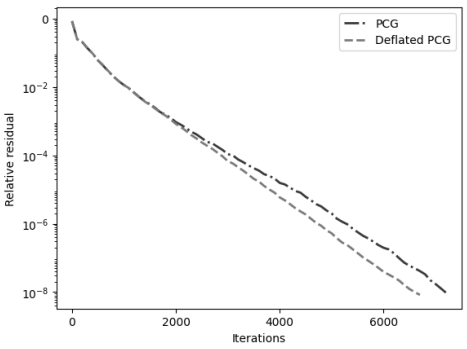


Figure 5.3: Convergence of residual on GUL-01M-00 for Jacobi PCG and Deflation PCG.

For a selection of the other problems, the results are similar to those seen for the GUL-01M-00 problem. The results of experiments on these problems can be seen in Table 5.3. The problems used in the comparison of the deflation preconditioner and the Jacobi preconditioner are mainly the smallest problems out of the ones presented in Section 5.1.1. The experiments performed here use parallel configurations as discussed in Section 4.4.2. Thus, 4 MPI processes are used for all experiments on the GUL-01M and YRK models and 16 MPI processes are used for the GUL-05M model.

For the convergence using the two different preconditioner methods, in general similar things are seen to what is discussed above for the GUL-01M-00 problem, but, depending on the observed problems, some differences are observed. Commonly, the method that uses the deflation preconditioner requires slightly less iterations than the one that uses the Jacobi preconditioner. In two cases, GUL-05M-00 and GUL-01M-00, the deflation preconditioner instead requires more iterations. Only for one of the observed problems, the YRK-00 problem, a significant improvement in terms of required iterations is obtained by using the deflation preconditioner. There, the number of iterations required to converge to a suitable solution is reduced by one third by using the deflation preconditioner.

Although the deflation preconditioner does offer the possibility to decrease the number of iterations required, in all but one of the cases the computation time when using the deflation preconditioner is higher. Due to the more complicated preconditioner, one iteration of the CG method takes more time when using the deflation preconditioner. This leads to a longer computation time for the iterative solver. Also, since the setup of the deflation preconditioner is more complicated, more time is spent on the setup of the method. This is insignificant for the total runtime, as for all experiments across the different problems and preconditioners the setup time is only a small portion of the total runtime.

Only for one of the problems, the YRK-00 problem, the deflation preconditioner gives an advantage for the runtime. Because of the significant improvement in the number of iterations required to converge to a suitable approximate solution, the slightly longer runtime of each iteration is not a problem for the total runtime. This means that for particular problems the deflation preconditioner can be very useful. Unfortunately,

Model	Preconditioner	T_{sup}	t_{itr}	n_{itrs}	T_{sol}	T_{tot}	M_{tot}
GUL-01M-00	Jacobi	4.0	0.039	7568	298.2	302.2	5.5
	Deflation	4.0	0.041	7233	298.1	302.1	6.0
GUL-01M-ST	Jacobi	4.0	0.040	6574	266.1	270.1	6.1
	Deflation	4.0	0.040	6720	271.9	275.9	6.6
YRK-00	Jacobi	5.0	0.040	4591	185.8	190.8	6.1
	Deflation	5.0	0.046	3005	138.9	143.9	6.7
YRK-ST	Jacobi	5.0	0.047	4575	182.3	187.3	5.9
	Deflation	4.9	0.055	4392	200.6	205.5	6.5
GUL-05M-00	Jacobi	10.7	0.047	6985	329.5	340.2	43.4
	Deflation	13.1	0.055	7028	383.4	396.5	50.7
GUL-05M-ST	Jacobi	10.4	0.040	6642	310.2	320.6	46.9
	Deflation	12.1	0.046	6235	341.2	353.3	54.2

Table 5.3: Results of the PCG method preconditioned with a Jacobi and with a deflation preconditioner applied to both the initialisation and simulation step of three different models

this is not the case for every problem and no clear distinction can be made between when the deflation preconditioner performs better than the Jacobi preconditioner and when it does not.

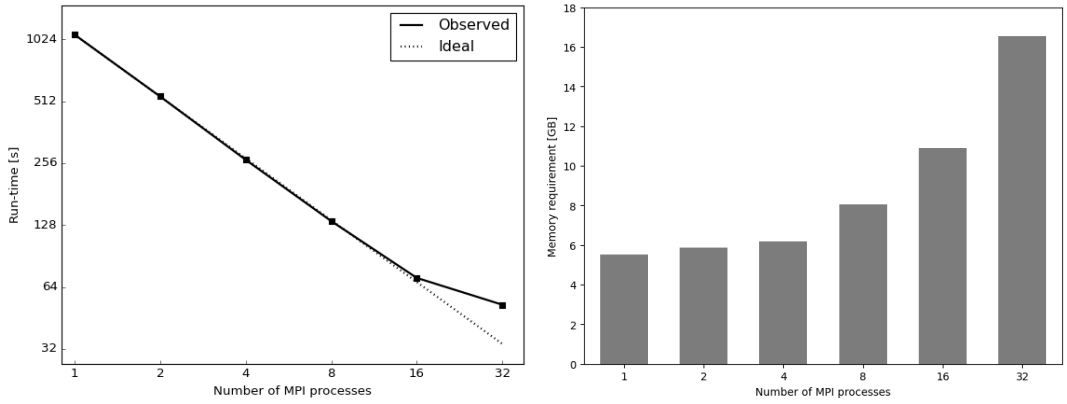
Lastly, the peak memory requirement increases when the deflation preconditioner is used. In all problems for which the two preconditioners are compared, the memory requirement of the deflation preconditioner is 10 to 20% higher than that of the Jacobi preconditioner. The numbers given here represent the total peak memory requirement in the simulation.

Based on these results, the choice is made here to use the more simple Jacobi preconditioner as a benchmark to compare against the AMG based preconditioners. As discussed, the deflation preconditioner does, in most of the cases considered here, not offer an improvement in runtime and also has a higher memory requirement.

For both the Jacobi and deflation methods, an important note has to be made. Theoretically, the expectation is that applying the PCG method with these preconditioners to a finer FEM discretisation of the same model results in more iterations being required to satisfy the same decrease in relative residual. However, when comparing the experiments on the GUL-01M case and the GUL-05M, this is not the case here. A thorough investigation has been performed to determine why this is the case, but no results different from what was obtained here have been found.

5.3.2 Strong Scalability

To test the strong scalability of the Jacobi preconditioned CG method, one of the problems, the GUL-01M-ST problem, is approached with several configurations of MPI processes. To start with, the simulation is performed without any parallel computing, so with a single MPI process. After this, the same simulation is performed with repeatedly doubling numbers of MPI processes. If the strong scalability of this method would be ideal, then doubling the number of MPI processes results in the runtime being halved.



(a) (b)

Figure 5.4: Strong scalability of runtime and corresponding memory usage for solving the linear problem with the Jacobi PCG method in one timestep of GUL-01M-ST using varying amounts of MPI processes. (a) The total runtime of the solver and (b) the memory requirement are shown here.

Figure 5.4a shows the results of this investigation in strong scalability. The runtimes shown here are runtimes of the full application of the linear solver, so both the setup of the solver and the iterative solver itself. The results that would have been obtained if the strong scalability of the method would have been ideal are given by the dotted line. As can be seen, the observed runtimes remain very close to this ideal strong scalability. Only when increasing to 32 MPI processes, the observed runtime is significantly higher than the runtime that would be observed if the strong scalability of the solver were to be ideal. This means that this method scales very well with the increase of the number of MPI processes.

Figure 5.4b shows the total peak memory requirements corresponding to the same configurations of MPI processes. Increasing the number of MPI processes results in a significant increase in the peak memory requirement. This is especially prevalent when using 16 or 32 processes, which can double or triple the peak memory requirement.

5.4 SAMG

After exploring the original preconditioning methods discussed in the previous section, the advancement to AMG-based preconditioners can be made. The first AMG-based preconditioner that is covered is the method provided through SAMG.

The AMG preconditioner using SAMG is applied in a single V-cycle. For coarsening, standard coarsening operators are used for all but the first coarsening operations. For the first coarsening operation, an A1-coarsening operation is applied. This changes only in the section where different coarsening operations are considered. The prolongation is performed through unknown-based prolongation. The smoother used, apart from in the investigation of different smoother methods, is the ℓ_1-Gauss-Seidel smoother. The AMG preconditioner is applied to an CG iterative solver provided through the AMG

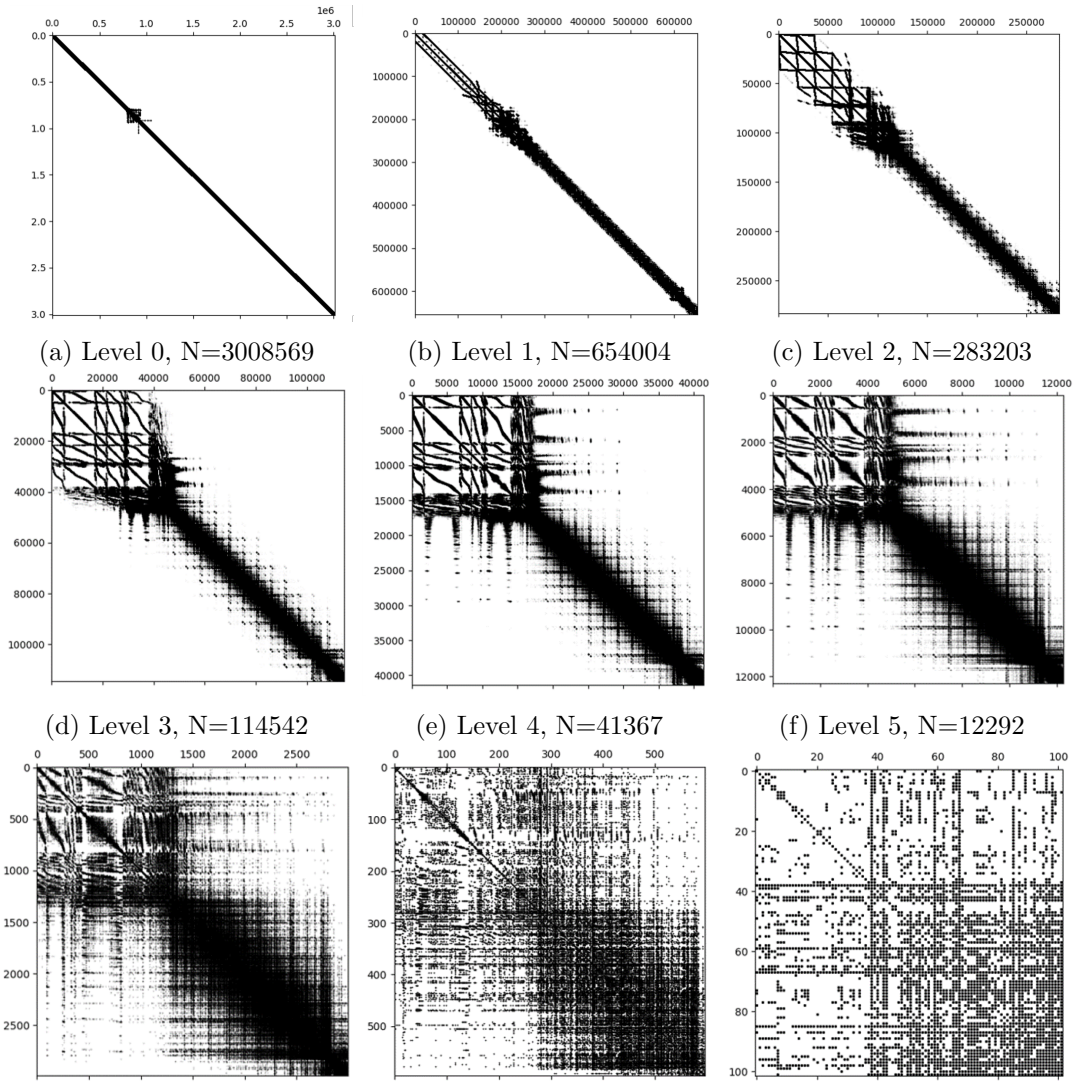
solver, with a small investigation being conducted into other solvers.

To gain insight into the AMG structure obtained in a practical problem, a multigrid structure obtained through SAMG is discussed. After this, some aspects of the setup of the SAMG preconditioner are discussed. Then, a study of the scalability of the SAMG methods is presented.

5.4.1 Multigrid Structure

Using SAMG a multigrid hierarchy is built. This hierarchy consists of multiple matrices, derived through interpolation from the original full-size matrix.

Figure 5.5 gives a simple overview of what such a multigrid hierarchy looks like



(a) Level 0, N=3008569 (b) Level 1, N=654004 (c) Level 2, N=283203

(d) Level 3, N=114542 (e) Level 4, N=41367 (f) Level 5, N=12292

(g) Level 6, N=2991 (h) Level 7, N=596 (i) Level 8, N=102

Figure 5.5: Overview of SAMG structure obtained using SAMG for GUL_01M_ST model. For every level the corresponding matrix size (N) is shown.

on one of the smaller models, the GUL-01M-ST model, which was also observed in Section 5.1.2. This hierarchy is built using an A1-aggressive coarsening operation to built the first level from the original matrix and using standard coarsening operations to built the subsequent levels. It can be seen that for the higher levels of the AMG hierarchy the size of linear systems decrease, but the number of nonzero elements does not decrease as quickly. This makes the matrices on the lower levels of the hierarchy more dense than the matrices on the higher levels.

A particular part that is interesting in the matrices in this multigrid hierarchy is the influence of the irregular part around the $[A]_{850,000;850,000}$ entry of the original matrix. This irregularity causes similar irregularities in the matrices at other levels of the hierarchy. This is because some of the irregular connections in the original coarsening heavily influence the coarsening operation at other levels of the hierarchy. This irregular part does not cause problems for the application of the multigrid method but can cause more computations to be required in the interpolation and restriction operations. This could slow down the multigrid method compared to when it is applied to a matrix with only elements in a set of diagonals.

It is observed that in the last matrix obtained here, on level 8, there is still a level of sparsity. This suggests that it might be possible to apply additional coarsening operations to this matrix to obtain an even smaller matrix that is completely dense. However, in practice this is usually not done, as the linear system on the highest level of the hierarchy is already small to effectively solve using a direct method.

For the matrices obtained on the different levels of the multigrid structure, many of the properties of the original matrix are preserved. This means that the matrices on all levels are symmetric. Furthermore, the matrices all have positive diagonal elements and are diagonally dominant. This means that the matrices on every level of the hierarchy are SPD.

5.4.2 Krylov method and Smoother

Before investigating the performance of the AMG preconditioner using SAMG software, it is necessary to find the best possible configuration of the AMG preconditioner. The first configuration investigated is the configuration of Krylov methods and smoothers to be used in the application of the SAMG preconditioner. SAMG offers a wide variety of Krylov methods and smoothers that can potentially be used in the application of the preconditioner. A selection of the most commonly used of these options is covered here.

The problems observed are SPD problems to which a standard AMG method is applied. According to the SAMG Team [35] **(p.38)**, all the available Krylov methods should be applicable to this type of problem. As the problems are SPD, the expectation is that the CG method is the best method for these problems. However, because of the occurrence of poor element shapes in the FEM discretisation, problems can be nearly singular. This raises the concern that the CG method might not always be applicable to these problems. For the smoother in the AMG method, no clear guidelines are given, but it can be desirable to use a smoother that is simple to compute.

In Table 5.4 an overview of the results of the investigation into the configuration

Krylov Method	Smoother	T_{sup}	t_{itr}	n_{itrs}	T_{sol}	T_{tot}	M_{itr}	M_{AMG}	M_{tot}
CG	$\omega-$Jacobi	3.82	0.501	36	18.0	21.8	0.09	1.28	9.1
	GS	4.69	0.500	27	13.5	18.2	0.19	1.28	9.1
	ℓ_1-GS	4.29	0.323	27	8.7	13.0	0.13	1.28	9.1
	ILU(0)	15.38	0.533	No Convergence					
	ILUT	7.77	0.304	No Convergence					
GMRES	$\omega-$Jacobi	4.15	0.584	46	26.9	31.0	0.76	1.69	9.5
	GS	4.75	0.526	41	21.6	26.3	0.86	1.79	9.6
	ℓ_1-GS	4.31	0.404	38	15.4	19.7	0.80	1.74	9.6
	ILU(0)	15.72	0.568	No Convergence					
	ILUT	7.79	0.349	No Convergence					
GCR	$\omega-$Jacobi	4.10	0.612	47	28.8	32.9	0.78	1.71	9.6
	GS	4.63	0.612	42	25.7	30.3	0.88	1.42	9.3
	ℓ_1-GS	4.37	0.445	39	17.4	21.7	0.83	1.76	9.6
	ILU(0)	15.25	0.619	No Convergence					
	ILUT	7.93	0.374	No Convergence					
BiCGSTAB	$\omega-$Jacobi	4.23	1.062	43	45.7	49.9	0.10	1.28	9.1
	GS	4.75	0.906	23	20.8	25.6	0.20	1.28	9.1
	ℓ_1-GS	4.35	0.614	22	13.5	17.9	0.15	1.28	9.1
	ILU(0)	13.97	1.022	No Convergence					
	ILUT	7.80	0.543	No Convergence					

Table 5.4: Comparison of application of SAMG with different Krylov method solvers and smoothers MPI = 2

of the smoothers and Krylov methods is shown. These results are obtained for the GUL-01M-ST problem using a tolerance in the relative residual of 10^{-4}. The use of a relative residual of 10^{-4} instead of the usually used 10^{-8} is to speed up the process of these experiments. Performing a selection of the experiments with a relative residual of 10^{-8} gives results similar to those observed here, but more iterations are required. For the MPI configuration, two MPI processes are used. This is done to signify the difference between the normal Gauss-Seidel smoother and the ℓ_1-Gauss-Seidel smoother. Without parallel computing, similar results are obtained, except that there is no difference between the Gauss-Seidel and ℓ_1-Gauss-Seidel smoother, as these smoothers are exactly the same if parallel computing is not used.

For the results in Table 5.4, the symbols are mostly as they were introduced in Section 4.4. There are two new results compared here that were not yet described in Section 4.4. The first, M_{itr}, is the peak memory requirement during the iteration of the Krylov method. The second, M_{AMG}, is the peak memory requirement during the application of the AMG method.

For the Krylov method, it is observed that the best results are obtained using the CG method. For every smoother applied that gives convergence, the CG method requires the shortest runtime to converge. In terms of number of iterations, the CG method requires less iterations than both the GMRES and GCR methods and only, for two of the smoothers, more than the BiCGSTAB method. However, the time of a

single iteration with the BiCGSTAB method is twice as long. For the peak memory requirement, there are some differences during the iteration of the Krylov method, with the CG method having the lowest requirement. This advantage is less prevalent in the total peak memory requirement.

The next aspect to consider are the different smoothing methods. Five different smoother methods are considered, of which three, Gauss-Seidel, ℓ_1-Gauss-Seidel, and ILU(0) **(Saad, p. 307 [26])**, are applied in a regular way. The other two, $\omega-$Jacobi and ILUT need further clarification. The $\omega-$Jacobi smoother refers to a damped Jacobi smoother with damping parameter ω. The value of the damping used here is $\omega = 0.4$. Other values have been investigated and gave similar results as long as the value of ω stays away from 0 and 1.

The ILUT **(Saad, p. 321 [26])** method is used with a fill-in parameter of 1 and a tolerance of 0.05. The fill-in parameter controls the level of fill-in allowed in the computation of the smoother. The tolerance causes entries of the smoother matrix to be dropped during the computation of the smoother if they are less than a relative tolerance that is based on the global tolerance and the norm of the row of the smoother matrix. The ILUT smoother was tested with different values of these parameters, but no parameter configuration resulted in convergence for the ILUT smoother. The ILU smoother, which is the same as using the ILUT smoother with 0 level of fill-in and a tolerance of 1, does not result in convergence with any of the Krylov methods either.

For the smoothers that converge, the two Gauss-Seidel smoothers significantly outperform the Jacobi smoother. The use of either of the two smoothers results in requiring fewer iterations and a shorter runtime than the use of the Jacobi smoother. Only the peak memory requirement is slightly higher during the iteration of the Krylov method, but this is insignificant to the total memory requirement.

Between the two Gauss-Seidel smoothers, the ℓ_1-smoother has a significant advantage over the normal Gauss-Seidel smoother, because the runtime required by the ℓ_1-smoother for a single iteration is much lower than that of the Gauss-Seidel smoother.

For these reasons, the choice was made to apply the SAMG method using the CG method with the ℓ_1-Gauss-Seidel smoother. This is the configuration that is used in all the experiments that follow.

5.4.3 Coarsening method

A second aspect in the setup of SAMG that is to be investigated is the use of aggressive coarsening. The recommended use of aggressive coarsening for these types of problems is to use a single A1-coarsening operation for the first coarsening operation **(the SAMG Team [35])**. Here an investigation is conducted into whether this aggressive coarsening operation is necessary and if applying more than one aggressive coarsening operation can result in a significant improvement. If aggressive coarsening is applied, the type of aggressive coarsening that is used is always A1-coarsening, as this is recommended for anisotropic FEM discretisations. The value of the strong connectivity threshold, θ_{ecg} as described in Definition 3.2, is set to 0.7. This is different from the default value used in SAMG, which is 0.25. Like the use of A1-coarsening, this is also a

recommendation specifically for problems derived from anisotropic FEM discretisations.

To obtain the matrix on a higher level of the multigrid hierarchy, a coarsening operation is applied. This coarsening operation can be a standard coarsening operation or an A1-coarsening operation. To investigate the effectiveness of aggressive coarsening, A1-coarsening operations are applied to a changing number of levels. When aggressive coarsening is applied, it is always applied on the highest possible levels.

For these experiments, the preconditioner is applied using A1-coarsening for the first n_{AO} operations. In terms of the prolongation I_{k+1}^{k} and restriction I_{k}^{k+1} operators given in Section 3.3.4, this means that operators $I_{1}^{0}, \cdots I_{n_{AO}}^{n_{AO}-1}$ and $I_{0}^{1}, \cdots I_{n_{AO}-1}^{n_{AO}}$ are obtained through aggressive coarsening. The other prolongation and restriction operators are obtained through standard coarsening. The value of n_{AO} changes from 0, which means that all coarsening operations are standard coarsening operations, to K, which means that all coarsening operations are A1-coarsening operations.

Table 5.5 shows the numbers of unknowns on the different levels of the AMG hierarchy when applying various numbers of A1-aggressive coarsening operations to the linear system of the GUL-01M-ST problem. The number of aggressive coarsening operations corresponds to the parameter p here. In the first column, no aggressive coarsening is applied, in the second one operation is applied, between level 0 and level 1 and so on. In the last column, all coarsening operations are aggressive. One level of aggressive coarsening is what is typically used and corresponds to the example discussed above in Section 5.4.1.

In Table 5.6 the results of the approached configurations of aggressive coarsening operations are shown, along with the AMG complexities of the used multigrid hierarchies. All results shown here were obtained without the use of parallel computing. There, it is observed that a massive advantage is obtained by using at least one level of aggressive coarsening. Although the change in number of iterations is small, the decrease in time required for each iteration is significantly reduced. Furthermore, the time required in the setup of the AMG method is reduced significantly by the use of aggressive coarsening. Lastly, the peak memory requirement of the AMG method is cut to a third by using at least one aggressive coarsening operation. This results in a

	Number of operations of A1-aggressive coarsening					
Level	0	1	2	3	4	5
0	3060775	3060775	3060775	3060775	3060775	3060775
1	1407219	654004	654004	654004	654004	654004
2	624557	283203	137006	137006	137006	137006
3	259911	114542	52555	23782	23782	23782
4	98370	41367	17358	7675	3108	3108
5	32933	12292	4641	1999	746	331
6	9140	2991	1069	400	143	
7	2150	596	206			
8	444	102				

Table 5.5: Numbers of unknowns on each level of the AMG structures for GUL-01M-ST with different aggressive coarsening applied to varying amounts of levels

n_{AO}	T_{sup}	t_{itr}	n_{itrs}	T_{sol}	T_{tot}	M_{AMG}	M_{tot}	ω_g	ω_a
0	36.94	2.325	145	337.1	374.1	4.90	12.47	1.8093	2.0804
1	20.90	1.688	138	232.9	253.8	1.78	9.35	1.3686	1.4147
2	16.91	1.553	139	215.9	232.8	1.78	9.34	1.2881	1.2874
3	16.37	1.541	139	214.2	230.6	1.78	9.34	1.2742	1.2636
4	16.32	1.541	139	214.2	230.5	1.77	9.34	1.2722	1.2612
5	17.96	1.535	140	214.9	232.9	1.77	9.34	1.2720	1.2610

Table 5.6: Results of SAMG on GUL-01M-ST with different amounts of levels to which aggressive coarsening is applied.

memory requirement that is 25% lower in total.

The use of two coarsening operations can further reduce the runtime of both the setup stage and the iteration stage. Here, a further reduction of around 20% of the runtime is obtained by using an extra aggressive coarsening operation.

Using more than two aggressive coarsening operations gives similar results as using two aggressive coarsening operations. In some cases, using too many aggressive coarsening operations can lead to a loss of information in the prolongation operator. This would lead to a loss in the effectiveness of the AMG preconditioner. However, this is not observed here, possibly because the used method for aggressive coarsening is not aggressive enough for this to happen.

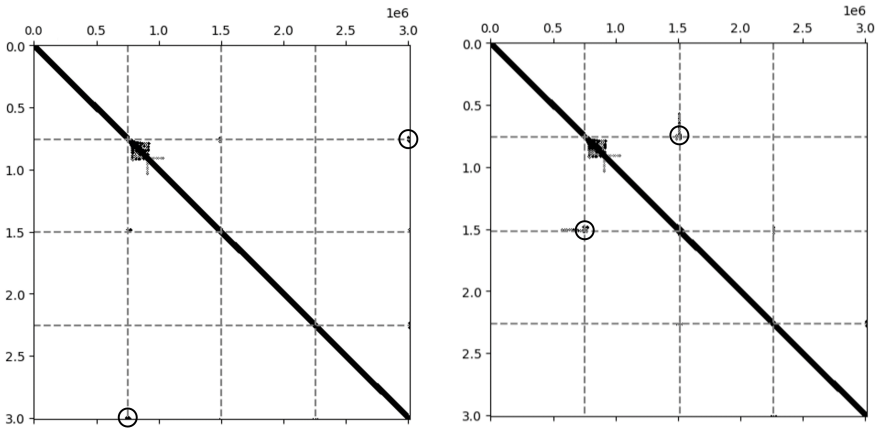
Although using more than one aggressive coarsening operation gives a further improvement over only using one, this improvement is not large. For this reason, the recommendation given by the SAMG Team [35] **(p.90)** is followed and only a single aggressive coarsening operation will be used for the experiments covered below.

5.4.4 Use of unknown redistribution

For efficient application of SAMG with parallel computing methods, it is essential to have a good domain partitioning. To improve this, ParMETIS can be used to redistribute unknowns over the used processors. The downside of the use of redistributing the unknowns is a more complicated setup of the method being required along with the requirement to copy the matrix an extra time, which results in a higher peak memory requirement.

Figure 5.6a gives an example of how a domain partition is applied to one of the matrices used in the experiments, with four MPI processes. This is the same matrix that was studied in Section 5.1.2, obtained from the GUL-01M-ST problem. The domain partition of the matrix over the four MPI processes is illustrated by the dashed lines. Blocks on the diagonal represent interactions within a single MPI process, blocks not on the diagonal represent interactions between different processes. Nonzero entries located in the off diagonal blocks results in the unknowns corresponding to the nonzero entry being in the halo. To reduce the communications between MPI processes, the size of this halo has to be kept as small as possible.

By reordering the unknowns, the size of the halo can be decreased. On this particular matrix, the influence of using matrix reordering is not large, as most of the entries in



(a) (b)

Figure 5.6: Overview of domain partition and changes made to it by redistributing the unknowns using ParMETIS on the GUL-01M-ST with 4 MPIs. (a) First the original domain partition is shown, (b) then partition after application of redistributing.

the halo are located in the locations where unknowns corresponding to the diagonal bands are distributed over the different processes. These locations are signified by the main diagonal band crossing the dotted lines into a block corresponding to a different process. Redistributing these unknowns typically does not reduce the halo, as for each unknown removed from the halo, a new one is added back in.

In some locations in this matrix, advantages can be obtained from reordering the unknowns. The elements in the matrix that correspond to this are marked by circles. Through the reordering of unknowns, these elements are moved to a new location in the matrix, as seen in Figure 5.6b. This means that some of the unknowns have been removed from the halo.

The effect of redistribution of unknowns is small for this problem. In larger problems, many more unknowns are located in the halo, which means that the effect of the redistribution of unknowns can be much bigger.

To investigate the possible advantages of redistributing the unknowns of the matrix and to determine whether the use of it is necessary to effectively apply the SAMG method, a comparison is made between the results obtained from experiments performed with and without the use of ParMETIS. Models of different sizes are investigated using various amounts of MPI processes. For all investigated cases, the default configuration of MPI processes, as described in Section 4.4.2, is investigated. Along with this, a select few other configurations are investigated for some of the models. By doing this, it can be confirmed that the possible advantages of redistributing the unknowns are more prevalent because of the problem size or because of the number of MPI processes.

Table 5.7 shows the problems, the investigated configurations of MPI processes for these problems and the sizes of the halo before and after the application of the reordering. There, N_{halo} refers to the number of unknowns in the halo before the unknowns are reordered and $N_{\text{halo R}}$ refers to the unknowns after the reordering is

Model	N	nnz	n_{MPI}	N_{halo}	$N_{\text{halo R}}$	$\frac{N_{\text{halo R}}}{N_{\text{halo}}}$
GUL-ST-01M	3,008,569	237,477,647	$*$ 4	134,183	115,282	0.86
			8	305,001	234,212	0.77
			16	619,584	384,956	0.62
GUL-ST-05M	14,986,663	1,197,035,813	16	2,949,442	1,203,206	0.41
GUL-ST-10M	29,443,571	2,358,175,787	$*$ 16	5,782,311	1,885,235	0.33
			$*$ 32	11,596,341	2,778,817	0.24
			64	23,290,476	3,786,155	0.16
GUL-ST-20M	59,149,691	4,746,831,353	$*$ 64	46,402,661	6,048,025	0.13
YRK-ST	3,379,736	267,476,018	$*$ 4	326,998	161,966	0.50
GRO-ST	33,580,236	2,685,914,088	$*$ 32	22,565,420	2,746,335	0.12

Table 5.7: Reduction of size of halo with and without METIS on several models. $*$ marks the default configurations of MPI processes.

performed. Along with this, a rate is given of how much the halo is reduced by the reordering. This is obtained by dividing the number of elements in the halo after reordering by the number of elements originally in the halo. It is observed that the size of the halo roughly doubles when either the size of the problem doubles or the number of MPI processes doubles if no reordering is used. This is what is expected to be observed when the unknowns are not reordered. When reordering is used, the number of elements in the halo still increases when increasing the number of MPI processes or the problem size, but less significantly than without reordering. This results in the reordering being especially effective when applied to large problems which use high numbers of MPI processes. The reduction of the halo can be very large, over 80% in the problems seen here. A great reduction of the halo can be expected to have a positive influence on the runtime as it decreases the amount of communications between processors.

The results obtained from the experiments with and without the reordering of the unknowns are presented in Table 5.8. In this table, the column 'Reor.' indicates whether the reordering of the unknowns is used in the experiments, with experiments labelled 'N' for not using reordering and 'Y' for using reordering. The column marked by T_{reor} shows the time spent applying the matrix reordering and is zero for all experiments that do not use matrix reordering.

For the models of limited size, such as GUL-01M-ST, GUL-05M-ST and YRK, no advantage is obtained through the application of matrix reordering. Some small improvements are seen in the setup stage and iteration stage of the solvers for these models, but this is nullified by the extra time spent on the application of matrix reordering. This is in line with what was seen in Table 5.7, as the reduction in the halo on these models is small.

For the GUL-05M-ST problem, the runtime required for the application of the reordering is longer than seen on any of the other models. This causes the total runtime to be much higher than the observed runtime without reordering. The high runtime during the reordering of unknowns for this particular problem is unexpected.

On the larger models, the influence of matrix reordering is more noticeable. Especially on GRO-ST, GUL-20M-ST and GUL-10M-ST with 32 processes the time

Model	n_{MPI}	Reor.	T_{reor}	T_{sup}	t_{itr}	n_{itrs}	T_{sol}	T_{tot}	M_{AMG}	M_{tot}
GUL-01M-ST	4 ∗	N	0.0	7.4	0.439	137	61.0	68.5	1.7	9.8
		Y	2.7	7.2	0.443	139	61.6	71.7	1.6	10.6
	8	N	0.0	5.4	0.231	142	32.8	38.3	2.6	11.8
		Y	1.8	5.2	0.237	140	33.2	40.6	1.8	12.0
	16	N	0.0	4.7	0.136	145	19.7	24.7	1.9	13.4
		Y	4.2	5.5	0.132	141	18.6	29.0	3.1	15.7
GUL-05M-ST	16 ∗	N	0.0	26.4	0.625	122	76.3	104.4	16.2	73.6
		Y	21.3	16.4	0.686	116	79.6	123.5	15.7	77.7
GUL-10M-ST	16	N	0.0	51.5	1.257	117	147.1	199.5	31.4	144.1
		Y	10.3	34.8	1.231	118	145.3	191.2	30.3	152.3
	32 ∗	N	0.0	58.4	0.650	120	78.0	138.0	61.9	219.5
		Y	6.7	27.3	0.700	116	81.2	116.9	59.2	226.0
	64	N	0.0	71.8	0.446	131	58.4	131.5	122.9	370.2
		Y	55.9	27.1	0.477	115	54.9	141.6	115.3	371.8
GUL-20M-ST	64 ∗	N	0.0	142.9	0.911	123	112.1	261.2	251.5	748.2
		Y	9.3	39.9	0.721	116	83.6	136.1	235.4	750.6
YRK-ST	4 ∗	N	0.0	12.2	0.516	128	66.0	78.3	3.2	11.9
		Y	3.7	8.9	0.508	123	62.5	75.2	2.7	12.4
GRO-ST	32 ∗	N	0.0	88.1	0.803	78	62.6	152.0	74.2	234.4
		Y	8.9	26.1	0.676	69	46.6	82.6	66.3	236.7

Table 5.8: Results of problems of different size of GUL model, solved using SAMG with and without reordering the unknowns beforehand with ParMETIS. ∗ marks the default configurations of MPI processes.

required in the setup of the method is reduced significantly, at the cost of some extra time being spent on the process of reordering, which together gives a clear reduction to the total runtime. For GUL-10M-ST with 64 processes, a similar clear reduction in setup time is obtained, but the advantage of this is lost because of the amount of time required to reorder the matrix. This could suggest that, for this particular problem, no more than 32 processes should be used.

For all problems, it is observed that the maximum memory requirement in the application of the solver is slightly reduced by reordering the matrix. However, this advantage is lost because the extra copy of the matrix is made to apply the reordering of the unknowns.

In the end, the use of reordering of the unknowns can provide a significant advantage for some of the larger models and does not cause problems for smaller models. The memory requirement when applying the reordering is slightly higher, but this is insignificant. For these reasons, the reordering of the unknowns using ParMETIS will be applied if possible.

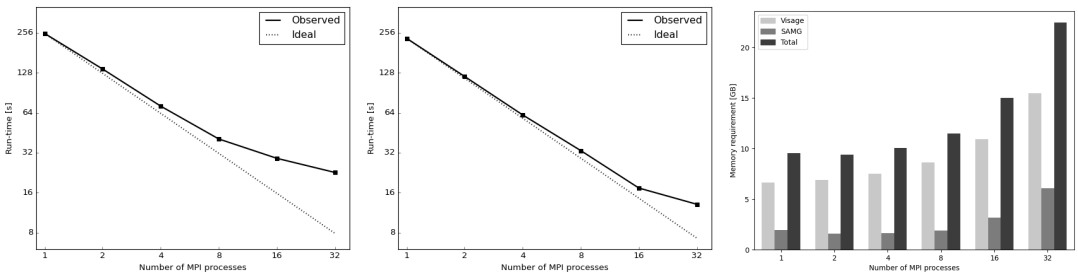
5.4.5 Strong Scalability

Similar to how this was done for the Jacobi preconditioned CG, the strong scalability of the linear solver obtained from SAMG will be studied as well. Again, experiments are performed on the GUL-01M-ST problem with repeatedly doubling numbers of MPI processes. For all configurations of MPI processes, the reordering of the unknowns using ParMETIS is applied to the linear system. For this small problem, the effect of reordering the unknowns is very small, so the results seen when applying matrix reordering are nearly the same.

Figure 5.7a shows the runtimes that are observed when using the SAMG method with different numbers of processes. Again, the dotted line presents the runtimes that would have been observed if the strong scalability of the method was perfect. It is observed that the strong scalability of the SAMG method is good for a low number of processes. For as much as eight processes, the observed runtime stays close to this line of ideal scalability. For more than eight processes very little improvement is observed when adding more processes.

Although the strong scalability of the entire method is not great for more than eight processes, the strong scalability of only the iterative solver is better. This is seen in Figure 5.7b, where the strong scalability of only the iterative solver stage is shown. The times shown correspond to the variable T_{sol}, as explained in Section 4.4. The much better strong scalability of the iterative solver suggests that the worse scalability of the full method can be largely attributed to the setup stage of the solver. In applications where several similar systems are used, it can be possible to reuse the setup of an earlier used AMG method. If this can be done, the scalability of the full method is closer to what is observed for only the iterative solver.

Lastly, Figure 5.7c shows the peak memory requirements for the different configurations of MPI processes used to observe the scalability of the methods. The peak memory requirements of the Visage simulator and the SAMG solver are shown separately along with the total memory requirement. It is again observed that using more MPI processes significantly increases the peak memory requirement, especially when using 16 or more processes. This increase is observed in the peak memory



(a) (b) (c)

Figure 5.7: Scalability of runtime and corresponding memory usage for solving linear problem using SAMG in one timestep of GUL-01M-ST using varying amounts of MPI processes. (a) The total runtime of the solver, (b) the runtime of only the iterative solver and (c) the memory requirement are shown here.

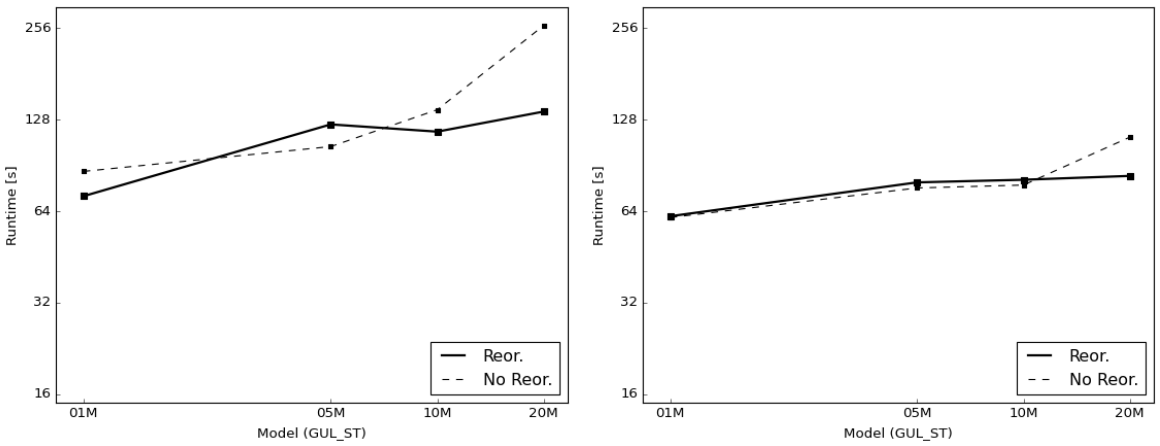
requirements of both Visage and SAMG.

5.4.6 Weak Scalability

A last aspect of the SAMG method that is considered is the weak scalability of the method. For this, the method is again applied both with and without the reordering of the unknowns. To study the weak scalability of the method, the solver is applied to discretisations of various grid sizes of the same model, in this case the GUL-ST model. For each of the problems, the default number of MPI processes is used, as discussed in Section 4.4.2.

To conduct an appropriate study of the weak scalability, it is desirable to use problems for which both the size of the problem and the number of MPI processes scale in the same way between different observed problems. Unfortunately, the available problems do not fully follow this. The smallest of the observed problems has roughly one million nodes and three million unknowns and is solved using 4 MPI processes. The second smallest problem, which has five million nodes and fifteen million unknowns and uses 16 MPI processes. This means that the second problem is five times the size of the smallest problem but only uses four times as many MPI processes. This means that the comparison between the smallest and second smallest problem is not completely valid. The results for the smallest problems are still included, as it gives some insight in the weak scalability, but should not be considered for the conclusions. For larger problems, the problems and the number of MPI processes used do scale in the desired way, with both doubling between each of the models.

Figure 5.8a gives an overview of the runtimes of the full method of SAMG for solving the linear problem with (SAMG) and without (No METIS) the use of matrix reordering. These results are the same as what was presented in Table 5.8. There,



(a) (b)

Figure 5.8: Scalability of runtime of SAMG with and without reordering of unknowns for solving the linear problems in GUL-ST models of increasing size. The scalability of (a) the total runtime required by the solver and (b) the runtime required in only the iterative solver are both shown.

it can be seen that the full method does not have great weak scalability, especially when matrix reordering is not applied. If the weak scalability would be ideal, the runtimes would be constant for the different problems, which is not the case for the SAMG method without matrix reordering. For the method with matrix reordering, the runtimes stay more close together, but it is hard to draw conclusions from this because of the unexpected result in the runtime of the application of ParMETIS for the 05M-problem, as seen in Table 5.8.

The weak scalability of only the iterative solver is much better. In Figure 5.8b a very good weak scalability is observed for the SAMG method with matrix reordering. The weak scalability of the iterative solver of the SAMG method without matrix reordering is also better than that of the full method, but there is still a significant increase in runtime observed in the largest problem.

5.5 hypre

The second method using an AMG preconditioner that is investigated is the BoomerAMG implementation of the AMG method provided through hypre. Here, this method is referred to most as 'the hypre method' or simply 'hypre'.

For fair comparison, it is attempted to apply this method using a setup that is as close as possible to the setup that was used during the investigation of SAMG.Like SAMG, hypre is applied using a single V-cycle. For the coarsening method, again, apart from in the investigation of different coarsening methods, A1-coarsening is used for the first coarsening operation and standard coarsening is used for the other coarsening operations. For the prolongation method, again unknown-based prolongation is used. Outside of the investigation of different smoother methods, a ℓ_1-Gauss-Seidel smoother is used. Parallel computing is used for the hypre method in the same way as for the other preconditioning methods. With hypre, if any parallel computing is used, reordering of the unknowns is applied.

An important note that has to be made about the method provided through hypre is that the output of certain information is not as clear as it is for SAMG and the originally used solvers. This especially gives difficulties in estimation of the peak memory requirement. Where the methods observed earlier gave clear estimates of the observed peak memory requirements, the only way to gain a similar estimate for the hypre method is by estimating it based on the percentage of total available memory used. The method used here to obtain this estimate gives percentages with an accuracy of one decimal, which means that there is some range of error in these estimates. The ranges, $[P_{\min}, P_{\max}]$, that estimate the peak memory requirements that are shown here are derived this percentage, with the minimum and maximum being obtained

$$M_{\min} = n_{\text{MPI}} \cdot (\alpha - 0.05) \cdot 1024$$
$$M_{\max} = n_{\text{MPI}} \cdot (\alpha + 0.05) \cdot 1024$$

where $P_{\min}$ and $P_{\max}$ are the minimum and maximum of the range respectively. α is the percentage of the total available memory of one of the used cores that is used in the application of the linear solver. This percentage is rounded to one decimal.

5.5.1 Choice of Smoother

Like SAMG, hypre also offers several different smoothers that can be used in the application of the AMG method. Because the SAMG method was applied using the ℓ_1-smoother, it is likely best for a fair comparison to apply the hypre method with the ℓ_1-Gauss-Seidel smoother as well. However, other smoothers are also investigated to confirm that the ℓ_1-Gauss-Seidel smoother also performs best for the hypre method. Unlike SAMG, hypre does not offer the option to use the AMG method as a preconditioner to other Krylov methods than the CG method. This is not a problem, as the CG method was also used with SAMG and is the method that is most desirable to use for the SPD problems encountered here.

For the smoothers investigated, once again the damped Jacobi, ℓ_1-Gauss-Seidel and ILUT smoother are approached. For the damped Jacobi method, once again the parameter used is $\omega = 0.4$. hypre does not offer the possibility to use a full Gauss-Seidel smoother when using multiple processes. This means that ℓ_1-Gauss-Seidel and Gauss-Seidel smoother can not be approached separately. The experiments performed here use only a single process, meaning that the observed ℓ_1-Gauss-Seidel smoother is the same as a full Gauss-Seidel smoother.

The results of the comparison of different smoothers are presented in Table 5.9. For this, the linear systems of the GUL-01M-00 and GUL-01M-ST problems are approached using a single MPI process. Because a single MPI process was used, the ℓ_1-Gauss-Seidel smoother is the same as the ℓ_1-Gauss-Seidel smoother.

It is observed that using the Gauss-Seidel smoother results in significantly faster convergence than using the Jacobi smoother. The number of iterations required for these methods to converge is for both these problems less than half of the number of iterations required to converge with the Jacobi smoother. This comes at the cost of the time required for one iteration being higher. In the end, both smoother methods were faster than the Jacobi smoother, but not by much. For this reason and because it was already used in the SAMG method, the ℓ_1-Gauss-Seidel method will also be used for the hypre method.

For the last smoother, the ILUT smoother, once again no convergence was obtained. This is similar to what was observed above in the corresponding investigation of SAMG.

Model	Smoother	T_{sup}	t_{itr}	n_{itrs}	T_{sol}	T_{tot}
GUL-01M-00	ω-Jacobi	8.47	1.123	112	125.8	139.3
	$\ell-$GS	8.73	1.915	54	103.4	117.3
	ILUT	No Convergence				
GUL-01M-ST	ω-Jacobi	8.38	1.095	320	350.3	363.6
	$\ell-$GS	8.77	1.864	141	262.8	276.5
	ILUT	No Convergence				

Table 5.9: Comparison of different smoothers for hypre on the initialisation and simulation models for GUL-01M

5.5.2 Coarsening method

The second element in the setup of the hypre method investigated is the coarsening methods and the use of aggressive coarsening. The study of the use of different numbers of coarsening methods is not carried out as extensively as was done for the SAMG method. Instead, only the use of no aggressive coarsening operations, one aggressive coarsening operation on the first multigrid level, and aggressive coarsening for every coarsening operation are approached. The method for aggressive coarsening operations is again A1-coarsening.

Along with this, the parameter for the threshold for strong connections is investigated. For the SAMG method, based on the information provided, the choice was made to use $\theta_{\text{ecg}} = 0.7$ for this parameter. hypre Project Developers [15] instead suggests a value of $\theta_{\text{ecg}} = 0.9$ for this threshold parameter. Both values are investigated for each different number of aggressive coarsening operations.

Table 5.10 shows the results of the use of these different configurations of coarsening methods. These results were obtained on the GUL-01M-ST problem with no parallel computing. Here, n_{AO} refers to the number of aggressive operations and n_L denotes the number of levels in the multigrid hierarchy. The last two rows, where 'All' is given for n_{AO}, use aggressive coarsening for all levels of multigrid hierarchy. ω_g and ω_a refer to the multigrid complexities discussed in Section 3.3.3. All other results covered here are as they are described in Section 4.4.

It is observed that the use of aggressive coarsening gives a significant improvement in runtime, similar to what was seen for the SAMG method. The number of iterations required remains roughly the same, but the runtime for a single iteration is significantly reduced. Together with the reduction in the setup time, this gives a clear reduction in total runtime. Again, the use of aggressive coarsening for every coarsening operation gives a small improvement over the use of a single aggressive coarsening operation. However, using more aggressive coarsening operations makes the method less stable and therefore it is advised not to use more than one coarsening operation **(hypre Project Developers [15])**. Thus, the choice is made to still use a single coarsening operation on only the highest level.

For the threshold parameter, the differences between the two choices for the parameter are small. The value of 0.7 requires fewer iterations of the linear solver, but the runtime of a single iteration is higher. This results in a shorter runtime for

n_{AO}	θ_{ecg}	T_{sup}	t_{itr}	n_{itrs}	T_{sol}	T_{tot}	n_L	ω_g	ω_a
0	0.7	18.21	2.702	137	370.14	393.41	11	1.7985	2.0885
	0.9	12.53	2.451	145	355.38	372.93	11	1.7790	1.8492
1	0.7	10.77	1.938	140	271.30	287.16	10	1.3659	1.4273
	0.9	8.64	1.861	141	262.37	275.93	10	1.3839	1.3819
All	0.7	8.39	1.752	143	250.54	264.07	7	1.2698	1.2634
	0.9	7.74	1.733	145	251.27	264.03	8	1.2974	1.2790

Table 5.10: Comparison of different methods of coarsening for hypre on the GUL-01M-ST model.

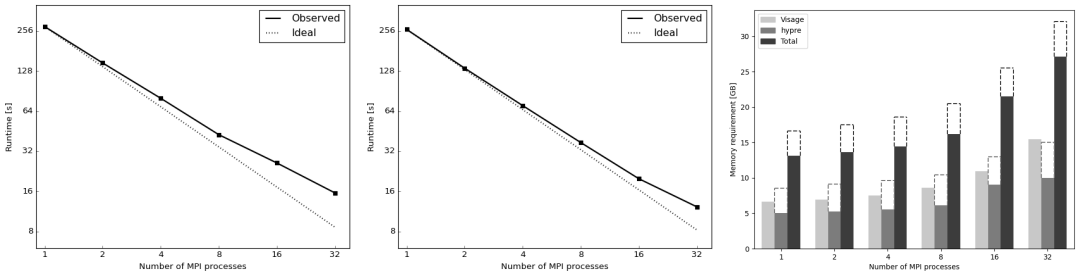
$\theta_{\text{ecg}} = 0.9$. Since $\theta_{\text{ecg}} = 0.9$ is the recommended value in hypre and the results it gives in this example are better, this is the value for θ_{ecg} that will be used here.

5.5.3 Strong Scalability

Last, the strong scalability of the hypre method will be investigated, in a similar way as in which this was done for the two other methods discussed above. Again, the linear solver method is applied to a particular problem, the GUL-01M-ST problem, with different numbers of MPI processes. The runtime observed with a single process is used as a benchmark for runtimes that would have been observed if the method had scaled perfectly.

In Figure 5.9a the runtimes of the solver using repeatedly doubling numbers of MPI processes are shown, together with the runtimes that would be observed if the method were to scale perfectly. There, it is seen that observed runtimes stay close to the line of ideal scalability. With more than eight MPI processes, the scalability of the full solver is less good. The scalability of only the iterative solver is better, as is shown in Figure 5.9b. This stays close to perfect upto 16 MPI processes and even for 32 MPI processes it scales close to perfectly.

The peak memory requirements of the hypre method observed for different numbers of MPI processes are shown in Figure 5.9c. The range of inaccuracy in the memory requirement is given by the dashed part. For eight or less processes, the differences in memory requirement are small. For more than eight processes, the memory requirement quickly increases. For 32 processes, the memory requirement is nearly double that of what is seen for less than eight processes.



(a) (b) (c)

Figure 5.9: Scalability of runtime and corresponding memory usage for solving linear problem using hypre in one timestep of GUL-01M-ST using varying amounts of MPI processes. (a) The total runtime of the solver, (b) the runtime of only the iterative solver and (c) the memory requirement are shown here.

5.6 PETSc

The last AMG-based method studied is the PCGAMG method provided by PETSc. However, for this method, it turned out not to be possible to obtain similar results as those seen for the other two methods. Due to limitations and the good results observed in the two methods discussed above, it was decided not to pursue this method beyond the stage in which some of the initial tests were performed. Although the PETSc software did not become useful as an AMG method, the implementation of PETSc was still useful in the spectral analysis of linear problems, as discussed in Section 5.2.

For the PCGAMG solver, the only available results are for the small GEE problem. These results are obtained on the local machine and do not use parallel computing. From these results, the expectation is that the PETSc method does not perform better than the other AMG methods or even the Jacobi preconditioner.

Table 5.11 shows the results of an experiment used to compare the PETSc solver with the other methods. First, it is observed that the setup of PETSc is much longer than that of the other methods. However, this might be attributed to an imperfect implementation of the PETSc method and cannot be used directly to draw conclusions. However, the run-time per iteration is also significantly higher than that of the other AMG methods, which results in a higher run-time of the iterative solver. This can not be attributed to poor implementation and instead suggests that the performance of the PETSc method is not as good as that of the other methods.

The peak memory requirement of the PETSc solver also seems higher than that of the other solvers. For this, it is difficult to draw a concrete conclusion, as the memory requirements found for both the hypre and PETSc methods are not fully accurate. These peak memory requirements are based on the memory usage reported during the application of these solvers, with a range of error. The given peak memory requirements of the Jacobi preconditioner and the SAMG method are accurate.

Preconditioner	T_{sup}	t_{itr}	n_{itrs}	T_{sol}	T_{tot}	m_{AMG}	m_{tot}
Jacobi	0.17	0.002	305	0.48	0.65	0.0	63.8
SAMG	1.04	0.017	17	0.28	1.33	29.0	141.1
hypre	0.10	0.024	6	0.15	0.29	29.0-58.0	141.1-170.0
PETSc	5.26	0.066	16	1.06	6.32	70.6-91.1	150.8-171.3

Table 5.11: Results on the smaller GEE for the four considered preconditioners.

5.7 Comparison of solvers

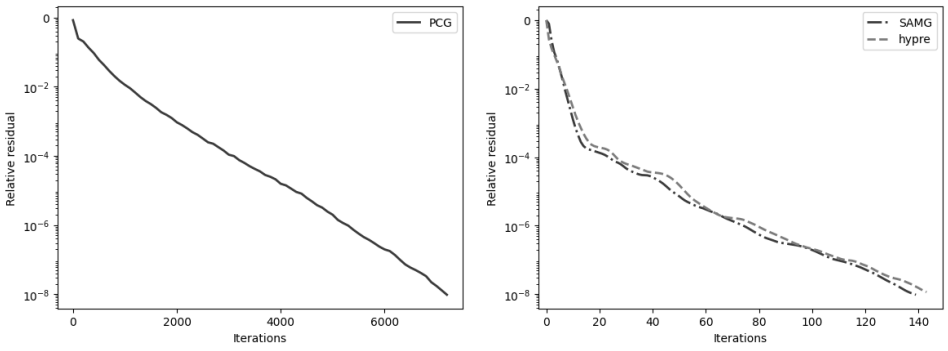
Above five different methods for preconditioning the CG method have been discussed, the two methods originally used in the Visage simulator and three methods based on Algebraic Multigrid. After the decision to use the Jacobi preconditioner over the deflation preconditioner and the decision to not use the AMG method provided through PETSc, three methods remain to be compared. These are the Jacobi preconditioned CG method that was originally used and the CG methods preconditioned by AMG as provided by SAMG and hypre. For a simple way to distinguish between the different methods, the three methods are in most cases referred to as 'Jacobi' or 'Jacobi PCG' for the Jacobi preconditioned CG method that was used originally in the simulator and 'SAMG' and 'hypre' for the two AMG based methods.

To compare these methods, first the convergence of the residual in the CG method for these different solvers is covered. Then, the performance of these methods on the different problems, with their default configurations of MPI processes, is discussed. After this, a comparison of the scalability of the methods, as discussed above, is made. Lastly, a short investigation is done in the performance of parallel computing using a combination of multiple MPI processes and multiple OpenMP threads.

5.7.1 Convergence of residual

The convergence of the residual is studied in the same way as was done in Section 5.3.1 for the Jacobi and deflation preconditioned CG methods. The relative residual is calculated at each iteration of the CG method as $\frac{\|\mathbf{r}_j\|}{\|\mathbf{r}_0\|}$ for iteration j.

Figure 5.10 shows the results of this on the GUL-01M-ST model for the three different preconditioned CG methods. It is difficult to make a direct comparison between the Jacobi preconditioned method and the two AMG method, since a single iteration of the Jacobi preconditioned CG method is vastly different from an iteration of an AMG preconditioned method. The methods that use an AMG preconditioner converge much faster, but a single iteration for these methods is computationally much



(a) (b)

Figure 5.10: Convergence of error on GUL-01M-ST for (a) Jacobi PCG and (b) the two AMG methods, SAMG and hypre.

more costly. For all methods faster convergence is seen in the first few iterations. The influence of this fast convergence is much more noticeable in the AMG methods, where very fast convergence is observed to a relative residual of 10^{-4}.

Between the two AMG methods, very little difference in their way of convergence is observed. For this particular example, the method using SAMG requires fewer iterations to converge, but the difference in total iterations is small. This also differs between different problems, with on some models hypre requiring less iterations and on some models SAMG requiring less iterations.

5.7.2 Performance on most commonly observed problems

For a direct comparison of how the methods would work in practice, the problems described above are approached using the default number of MPI processes, as described in Section 4.4. For the comparison of the methods the focus is kept on problems which are used in the simulation steps, marked by 'ST' here. In actual simulations, the solvers need to be applied to these problems multiple times, meaning that potential improvements on these problems will in practice be the most important. Although only the results on these models are discussed here, the results seen on the initialisation models are similar. The linear solver methods are applied using the setups discussed above.

Table 5.12 shows the results of the performed experiments on these problems with the three methods which are to be compared. All experiments were performed using parallel computing, with the number of MPI processes being as they are described in Section 5.1. The methods all make use of the CG method that is preconditioned differently. Because the implementation of the CG method may not be the exact same across the studied methods, there can be small differences in the iterative method, but these should not influence the results.

What is immediately evident from these results is that the total runtime required to apply the linear solver with the AMG methods is much lower than that of the Jacobi preconditioned method. Depending on the observed problem, the total runtimes of the AMG solvers can be as low as one fifth of the total runtime of the Jacobi PCG method. Especially in the stage of the iterative solver large improvements are seen for the AMG methods. The setup of the AMG solvers can in some cases, especially for the smaller problems, take longer than that of the Jacobi PCG method.

The AMG preconditioner is expected to lead to far less iterations being required for the CG method to converge than the Jacobi PCG method, at the cost of more time expensive iterations. Here, this is observed as the AMG methods require fewer iterations than the Jacobi PCG method, but each single iteration takes much longer. This still results in a much shorter total time required for the iterative solver.

The main downside of the AMG based methods is the higher peak memory requirement. This higher peak memory requirement was anticipated for these methods, as these methods require the use of a memory-expensive multigrid hierarchy. In most cases, the peak memory requirement of the methods using an AMG preconditioner is close to two times the peak memory requirement of the Jacobi PCG method on the same problem. In some cases, this can even be higher, especially for the hypre solver.

Model	Method	T_{sup}	t_{itr}	n_{itrs}	T_{sol}	T_{tot}	M_{AMG}	M_{tot}
GUL-01M	Jacobi	4.0	0.040	6574	266.1	270.1	0.0	6.1
	SAMG	9.8	0.443	139	61.6	71.7	1.6	10.6
	hypre	8.3	0.489	144	70.4	79.9	5.5-9.6	14.4-18.6
GUL-05M	Jacobi	10.4	0.047	6679	311.3	321.7	0.0	46.9
	SAMG	37.7	0.686	116	79.6	123.5	15.7	77.7
	hypre	18.7	0.688	135	92.9	113.7	45.2-61.7	107.3-123.8
GUL-10M	Jacobi	20.2	0.048	6625	318.4	338.6	0.0	134.8
	SAMG	34.0	0.700	116	81.2	116.9	59.2	226.0
	hypre	20.0	0.702	123	86.3	108.7	80.7-113.7	247.6-280.6
GUL-20M	Jacobi	49.4	0.056	6858	382.5	431.9	0.0	442.2
	SAMG	49.3	0.721	116	83.6	136.1	235.4	750.6
	hypre	25.6	0.741	126	93.4	123.4	178.0-244.0	693.2-759.2
YRK	Jacobi	5.6	0.040	4575	182.3	187.9	0	5.9
	SAMG	12.4	0.508	123	62.5	75.2	2.6	11.9
	hypre	9.3	0.597	152	90.8	101.4	[1] 7.5-15.6	[1] 17.3-25.4
GRO	Jacobi	33.2	0.057	6748	384.2	417.4	0	136.9
	SAMG	34.6	0.676	69	46.6	82.6	66.3	216.9
	hypre	21.9	0.785	77	60.5	84.9	[1] 85.0-127.4	[1] 241.0-278.0

Table 5.12: Results on all observed models with the three different preconditioners, the originally used Jacobi, SAMG and hypre. The number of MPI processes used is the default number described in Section 4.4.2.
[1] Because of technical difficulties, no estimate of the memory requirement for hypre was obtained on the YRK and GRO models. The estimate shown here is obtained form the estimate of GUL models of similar sizes, combined with the results seen for SAMG.

Between the two AMG methods, SAMG and hypre, the differences are small. The setup of the hypre method is faster than that of the SAMG method, but the iteration stage of the SAMG method is faster. In the end, both methods are faster on some of the observed problems, but there is no clear difference as to for what type of problem which method is fastest. Especially in smaller models, the peak memory requirement of the hypre method is much higher than that of the SAMG method. On larger models, the differences between the two AMG methods are smaller, with the peak memory requirement of the hypre method in some of the large models possibly being lower. It again has to be noted that there is some range of inaccuracy in the peak memory requirements of the hypre method due to the inaccurate method of obtaining this number described above.

5.7.3 Comparison of scalability

For each of the methods considered, investigations have been carried out on the scalability of the method. Here, the scalability of the three preconditioning methods is compared. This is used to determine how well parallel computing works for each of these methods.

5.7.3.1 Strong Scalability

The first type of scalability that is approached is the strong scalability of the different methods. This is covered in Section 5.3.2, Section 5.4.5 and Section 5.5.3 individually. Here, these results are combined and the strong scalability of the methods is compared. Again, the problem on the GUL-ST-01M model is observed with varying numbers of MPI processes.

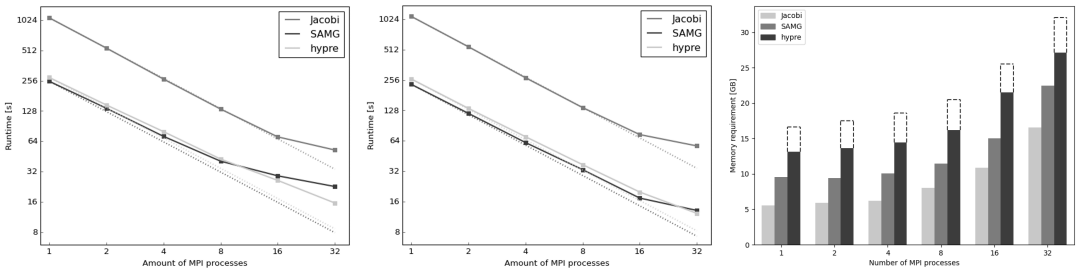
In Figure 5.11 the results of solving the same problem with the three solvers and an increasing number of MPI processes are shown. This is a combination of the three figures, Figure 5.4, Figure 5.7 and Figure 5.9.

Figure 5.11a shows the runtimes of the solvers for a increasing numbers of MPI processes. It is again seen that the AMG methods give a significant improvement in terms of runtime over the Jacobi PCG method and that this is seen for all numbers of MPI processes. The scalability of the AMG methods is not as good as that of the Jacobi PCG method. Where the Jacobi PCG method scales well up to 16 processes, the AMG methods only scale well up to eight processes. Even though the Jacobi PCG method scales better, for no possibly used number of MPI processes is it faster or expected to be faster than the AMG methods.

Although the full methods do not scale so well for the AMG methods, the iterative solver scales nearly as well as the iterative solver of the Jacobi PCG method, as seen in Figure 5.11b. This means that the main reason for the poorer scalability of the AMG methods is the more expensive setup of the methods, which was already seen in Section 5.7.2.

Increasing the number of MPI processes causes an increase in the peak memory requirements, with similar increases being seen across the methods. In Figure 5.11c, it is seen that the results are similar between the changing number of MPI processes. For each experiment, the peak memory requirement of the AMG methods is higher than that of the Jacobi PCG method. The only difference between different numbers of processes is that the increase of the peak memory requirement in the application of AMG methods is not as large for configurations with more MPI processes.

The differences in scalability between the two AMG methods are small. For low



(a) (b) (c)

Figure 5.11: Scalability of runtime and corresponding memory usage for solving linear problem using the three different methods, Jacobi, SAMG and hypre. All are applied to one timestep of GUL-01M-ST using varying amounts of MPI processes. (a) The total runtime of the solver, (b) the runtime of only the iterative solver and (c) the memory requirement are shown here.

numbers of MPI processes, SAMG seems to scale better and stays very close to ideal scalability, especially when only the iterative solver stage is considered. hypre scales better on larger numbers of MPI processes, and the trajectory of the scalability of hypre is very close to being logarithmically linear. For 16 or more processes, hypre even clearly outperforms SAMG.

5.7.3.2 Weak Scalability

Next, the weak scalability of the linear solver methods is considered. This is done in a similar way to the way it was done before for the SAMG method in Section 5.4.6. The results of the different solvers on models of the same problems with discretisations of varying sizes are approached. Each problem is solved with a scaling number of MPI processes, which are the default numbers of processes used throughout this discussion. The weak scalability of a method is then considered to be good if the runtime required across the problems of changing size remains similar.

Figure 5.12a shows the runtimes compared with each other for these solvers. These results are the same as what is given in Table 5.12 for the total runtime of the solvers. It is seen that the weak scalability of none of the methods is great, the runtimes for all solvers clearly increase when approaching the same problem with more fine discretisations and a scaling number of MPI processes. However, this increase is not large which means that the scalability of these methods is not poor either.

For the AMG methods, a decrease is observed when going from the 05M model to the 10M model. For SAMG, this is likely because of the poor application of METIS which gave a significant increase to the runtime for this problem. For hypre, the unexpected high runtime is also observed when only considering the iteration stage. This suggests poor performance of the hypre method on the specific problem obtained from the 05M

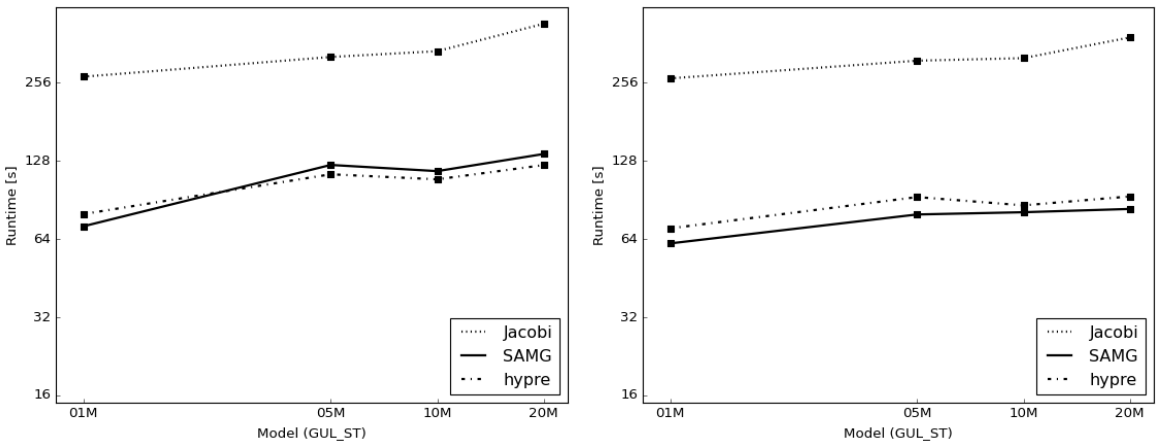


Figure 5.12: Scalability of runtime of the Jacobi PCG, SAMG and hypre method, for solving the linear problems in GUL-ST models of increasing size. The scalability of (a) the total runtime required by the solver and (b) the runtime required in only the iterative solver are both shown.

model.

The runtimes of only the iteration stage of the solvers are visible in Figure 5.12b. There, it is seen that for the iteration stage the AMG methods scale very well and better than the Jacobi PCG method. The only point where either of the methods does not scale well is the 05M problem for the hypre method.

5.7.4 Hybrid MPI + OpenMP

The results that have been obtained using parallel computing so far have all been obtained using methods that purely use MPI processes. It is also possible to use both multiple MPI processes and OpenMP threads in hybrid configuration. This can result in large differences in the performance of certain methods.

In most of the experiments, only parallel methods using multiple MPI processes were approached. Initial tests showed that parallelisation purely with multiple MPI processes gave the largest decrease in required runtime for the solvers. Because the focus was on decreasing the runtime as much as possible, the use of multiple OpenMP threads was not researched deeply.

Several experiments into hybrid configurations of multiple MPI processes and multiple OpenMP threads were conducted. These were performed on the GUL-10M-ST problem and always used 32 cores in total, which is the default number of cores described for this problem. The 32 cores were distributed in different ways over MPI processes and OpenMP threads.

The results of this are shown in Table 5.13. There n_{MPI} and n_{OMP} refer to the number of MPI processes and the number of OpenMP threads used respectively. The different configurations were applied to the Jacobi PCG method and the two AMG preconditioned CG methods.

It can be seen that the use of a hybrid configuration gives a longer runtime for all of the solvers. In general, as the number of OpenMP threads increases and thus the

Prec.	n_{MPI}	n_{OMP}	T_{sup}	t_{itr}	n_{itrs}	T_{AMG}	T_{tot}	M_{AMG}	M_{tot}
Jacobi	32	1	20.2	0.048	6625	318.4	338.6	0.0	134.8
	16	2	20.5	0.055	6625	365.8	386.3	0.0	93.4
	8	4	24.0	0.058	6625	382.3	406.2	0.0	74.0
	4	8	38.8	0.072	6625	477.6	516.5	0.0	66.9
SAMG	32	1	34.0	0.700	116	81.2	116.9	59.2	226.0
	16	2	52.3	0.983	121	118.9	173.9	30.3	153.7
	8	4	67.1	0.996	117	116.5	186.8	23.9	126.7
	4	8	84.3	1.320	116	153.1	241.0	22.7	118.0
hypre	32	1	20.1	0.702	123	86.3	108.7	80.7-113.7	247.6-280.6
	16	2	24.7	0.803	135	108.5	137.5	82.9-99.4	206.3-222.8
	8	4	36.0	0.822	133	109.4	150.7	74.5-82.8	177.4-185.7
	4	8	54.9	1.176	136	160.0	228.3	71.9-76.0	167.1-171.2

Table 5.13: Results obtained using different hybrid configurations of MPI and OpenMP on the GUL-10M-ST model

number of MPI processes decreases, the runtime of the solvers increases as well. For the configuration with the most OpenMP threads and the least amount of MPI processes, the runtime is nearly double the runtime of pure MPI methods. Both the setup time and the iterative solver time are much higher for configurations with fewer MPI processes, with an especially steep increase in the setup time for the AMG methods.

The advantage of using a hybrid configuration is the significantly lower peak memory requirement. Using more OpenMP threads and fewer MPI processes gives a significant decrease in the peak memory requirements for all three methods. The method with the most OpenMP threads observed here even cuts the peak memory requirement in half when compared with the pure MPI configuration.

The differences between the different solver methods remain the same across the different configurations of MPI processes and OpenMP threads. The AMG methods are much faster and have a higher peak memory requirement than the Jacobi PCG method. What is interesting is that some of the configurations give an improvement over the originally used solver in both these areas. The configuration of eight MPI processes and four OpenMP threads with the SAMG method has a lower peak memory requirement than the originally used Jacobi PCG method, while also requiring less than half the runtime.

5.8 Improvement in coupled simulations

In the experiments discussed in this chapter, only the Visage geomechanics simulator was considered. The results shown only consider the linear solver method in this simulator. From these results, it is concluded that the AMG preconditioners can give major advantages in simulations performed by the Visage simulator.

However, so far no results have been shown that represent the actual application for which the Visage simulator is practically used. The approached AMG-preconditioners are mainly considered for simulations in which the Visage simulator is coupled with the Intersect simulator. There, the Intersect simulator performs a simulation on the flow of fluid or gas in a reservoir and Visage performs the corresponding simulation of the stresses in the rock surrounding the reservoir.

These simulations were not studied in this research, which means that the available results are limited. However, the results shown here should give a good insight in the improvement obtained from the application of the AMG preconditioner to these simulations.

In Figure 5.13 the runtimes of such a coupled simulation can be seen. The stress simulation performed by Visage in this project uses 2.3 million cells, which leads to around 6.9 million unknowns. In this project, eight load steps are performed. Parallel computing is used with 16 MPI processes. The AMG preconditioner used in the Visage simulator in this simulation is the method provided by SAMG.

It is observed that in the simulation that uses the Jacobi preconditioner, the runtime is severely dominated by the Visage simulator. More than 90% of the total runtime is spent on the Visage simulation. In the simulation that uses the AMG preconditioner, the runtime of the Visage simulator is decreased to a fourth of the runtime of the simulation with the Jacobi preconditioner. This leads to the total simulation being decreased to a third of what was observed previously.

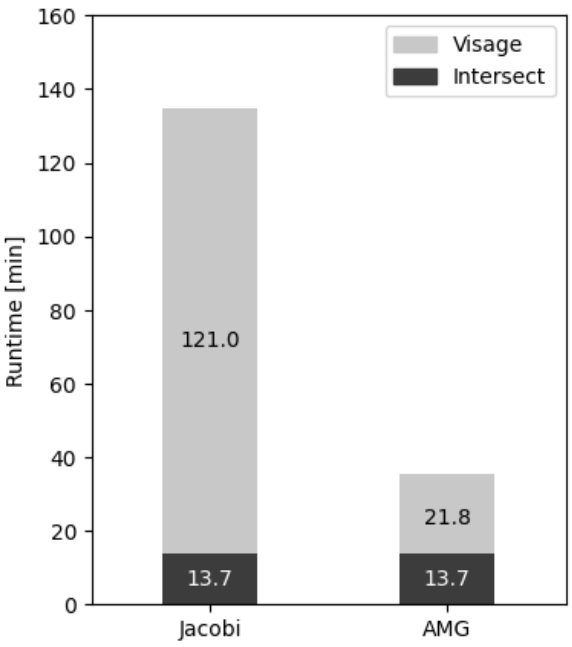


Figure 5.13: Runtimes of Visage and Intersect in coupled simulation using the Jacobi preconditioned CG method and the AMG preconditioned CG method using the SAMG method on a particular problem

6 Summary and Conclusions

To form an accurate model of subsurface deformations, geomechanical simulations are an essential part. The main problem with these simulations is their high computational costs in large simulations, which results in high runtimes of the simulations. In this research, the computational cost of these geomechanical simulations is studied, with a specific focus on geomechanical models of reservoirs with a high flow of fluids or gasses. This study of geomechanical simulations is done through the Visage geomechanical simulator.

In these simulations, the global equilibrium of internal stresses and external forces is attempted to be solved. This continuous problem is discretised using the Finite Element Method. This discretisation leads to a nonlinear problem, to which an approximate solution is obtained using the Newton-Raphson method. In the application of the Newton-Raphson method, a Symmetric Positive Definite linear problem is required to be solved.

Solving linear problems with Symmetric Positive Definite matrices can be done in various ways. In small models, a direct method can be applied to obtain an exact solution to the linear problem. However, for the large models that are typically observed in geomechanical simulations, these methods are rarely applicable. Instead, an approximation of the solution of the large linear systems is obtained using iterative methods.

The main method that is considered for iteratively solving large linear problems is the Conjugate Gradient method. This method is typically the most efficient method for solving large Symmetric Positive Definite linear problems. To further improve the convergence of this method, a preconditioner can be applied to the linear system which is solved using the Conjugate Gradient method. For these preconditioner methods, various methods exist, with more advanced methods giving bigger improvements, but being more difficult to apply.

In this research, the Algebraic Multigrid method is studied to be used as a preconditioner to the Conjugate Gradient method. This method is able to give large improvements to speed of convergence of the Conjugate Gradient method, but requires a problem-specific setup configuration to be used effectively. It is studied how this preconditioner is applied effectively within the geomechanical simulation.

The Algebraic Multigrid method is approached through different software distributions, namely those in SAMG, hypre and PETSc. Experiments with the preconditioner methods obtained from these software distributions are performed using the Visage simulator. For these experiments, problems with at least three million unknowns are approached. The experiments are carried out on a state-of-the-art High Performance Computing cluster.

On the basis of the results of these experiments, a comparison of the different preconditioning methods is made. The Algebraic Multigrid methods are compared with

each other and with the Jacobi preconditioner, which is the preconditioning method previously used by the Visage simulator. For this comparison, the focus is on the runtimes and peak memory requirements of the different solver methods in the carried out experiments.

For one of the three AMG methods, the PETSc method, it was not possible to make a complete comparison with the other two methods. It was, in this research, not possible to apply this method to the approached linear problems. The other two methods are applicable to the approached problems and the results are compared to the Jacobi preconditioner.

It is observed that Algebraic Multigrid preconditioners can give a significant improvement over Jacobi preconditioners. A massive reduction of the required number of iterations is observed, with the use of the Algebraic Multigrid preconditioners leading to up to 80 times less iterations being required for the Conjugate Gradient method to converge. While a single iteration of the Algebraic Multigrid preconditioned method requires a longer runtime, the total runtime of the linear solver method is a lot shorter. For the total runtime, improvements of up to five times are observed by the application of the Algebraic Multigrid method. A downside of the Algebraic Multigrid preconditioners is the increased peak memory requirement. The peak memory requirement for the Algebraic Multigrid method is in the worst case double that of the Jacobi preconditioned method.

6.1 Recommendations on Linear Solvers

Based on this research, the main recommendation regarding the application of the Conjugate Gradient method in linear problems obtained through geomechanical simulations is to use the Algebraic Multigrid preconditioner. This preconditioner gives significant improvements over the Jacobi preconditioner in the observed linear problems obtained from large geomechanical simulations.

In terms of runtime, the AMG preconditioners are much more efficient. Runtimes with the AMG preconditioners are as low as a fourth of the runtime of the Jacobi preconditioner. The reduction of runtimes can be even bigger when only the time of the iterative solver is considered. The time to set up the AMG-based preconditioners is typically higher, because the preconditioner is more complicated.

The downside of AMG preconditioners is their higher peak memory requirement. In most cases, the memory requirement of the linear solver is doubled when applying the AMG-based preconditioners. This means that these preconditioners might not be applicable if the available memory of the used system is limited.

Between the two methods for which results are obtained on the observed problems, the observed differences are small. The runtimes of these methods are, in most cases, very similar. Some differences in peak memory requirements are observed, with the SAMG method typically having a significantly lower requirement on smaller models. However, on larger models this difference is not observed. An advantage of the SAMG method is the more accurate estimate of the memory requirement that is obtained. An advantage of the hypre method is the potential use of GPU acceleration. This might reduce the runtime even further, but this has not been approached in this research.

The method provided through PETSc is only considered in a small model, where it does not give better results than the other two AMG preconditioners. In this research, this leads to the suggestion to not consider PETSc further. PETSc does have the advantage of being useful in the spectral analysis of problems. Furthermore, if it can be applied to the observed problems, it also provides the possibility to use GPU acceleration.

For the specific configurations of the AMG methods, different options are approached. In most cases, the result of this was that the options recommended by the distributors of the AMG methods were the best performing.

Large reductions in runtime can be obtained through the use of parallel computing. This was primarily applied using multiple MPI processes. The AMG preconditioners generally scale well when using parallel computing. The Jacobi preconditioner scales even better than the AMG preconditioners, but this method is for no number of MPI processes faster than the AMG preconditioners.

When parallel computing is used with the AMG preconditioners, crucial advantages are obtained through the redistribution of the unknowns using ParMETIS. In the largest models, where the highest number of MPI processes is used, the AMG methods perform much worse without the use of unknown redistribution. In these models, it is recommended to use such a redistribution of the unknowns. On smaller models with fewer MPI processes, the advantage is smaller and can even be not noticeable.

The use of parallel computing with multiple OpenMP threads has so far not been thoroughly researched. For a small set of experiments, the hybrid use of multiple OpenMP threads and MPI processes has been approached. From this it is observed that these hybrid configurations are not as fast as pure MPI configurations but have lower peak memory requirements.

6.2 Future Research

In this report, different ways of applying AMG-based preconditioners are presented. In the future, more research can be done to further improve these preconditioners.

The main concept that can possibly provide further improvements is the application of GPU acceleration. The hypre method that was investigated here provides this possibility, but this was not approached so far. In many other applications the use of GPU acceleration gives significant advantages and it might do so as well in the application of AMG-based preconditioners.

Next, it might be worth considering further investigating the use of hybrid configurations in parallel computing. So far these showed a higher runtime, but lower memory requirements. If a trade-off between these two has to be made, hybrid configurations could give advantages.

Finally, the use of AMG-based preconditioners is currently only considered in geomechanical simulations coupled with the simulations of fluids or gasses in reservoirs, but it is likely useful in other geomechanical applications as well. The Visage simulator has several other applications, in which AMG-based preconditioners can provide advantages.

Bibliography

[1] Allawi, R. H., & Al-Jawad, M. S. (2021). Wellbore instability management using geomechanical modeling and wellbore stability analysis for zubair shale formation in southern iraq. *Journal of Petroleum Exploration and Production Technology*, *11*, 4047–4062.

[2] Amdahl, G. M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. *Proceedings of the April 18-20, 1967, spring joint computer conference*, 483–485.

[3] Baker, A. H., Falgout, R. D., Kolev, T. V., & Yang, U. M. (2011). Multigrid smoothers for ultraparallel computing. *SIAM Journal on Scientific Computing*, *33*(5), 2864–2887.

[4] Balay, S., Abhyankar, S., Adams, M. F., Benson, S., Brown, J., Brune, P., Buschelman, K., Constantinescu, E. M., Dalcin, L., Dener, A., Eijkhout, V., Faibussowitsch, J., Gropp, W. D., Hapla, V., Isaac, T., Jolivet, P., Karpeev, D., Kaushik, D., Knepley, M. G., ... Zhang, J. (2024). PETSc Web page. https://petsc.org/

[5] Brandt, A., Brannick, J., Kahl, K., & Livshits, I. (2015). Bootstrap algebraic multigrid: Status report, open problems, and outlook. *Numerical Mathematics: Theory, Methods and Applications*, *8*(1), 112–135.

[6] Briggs, W. L., Henson, V. E., & McCormick, S. F. (2000). *A multigrid tutorial*. SIAM.

[7] Cleary, A. J., Falgout, R. D., Henson, V. E., Jones, J. E., Manteuffel, T. A., McCormick, S. F., Miranda, G. N., & Ruge, J. W. (2000). Robustness and scalability of algebraic multigrid. *SIAM Journal on Scientific Computing*, *21*(5), 1886–1908.

[8] Clees, T. (2005). *Amg strategies for pde systems with applications in industrial semiconductor simulation*. Shaker.

[9] Falgout, R. D. (2006). *An introduction to algebraic multigrid* (tech. rep.). Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States).

[10] Fossen, H. (2016). *Structural geology*. Cambridge university press.

[11] Fraunhofer SCAI. (2024). Samg - efficiently solving large linear systems of equations. Retrieved September 16, 2024, from https://www.scai.fraunhofer.de/en/business-research-areas/fast-solvers/products/samg.html

[12] Fredrich, J. T., Arguello, J., Deitrick, G., & de Rouffignac, E. P. (2000). Geomechanical modeling of reservoir compaction, surface subsidence, and casing damage at the belridge diatomite field. *SPE Reservoir Evaluation & Engineering*, *3*(04), 348–359.

[13] Gustafson, J. L. (1988). Reevaluating amdahl's law. *Communications of the ACM*, *31*(5), 532–533.

[14] Horn, R. A., & Johnson, C. R. (2012). *Matrix analysis*. Cambridge university press.

[15] hypre Project Developers. (2024). *hypre Documentation*. Lawrence Livermore National Laboratory.

[16] Jönsthövel, T. B. (2012). *The deflated preconditioned conjugate gradient method applied to composite materials* [Doctoral dissertation, Delft University of Technology].

[17] Jönsthövel, T. B., Van Gijzen, M., MacLachlan, S., Vuik, C., & Scarpas, A. (2011). Comparison of the deflated preconditioned conjugate gradient method and parallel direct solver for composite materials. *Reports of the Department of Applied Mathematical Analysis, 11-05*.

[18] Karypis, G., & Schloegel, K. (2024). Slepc, scalable library for eigenvalue problem computations. https://github.com/KarypisLab/ParMETIS?tab=readme-ov-file
[19] Lawrence Livermore National Security. (2024). Documentation for hypre. Retrieved September 16, 2024, from https://hypre.readthedocs.io/en/latest/ch-intro.html
[20] Li, X., Li, Q., Bai, B., Wei, N., & Yuan, W. (2016). The geomechanics of shenhua carbon dioxide capture and storage (ccs) demonstration project in ordos basin, china. *Journal of Rock Mechanics and Geotechnical Engineering*, *8*(6), 948–966.
[21] Metz, B., Davidson, O., De Coninck, H., Loos, M., & Meyer, L. (2005). Carbon dioxide capture and storage. summary for policymakers.
[22] Oosterlee, C. W., & Washio, T. (1998). An evaluation of parallel multigrid as a solver and a preconditioner for singularly perturbed problems. *SIAM Journal on Scientific Computing*, *19*(1), 87–110.
[23] Rauber, T., & Rünger, G. (2013). *Parallel programming*. Springer.
[24] Roman, J., Campos, C., Dalcin, L., Romero, E., & Tomás, A. (2024). Slepc, scalable library for eigenvalue problem computations. Retrieved September 16, 2024, from https://slepc.upv.es/
[25] Ruge, J. W., & Stüben, K. (1987). Algebraic multigrid. In *Multigrid methods* (pp. 73–130). SIAM.
[26] Saad, Y. (2003). *Iterative methods for sparse linear systems*. SIAM.
[27] Saad, Y. (2011). *Numerical methods for large eigenvalue problems: Revised edition*. SIAM.
[28] SLB. (2023). *Visage Finite-element geomechanics simulator*. Schlumberger Oilfield UK Limited.
[29] SLB. (2024a). Intersect high-resolution reservoir simulator. Retrieved September 18, 2024, from https://www.slb.com/products-and-services/delivering-digital-at-scale/software/intersect-high-resolution-reservoir-simulator/intersect
[30] SLB. (2024b). Visage finite-element geomechanics simulater. Retrieved September 18, 2024, from https://www.slb.com/products-and-services/delivering-digital-at-scale/software/visage
[31] Stüben, K. (1999). *Algebraic multigrid (amg). an introduction with applications*. GMD Forschungszentrum Informationstechnik.
[32] Stüben, K., et al. (2001). An introduction to algebraic multigrid. *Multigrid*, 413–532.
[33] Stüben, K. (2001). A review of algebraic multigrid. *Numerical Analysis: Historical Developments in the 20th Century*, 331–359.
[34] Süli, E. (2012). Lecture notes on finite element methods for partial differential equations. *Mathematical Institute, University of Oxford*.
[35] the SAMG Team. (2024). *SAMG User's Manual*. Fraunhofer SCAI. Schloss Birlinghoven 1, 53757 Sankt Augustin, Germany.
[36] Timoshenko, S., & Goodier, J. (1951). *Theory of elasticity*. McGraw-Hill.
[37] Vanek, P., Mandel, J., & Brezina, M. (1996). Algebraic multigrid by smoothed aggregation for second and fourth order elliptic problems. *Computing*, *56*(3), 179–196.
[38] Vuik, C. and Lahaye, D.J.P. (2019). *Scientific Computing (wi4201) Lecture Notes*.
[39] Zienkiewicz, O. C., & Taylor, R. L. (2005). *The finite element method set*. Elsevier.
[40] Zienkiewicz, O. C., & Taylor, R. L. (2000). *The finite element method: Solid mechanics* (Vol. 2). Butterworth-heinemann.