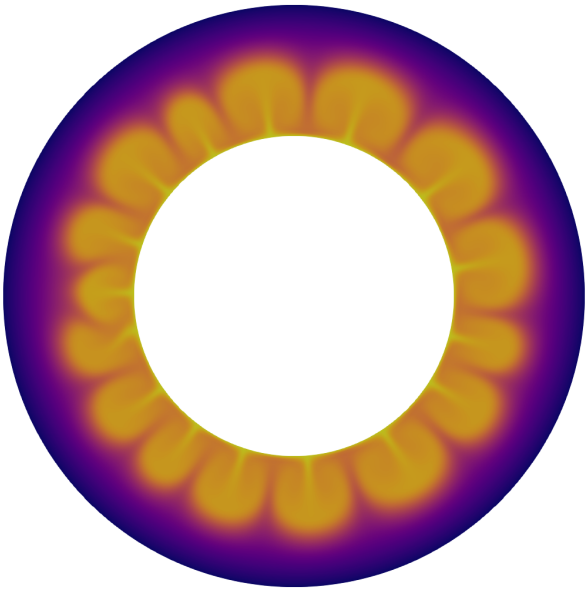

Master Thesis

Performance optimization of solving methods in ELEFANT and development of simPyFEM: An efficient educational tool for geodynamic simulations using Python

Job Mos BSc

Supervisor: dr. Cedric Thieulot

Second supervisor: Lukas van de Wiel MSc

Utrecht University

March 1, 2018

Abstract

Numerical models are used to develop a better physical understanding of small and large scale geodynamic processes over geological time scales. Through continuous development of computational techniques and methods, the accuracy and resolution of the models increases each year. Simulating geodynamic processes such as mantle convection or subduction requires the solve of a linear system in which efficient solvers and preconditioners can be used to yield low computation times and thus optimal performance. Many state of the art codes are fully parallel but even then can require multiple days or weeks to run large simulations. Implementing an efficient solver and preconditioner into ELEFANT is the first step into optimizing its performance. A free and open source package for scientific computing (PETSc) is used to facilitate this process. The second part of this thesis consists of the development of an educational tool for geodynamics called simPyFEM. By translating and optimizing a pre-exisiting code (simpleFEM) from Fortran to Python, its usability and versatility are improved. This modernization is inviting to students who are new to geodynamic modelling and want to experience the effect, on an educational level, of changing parameter such as viscosity, density and gravity.

Contents

1 Introduction

1.1 Geodynamic modelling

Through the interpretation of geological, geophysical, geochemical and other data we gain a better understanding of the fundamental geodynamic processes linked to the tectonic evolution of the Earth. Reconstructing and simulating tectonic progression of Earth's history (Van Hinsbergen et al., 2012) allows us to develop an explanation for the formation and evolution of both small and large scale structures and dynamical processes. The scale of these structures ranges from grain-size evolution (Thornton, 2000) to 10-meter scale shear zones (Mancktelow, 2002) to whole Earth mantle convection (Schubert et al., 2001; Ismail-Zadeh and Tackley, 2010; Zhong et al., 2007).

Tomographical data, obtained through seismic stations, reveals the relative velocity structure of the Earth (Bijwaard et al., 1998). This structure can be linked to density and temperature of convecting mantle material. Combining these observations and interpretations with laboratory High Pressure/Temperature (HPT) experiments (Murakami et al., 2004), geological data logged in the field and paleomagnetic data (de Groot et al., 2016) are the basis of producing accurate numerical models. Ultimately, a starting point is required when developing a model (e.g. initial slab angle and plate motion velocity when simulating plate subduction).

Numerical models are able to simulate both (in time and space) small and large scale processes employing forward modelling. Contrary to seismics, where a model is produced out of a set of observational data (Floricich et al., 2011), numerical modelling is grounded by a set of equations, boundary conditions and an initial setup. These physically substantiated equations describe the interaction between a set of variables such as pressure, velocity, viscosity, density and temperature (a detailed explanation can be found in books such as Gerya (2010) and Lowrie (2007)). Consequently, the set of these variables allows for an interpretation of the resulting geodynamic behaviour.

The complexity of numerical models is increasing rapidly. Two-dimensional models were the first to arrive in the geodynamic modelling field, some of which are still in active development (Sopale (Fullsack, 1995), I2ELVIS (Gerya and Yuen, 2003), SULEC (Quinquis et al., 2011)), and evolved to complex pieces of software which are able to model thermomechanically coupled geodynamics in full 3D at high resolutions and small time steps showing large jumps in viscosity (Kronbichler et al., 2012; Zhong et al., 2000). Modern codes such as ASPECT (Kronbichler et al., 2012) and StagYY (Tackley, 2008) exploit mesh refinement (May et al., 2013), yin-yang type elements (Guerri et al., 2016), optimal solving strategies (May et al., 2016) and multi-core parallelism (Bangerth et al., 2012) further enhancing computational performance.

Continuous development of these types of techniques and constantly increasing needs on numerical modellers drives researchers to further improve performance and intricacy of pre-existing or newly developed models (May et al., 2013, 2014, 2015; Ho et al., 2015). Modelling geodynamical processes at high performance promotes exploiting high resolutions and thus allows for a more detailed description of the physical evolution. However, high resolution comes at the price of long computational time. Combining high resolution with large contrasts in viscosity, caused by plumes when modelling mantle convection, also requires switching from a direct solving technique to an iterative solver approach. Iteratively solving a system of equations facilitates the use of preconditioners which beneficially manipulate the equations (Saad, 2003) and an increase in computational performance is expected.

1.2 ELEFANT

Not all numerical codes focus on identical problems. For example, ASPECT (Advanced Solver for Problems in Earth's ConvecTion (Kronbichler et al., 2012)) is a state of the art mantle convection code specialized in high performance and high resolution models. Running the code on simple models at low resolution may yield an accurate solution, but is unreasonable/inefficient because the same result can also be obtained by a simpler

and less complex numerical code in a fraction of the time. Benchmark tests in the modelling community (Tosi et al., 2015) compare the many numerical codes which are instructed to run the same model. This provides a performance and accuracy overview of the many codes available today (e.g. ASPECT, CitcomS (Zhong et al., 2000), ELEFANT and others).

Like ASPECT is aimed at mantle convection, other codes will focus more on smaller scale structures in the crust and lithosphere. ELEFANT focusses on larger scale structures such as modelling subduction and convection. ELEFANT (ELElements Finis Appliques au Numerique en TectOnique (Thieulot, 2014)) is a descendant of FANTOM (Thieulot, 2011) (Finite Element method Applied to Numerical TectOnics Modelling), thus borrowing a lot of features from it. It is a multi-purpose thermomechanically coupled finite element Arbitrary Lagrangian Eularian geodynamical code and it allows for different boundary conditions (e.g. no-slip, free-slip and a free surface). The code is still under active development and usage (Maffione et al., 2015; Tosi et al., 2015; Thieulot, 2017, 2018). ELEFANT offers a variety of solving methods. MUMPS (MUltifrontal Massively Parallel sparse direct Solver) is the integrated direct solver (Amestoy et al., 2001). Numerous iterative solvers and preconditioners are also available as on/off toggles in the code.

Not only regularly spaced quadrilateral (in 2D) and hexahedrons (in 3D) are supported. Many other element geometries can be toggled with a single variable. The range from a 2D box with regular square elements to a 3D hollow sphere containing up of twelve individual parts that communicate through a complex connectivity setup form a versatile base layer for many types of numerical simulations. However, the main focus is on regular Q_1P_0 elements. To efficiently solve the set of equations, an outer-inner solve technique known as the Schur complement method (Horn and Zhang, 2005) is used. ELEFANT is constructed to be installed and set up relatively easily. The code and subroutine structure are built to be understood by students with a little background in finite elements, modelling and Fortran. By breaking up the code in many subroutines, every routine serves one job only. This further supports understandability for outside users.

ELEFANT is in continuous development. Performance improvements such as parallelism could have a great impact on its future use. Often, more than 90 percent of runtime consists of the solve step. Especially in large runs with high resolution and high viscosity contrasts, this problem becomes prominent. Direct solvers are always the preferred method for solving smaller 2D problems because of relatively small amounts of CPU and RAM (Random Access Memory) requirements compared to 3D. Upscaling to 3D can present computational complications, such as long computation times, when using direct solvers. For high resolution 3D problems, iterative solvers provide a better solution. These solvers require a small amount of RAM compared to direct solvers. Trough the continuous development of ELEFANT it grows to be a more robust, versatile and powerful code.

1.3 Governing equations

On the geological time scale, flow of mantle and lithosphere can be estimated by assuming continuous fluid behaviour (Karato, 2008). Therefore the Stokes equations are solved to model Earth's material flow. In the case of geodynamics, the momentum conservation equation is used to describes the mechanics of the litho- and asthenosphere

$$\boldsymbol{\nabla} \cdot \boldsymbol{\sigma} + \rho \vec{g} = \vec{0} \tag{1}$$

where $\boldsymbol{\sigma}$ is the stress tensor (Pa), ρ the mass density (kg/m^3), and $\vec{g}$ the gravitational acceleration vector (m/s^2).

To preserve mass conservation (i.e. mass outflow is equal to mass inflow in a bounded system), incompressibility is assumed. The incompressibility constraint, resulting in a constant overall density writes

$$\boldsymbol{\nabla} \cdot \vec{v} = 0 \tag{2}$$

where $\vec{v}$ is the velocity vector (m/s). The stress tensor is given by

$$\boldsymbol{\sigma} = -p\boldsymbol{I} + \boldsymbol{\tau} \tag{3}$$

where p is the hydrostatic pressure (Pa), $\boldsymbol{I}$ the identity matrix and $\boldsymbol{\tau}$ the deviatoric stress tensor (Pa). The latter is expressed in terms of dynamic viscosity μ (m^2/s) and strain rate

$$\boldsymbol{\tau} = 2\mu\dot{\boldsymbol{\epsilon}} \tag{4}$$

where the $\dot{\boldsymbol{\epsilon}}$ is the strain rate

$$\dot{\boldsymbol{\epsilon}} = \frac{1}{2}\Big[\boldsymbol{\nabla}\vec{v} + (\boldsymbol{\nabla}\vec{v})^T\Big] \tag{5}$$

expressed in terms of velocity gradients (s^{-1}). Combining these five equations results in the Stokes equation

$$\boldsymbol{\nabla} \cdot (2\mu\dot{\boldsymbol{\epsilon}}) - \boldsymbol{\nabla}p + \rho\vec{g} = \vec{0} \tag{6}$$

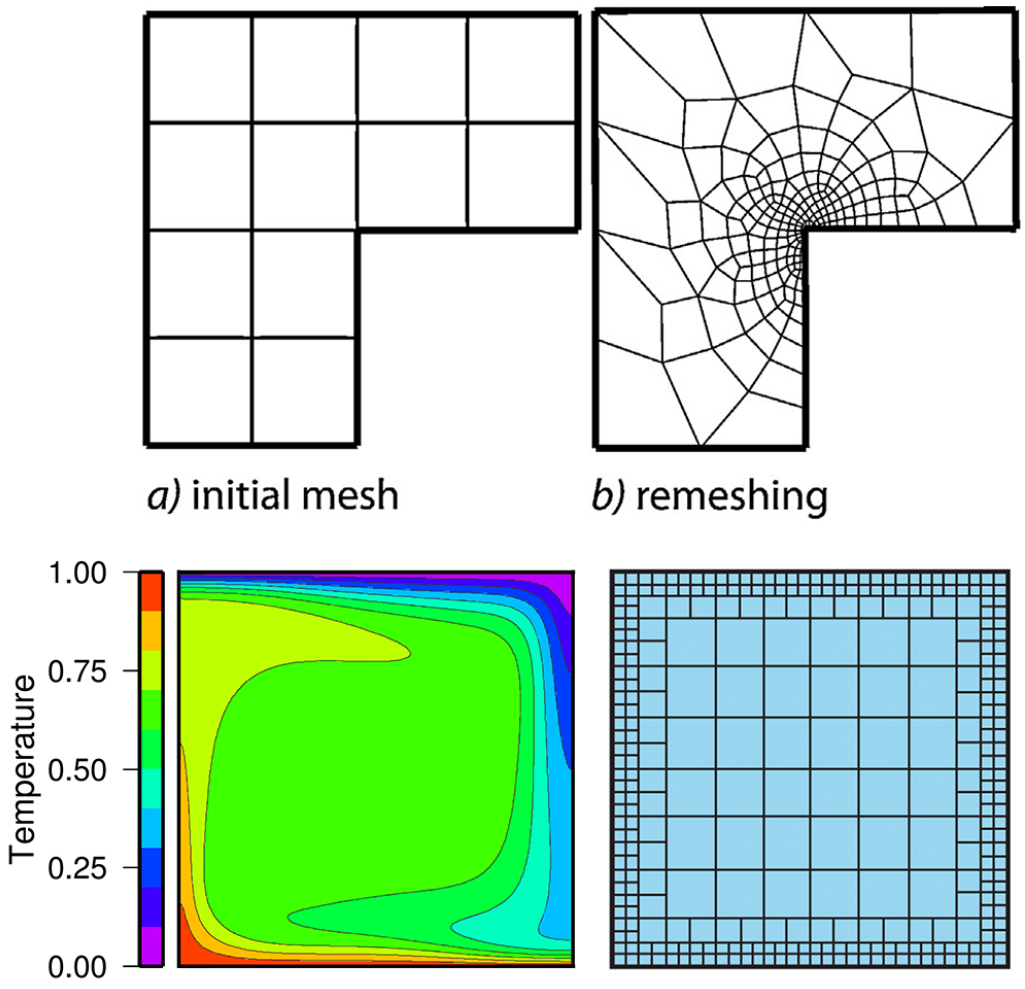

Figure 1: Top: comparison of a regularly spaced rectangular mesh and adaptive mesh. In this case, more detail is required in the center and thus mesh is adapted. Bottom: simple dimensionless convecting system and associated mesh with higher resolution in areas of higher temperature gradients. Top two and bottom right figure from May et al. (2013), bottom left from Tosi et al. (2015).

1.4 Finite elements

To be able to solve the Stokes equations, the Finite Element Method (FEM) is used. This is a powerful numerical technique to solve Partial Differential Equations (PDE) that arise in fields of engineering and applied science

(Reddy and Gartling, 2010). The method is based on the variational and weighed residual methods (Reddy, 1986, 2002, 2004; J N Reddy, 2005; Heinrich, 1965; Mikhlin, 1971) through which a differential equation and its solution are represented as a set of unknowns and corresponding functions. The functions are an approximation such that they abide the boundary conditions.

In FEM, the domain Ω is subdivided into smaller geometrically shaped subdomains called finite elements, each able to satisfy different boundary conditions. For every element it is possible to produce an approximate function required for solving the PDE. FEM allows for irregularly shaped elements, hereby making it possible exploit the use of mesh refinement techniques (May et al., 2013). Areas where gradients in parameters (temperature, velocity, viscosity) are largest are assigned smaller elements (Figure 1). In lithosphere modelling a fault geometry with high gradients in viscosity would be an ideal place for mesh refinement.

The main steps that are carried out during FEM calculation, in order, are discretization of the PDE, construction of the weak form, setup of the equations and unknowns, assembly of the matrix/matrices, boundary condition imposing and solving the system. Discretization subdivides the domain Ω, where $\Omega \cup \Gamma$ (Figure 2), into element domains Ω^e bounded by Γ^e. For every element, an elemental matrix and elemental right-hand side are constructed. This is carried out through a numerical integration. This integration can be of linear, quadratic or higher order. If calculating the unknown pressure, P, in a single element, the FEM expression is

$$P(x,y) \approx P^e(x,y) \sum_{i=1}^{n} P_i^e \phi_i^e(x,y) \tag{7}$$

where P^e is the approximation of P for a single element, ϕ^e are the functions related to the element and n is the number of nodes that make up the element ($n = 4$ for the dark grey element in Figure 2).

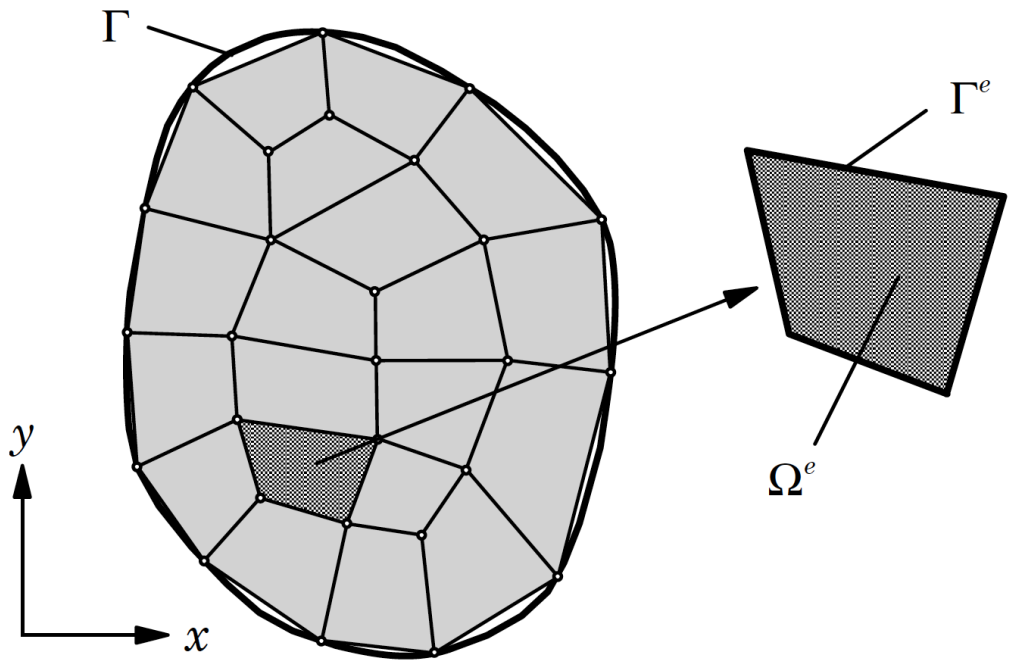

Figure 2: Discretisation of the domain Ω using irregularly shaped elements. From Reddy and Gartling (2010).

In order to numerically integrate a PDE into a computer, its weak form needs to be constructed. The weak form is equivalent to the actual PDE in addition to supporting some types of boundary conditions (Reddy and Gartling, 2010). For an extensive background on the weak formulation, the reader is referred to Pinchover and Rubinstein (2005). After integration, simple boundary conditions can already be applied to the weak form over Ω. Finally, the weak form is decomposed to the element-based equations.

The resulting polynomials can be integrated over a number of the integration points, known as quadrature

points (nodes that are at the corner of the element). Shape functions describe the geometry of the element in a unique coordinate scheme and are used in the weak form of the PDE. Continuity in the system is maintained as neighbouring elements share some amount of nodes depending on element geometry. As an example, a grid of quadrilateral elements is set up (Figure 3). In the case of 2D, the resolution is defined as the amount of nodes in the x and y direction, nx and ny respectively. In this case, 4 nodes make up 1 element. For hexahedral elements in 3D, 8 nodes make up an element. In order to solve PDEs with FEM, it must be known which local

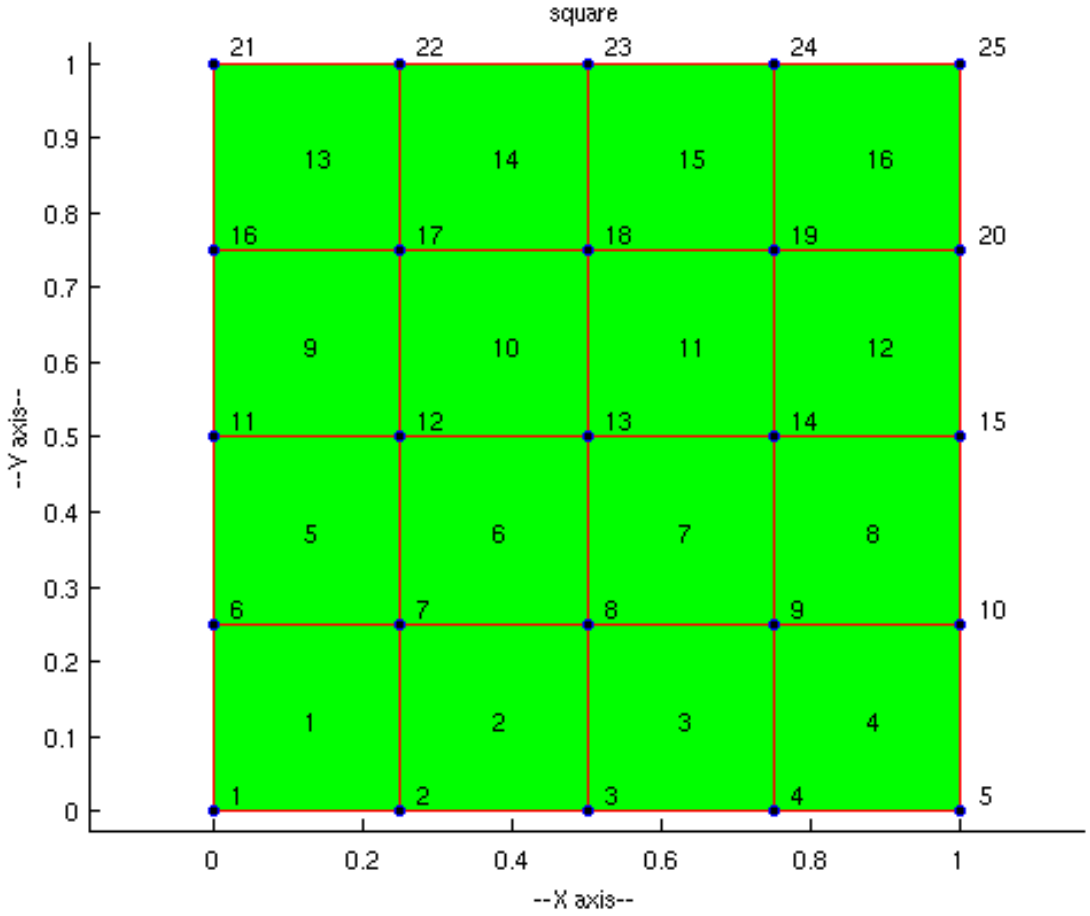

Figure 3: FEM mesh numbering of nodes and elements in a rectangular grid. From GNU (2014).

nodes belong to which element. This link between node and element numbering is called connectivity. A correct connectivity setup is crucial. From Figure 3 it is clear that the corners of element number 1 are nodes 1,2,7 and 6. Local ordering of nodes belonging to elements is done counter clockwise starting from the bottom left.

A quadrilateral element surrounded by other elements shares two nodes with the up, down, left and right elements and one node with the neighbouring top-right, top-left, bottom-right and bottom-left elements. In Figure 3, element 11 shares global nodes 13 and 18 with element 10 and shares only global node 13 with element 6. Sharing in this case indicates that the parameters at local node 1 of element 11 are equal to the parameters of local node 3 of element 6.

The connectivity and set of equations are cast into a linear system of shape $\boldsymbol{A}\vec{x} = \vec{b}$ in which $\boldsymbol{A}$ and $\vec{b}$ are known and $\vec{x}$ is the vector of unknowns (Ho et al., 2015). In the case of the Stokes equation, they can be written in vector notation as Block Stokes:

$$\begin{bmatrix} K & G \\ G^T & 0 \end{bmatrix} \begin{bmatrix} v \\ p \end{bmatrix} = \begin{bmatrix} f \\ 0 \end{bmatrix} \tag{8}$$

where K is discrete gradient of the deviatoric stress tensor, G is the gradient operator, G^T is the divergence

operator, v is the velocity, p is the pressure and f is the righthand side vector describing the force field (May et al., 2015). This notation of Stokes, as opposed to the more simple $\boldsymbol{A}\vec{x} = \vec{b}$ formulation, allows for higher performing and more robust iterative solving techniques.

1.5 Challenges and goal

Models at a high resolution result in a large system of linear equations with dimensions of $\boldsymbol{A}$ as high as millions. In many cases, solve time comprises most of the model's runtime. A percentual reduction of solve time is necessary to increase the number of geodynamic models run per time unit. In this thesis I will present iterative solver methods combined with preconditioning which can improve performance of ELEFANT. Consequently, models of mantle convection, plate subduction, and slab tear are able to run at higher resolution and results can be interpreted with more accuracy and detail.

Additionally, an educational FEM code is rewritten from Fortran to Python. This opens up easy-to-use, but powerful functionality in the space of large numerical calculations. A penalty method, as well as using the Schur complement, are compared on a performance level. Conjugate Gradient and Preconditioned Conjugate Gradient methods will be explored to assess performance for two tests. The aim is to provide a powerful tool for Earth Sciences students (not only those of Universiteit Utrecht) to explore geodynamics.

2 Methods

To get an understanding of how the models are set up, certain aspects of a numerical code need to be addressed. What techniques can be used to solve a system and how do these work? How is an external solving package integrated into ELEFANT? These questions are answered in this section.

2.1 Matrices

Solving a discrete PDE implicitly yields a system of linear equations

$$\boldsymbol{A}\vec{x} = \vec{b} \tag{9}$$

where $\boldsymbol{A}$ is the $n \times n$ linear operator, $\vec{b}$ is the solution vector and $\vec{x}$ is the vector of unknowns, both of length n (number of degrees of freedom). FEM discretisation of PDE's results in a sparse linear operator with, less than 1% nonzero entries. As of this discretization, $\boldsymbol{A}$ is symmetric. Its symmetry is defined as

$$\boldsymbol{A}^T = \boldsymbol{A} \tag{10}$$

and is Symmetric Positive Definite (SPD) because

$$\vec{w}^T \boldsymbol{A}\vec{w} > \vec{0} \quad \forall \quad \vec{w} \neq \vec{0} \quad for \quad \vec{w} \in \mathbb{R}. \tag{11}$$

The combination of sparsity and SPD can yield a large computation time and space (RAM) saving. First, only the upper or lower half (including the diagonal) of $\boldsymbol{A}$ has to be stored to capture the whole matrix because it is symmetric. Second, zero entries can be disregarded and only non-zero entries have to be stored. The non-zero structure of SPD matrices can be predicted and is dependent on number of elements, degrees of freedom, resolution and connectivity.

For a 2D quadrilateral grid, the nodes that are not located on the boundary communicate to 8 adjacent nodes. In the assembly of $\boldsymbol{A}$, this results in a maximum of 9 non-zeros per row. A corner along the boundary is communicating with 3 nodes and results in 4 non-zeros per row in $\boldsymbol{A}$. For simplicity, let us assume that every row contains 9 non-zero entries. The amount of sparsity (percentage of non-zeros in a matrix) is

$$size(\boldsymbol{A}) = n \times n, \quad sparsity = \frac{9n}{n^2}. \tag{12}$$

Thus, for a matrix size of 1000×1000, $sparsity = 0.9\%$. Increasing matrix dimensions would further decrease the percentage of non-zeros. Storing the zero-entries of $\boldsymbol{A}$ is highly inefficient as they do not contribute to the solution. The critical information of $\boldsymbol{A}$ is the combination of the location and value of the non-zero entries. Compressed Row Storage (CSR) stores the matrix in a more compact format using three one-dimensional arrays. The first and second arrays contain row-structured integer pointers of the non-zero values while the third array contains the matrix values (Figure 4).

$$A = \begin{bmatrix} a & 0 & b & 0 & 0 & c \\ d & e & f & 0 & 0 & 0 \\ 0 & 0 & g & 0 & h & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & i & 0 \\ 0 & 0 & j & k & l & 0 \end{bmatrix}$$

$$\text{row pointer} = \begin{bmatrix} 0, 3, 6, 8, 8, 9, 12 \end{bmatrix}$$

$$\text{column index} = \begin{bmatrix} 0, 2, 5, & 0, 1, 2, & 2, 4, & 4, & 2, 3, 4 \end{bmatrix}$$

$$\text{value} = \begin{bmatrix} a, b, c, & d, e, f, & g, h, & i, & j, k, l \end{bmatrix}.$$

Figure 4: Structure of CSR storage consisting of three vectors. In this example, zero-indexing is used. From Liu and Vinter (2015).

2.2 Time complexity

Scaling (Computation as a function of resolution, processors or cores) is important when assessing performance. The slope in a log-log plot indicates how well a certain method/technique scales. When measuring computation time, a slope of 1 means that computing time scales linearly with the degrees of freedom (i.e. increasing resolution by two results in computation time increase of two). Theoretically, multigrid computation time scales with $\mathcal{O}(N)$ while other methods might scale with $\mathcal{O}(N \log N)$ or $\mathcal{O}(N^2)$, where $\mathcal{O}(N^2)$ is the slowest. We can visualise this by plotting N, $N \log N$, and N^2 (Figure 5).

2.3 Solvers

To obtain the solution to the Stokes equation, a linear system has to be solved (Eq. 9). It is impossible for large systems to carry out $\vec{x} = \boldsymbol{A}^{-1}\vec{b}$ because the inverse of a large matrix is very costly to compute. The valid strategy to solving such a system is to apply a solver which takes $\boldsymbol{A}$ and $\vec{b}$ as input and returns solution $\vec{x}$. Direct solvers aim to calculate a solution in a single iteration through Gaussian elimination. When using such an approach, $\boldsymbol{A}$ first has to be factorized. In linear algebra, factorization is the decomposition of a matrix into a product of 2 or more separate matrices so that the product of the factorization equals the original matrix. An example of a decomposition is LU, where $\boldsymbol{A}$ is split in a lower and upper triangular matrix so that

$$\boldsymbol{L} \cdot \boldsymbol{U} = \boldsymbol{A}. \tag{13}$$

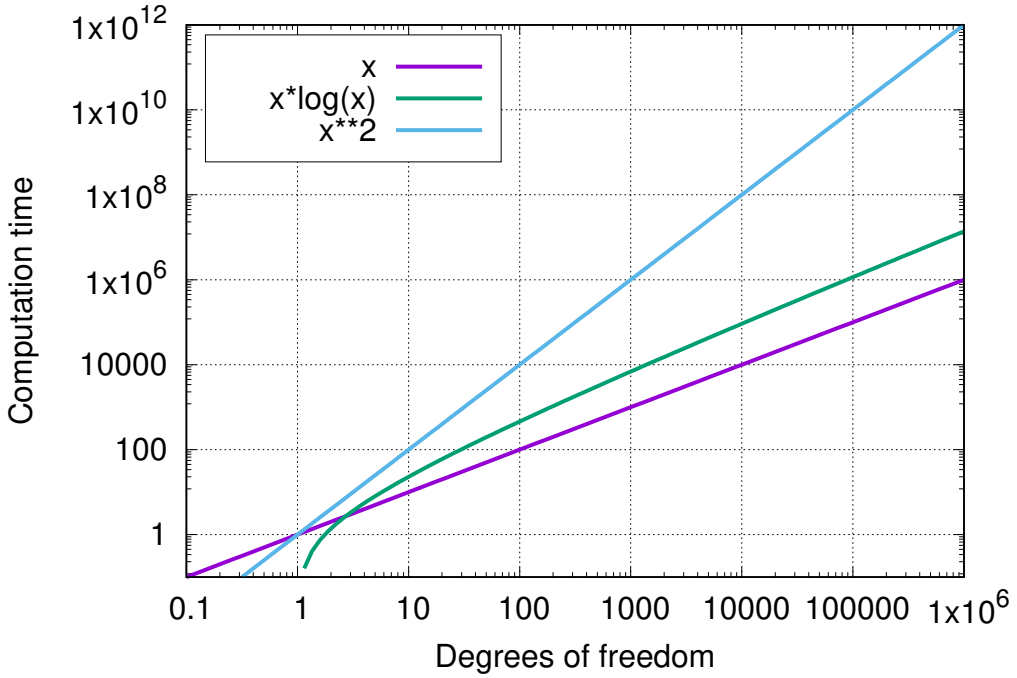

Figure 5: Time complexity of three functions. x^2 scales with the steepest slope.

A related factorization is the incomplete LU decomposition (ILU) which uses an approximation so that

$$\boldsymbol{L} \cdot \boldsymbol{U} \approx \boldsymbol{A}. \tag{14}$$

ILU factorization is more time efficient as L and U require less memory to be stored but is not valid for solving equations as the factorization is not exact. Some factorization software packages include UMFPACK (Davis, 2004),SuperLU (Li, 2005), PARADISO (Schenk and Gartner, 2006) and MUMPS (Amestoy et al., 2001). MUMPS is one of the packages implemented in ELEFANT. It supports sparse large linear systems and is able to run in parallel through MPI. At its core, it uses BLAS and LAPACK to carry out small matrix calculations. The deficiency of MUMPS is in the non-linear scaling RAM and CPU requirements as the size of the system increases. Hence, increasing the resolution by two means RAM and CPU demands increase by a factor of larger than two (COMSOL, 2014).

Due to this limitation different methods need to be used for high resolution (and especially 3D) models. In contrast to direct solving methods, iterative methods produce approximate solutions with which consecutive solution are computed, hereby approaching the exact solution vector. At each iteration, the stopping condition is tested. This generally involves calculating the residual vector which is defined as

$$r = b - Ax^k \tag{15}$$

where b is the righthand side vector of $\boldsymbol{A}\vec{x} = \vec{b}$ and x^k is the solution vector at iteration k in an iterative solve. From this time forth, $\boldsymbol{A} = A$, $\vec{b} = b$, $\vec{x} = x$ etc. When an iterative method is converging, the residual will decrease after every iteration. The stopping condition itself is based on setting a tolerance. Low tolerance results in a more iterations and an accurate solution whereas a high tolerance allows for inaccurate results but at a lower computational cost.

The disadvantage of direct solvers, regarding RAM issues, is absent here. Contrary to direct solvers, iterative solvers do not require A to be stored explicitly. It only requires the solution matrix-vector products by A (Saad, 2003). This feature vastly decreases storage requirements.

Iterative methods start with an initial guess x^0. Often, when there is no pre-knowledge about the solution, a zero vector is used $x^0 = 0$. x^1, the vector used in the second iteration, is closer to the exact solution than x^0 and thus the residual decreases.

Algorithm 1: Jacobi method

Input : A,b,Initial guess x^0

Output: Solution vector x

decompose A

$k = 0$, $x^k = 0$

while *not converged* **do**

  $x^{k+1} = D^{-1}(b - (L + U)x^k)$

  $x^k = x^{k+1}$

  $k = k + 1$

end

Many types of iterative solvers exist. They can be subdivided into two classes. The first are stationary methods, which are based on the residual and correct the solution at each iteration accordingly. Jacobi (Jacobi, 1845), Gauss-Seidel (Seidel, 1874), and Successive Over-Relaxation (SOR,(Young, 1954)) are among the most well known stationary methods. The second class is Krylov subspace methods which computes

$$A^i b \quad i = 1, 2, ..., m - 1 \tag{16}$$

where m is the subspace dimension (Saad, 2003). The Conjugate Gradient (CG) is the archetypal method. Other Krylov subspace methods include Generalized Minimal REsidual Method (GMRES, Saad and Schultz (1986)) and BiCGSTAB (van der Vorst, 1992). To be able to understand the principles of iterative solvers, a simple method known as Jacobi is first explained.

Jacobi

Not all solvers are able to handle all types of matrices. For example, the Jacobi solver requires A to be fully non-zero along the diagonal. Furthermore, diagonal elements must be at least strictly diagonally dominant over off-diagonal elements. Strict diagonal dominance of A at any element along the diagonal is given in Saad (2003) as

$$|a_{jj}| > \sum_{i=1}^{i=n} |a_{ij}| \text{ for } i = 1, ..., n \text{ and } i \neq j \tag{17}$$

where a indicates an element of matrix A. This feature restricts the method's usability and practicality. However, when solving well defined and relatively uncomplicated problems, this method could be useful.

The method is based on calculating the i-th vector component of the $k + 1$ iteration to reduce the ith component of the residual vector (Saad, 2003). The first step is the decomposition of A into the diagonal D and the strictly upper and lower half U and L respectively in a similar manner as the LU decomposition (Eq. 13). The iteration is written as

$$x^{k+1} = D^{-1}(b - (L + U)x^k). \tag{18}$$

Assuming that D^{-1} is stored as a vector, one matrix multiplication, $(L+U) \cdot x^k$, is carried out at each iteration, taking up most of the computation time. The Jacobi algorithm in pseudocode is given in Algorithm 1.

Conjugate Gradient

According to Dongarra and Sullivan (2000), Krylov subspace methods are in the top 10 of most influential algorithms alongside methods like FFT (Fast Fourier Transform) and Quicksort. Some large sparse linear

systems can be solved in a more efficient manner using this method compared to simple iterative methods. Among the most used Krylov methods, we find the Conjugate Gradient (CG), Generalized minimal residual (GMRES), and multigrid methods. Particular geodynamic models may apply a combination of GMRES and CG with a multigrid preconditioner (ASPECT), showing there is not a single best method for every problem. CG was originally presented by Hestenes and Stiefel (1952).

Optimizing the solving procedure is critical when analyzing and improving performance of numerical codes. Direct methods are not dependent on values of the matrix. In contrast, iterative solver performance depends solely on the matrix entries.

Solving large linear systems (i.e. matrix dimensions of $10^6 \times 10^6$) proves to be computationally costly when utilising suboptimal solvers. Exploiting the sparsity and SPD nature of the linear operator (A in Eq. 9) could not be enough to achieve efficient solves. In view of the fact that the solve step is the main contributor of computation time (Figure 6), it requires the use of preconditioners. For the understanding and explanation of preconditioning techniques, the excellent presentation and slides from Bangerth (2013) are used.

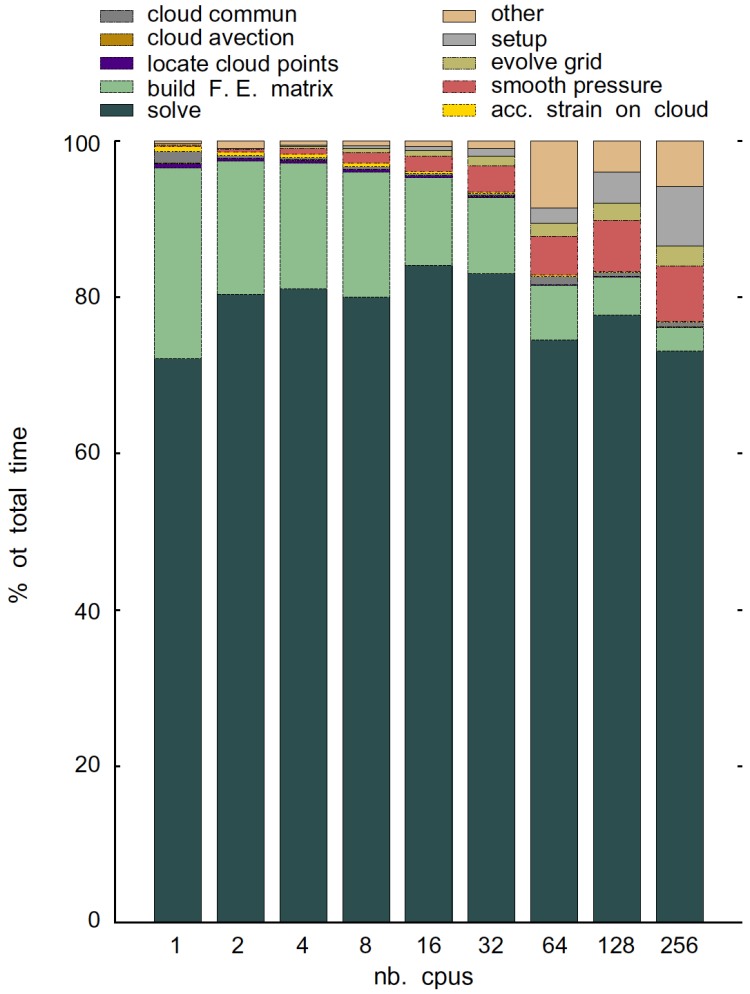

Figure 6: Overview of main time contributors in a FEM simulation showing that the solve step is always the most computationally expensive. From Thieulot (2011).

The controlling factor on iterative solve time is the convergence rate. This is the rate/time in which an iterative solver arrives at the convergence criterion. Setting a high accuracy will result in longer solve times while a low accuracy is cheaper to compute but yields a less accurate solution. The linear operator in Eq. 9 is what dominates performance of solvers. Every matrix has a corresponding condition number which is the ratio between the largest and smallest singular values of A in Eq. 9. Generally, for a square matrix (i.e. matrix of

dimensions $n \times n$), the eigenvalues are calculated as

$$s^i = |A - \lambda^i I| \quad where \quad i = 1, 2, ..., n \tag{19}$$

where $|...|$ indicates a determinant. The condition number then is

$$C = \frac{s_{max}}{s_{min}}. \tag{20}$$

An infinite condition number indicates a singular (i.e. non-invertable) matrix. Ill-conditioned matrices correspond to a large condition number relative to the accuracy of matrix entries.

When A is SPD and using CG as an iterative solver, the residual is reduced by

$$r = \frac{\sqrt{C} - 1}{\sqrt{C} + 1} < 1. \tag{21}$$

m iterations are needed to arrive at tolerance ϵ so that

$$r^m = \epsilon \rightarrow m = \frac{\log(\epsilon)}{\log(r)}. \tag{22}$$

For Laplace equations, the condition number is h^{-2} where h is the internodal distance of the mesh. Substituting into Eq. 21 yields

$$r = \frac{C - h}{C + h} < 1 \tag{23}$$

and thus the amount of iterations scales as

$$m = O(\log(\epsilon)/h). \tag{24}$$

When the mesh resolution is increased, h decreases, $r \rightarrow 1$, resulting in slower convergence rates of CG. A resolution increase of 2 thus requires twice as many CG iterations. The advantage of CG is that the number of iterations required for convergence is predictable.

Multigrid

As the resolution and the degrees of freedom in a system grows, the number of iterations needed to compute the solution increases. The rate at which this increases varies per method. In a log-log plot of degrees of freedom versus solve time, an ideal slope would be 1. When solving 3D Stokes, direct methods are known to not scale well, showing high slopes, while an iterative method like GMRES with an ILU preconditioner has a more shallow slope (Figure 7 in May et al. (2013)). However, correctly tuning a technique known as multigrid will result in a high level of performance for particular types of problems.

The user might choose to not use multigrid as it is more complex than other iterative methods. However, when implemented correctly, it can greatly improve the speed of solving the linear system. Fedorenko (1964) first explicitly documented this method and since then is used progressively more as it evolves. Extensive background on the use of this method can be found in Saad (2003); Wesseling (2004); Borzi and Schulz (2009); Falgout (2006) and Meurant (1999).

Every iterative method uses the original grid to get a solution on the next iteration. Multi-level methods, however, exploit the interaction of multiple coarser and finer meshes to reach faster convergence, hence the name. Residuals (Eq. 15) at each iteration have a certain wavelength when projected onto the mesh (Figure 7). When iterating, residuals with a short wavelength get annihilated faster than long term residuals. The goal is to reduce residuals of all wavelength simultaneously. By solving the system at increasingly coarser grids and exchanging information between them, residuals of all wavelengths get annihilated in the same number of iterations. From Figure 7 it is observed that multigrid does a better job of decreasing residuals than GS.

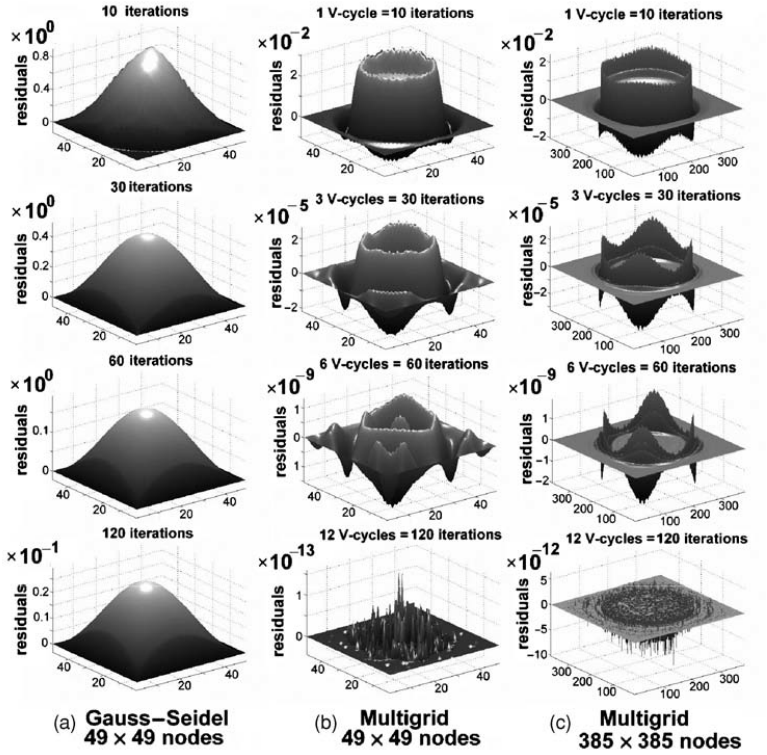

Figure 7: Distribution of the residual with a) Gauss-Seidel for a low resolution, b) multigrid for a low resolution and c) multigrid for a high resolution. Note that the multigrid high resolution mesh residuals decreases only slightly slower than the multigrid residuals of the lower resolution mesh. From Gerya (2010).

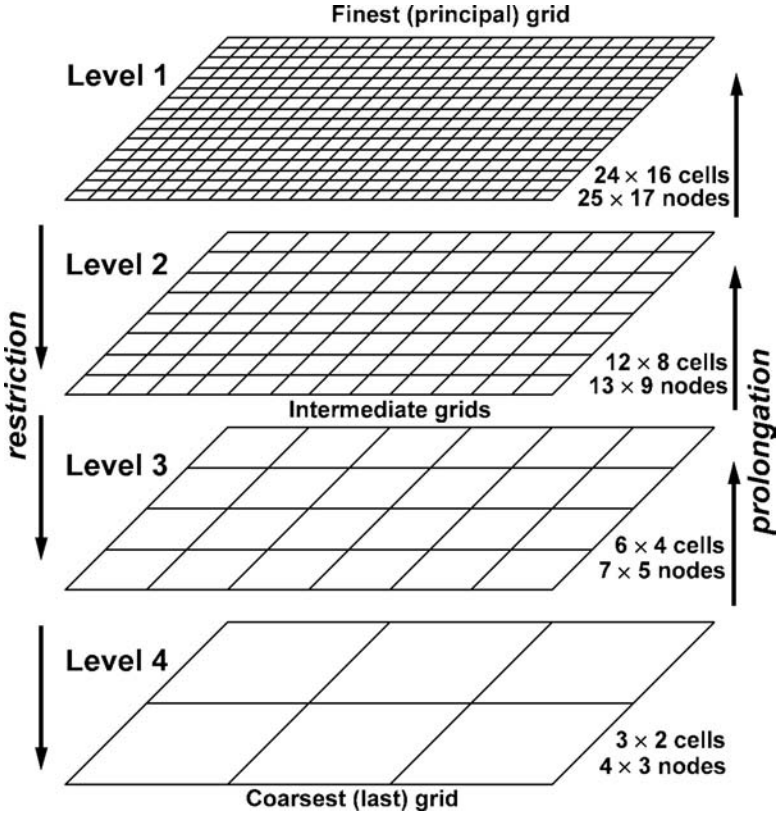

Figure 8: Structure of the coarse and fine multigrid meshes showing restriction and prolongation directions. From Gerya (2010).

Furthermore, using the multigrid method at a higher resolution performs better than GS at low resolution (Figure 7c).

During an iteration cycle of an iterative method, the information about the unknowns is propagated only by neighbouring cells (Gerya, 2010). If the grid is coarsened, the information has to travel a smaller distance in order to communicate with the total domain and thus information is propagated faster per iteration. By back-propagating the information of the coarse grid residuals onto the finer grids, residuals are smoothened. The information exchange from fine to coarse grids is called restriction requires a restriction operator. Prolongation is the operation of back-propagation of the information onto the fine grids (Figure 8) and requires an interpolation operator. This method proves to be very robust for models with sharp viscosity contrasts such as mantle convection and subduction. However, for large system any iterative solver becomes very costly. Preconditioning helps to alleviate this problem and reduces the condition number.

2.4 Preconditioners

The perfect preconditioner is defined as $P^{-1} = A^{-1}$ so that P applied to A would yield an identity matrix, thus $C = 1$ and 1 iteration is required to reach convergence. Calculating the inverse of large matrices becomes nearly impossible when working with large matrices. Therefore, the approach is to find a preconditioner which needs to meet three basic requirements. First, it needs to be cheap to compute

$$Px = b. \tag{25}$$

This matrix-vector multiplication is carried out every iteration and thus must remain cheap. Second, P must be non-singular (i.e. its inverse exists). Third, P must be similar to A so that P^{-1} applied to A is close to identity and the condition number is close to 1. Preconditioning a simple iterative method results in

$$x^{k+1} = x^k - P^{-1}(Ax^k - b) \tag{26}$$

where $Ax^k - b$ is the residual. If $P^{-1} = A^{-1}$ only 1 iteration is required as $P^{-1}A = I$ and

$$x^{k+1} = A^{-1}b. \tag{27}$$

Let us take the Jacobi preconditioner where $P^{-1} = \omega D^{-1}$ where D^{-1} is the inverse of the diagonal of A and ω is a relaxation factor so that

$$x^{k+1} = x^k \omega D^{-1}(Ax^k - b). \tag{28}$$

The preconditioner is easy to compute but does not converge very fast and only converges if ω is sufficiently small. In this case, ω needs to be smaller than $(s_{max})^{-1}$. When using CG, ω can be neglected as the convergence is not affected by it. Gauss Seidel (GS) preconditioning ($P^{-1} = (D + \alpha L)^{-1}$) is slightly better than Jacobi and results in

$$x^{k+1} = x^k \omega (D + L)^{-1}(Ax^k - b). \tag{29}$$

This preconditioner is cheap to apply as its triangularity can be exploited in matrix multiplications. However, this preconditioner is asymmetric and its inverse is hard to compute. Applying an symmetric preconditioner to a symmetric matrix yields an asymmetric linear operator in the iterative solve. CG is only compatible with symmetric matrices and thus GS preconditioning is not an option. The Successive Over Relaxation (SOR) preconditioner applied to an iterative solve gives

$$x^{k+1} = x^k \omega (D + \alpha L)^{-1}(Ax^k - b). \tag{30}$$

and is, again, asymmetric. However, Symmetric SOR (SSOR) can be used in stead. By not only including L and D, but also U in the preconditioner, symmetry is restored and integration with CG is possible. The

condition number for this method reduces the condition number by $O(1/\sqrt(h))$, rather than $O(1/h)$. Jacobi, GS and (S)SOR are always dependent on mesh resolution. These preconditioners could be useful for small systems but may present problems for larger systems.

A better preconditioner would be to approximate A^{-1} by using ILU factorization. This method provides high convergence rates as well as increased robustness (i.e. convergence probability). An ILU preconditioner allows for a denser matrix by adding levels of fill. Fill increases the sparsity pattern of the preconditioner to A^2 when one level of fill is used A^3 when two levels of fill are used etc. Adding more levels increases robustness at the cost of an increase in matrix multiplication time using the preconditioner.

Returning to Jacobi; the vector multiplications of A by x means that the approximations of x are coupled to the matrix entries of A (the mesh). If an entry of x^k is a bad approximation, it can only be corrected by its direct neighbours in the mesh. If we take a value of x at the right of the domain which needs to be corrected (assumed that the left of the domain is correct), it would take at least n iterations to reach the right column of the domain, as a maximum of 1 column is corrected at every iteration.

So far, we have seen that not all preconditioners provide good solutions to the problem of efficiently iteratively solving a linear system of equations. Mesh dependency on condition number is the major disadvantage of many preconditioners. The multigrid method presents a less conventional way of preconditioning which is independent of mesh size (equivalent when using it as a solving method seen in Figure 7) and, when correctly tuned and using CG as solver, should provide textbook performance computing the solution in less than 10 iterations (Balay et al., 2017a). Parameters of multigrid which can be tuned include the amount of multigrid levels used, smoothing factor (which controls the amount of residual smoothing carried out at each iteration) and using a v/w cycle (controlling how coarse and fine grids hierarchically communicate)

2.5 PETSc

Portable, Extensible Toolkit for Scientific Computation (PETSc) (Balay et al., 1997, 2017a,b) is a set of data structures, routines and data types aimed at solving partial differential equations (PDE). It supports (scalable) parallel computation through MPI, OpenCL and GPUs (CUDA). PETSc can be downloaded from the website or through GitHub. The latter allows the user to update to newer versions through the command `git pull`.

After downloading or cloning using Git, PETSc is configured with or without options to install external packages. These external packages include compilers such as `g++`, `gfortran` and `gcc` as well as linear algebra packages such as `LAPACK`. If the user already has these installed there is no need to configure with those extra options. Next, two `PATH`s need to be added to either ∼`/.bashrc` (when using a Linux distribution) or ∼`/.bash_profile` (on OSX). The first path directs to the installation directory of PETSc while the second is the architecture type found in this folder (e.g. `arch-linux2-c-debug` when using a Linux distribution).

A file called `variables` needs to be referenced in the Makefile of a program as it holds the reference to the location of `PETSC_LIB`. Finally, when one wishes to use the version of `mpif90` that was configured with PETSc, its location on the machine must be indicated. To integrate PETSc into a program, two statements need to be called:

```
#include <petscXXX.h>
```

followed by

```
use petscXXX
```

where `XXX` specifies type of PETSc functionality such as `petscksp` for linear iterative solvers, `petscvec` for the use of PETSc vectors or `petscsnes` for nonlinear solver functionality.

Variable types such as integers and reals correspond, in the PETSc data structure, as PetscInt and PetscReal

respectively. Matrices and vectors are also supported. These objects can be declared as

$Mat :: A$

or

$type(tMat) :: A.$

Similarly, more types such as `VEC, KSP, PC` and `DM` are declared. Passing a null as an argument is significantly easier in C than it is in F90. In C a null value is passed as `NULL` while in `F90` the user must specify the type of null (integer, real, character etc.). This is done by passing `PETSC_NULL_XXX` where `XXX` specifies the null type.

Routines from PETSc are powerful but can be confusing to use. Many options can be set for every call to a function. To organize nearly all function options, the "options file" or "options database" can be used. In this file, the user puts a list of function parameters in order to avoid having to hardcode them. A second advantage is that function options can be changed without having to recompile the program. For a matrix, its type might be specified in the options file as

```
-mat_type MatType
```

where MatType is one of many matrix types. Complete lists of these options can be found in the PETSc manual. However, it remains possible to also hardwire these options into the code using

$MatSetType(Matrix, MatType)$

but one can imagine that using many of these routines in an already large and complex code will not improve its readability. Therefore, the use of the options database is preferred over hardwiring into the code. The user can supply the name and location of the options file at runtime. Every PETSc code should start with the command

$PetscInitialize()$

and end with

$PetscFinalize()$

to indicate the beginning and end of a program. The options file can be given as an input in the PetscInitialize routine. Applying the options file is carried out by the routine

$XXXSetFromOptions()$

where XXX is the type to apply the options to. Taking a matrix as an example, the routine becomes

$MatSetFromOptions()$

and thus reading the matrix type from the options file. Monitoring the behaviour of a code (i.e. monitoring convergence of solving a linear system) is incredibly important. PETSc offers a suite of monitoring commands which can track all the necessary information for the user. Specifying

```
-ksp_monitor_lg_residualnorm
```

will open a X11 linegraph (lg) which plots live information throughout runtime in stead of printing it to the terminal. Fortran and C do not use the same indexing. In an array of values, the first entry in a C array is 0 (0 indexed) while in Fortran the first spot in an array is at 1 (1 indexed). Feeding 1 indexed arrays to PETSc will cause instabilities and crashes. To overcome this problem the user can give array indices to PETSc routines as

$PETScRoutine(argument, argument, i - 1)$

where PETScRoutine can be any PETSc call and i is the Fortran (1 indexed) index which is converted to 0 indexed.

The debugging procedure is made easier with the error variable. This variable can be given as an input/output in every PETSc routine. The routine returns either a 0 or a non-zero. A return value of 0 indicates that the routine was carried out without error whereas a non-zero value means that something went wrong. Thus, checking the error return at every PETSc call is advisable.

To gain familiarity with PETSC subroutines and data structures, it is first implemented in simpleFEM. When simpleFEM was confirmed to work properly with PETSc, it was put in ELEFANT. A graphic overview is given in Figure 9. The basic version of simpleFEM has no external subroutines. This is why every part in Figure 9 is structured in a sequential list.

As mentioned before, every PETSc program should begin with PetscInitialize. Next, PETSc vectors are created. A vector can be duplicated if one desires the duplicant to have the same dimension. For example, in 9, vector b can be created and is duplicated under the name of x. This is allowed because b and x have the same length.

The solver environment is created next with calls to KSPCreate and KSPSetFromOptions. The latter reads user input from the file specified with PetscInitialize. CSR storage is required by MatCreateSeqAIJWithArrays. The basic version of simpleFEM does not include CSR storage, so this needed to be copied from ELEFANT first. MatCreateSeqAIJWithArrays creates a sequential (SeqAIJ) matrix (Mat) out of CSR (WithArrays) arrays. It takes care of preallocating storage for the user. According to the manual (Balay et al., 2017a), preallocating arrays is a simple but efficient performance booster: "By setting these parameters accurately, performance during matrix assembly can be increased by more than a factor of 50". The second advantage of using this routine is that the user can freely change the entries in the CSR arrays without having to call the subroutine again. It is done automatically. Thus, MatCreateSeqAIJWithArrays only has to be called once, at the beginning of the program. In order to efficiently access and change entries in PETSc vectors VecGetArrayF90 is used. From the name, it is deducted that this is a Fortran 90 specific subroutine. It uses a pointer array for pointing to entries in the vectors. After the F90 right hand side vector is properly inserted into the PETSc vector, it is ready for assembly. Matrix assembly is next, followed by specifying the block size. The block size (number of degrees of freedom per node) of the system has to be specified for increasing efficiency of the algebraic multigrid preconditioner. Just before the solve, the user specifies which matrix is used for solving and which is used for preconditioning with KSPSetOperators. These matrices don't necessarily have to be the same. Finally, the F90 solution vector is extracted from the PETSc vector and PETSc structures are properly destroyed and closed.

It is not very efficient to only use PETSc in small parts of a program in stead of using it as much as possible everywhere in the code. For example, using a F90 matrix and a PETSc (C) matrix simultaneously, even though they have the same exact entries, makes little sense. However, because the main structure of simpleFEM and ELEFANT still had to remain the same, it was chosen not to completely rewrite the code. This gives us the possibility to toggle PETSc on and off using one variable in the code.

3 Results / Numerical experiments

Practically testing these is an important step in the process of optimizing efficiency for any given problem. A set of tests has been set up which offers a wide spectrum of geodynamic (related) simulations. Starting with a 2D Stokes problem, complexity increases to a full 3D subduction/rollback system with inclusion of sticky air and a weak zone accompanied by a non-linear rheology. All tests were run on an HP workstation with a 2.6Ghz Intel Xeon E5 and 128GB of 2133Mhz DDR4 memory. Table 1 provides an overview of the experiments that were carried out.

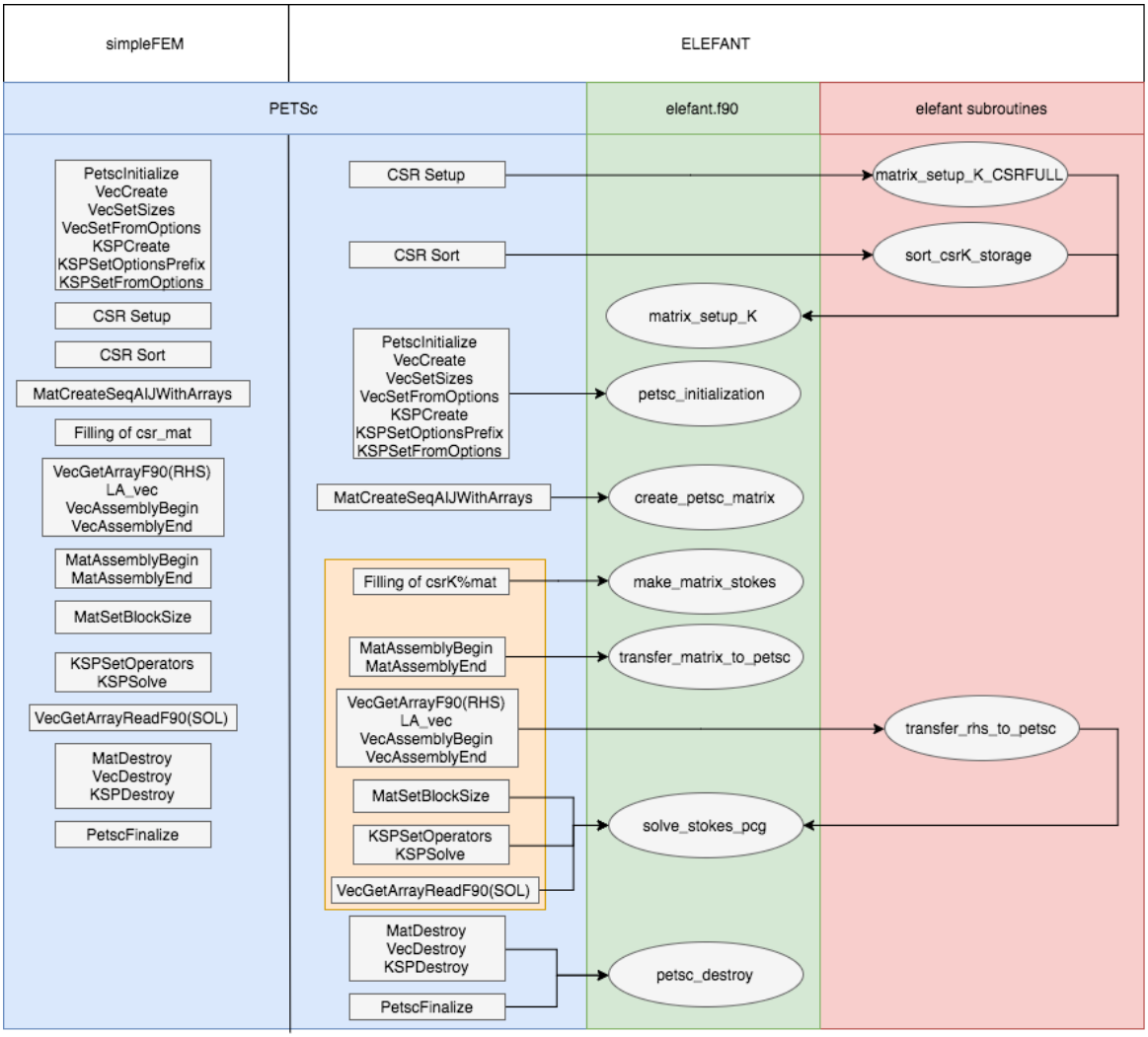

Figure 9: Overview of PETSc implementation in both simpleFEM and ELEFANT.

	2D			3D	
Experiment	NSinker	Tosi	Annulus	Cube	Subduction
Viscosity	Heterogeneous	Heterogeneous	Heterogeneous	Homogeneous	Heterogeneous
Grid type	Square	Square	Trapezoidal	Cubic	Cubic
Time stepping	No	Yes	Yes	Yes	Yes

Table 1: Overview of the performance experiments.

3.1 NSinker

The first test to examine preconditioner performance consists of a weak matrix experiencing pure shear (Duretz et al., 2011). The original benchmark and analytical solution were presented by Schmid (2002) and Schmid and Podladchikov (2003). Instead of one, multiple inclusions are implemented here. Consequently, no analytical solution exists and thus this is not a true benchmark. The domain spans $[0, 1] \times [0, 1]$, uses Q_1P_0 square quadrilateral elements and employs free slip boundary conditions. The shape of the circular inclusion is something that could also be observed in large geodynamic models such as subduction and mantle convection, making it a relevant test.

In the NSinker test, the Stokes equation is solved for a 2D box with highly contrasting viscous and dense circular inclusions with respect to the background material. The viscosity and density of the background material are set to 1 and 0 respectively (Figure 10). High contrasting, sharp boundaries are known to have

a negative effect on solve time and are therefore a good test for testing preconditioners. All of the circular inclusions are more dense and more viscous than their surroundings causing them to sink in the direction of gravity. Gravity, here, is directed downwards (in the negative z-direction). The direction of gravity has no impact on the accuracy of the solution.

Visualising z-velocities reveals that the inclusions are sinking towards the bottom (Figure 10). The 3 most dense inclusions sink the fastest. The lightest disk (with a density closest to the surrounding material) sinks the slowest, which could also be attributed to its relatively small size. The most dense inclusion, in the bottom left, is very small and has a very low viscosity. It is not the fastest sinking body, even though it is so heavy.

Pressure distribution shows a gradient from high pressures at the bottom to low, and sometimes negative, pressures at the top. Pressure is highest under faster sinking inclusions. The pressure under the largest body is the highest overall.

Figure 11 shows computation time of the Stokes solve as a function of resolution. Excluding the direct solving approach, multigrid preconditioning is the best option for this benchmark as it shows the lowest computation time and scales with a shallow slopes compared to ILU. Increasing the number of multigrid levels from two to four has a negative impact on the computation time. Therefore, applying the cheapest multigrid option, one level of coarsening (i.e. two multigrid levels), yields the best results. The regression lines drawn in black in Figure 11 are a best-fit scenario. As the coloured lines of, especially, multigrid are not perfectly linear in log-log, the slopes of the regression lines are subject to some error. However, it remains certain that multigrid preconditioning with two levels performs best when comparing iterative solving approaches.

Solving the system with a direct solver is more than an order of magnitude faster than using (preconditioned) iterative solvers. Its slope is steeper than multigrid + CG, indicating that the performance trends of the direct approach and multigrid preconditioning will intersect at very high resolution. This intersection point will be at a resolution at which the model is not anymore sensible to run and thus the relatively high slope of the direct solving approach is disregarded as a problem.

ILU preconditioners with and without infill are outperformed by both multigrid + CG and direct solvers. Using higher levels of infill speeds up calculation by a minor factor. The slopes between ILU and multigrid are so different that only using ILU at very low resolutions might be beneficial over multigrid.

3.2 Tosi

This benchmark has been extensively tested in Tosi et al. (2015). In that paper, many geodynamic codes were tested and solutions were compared across a set of different tests. A few different modes were explored: a mobile lid, in which the uppermost part of the 2D Cartesian box is actively part of the convecting regime, a stagnant lid, in which the topmost part of the domain is not taking part in convection, and finally a periodic lid where the system alternates between stagnant and mobile lids.

The version of this benchmark is a slight modification of the benchmark originally presented in Blankenbach et al. (1989). The goal of this benchmark is to test simple thermal convection for a constant and temperature-dependent viscosity. Various modern geodynamic-related codes have used this benchmark to validate their results (Gerya and Yuen, 2003; Kronbichler et al., 2012; Davies et al., 2011; Leng and Zhong, 2008).

In the benchmark, the non-dimensional Boussinesq convection equations are solved. The inertia in the fluid is considered negligible. Constant thermal diffusivity and thermal expansivity are assumed. The fluid is isothermally heated from the bottom and cooled from the top using non-dimensionality resulting in 1 and 0 as temperatures for the bottom and top respectively. The left and right walls of the domain satisfy $\frac{\partial T}{\partial x} = 0$, simulating insulation. Free slip boundary conditions are imposed with no material being allowed to either leave or enter the domain.

The setup that is used is Case 1 from Tosi et al. (2015) where the initial temperature profile is sinusoidal

and writes

$$T_{initial} = (L_z - z) + \frac{\cos(\pi x)\sin(\pi(L_z - z))}{100} \tag{31}$$

where x and z are the spatial coordinates (in the x/z coordinate system) and L_z is the domain size in the z-direction.The Rayleigh Number is set to $Ra = 10^2$ and the parameters controlling the viscosity contrast due to temperature and pressure are 10^5 and 1 respectively. A linear viscous rheology is assumed. This setup results in stagnant lid behaviour (Figure 12).

Plotting computation time versus resolution reveals that multigrid performs best (Figure 13). Regression lines shown in black indicate a slope of 1.16 for multigrid with one level of coarsening. This behaviour is similar to the NSinker results where the performance of preconditioners scaled best when using a multigrid preconditioner. A kink is observed in the multigrid results. The regression line was calculated with excluding the data after the kink. If we were to take this data into account, the slope would increase. A similar kink is observed in the ILU data. ILU preconditioners, in general, show steeper slopes. Increasing the levels of fill, hereby increasing density of the preconditioning matrix and consequently increasing the computation cost of applying it, yields an increase of performance, but does not outperform multigrid.

3.3 Annulus

The third test consists of an annulus in which two-dimensional whole Earth convection is simulated. The core of the Earth, with a radius of approximately half that of Earth (Dziewonski and Haddon, 1974), is considered as the hollow component of the annulus (Figure 14). The mesh is made up of regular quadrilateral elements which are stretched along their radial axis as their distance to the core increases (Figure 15). The initial temperature profile is described using a sinusoidal perturbation:

$$T_{initial} = -\frac{r - R_{core}}{R_{Earth} - R_{core}} * (T_{cmb} - T_{surface}) + T_{cmb} + \frac{\zeta}{100} \tag{32}$$

where r is the distance from the core-mantle boundary (cmb) in km, R_{core} is the radius of the core (in km), R_{Earth} is the radius of the Earth in km, T_{cmb} is the temperature at the cmb, $T_surface$ is the temperature at the surface of the earth and $zeta$ is a random number between 0 and 1. The randomness is introduced in order to stimulate asymmetrical convection. Temperature boundary conditions are set to T_{cmb} and T_{surf} for the cmb and surface of the Earth respectively.

Essentially, the "Case 1" model from Tosi et al. (2015) is replicated in which temperatures are dimensionless. Thus, T_{cmb} is 1 and T_{surf} is 0. The Courant-Friedrich-Lewy condition (Courant et al., 1928) is set to 0.5. Velocity boundary conditions are set to free-slip for both the cmb and surface of the Earth. The temperature dependent viscosity is purely linear and is described as

$$\mu_{eff} = \mu_{linear} = \exp(\gamma_T * T) \tag{33}$$

where μ_{eff} is the effective viscosity in Pas, γ_T is a parameter governing the viscosity contrast owing to temperature. Because of the dependency of viscosity on temperature, the initial viscosity structure is equal to the initial temperature distribution.

The setup of temperature and viscosity evolves into a set of plumes which grow radially outward in a mushroom type shape (Figure 14). Root Mean Square (RMS) velocities rapidly increase at the start of time stepping. Asymmetry, caused by the randomness term (ζ in Eq. 32), stimulates neighbouring plumes to coalesce into a single larger plume. These coalescing plumes correspond to bumps in the v_{rms} profile as high velocities are observed during fusion.

4 different preconditioners with CG are tested in this experiment (Figure 16). Using an ILU preconditioner here results in long computation times. Increasing the levels of fill decreases computation time and shows a marginally shallower slope of 1.49. A shallow slope, compared to ILU preconditioners, is observed if using

multigrid. This slope, however, is not linear in the log-log scale axis frame. It decreases at higher resolutions. The slope of multigrid at lower resolutions is comparable to that of ILU at high resolutions.

Similar results are observed for the classic 2D box benchmark (Figure 13). The annulus test is merely a change in geometry, while boundary conditions, viscosity and other parameters remain equivalent to that of Tosi et al. (2015).

3.4 Convecting cube

A first step into modelling 3D subduction models is to run a simulation of an isoviscous cube. In this cube, the initial temperature profile is linear from top to bottom and is includes a sinusoidal perturbation to stimulate the evolution of up- and downwellings (Figure 17) and writes

$$T_{initial} = T_{top} + \frac{L_z - z}{L_z * (T_b - T_{top})} - 100 \cos\left(\frac{3.123\pi x}{L_x}\right) \sin\left(\frac{\pi z}{L_z}\right) \cos\left(\frac{3.26\pi(0.8y + 0.2x)}{L_y}\right) \tag{34}$$

where T_{top} and T_b are the initial temperature at the top and bottom of the cube (in degrees K) and L_x,L_y and L_z specify the spatial dimensions of domain in the x, y and z direction (in km). The domain is of size $1000 \times 1000 \times 1000$km. The imposed temperatures are 500K at the top of the domain and 4000K at the bottom . Along the six faces of the cube free slip boundary conditions are imposed.

Viscosity in this test is set to $\mu_{eff} = 10^{21}$Pa s. By prescribing a non-uniform density, convection is stimulated. The initial density distribution is linear with temperature and writes

$$\rho_{init} = 3300(1 - 10^{-5} T_{init}) \tag{35}$$

where the density (ρ) is expressed in kg/m^3. The result of this setup is a vigorously convecting domain (Figure 17) caused by the buoyancy forces that result from the temperature dependent density changes.

From Figure 11 it is observed that a direct solver can provide a fast solution to a system of linear equations. However, this figure only visualizes raw computation speed. Operations required to carry out the solves of a direct solver are $N \log N$ and $N^{4/3}$ in 2D and 3D respectively (Li and Widlund, 2007). RAM usage required for directly solving a system scales equally (Calo et al., 2011). Similarly, CPU times for running 3D simulations are vizualised in May et al. (2013) and show a similar behaviour to memory requirements. Not only does the size of the matrix increase when switching from 2D to 3D, but the sparsity of it decreases as well. Thus, when comparing solving speeds using any solver of a 2D FEM matrix versus a 3D FEM matrix of the same size, the 2D system would show a lower solving time. It is for this reason, direct solvers are not the preferred method when running 3D models at even a moderate resolution, even when using the most powerful parallel supercomputer.

In order to test preconditioner performance five time steps of this convection test were carried out. The results in Figure 18 represent the total computation time required for purely the solving part of the calculation. The results reveal that ILU with 0 levels of infill performs the best. This preconditioner is the cheapest to apply. increasing the levels of fill, hereby increasing the accuracy of the preconditioner, yield slower computation times. Multigrid with one level of coarsening shows the highest slope, meaning it is not the preferred method for this simulation, especially at higher resolutions. Finally, an overlap of computation time results is observed between multigrid with more than one level of coarsening and ILU(2).

3.5 Subduction

So far, only simplified simulations of the realistic, large scale systems such as subduction or whole Earth mantle convections, have been tested. These simple models are a critical factor in understanding the response of preconditioners and solvers to varying conditions (i.e. viscosity, geometry of elements, 2D/3D etc.). A final test to examine preconditioner performance is to run a realistic full 3D thermomechanically coupled subduction model.

In a paper by Schellart and Moresi (2013), the authors describe a $4000 \times 1600 \times 660$ km domain in which a one-layer overriding plate and a three-layer subducting oceanic plate sit on top of the low viscous upper mantle ($\mu_{um} = 1.57 \times 10^{20}$). As this model provides an accurate description of subduction dynamics (according to the authors), and the paper gives a detailed description of the setup used, this model is replicated in ELEFANT to test preconditioner performance of a top of the line subduction model. The overriding plate is set to $\mu_{op} = 200\mu_{um}$ and the linear viscosities of the three-layered subducting plate are set to $\mu_{max} = 10^{23}$ at the top, followed by a viscous strong layer of $700\mu_{um}$ and a lower weak layer of $40\mu_{um}$. An overview of this setup is visualized in Figure 19, in which viscosities are dimensionless.

Along the side-faces and top face of the domain, free-slip boundary conditions are imposed. On bottom boundary, representing the 660 km discontinuity, a zero-velocity is imposed. This boundary is impenetrable and will encourage stagnation of the subducting plate. The initial slab angle is set to 29°. To show the evolution of the system (Figure 20), a resolution of $100 \times 41 \times 17$ is used leading to 69700 elements, 72144 nodes and a matrix size of 229068×229068. At this resolution, using a single thread of the HP workstation, calculating 450 time steps took 46 hours with a multigrid + CG inner solving approach.

For this type of modelling, the resolution is considered as low, with elements of size $38 \times 38 \times 38$ km (when using a resolution of $100 \times 41 \times 17$ elements). In order to obtain a more detailed description of material behaviour, the Marker and Cell method (MAC) is used (Harlow and Welch, 1965; van der Giessen and Aref, 2003). By using massless particles distributed over the whole domain, materials with varying parameters can be traced throughout even a highly deformed fluid. Within a single element, multiple (often tens or hundreds) of these particles can be placed to get a more detailed picture than purely using the elements. Computing the advection and building the markers is computationally costly. In this simulation 3.5% of total computation time consists of marker calculations. For the analysis of solver performance only the pure solve time is taken into account (i.e. marker calculation time is excluded).

The evolution of the subduction simulation is shown in Figure 20. The subducting slab sinks into the upper mantle and stagnates at the 660 km discontinuity. The overriding plate forms a wedge in the trench zone. The rate of thickening of the overriding plate in the wedge is considerably higher than in Schellart and Moresi (2013). Furthermore, the bottom layer of the subducting plate shows a high amount of stretching along the side of the plate. This behaviour is not observed in the original paper. In this paper, the slab folds over itself when stagnating against the 660 km discontinuity. The results in Figure 20 do not show these geometries. Rather, the subducting slab collapses onto itself. Interpretation of the results is limited by the resolution and marker density. The thinning of the subducting plate (especially at time steps higher than 250) results in the visibility of the individual markers. The resolution that is used is low and thus the amount of markers per cell is low. Increasing the resolution and amount of makers could yield more accurate and better interpretable results.

The accumulative solve time of 5 time steps as a function of resolution is visualized in Figure 21. As only three resolutions are tested, the black regression lines may not provide an accurate interpretation of the data points. At lower resolutions, the relatively cheap ILU(0) preconditioner is preferred over multigrid, ILU(2) and ILU(4). Higher levels of fill of the ILU preconditioner result in more expensive calculations. However, the absolute number of iterations required for convergence decreases as more levels of fill are used (Figure 22). The cost of the individual calculations required for ILU(2) and ILU(4) is not fully counteracted by the decrease in iterations because ILU(2) and ILU(4) yield the slowest computation times (Figure 21). In contrast to ILU preconditioners, multigrid preconditioners do not require many more inner iterations when increasing the resolution. Inner iteration count does not increase substantially (from 9 iterations at a low resolution to a maximum of 12 iterations at a higher resolutions). This feature separates multigrid from many other forms of preconditioning. Increasing the number of multigrid levels only results in a 1-iteration difference (11 iterations compared to 12 for 2 and 4 multigrid levels respectively). Computation times of multigrid preconditioners are similar. Multigrid with 1 level of coarsening outperforms the other multigrid approaches. Similar to ILU, the preconditioner that is cheapest to apply, is the fastest.

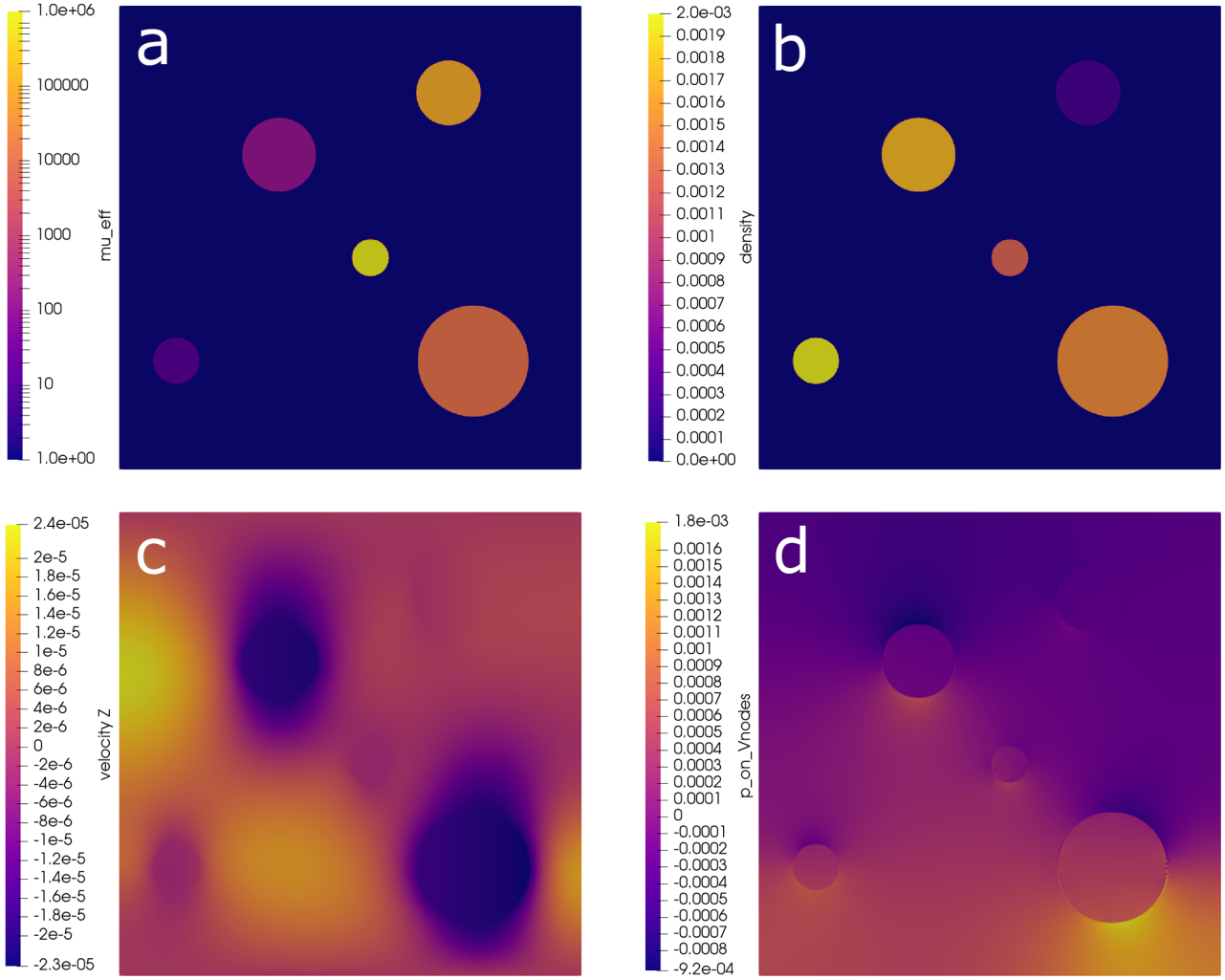

Figure 10: Solution of the NSinker test using a resolution of 700 × 700 elements showing a) viscosity b) density c) velocities in the z-direction d) pressure distribution with high pressures at the bottom and low pressures at the top.

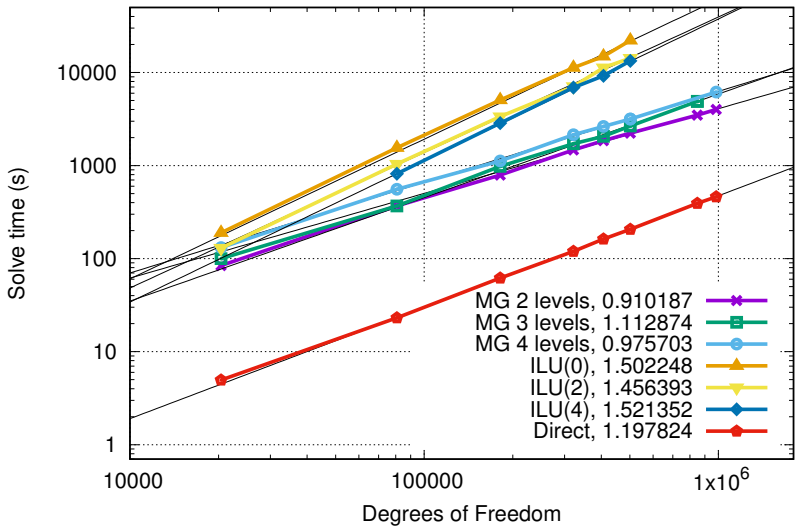

Figure 11: Performance scaling of 6 preconditioners with a CG solver and 1 direct solver case as a function of degrees of freedom/matrix size.

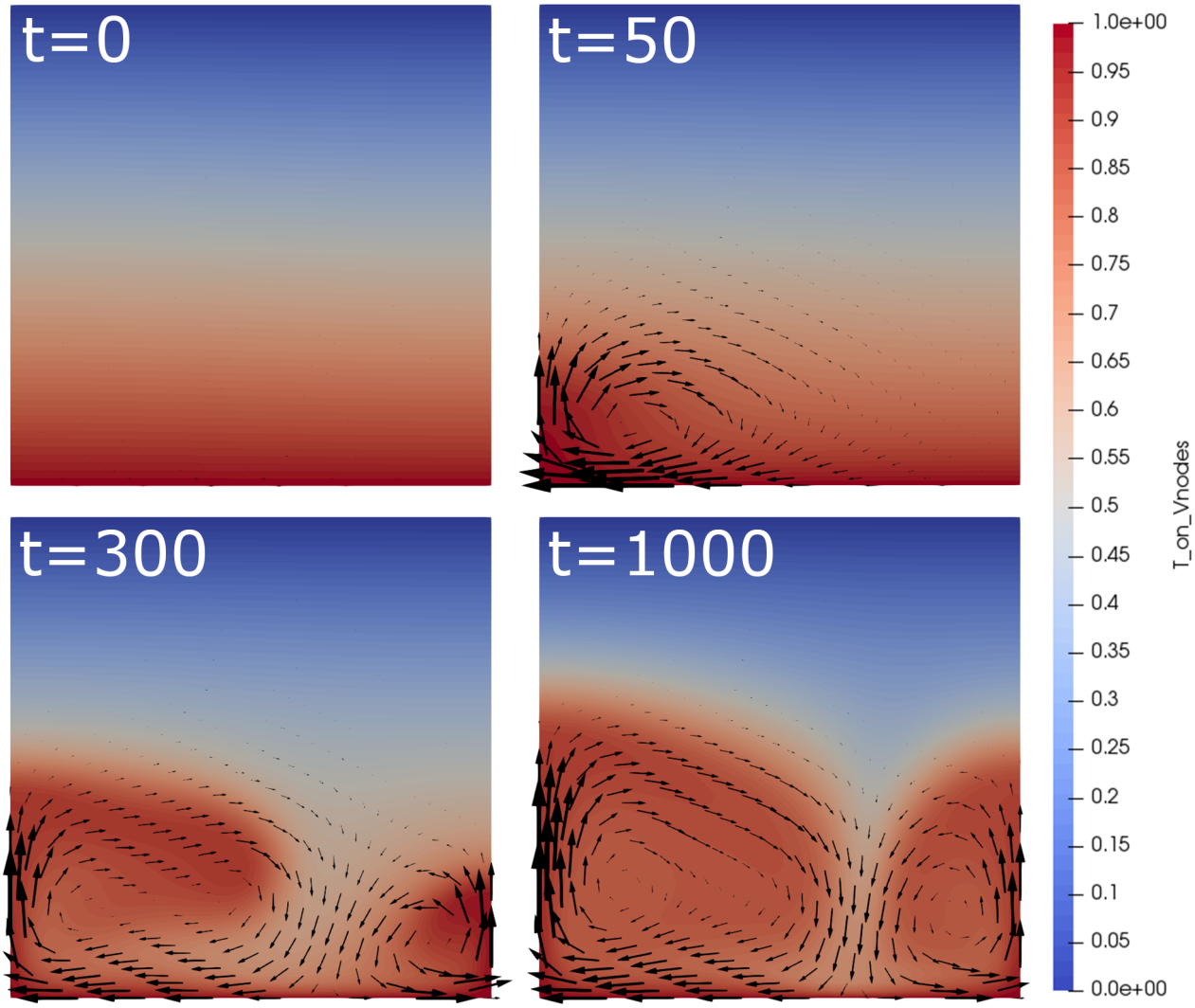

Figure 12: Temperature evolution of the stagnant lid 2D box convection benchmark of 64×64 elements. The black arrows visualize the velocity field. A convection cell forms at the bottom left and right of the domain

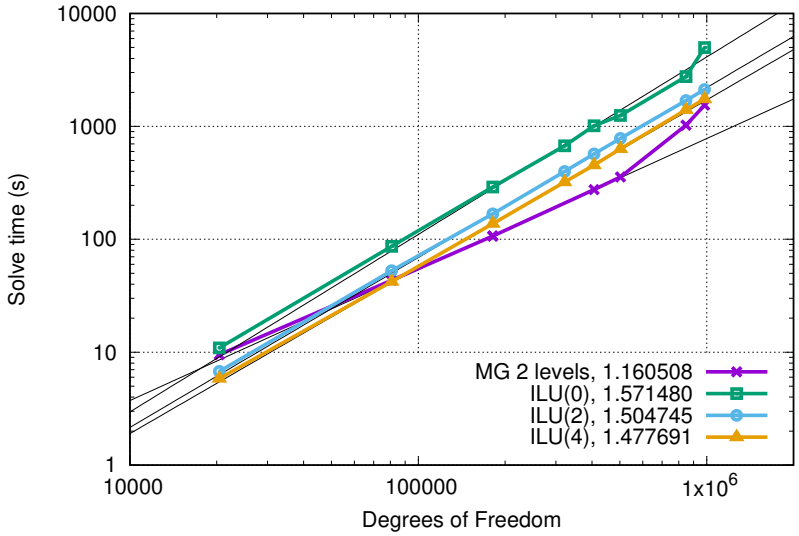

Figure 13: Performance scaling of 4 preconditioners with a CG solver when calculating a single time step.

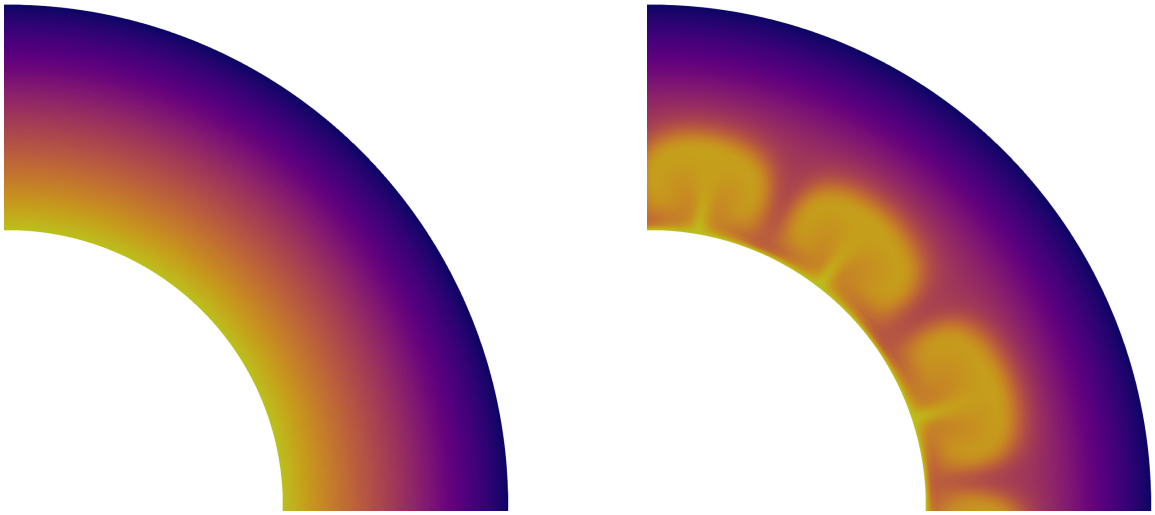

Figure 14: Left: quarter section of the annulus showing the initial temperature distribution. Right: Temperature distribution after 200 time steps.

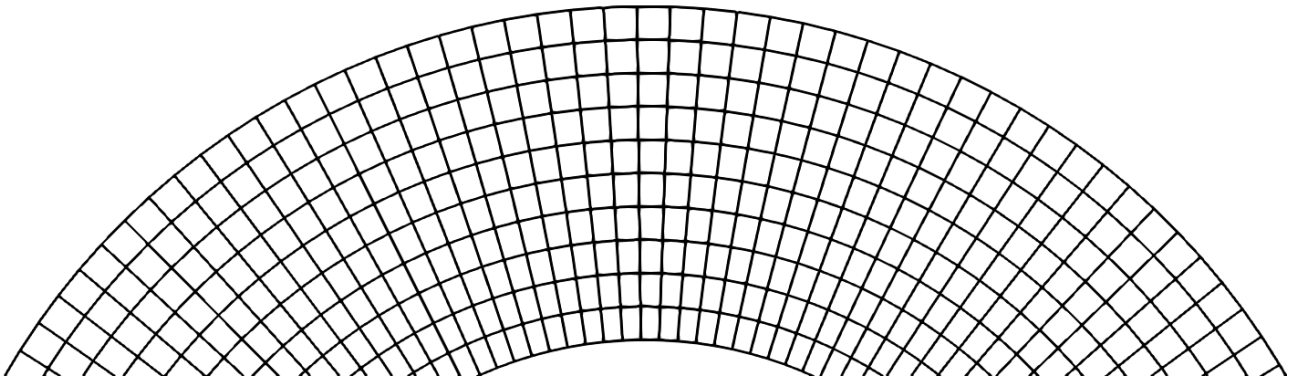

Figure 15: Wireframe representation of the annulus mesh showing stretched elements towards the center and near-square elements on the outer boundary.

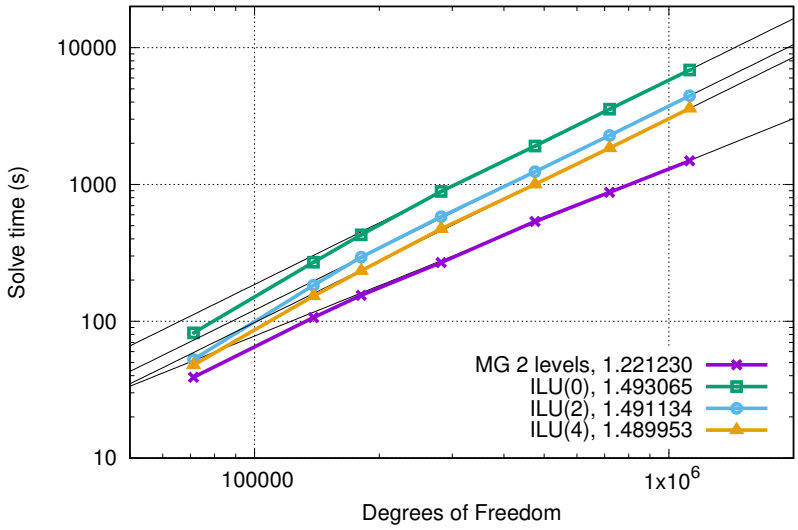

Figure 16: Performance scaling of 4 preconditioners with a CG solver as a function of degrees of freedom/matrix size for a single time step.

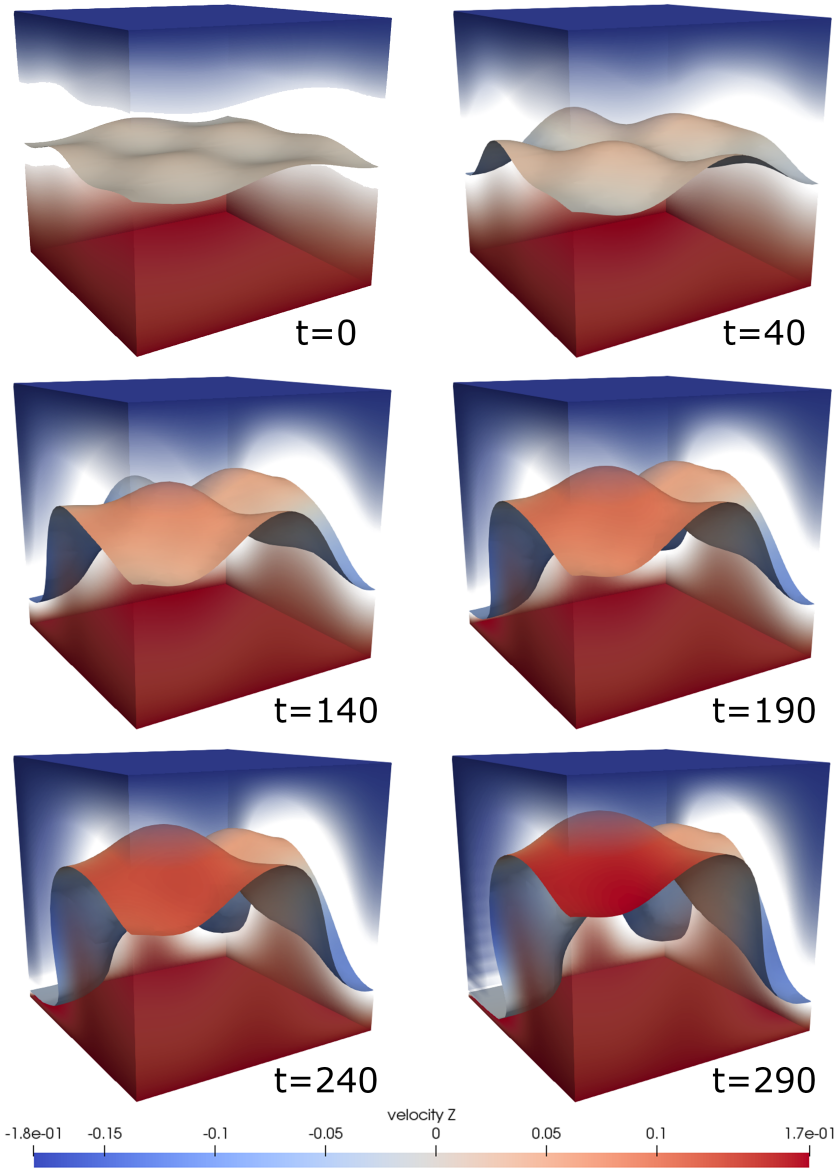

Figure 17: Time evolution of a convecting cube ($30 \times 30 \times 30$ elements) showing a 2250K isocontour coloured by z-velocity. The colours on the faces of the cube represent temperature. The initial sinusoidal temperature profile evolves into a system of multiple up- and downwellings.

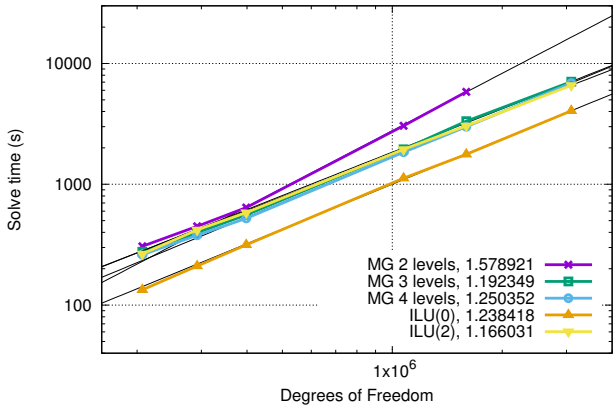

Figure 18: Solve time of five preconditioners with a CG solver for the convecting cube test when calculating 5 time steps.

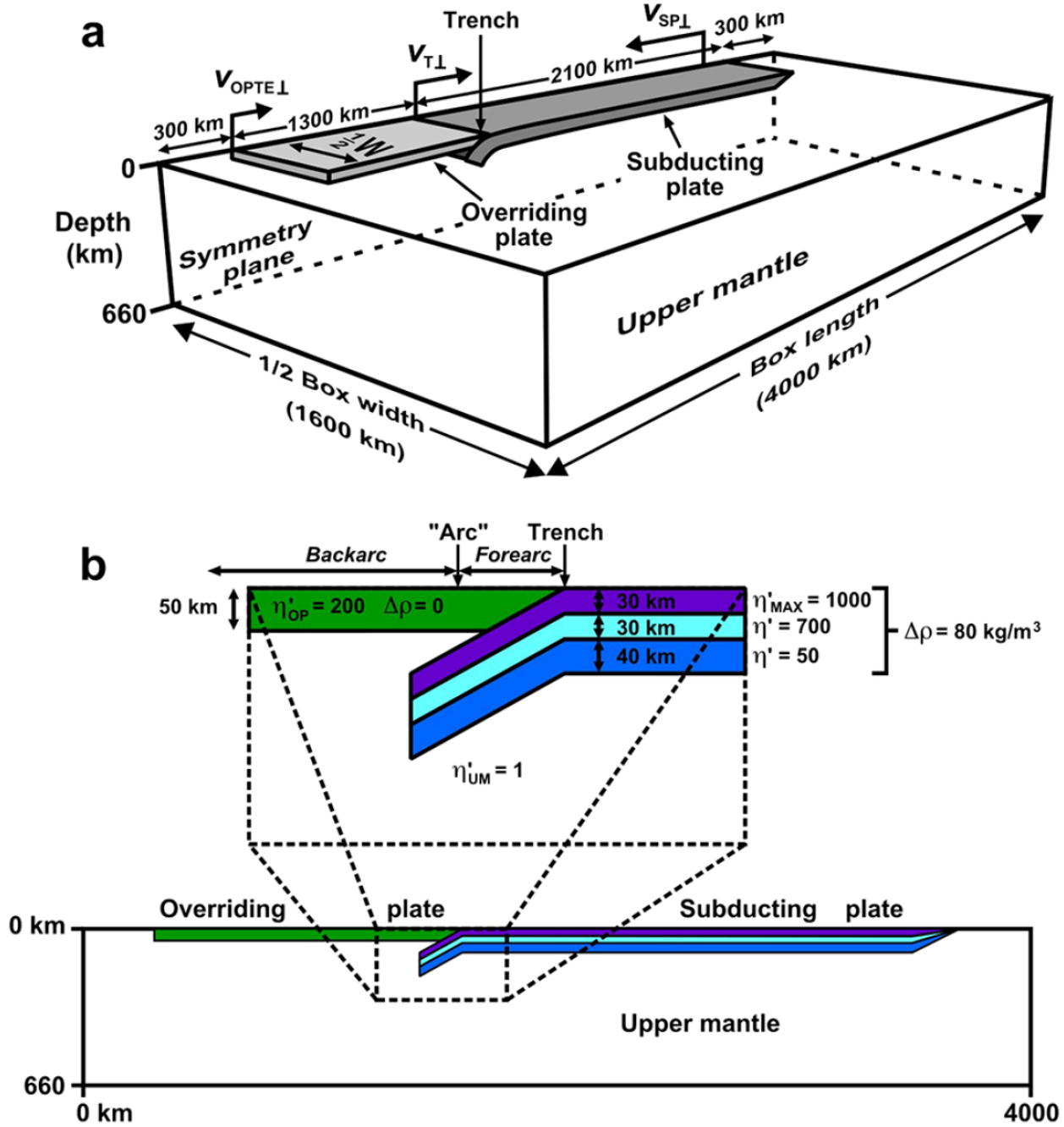

Figure 19: a) Overview of the initial domain showing the two plates in the upper mantle material. b) Side view of the plates showing three-layered subducting plate and single-layered overriding plate as well as viscosity values used in the original model. From Schellart and Moresi (2013).

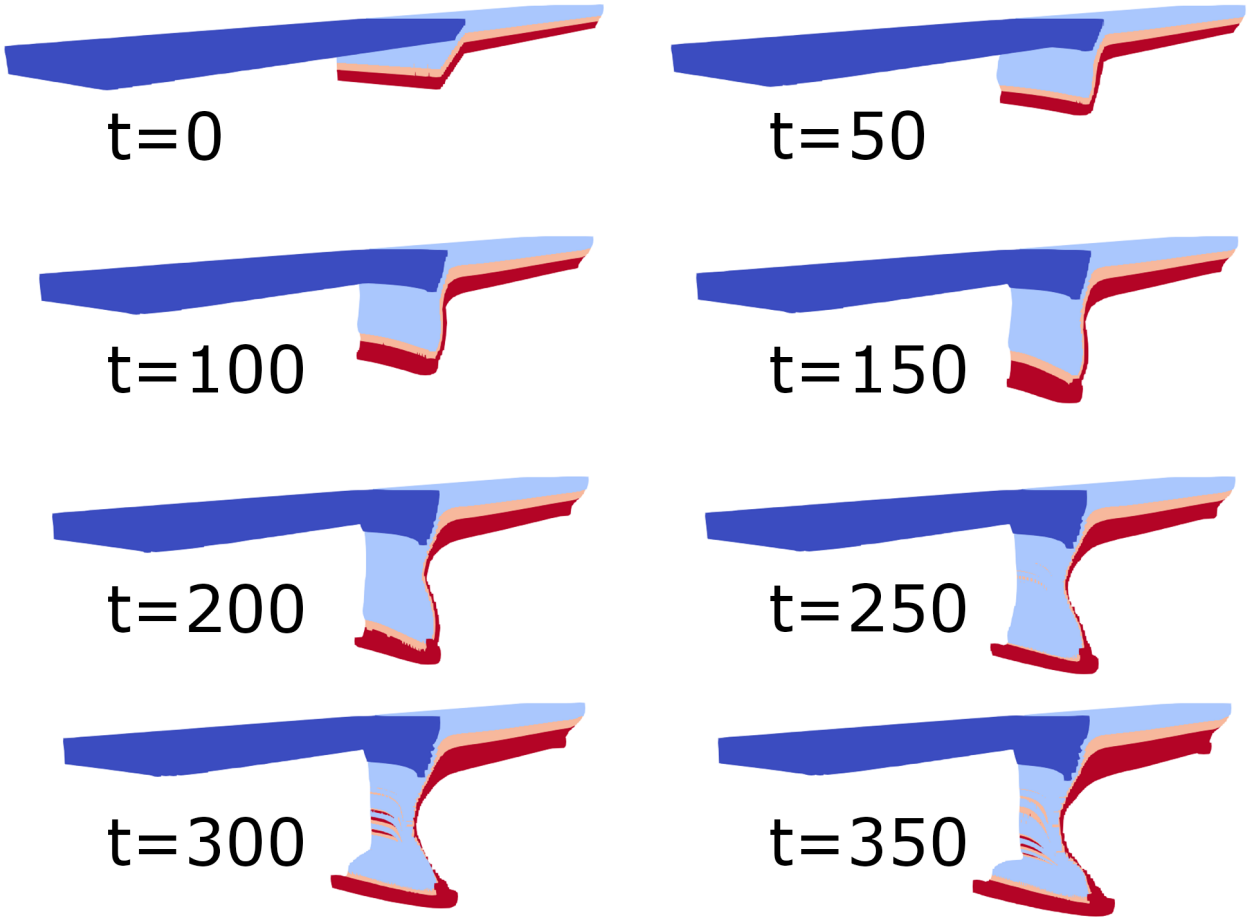

Figure 20: Evolution of the subduction model. The colours represent the different materials. Four of the five materials are visualized here. The upper mantle material is excluded for readability.

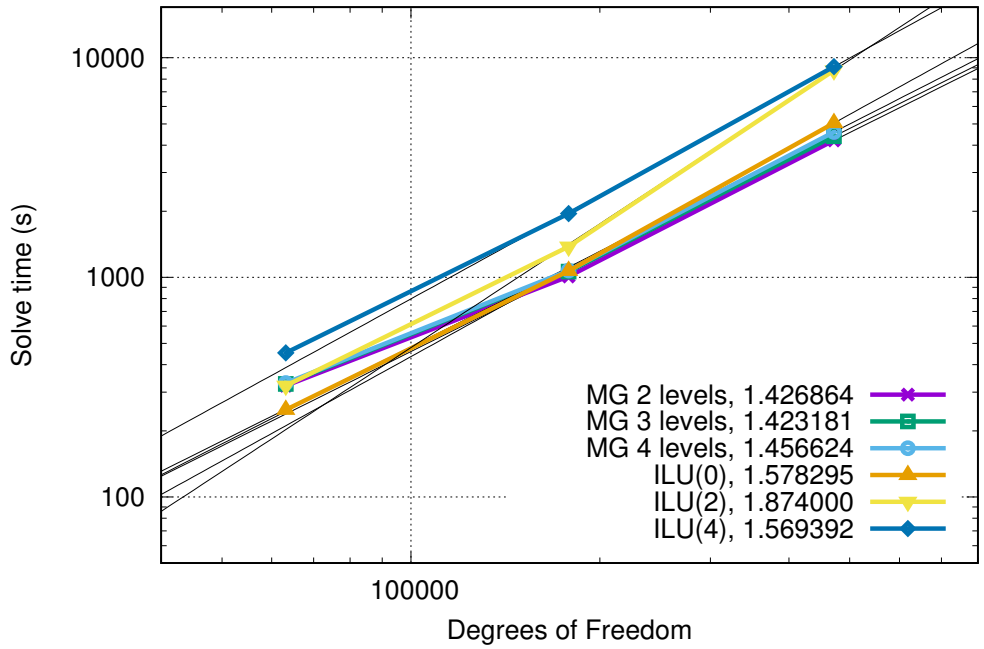

Figure 21: Performance of 6 preconditioners with a CG solver as a function of degrees of freedom for the subduction simulation. Regression lines are plotted in black.

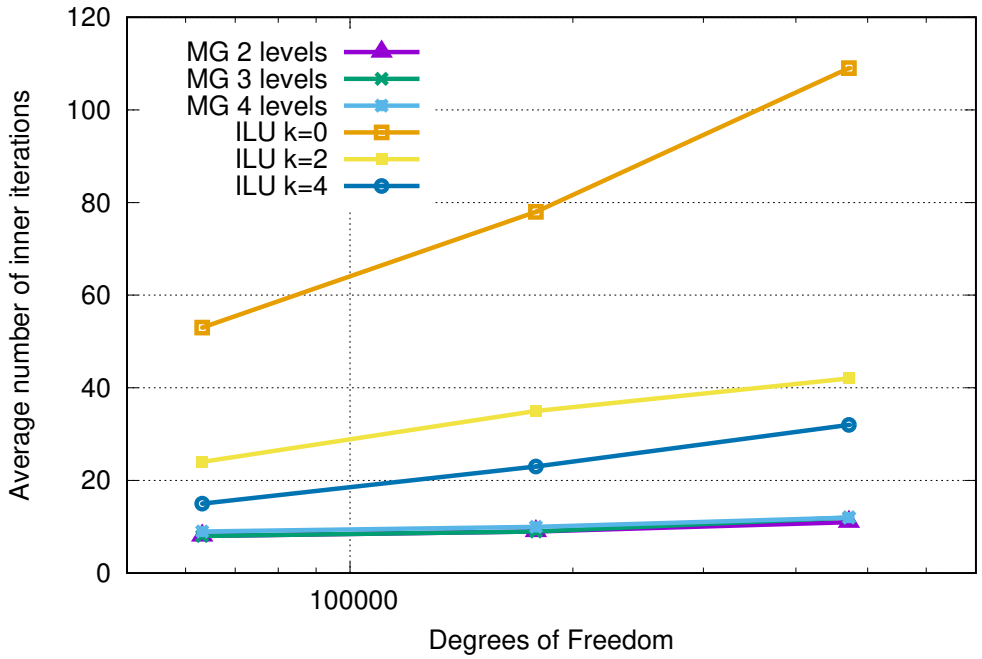

Figure 22: Inner iterations as a function of resolution for the subduction simulation. Multigrid requires the least iterations and ILU(0) the most.

4 Discussion and conclusions

Preconditioning a system of linear equation is a tradeoff between accuracy and cost of setup and execution. A very simple preconditioner (diagnoal/Jacobi) is very cheap to apply but only works for small and simple problems. This simple preconditioner can work for isoviscous cases when the diagonal elements of A are all in the same order of magnitude. Variations in the diagonal cause the preconditioner to perform very badly, even though it is the cheapest preconditioner to apply.

Direct solvers are limited to a certain resolution of the system because of RAM and CPU usage issues. RAM and CPU power that is needed for solving a system does not scale linearly with resolution. Thus, solving a $100 \times 100 \times 100$ cube would be impossible because it requires an excessive amount of RAM. Furthermore, preconditioning a system when using a direct solving method is useless because a direct solve is based on Gaussian elimination, making it independent of matrix and vector entry values.

Assuming a different non-zero structure (with respect to the operator in $\boldsymbol{A}\vec{x} = \vec{b}$) in the preconditioner is the approach of non-zero fill-in types of Incomplete Lower Upper factorization preconditioning. Comparing ILU(0) and ILU(2) in the case of a 3D isoviscous box reveals that making the matrix more dense, by applying a denser preconditioner (relative to the matrix), makes the system too complex. Calculations carried out with the non-zero structure of the matrix for 3D systems is already very costly. Thus, increasing the density further complicates the computation.

Multigrid methods are per definition not the best for all problems. Tackling isoviscous problems proves to be very inefficient with a multigrid preconditioner (Figure 18). However, when solving a system with (sharp) viscosity contrasts, multi-grid preconditioning proves to be useful. The fact that efficient multi-grid implementation results in an independency on resolution is advantageous for high resolution simulations. Some tuning could improve performance of the multi-grid preconditioner. The amount of levels used as well as the smoothing factor of the residual are parameters which can be conveniently tweaked using PETSc.

Some preconditioner performance results show a kink in their slopes (Figures 13,16, 18, 21). This could have multiple causes. First, the HP workstation used to run the models is under variable load at all times. At a certain hour, all its cores may be unused while the next hour all of the 40 available threads are under 100% load

because of a separate parallel computation. Second, performance results of models which include time stepping should be averaged over all the time steps. In many cases only a small amount of time steps were used because of large computation times. To be able to acquire more accurate performance results, more time steps should be carried out and an average should be calculated.

The advantage of multigrid over other preconditioners when simulating systems with viscosity contrasts becomes obvious in Figure 22. The iterations required for inner convergence are significantly lower for multigrid methods than ILU methods. This results in a direct increase in solve time. However, individual iterations of multigrid preconditioned CG are more computationally expensive. Thus, five times less iterations does not result in 5 times faster solve times. The lowering of iteration count is due to lowering of the condition numbers of the matrix. Multigrid methods do a better job in approximating A^{-1} and when applying the preconditioner to the matrix yields a matrix closer to the identity matrix.

The resolution for which the results of the subduction simulation are shown (Figure 20) is too low. In order to acquire more accurate results, the resolution needs to increase. At each time step, the markers within a cell get averaged to yield a cell-averaged variable (such as density or viscosity) for that cell. Increasing the amount of markers can yield a more detailed picture of the model, however, there is a limit. If the standard resolution of the model is not high enough, increasing the amount of markers will not add any more insights into the dynamics of the system. The amount of markers used, as well as the accuracy of the model results, are separate from the performance results. The fact that multigrid + CG performs the best when modelling subducting still holds, independent of markers and accuracy of the model outcome compared to the paper by Schellart and Moresi (2013).

The integration of PETSc into ELEFANT opens the door to experimentation with preconditioners and solvers. By documenting the capabilities of PETSc I have shown that correctly tuned preconditioners and solvers are able to achieve high levels of performance as well as increasing robustness of the solve step.

5 simPyFEM

Drawing more attention to the intricacies of modelling in Earth Sciences is an important step in capturing student's attention. By giving out a versatile and easy to understand tool, which can help to be the first step into this field their interest is sparked. To make an already flexible code appealing to all the more people, simpleFEM was rewritten to Python.

From the website of C. Thieulot (`http://cedricthieulot.net/simplefem.html`): "SimpleFEM is a Fortran code built for educational purposes. It showcases functional illustrations of the various features of a FEM code: the mesh and its connectivity, the shape functions and their derivatives, the numerical integration, the assembly phase, imposing the boundary conditions, and the system solve". Fortran (90) is still a widely used programming language in the scientific and banking community. It is a relatively simple and barebones language experiencing fast purely numerical calculations compared to other languages.

SimpleFEM is an adaptive piece of code. Many different versions exist (Figure 23), with more renditions about to arrive. The unadorned simpleFEM involves a 2D box with a uniform viscosity, density and temperature. As a result, it is built with a clear structure and is easily adaptable. The simultaneous simplicity and versatility make it well suited for educational purposes. A flaw is that a new version has got to be created when extending simpleFEMs features. Thus one can not simply toggle features on and off like one can in ELEFANT. Python can offer the user some of these on-off switch features without overcomplicating the code.

Python provides a more user-friendly language and structure than F90. It offers a wide variety in "modules" the user is able to import. Some of these modules include NumPy, a scientific computing package which offers tools such as matrix multiplication, Fourier transforms and various solving methods for linear systems and makes use of NumPy arrays (van der Walt et al., 2011) and MatPlotLib which is a plotting library with a MATLAB-like interface.

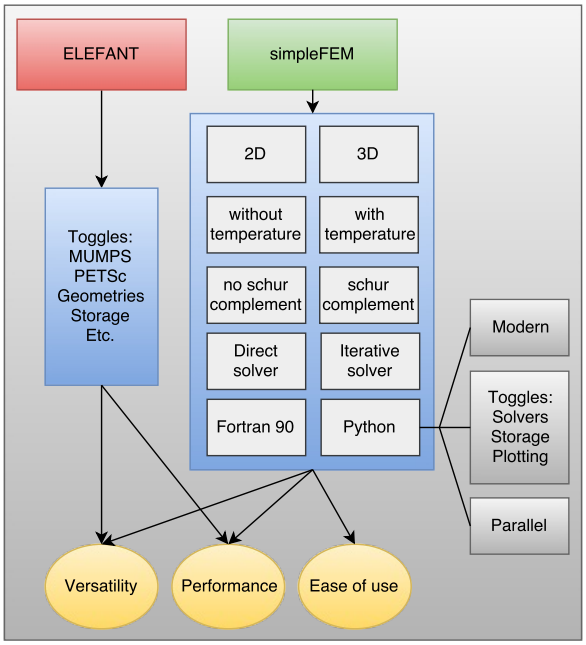

Figure 23: Code overview of ELEFANT and simpleFEM showing the strong points of both codes. Where ELEFANT uses toggles, simpleFEM branches out into separate versions of the same program.

Python forces the user to indent code to indicate looping or if-else structure. Especially, for less experienced programmers, this is a good habit to pick up early on. When exploiting some of the packages, features and parallel capabilities of Python, one can speed up numerical codes through a relatively simple procedure.

As Fortran 90 is not used by many organisations outside of the scientific and banking communities, teaching students of Earth Sciences a second language is a smart choice. To kickstart this, some pre-existing code has

to be rewritten. To achieve greater efficiency, and make the experience for students more streamlined, solver and preconditioners will be implemented in the Python code, alongside the pre-existing direct BLAS/LAPACK implementation. In this thesis, I will present a set of simpleFEM versions in Python and improve performance of the Python code and compare it to its F90 predecessor. The code is called simPyFEM and instructions and download links can be found on my GitHub (`https://github.com/jobmos/simPyFEM`).

In the end, if the user wants to run simPyFEM, nothing more is needed than the install of Anaconda (Anaconda, 2018) which installs, additionally to Python, the most commonly used packages. No external packages such as LINPACK, BLAS or gnuplot are required. Everything is contained in 1 file. This avoids confusion for the students as well as them having the ability to easily tinker with the code at home. One of the more easy ways to get Fortran code compiled and running on Windows machines is to install a UNIX terminal emulator such as CygWin. This has proven to be cumbersome for many students. Python has its own program for all operating systems, further increasing versatility. Python is an interpreted language; the program does not need a preceding compilation step.

5.1 Methods

The basic version of simpleFEM directly communicates with LINPACK and BLAS to solve the linear system. Two calls are required. The first call factors the matrix by gaussian elimination and estimates the condition number of the matrix. The second carries out the solve of the linear system. Some additional variables, which can be confusing for the user, are also fed as input/output. Python provides a more neat solution. One first imports a module, in this case NumPy. The solve call to Numpy is as follows

```
x=numpy.linalg.solve(A,b)
```

and takes cares of everything behind the scenes. In the end, NumPy is still built on calls to LINPACK, but the user is not exposed to this directly. Not only a direct method is included in the modules. Iterative solvers can also be applied to linear systems using SciPy. Calling the conjugate gradient solve subroutine is handled in a similar manner:

```
x,info=scipy.sparse.linalg.cg(A,b)
```

where x is the solution and info is a return integer providing information on solve convergence (0 meaning a successful solve, and anything other than 0 indicates a non-convergent/divergent pattern in the residual). Parameters which are not mandatory but may improve performance (initial guess, tolerance, maximum number of iterations, preconditioners) can be given as additional input to the call. Many other popular iterative methods, such as GMRES, MINRES, and biCGStab are also supported. For further documentation see the SciPy documentation (`https://docs.scipy.org/doc/`).

Here I will present a number of tests comparing the performance of various solving techniques including direct and iterative solves as well as penalty versus the Schur complement method using a diagonally preconditioned conjugate gradient method and direct method as inner solve. In the measurements of solve time, the setup of the program (setting boundary conditions, setting up the matrix and converting the matrix to CSR) will be excluded, essentially producing a clean comparison of pure solve times. Tests are run on a 2013 MacBook Pro with a dual core 2.6GHz Intel Core i5 (hyperthreaded up to 4 threads) and 8 GB of 1600MHz DDR3 RAM unless mentioned otherwise.

When directly translating Fortran simpleFEM to Python simPyFEM, one has to account for indexing. Fortran starts counting array indices at 1 while Python start at 0. Looping over the length of an array in Fortran is: $i = 1, 2, ..., n$. In Python, a loop over array indices would be: $i = 0, 1, ..., n - 1$.

To stimulate experimentation with the code, some simple argument passing has been integrated. The user can provide a resolution at which the model should be ran (Listing 1).

```
    if int(len(sys.argv)==3):
      nnx=int(sys.argv[1])
      nny=int(sys.argv[2])
    else:
      nnx=32
      nny=32
```

Listing 1: Piece of Python code allowing for command line passing of resolution.

```
def solve_cg(A,B,guess=None):

    solVecNorm=numpy.dot(B,B)
    residual = []
    max_its=1000

    if not guess:
        guess = numpy.zeros(Nfem)

    r = B - sparse.csr_matrix.dot(A, guess)
    p = r

    for i in range(max_its):
        Ap = sparse.csr_matrix.dot(A,p)
        alpha = numpy.dot(r,r) / numpy.dot(p, Ap)
        guess += alpha * p
        r_new = r- alpha * Ap

        if residual[i] < 1e-8 or i == max_its-1 :
            if i == max_its-1:
                print("No convergence, maximum iterations reached")
            else:
                print("Convergence reached: ",i," iterations.")
            break

        beta = numpy.dot(r_new,r_new)/numpy.dot(r,r)
        p_new = r_new + beta * p

        p=p_new
        r=r_new
    return guess
```

Listing 2: Python function for the Conjugate Gradient algorithm. Translated from Algorithm 6.18 in Saad (2003).

Algorithm 2: Uzawa iteration

Input : $\boldsymbol{A}$,$\boldsymbol{G}$,$\vec{f}$,$\vec{h}$

Output: $\vec{v}$,p

Choose v^0,p^0

while *not converged* **do**

  $v_{k+1} = A^{-1}(f - Gp_k)$

  $p_{k+1} = p_k + \alpha(G^T v_{k+1} - h)$

end

```python
def solve_pcg(A,B,guess=None):

    solVecNorm=numpy.dot(B,B)
    residual = []
    max_its=1000

    if not guess:
        guess = numpy.zeros(Nfem)

    r = B - sparse.csr_matrix.dot(A, guess)
    z = sparse.csr_matrix.dot(M,r)
    p = z

    for i in range(max_its):
        Ap = sparse.csr_matrix.dot(A,p)
        alpha = numpy.dot(r,z)/numpy.dot(Ap,p)
        guess += alpha * p
        r_new = r - alpha * Ap

        residual.append(numpy.dot(r_new,r_new)/solVecNorm)

        if residual[i] < 1e-8 or i == max_its-1 :
            print("No convergence, maximum iterations reached")
        else:
            print("Convergence reached: ",i," iterations.")
        break

        z_new = sparse.csr_matrix.dot(M,r_new)
        beta = numpy.dot(r_new,z_new) / numpy.dot(r,z)
        p_new = z_new + beta * p
        p=p_new
        r=r_new
        z=z_new
    return guess
```

Listing 3: Python function for the Preconditioned Conjugate Gradient algorithm. Translated from Algorithm 9.1 in Saad (2003).

When running a Python code in the terminal, the structure is as follows:

```
Python3 name_of_code.py argument1 argument2
```

and thus `argv[0]` is `name_of_code.py` and `argv[1] & argv[2]` correspond to the command line inputs `argument1` and `argument2` respectively. `length(argv)=3` in the case that resolution is specified. Not stating resolution results in setting the default of 32×32 nodes.

The most convenient and easy way to solve the system of linear equations is to use the direct NumPy approach `numpy.linalg.solve(A,b)`. This takes the dense matrix A and vector b and returns the solution vector. A better approach would be to execute a sparse direct solve which takes not the dense form of A but rather takes the CSR form of A. Matrices can easily be converted from dense to sparse by using `scipy.sparse.csr_matrix(A)` and are solved by `scipy.sparse.linalg.spsolve(A,b)`. In this manner, no further knowledge or complex subroutines are exposed to the user. From a performance point of view, this is not ideal. However, the code must remain easy to understand so a conversion is the preferred method.

A direct solver is not the only method of finding the solution to $Ax = b$ in Python. An iterative method, such as CG, is also supported. The routine `scipy.sparse.linalg.cg` supports both dense and sparse versions of the linear operator, making it more versatile. To gain a better understanding of CG (and to optimize it), I wrote out the algorithm as a Python function which is available as a function in the code and is open to

```
def uzawa1(KKK,G,Nfem,nel,rhs_f,rhs_h):
  alpha=1e3
  tol=1e-6
  niter=250
  KKKmem=KKK
  V_diff = []
  P_diff = []
  for iter in range(0,niter):
    B=rhs_f-numpy.matmul(G,Psol)
    B=numpy.linalg.solve(KKK,B)
    Vsol=B
        Psol=Psol+alpha*(numpy.matmul(G.transpose(),Vsol)-rhs_h)
        V_diff.append(numpy.max(abs(Vsol-Vsolmem))/numpy.max(abs(Vsol)))
        P_diff.append(numpy.max(abs(Psol-Psolmem))/numpy.max(abs(Psol)))
        Psolmem=Psol
        Vsolmem=Vsol
        if max(V_diff[iter],P_diff[iter]) < tol or iter == niter-1 :
            return (Vsol,Psol,V_diff,P_diff)
```

Listing 4: Python code of the basic Uzawa algorithm.

exploration and experimentation (Listing 2). Similarly, the Jacobi preconditioned conjugate gradient algorithm in translated to Python in Listing 3.

The above methods can be directly applied to the system $Ax = b$ when a penalty method is applied. This method is used for solving constrained optimization problems. It turns constrained problems into a set of unconstrained problems. For the case of solving a system of linear equations, the solution obtained through this method is not an exact one.

Inexact solutions may be appropriate for solving particular kinds of problems. For solving PDEs, however, finding the exact solution is necessary. The parameters/variables of a PDE govern its spatial geometry which, if too complex, results in non-convergence. An additional drawback of the penalty formulation is that decreasing the penalty tolerance, hereby increasing accuracy of the solution (Bryan and Shibberu, 2005), yields an ill-conditioned problem (Eq. 20 when the condition number $C \gg 1$).

A second method of solving the Stokes equations is to solve the saddle point Stokes (Eq. 8) using the Uzawa method (Uzawa and Arrow, 1989). In Saad (2003), Algorithm 8.6 shows that the Uzawa iteration essentially breaks down into two lines of calculations (Algorithm 2), wherein G denotes the gradient operator and G^T the divergence operator.

When substituting v (line 3 in Algorithm 2) into p (line 4 in Algorithm 2), we obtain

$$Sp = G^T A^{-1} f - h \tag{36}$$

where $S = G^T A^{-1} G$ denotes the Schur complement. Thus essentially, we are solving the system

$$\begin{bmatrix} A & G \\ 0 & -S \end{bmatrix} \begin{bmatrix} v \\ p \end{bmatrix} = \begin{bmatrix} f \\ h - G^T A^{-1} f \end{bmatrix} \tag{37}$$

An important observation is that the Schur complement needs not to be explicitly formed. Rather, the computation of A^{-1} is avoided by an inner solve which is allowed to be either a direct or iterative approach such as CG to the problem

$$Ax_{k+1} = f - Gp \tag{38}$$

If line 4 in Algorithm 2 is obtained through CG, the system can be properly solved. The resulting algorithm in Python is given in Listing 4. In this case the direct Python solve is used as an inner solve. Algorithm 2 suggests an initial value for α. According to Braess (2007), the step size parameter α is assumed to be sufficiently small

```python
def uzawa2(KKK,G,Nfem,nel,rhs_f,rhs_h):
    alpha=1e3
    tol=1e-6
    niter=250
    V_diff = []
    P_diff = []

    B=rhs_f-numpy.matmul(G,Psol)
    B=numpy.linalg.solve(KKK,B)
    Vsol=B

    for iter in range(0,niter):
        q=rhs_h-numpy.matmul(G.transpose(),Vsol)
        phi=numpy.matmul(G,q)
        B=phi
        B=numpy.linalg.solve(KKK,B)
        h=B
        alpha=numpy.dot(q,q)/numpy.dot(phi,h)
        Psol=Psol-alpha*q
        Vsol=Vsol+alpha*h
        V_diff.append(numpy.max(abs(Vsol-Vsolmem))/numpy.max(abs(Vsol)))
        P_diff.append(numpy.max(abs(Psol-Psolmem))/numpy.max(abs(Psol)))
        Psolmem=Psol
        Vsolmem=Vsol

        if max(V_diff[iter],P_diff[iter]) < tol or iter == niter-1:
            return (Vsol,Psol,V_diff,P_diff)
```

Listing 5: Python code of the Uzawa algorithm including calculation of the optimal α.

in order to ensure convergence of the algorithm. We are able to calculate the optimal α exactly at each iteration as follows

$$\alpha_k = \frac{q_k{}' q_k}{(Gq_k)' A^{-1} G q_k} \tag{39}$$

indicating that a system of the type $Ax = Gq_k$ needs to be solved. To solve this, a supplementary vector ϕ_k is introduced yielding Listing 5.

Because the condition number of $GTA^{-1}G$ is high (Braess, 2007), a conjugate gradient approach is more effective. The resulting algorithm is called the Uzawa Algorithm with Conjugate Directions in Braess (2007) and, with the integration of sparse matrices is given in Listing 6.

5.2 Results

To document on the performance of simPyFEM, the three Uzawa methods as well as penalty formulations with direct, CG and preconditioned CG will be critically tested. A test is run for which the analytical solution is known. Body forces are described in the x and y direction as

$$\begin{aligned} b1 =&((12.0 - 24.0y)x^4 + (-24.0 + 48.0y)x^3 + (-48.0y + 72.0y^2 - 48.0y^3 + 12.0)x^2 + \\ &(-2.0 + 24.0y - 72.0y^2 + 48.0y^3)^x + 1.0 - 4.0y + 12.0y^2 - 8.0y^3) \end{aligned} \tag{40}$$

$$\begin{aligned} b2 =&((8.0 - 48.0y + 48.0y^2)x^3 + (-12.0 + 72.0y - 72y^2)x^2 + \\ &(4.0 - 24.0y + 48.0y^2 - 48.0y^3 + 24.0y^4)x - 12.0y^2 + 24.0y^3 - 12.0y^4) \end{aligned} \tag{41}$$

The Uzawa methods that are described in this thesis do not converge equally fast. Outer solver convergence (i.e. the convergence rate of the three Uzawa methods) is tested with the SciPy sparse direct inner solve. For testing

```python
def uzawa3(KKK,G,Nfem,nel,rhs_f,rhs_h):
    tol=1e-6
    niter=250
    V_diff = []
    P_diff = []

    B=rhs_f-sparse.csr_matrix.dot(G,Psol)
    B=spsolve(KKK_sparse,B)
    Vsol=B
    q=rhs_h-sparse.csr_matrix.dot(GT,Vsol)
    d=-q

    for iter in range(0,niter):
        phi=sparse.csr_matrix.dot(G,d)
        B=phi
        B=spsolve(KKK_sparse,B)
        h=B
        alpha=numpy.dot(q,q)/numpy.dot(phi,h)
        Psol=Psol+alpha*d
        Vsol=Vsol-alpha*h
        qkp1=rhs_h-sparse.csr_matrix.dot(GT,Vsol)
        beta=numpy.dot(qkp1,qkp1)/numpy.dot(q,q)
        dkp1=-qkp1+beta*d

        V_diff.append(numpy.max(abs(Vsol-Vsolmem))/numpy.max(abs(Vsol)))
        P_diff.append(numpy.max(abs(Psol-Psolmem))/numpy.max(abs(Psol)))
        Psolmem=Psol
        Vsolmem=Vsol
        d=dkp1
        q=qkp1

        if max(V_diff[iter],P_diff[iter]) < tol or iter == niter-1 :
            return (Vsol,Psol,V_diff,P_diff)
```

Listing 6: Python code of the Uzawa algorithm known as Uzawa Algorithm with Conjugate Directions.

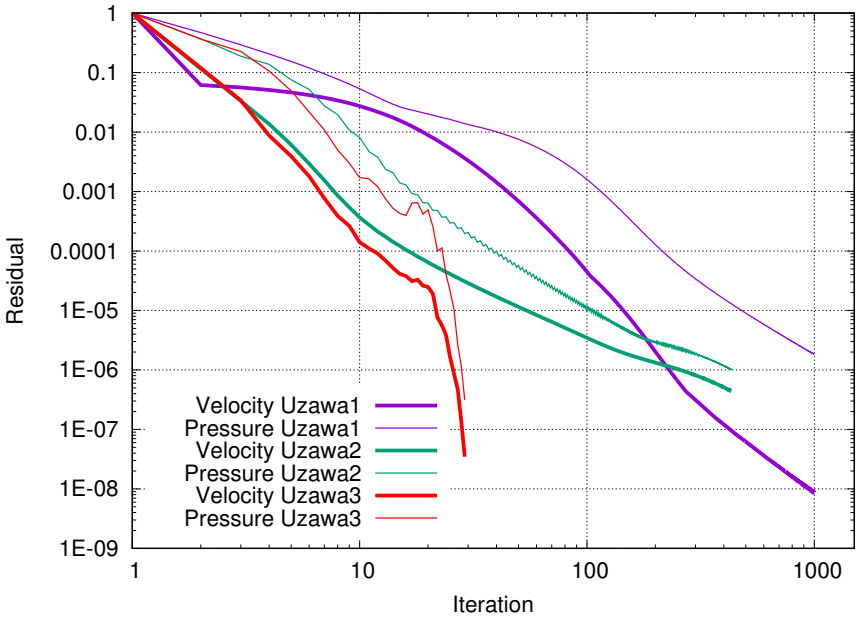

Figure 24: Residuals of velocity and pressure for the three different Uzawa methods (Listings 4,5 and 6).

Variable	Value
Matrix viscosity	$1\ Pa \cdot s$
Viscosity inclusion	$10^3\ Pa \cdot s$
Radius inclusion	0.25
Density inclusion	$10^3\ kg/m^3$
Midpoint inclusion (x)	0.5
Midpoint inclusion (y)	0.5

Table 2: Variables used in the circular inclusion test.

purposes, the maximum number of outer iterations is set to 1000. More than 1000 does not seem reasonable in this case. The first Uzawa method does not converge (Figure 24) in less than the maximum number of iterations. Velocity and pressure residuals decrease by approximately the same pattern but are more than two orders of magnitude different at the final iteration.

The second Uzawa method is able to converge in less than 1000 iterations. Absolute pressure and velocity residuals comparatively close and the difference between them decreases as the iterations continue. The last Uzawa method is capable of performing convergence in 52 iterations and experiences a velocity residual decrease of 4 orders of magnitude in the first 10 iterations. At iteration 30, a small plateau is reached with a subsequent rapid decrease of 2 orders of magnitude in 15 iterations. For the coming tests, the third Uzawa method will be used as it leads in performance.

To be able to explain CG convergence rates in the following part, condition number tests first need to be carried out. Two tests are set up accordingly. The first consists of an isoviscous and isodense material inside of a 2D box. Because of the body forces, a velocity field is generated. The second test consists of a 2D box in which the user can specify the location, size, density and viscosity of any number of circular inclusions. To promote experimentation, the very top of the simPyFEM code is dedicated to setting up the inclusions in a clear and simple way (Listing 7).

As discussed previously, higher condition numbers of matrices used in iterative solving techniques result in a slower convergence rate and thus longer solve times. A Python function was written to calculate the condition number (Listing 8). Computing eigenvalues of large matrices is computationally costly. For this reason, relatively small resolutions (from 16×16 to 64) were used. A single dense and viscous inclusion was placed in the center of the domain. Results are shown in Figure 25. Condition numbers for the penalty method are substantially higher than the Schur complement method matrices. Applying a preconditioner to an isoviscous system has no effect on the condition number (and thus the amount of iterations needed to reach convergence). Viscosity contrasts are expressed in the matrix by variations in the diagonal. Thus, diagonally preconditioning a matrix which has a constant diagonal (i.e. is isoviscous) only scales the eigenvalues by a constant factor. The scaling of eigenvalues has no effect on the condition number as the ratio between s_{max} and s_{min} remains constant.

The penalty method is tested first. Resolution is increased in powers of 2 from 32×32 to 128×128 elements in a unit square domain. Both tests (isoviscous & isodense and circular inclusion) were carried out in order to investigate the effect of the condition number on iteration number and the convergence rate and solve times. The variables used in the circular inclusion experiment are shown in Table 2. Velocity fields of both tests are visualized in Figure 26.

High condition numbers are observed in Figure 25 when using the penalty method. This is directly reflected in the number of iterations required for convergence (Figure 27). As discussed earlier applying a diagonal preconditioner on an isoviscous system has no effect on the convergence. As expected, tests results show that CG and PCG convergence are equal and only PCG is plotted for non-redundancy. High convergence rates appear to be not only related to pure condition number. When comparing the condition numbers of preconditioned A for an isoviscous case and circular inclusion case, the isoviscous condition number is higher by some orders

```
def variable_setup():
    #Resolution: you can also specify this at run-time
    if (len(sys.argv)==3):nnx=int(sys.argv[1]);nny=int(sys.argv[2])
    else: nnx = 32 ; nny = 32
    #Spatial dimensions of the system:
    Lx = 2.0 ; Ly = 1.0
    #Gravity:
    gx = 0.0 ; gy = -1.0
    #Plotting solution(s):
    plot_solution = True
    #Sinkers:
    use_sinkers = True
    return (nnx,nny,Lx,Ly,gx,gy,plot_solution,use_sinkers)

def sinker_setup():
    rho_matr = 1.0                #density of the matrix material
    vis_matr = 1.0                #viscosity of the matrix material
    n_sinker = 2                  #number of inclusions

    #Setup_sinker_arrays----------------------------#
    vis_sinker    = numpy.zeros(n_sinker,dtype=float)#
    rho_sinker    = numpy.zeros(n_sinker,dtype=float)#
    rad_sinker    = numpy.zeros(n_sinker,dtype=float)#
    mid_sinker_x  = numpy.zeros(n_sinker,dtype=float)#
    mid_sinker_y  = numpy.zeros(n_sinker,dtype=float)#
    #-----------------------------------------------#

    #Sinker 1
    vis_sinker[0] = 1.0e3         #viscosity of the inclusion
    rad_sinker[0] = 0.25          #radius of the inclusion
    rho_sinker[0] = 1.0e3         #density of the inclusion
    mid_sinker_x[0] = 0.5         #midpoint of the inclusion, x
    mid_sinker_y[0] = 0.5         #midpoint of the sinker, y

    #Sinker 2
    vis_sinker[1] = 1.0e3         #viscosity of the inclusion
    rad_sinker[1] = 0.15          #radius of the inclusion
    rho_sinker[1] = 2.0e3         #density of the inclusion
    mid_sinker_x[1] = 1.5         #midpoint of the inclusion, x
    mid_sinker_y[1] = 0.5         #midpoint of the inclusion, y

    return (rho_matr,vis_matr,n_sinker,vis_sinker,rad_sinker,rho_sinker,
   mid_sinker_x,mid_sinker_y)
```

Listing 7: First lines of simPyFEM code, consisting of two functions which set up parameters such as resolution and dimensions of the system as well as sets up the geometries and parameters of the inclusions.

```
def compute_con_number(A):
    eig_values  = numpy.linalg.eigvalsh(A)    #computation of eigenvalues
    eig_val_min = abs(numpy.min(eig_values))  #minimum of eigenvalues
    eig_val_max = abs(numpy.max(eig_values))  #maximum of eigenvalues
    con_number  = eig_val_max / eig_val_min   #condition number
```

Listing 8: Python code to calculate the condition number of input matrix A.

of magnitude. This would imply faster convergence rates for the circular inclusion case. Comparing Figure 27 and Figure 29 reveals the opposite. PCG on an isoviscous box is able to converge in less than 2000 iterations, whereas PCG applied to the circular inclusion case is not converging.

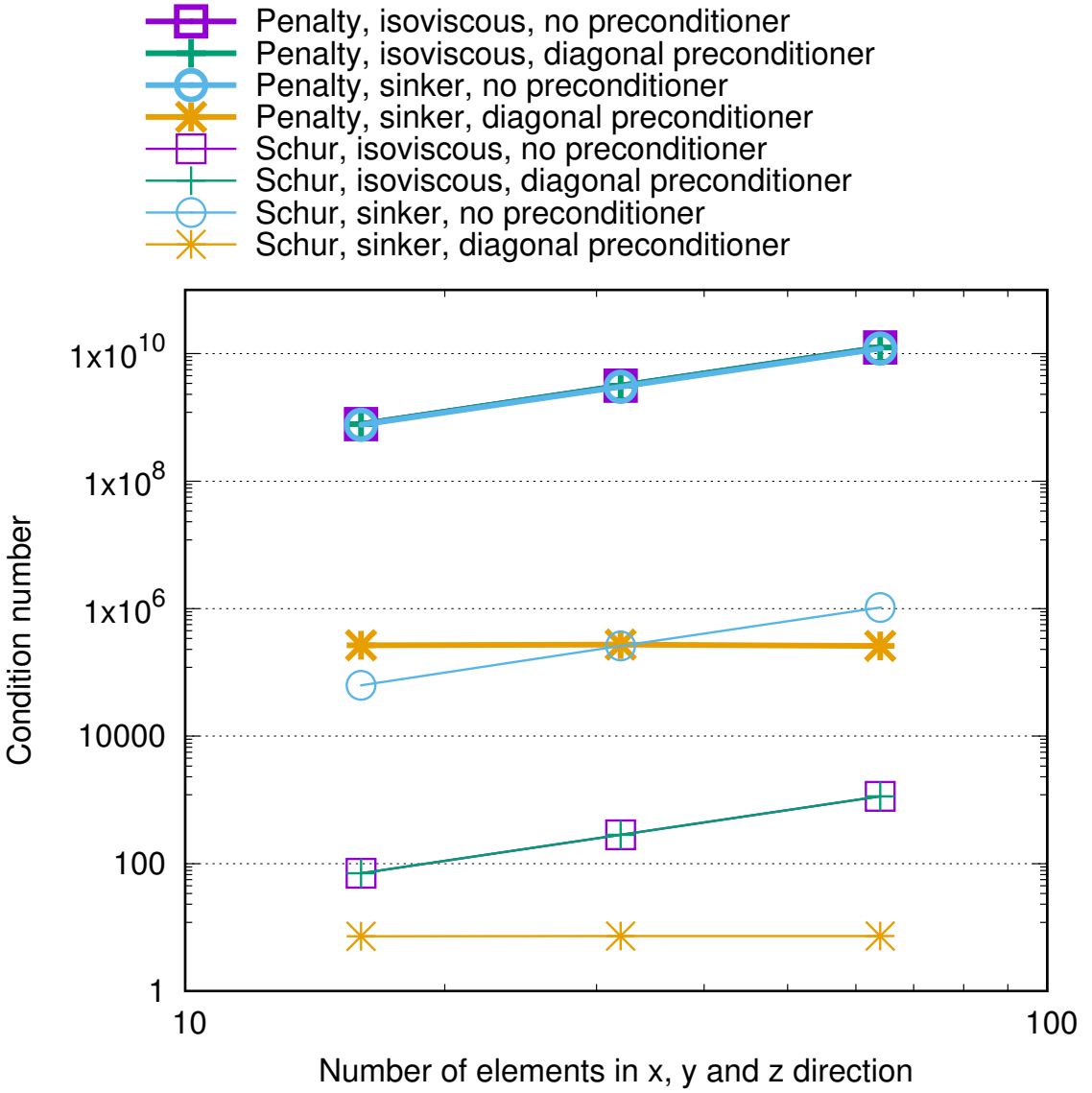

Figure 25: Condition number scaling across small resolutions. For unpreconditioned CG solves, the condition number increases with resolution. Applying the Jacobi preconditioner results in a constant condition number.

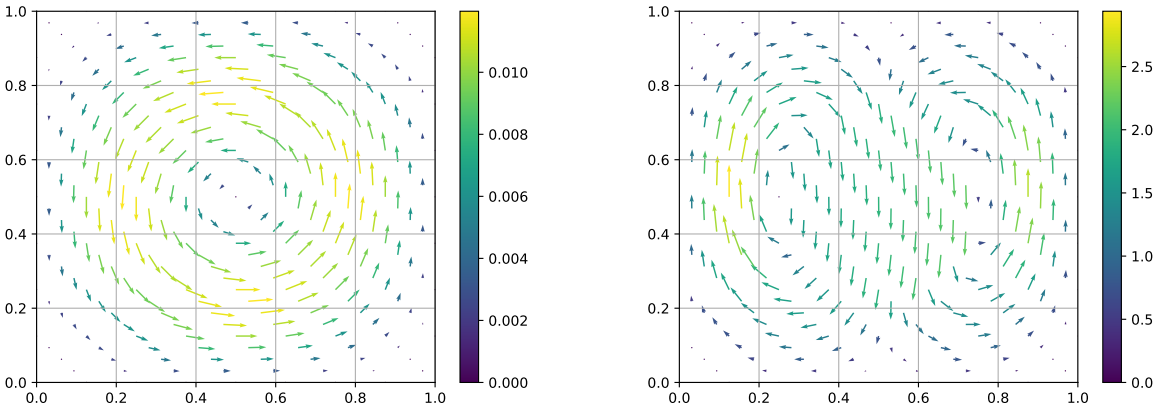

Figure 26: Left: Velocity field in case of an isoviscous and isodense system. Right: Velocity field with a dense and viscous circular inclusion placed in the center of the domain.

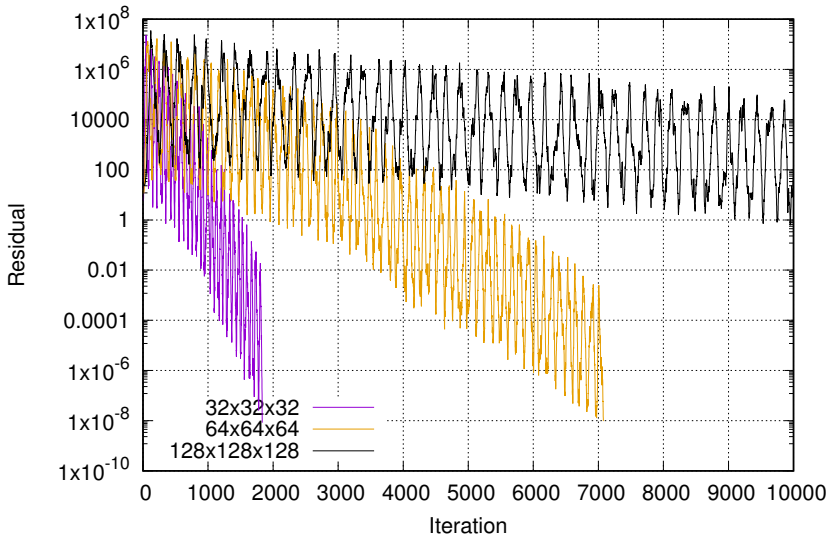

Figure 27: Convergence of Preconditioned Conjugate Gradient for an isoviscous box using the penalty method.

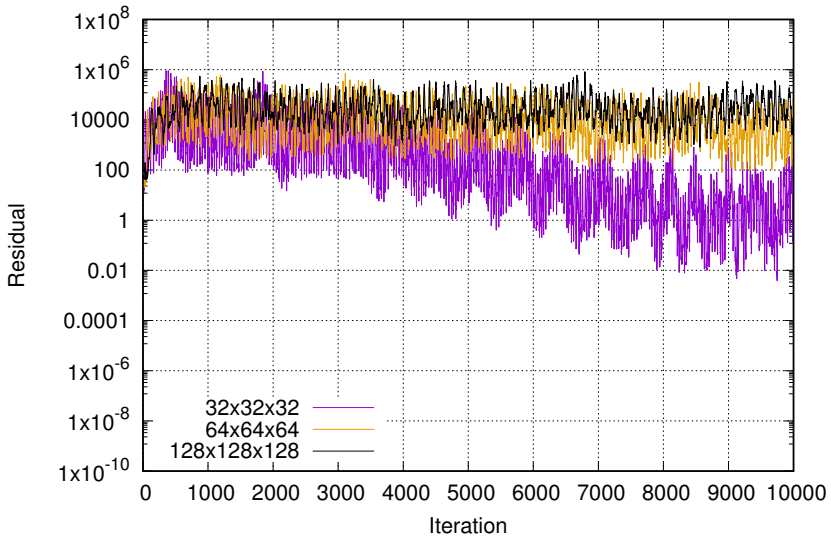

Figure 28: Convergence of Conjugate Gradient for three resolutions for the circular inclusion test using the penalty method.

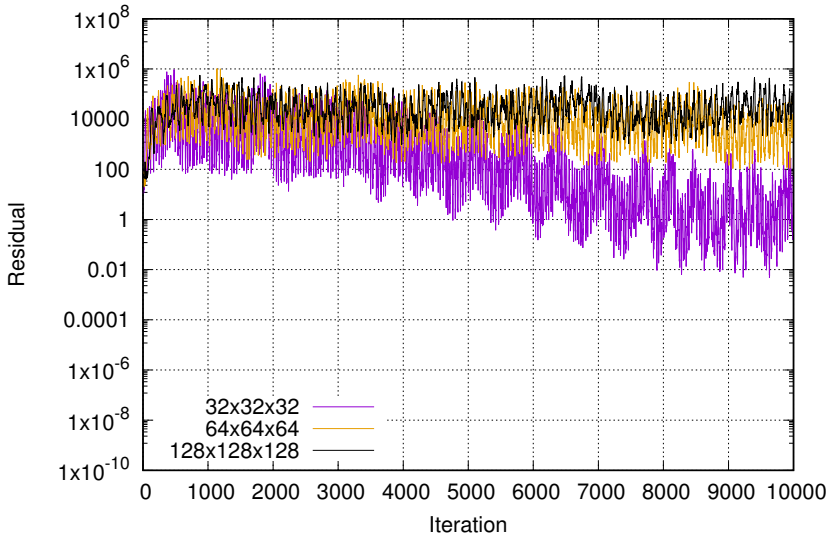

Figure 29: Convergence of Preconditioned Conjugate Gradient for the circular inclusion test using the penalty method.

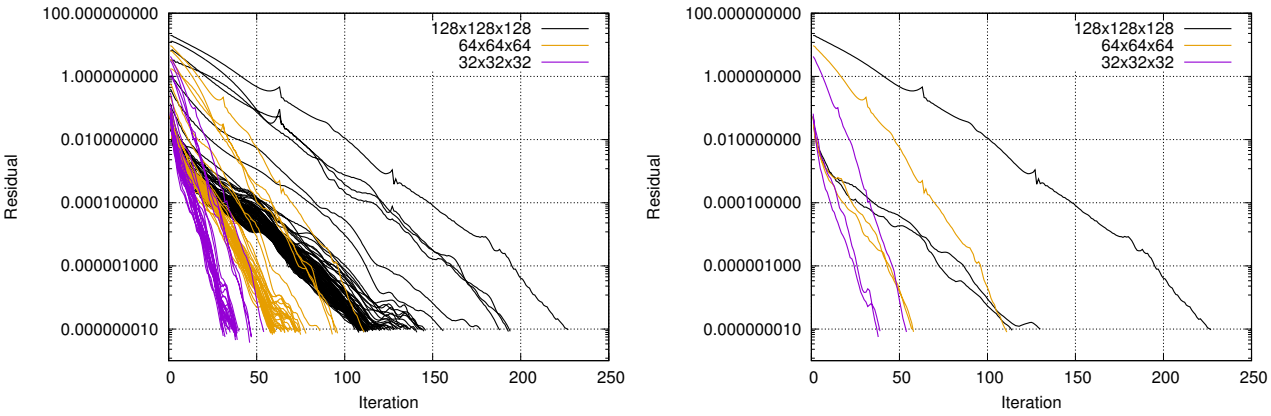

Figure 30: Convergence of Preconditioned Conjugate Gradient for an isoviscous box using the Schur complement method. Left: all data. Right: selected data for readability.

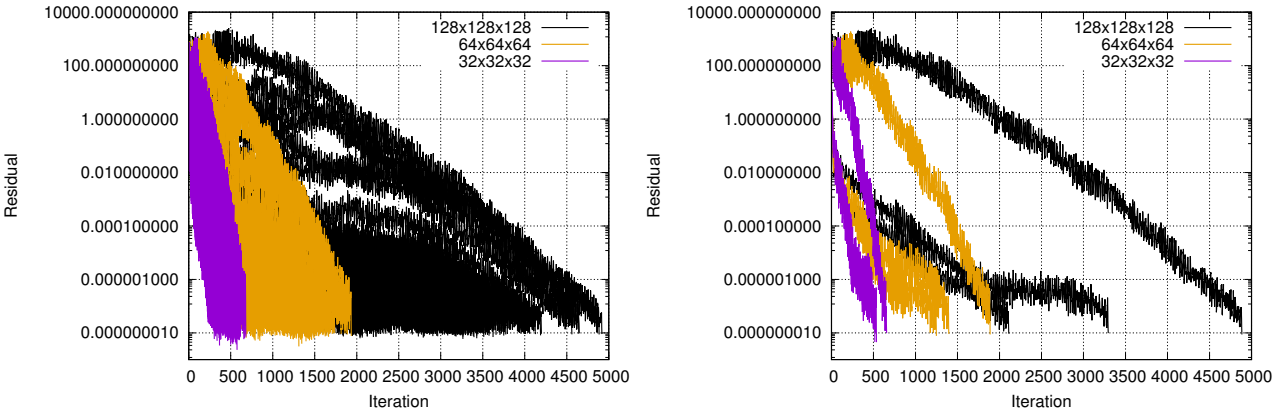

Figure 31: Convergence of Conjugate Gradient for three resolutions for the circular inclusion test using the Schur complement method. Left: all data. Right: selected data for readability.

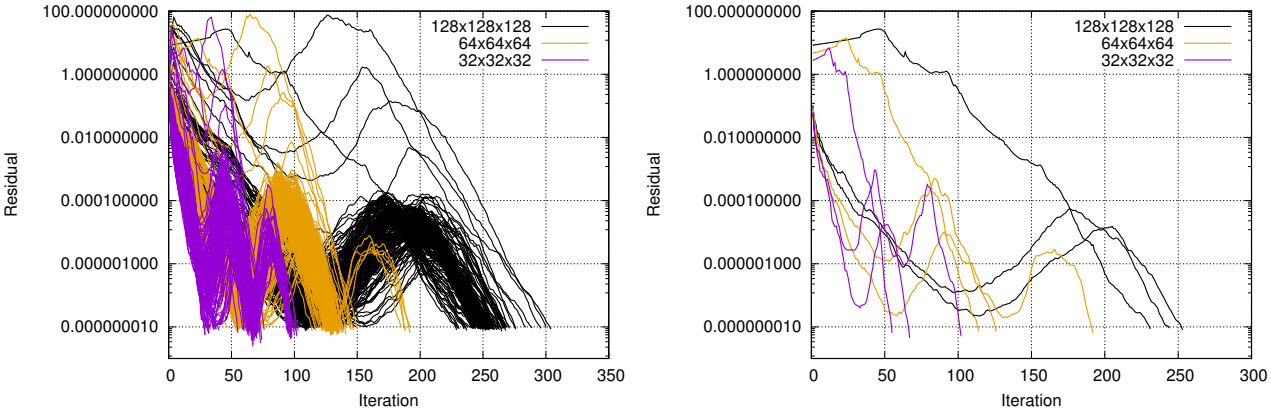

Figure 32: Convergence of Preconditioned Conjugate Gradient for the circular inclusion test using the Schur complement method. Left: all data. Right: selected data for readability.

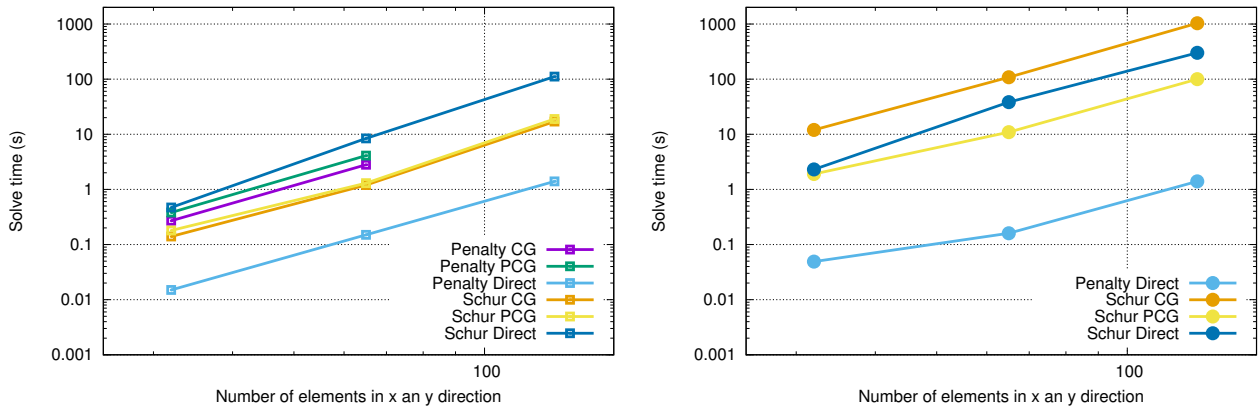

Figure 33: Computation time of the solve. Left: isoviscous box where the Schur complement method with a direct inner solver performs the worst. Right: circular inclusion test where the Schur complement method with a CG inner solve performs the worst.

Similarly, the lower condition number of preconditioned A compared to non-preconditioned A in the case of the circular inclusion test suggests faster convergence rates of PCG. However, convergence rate patterns visualized in Figure 28 and 29 are nearly identical. In reality, when looking at the raw data, they are not equal. Nevertheless, the premise that convergence rates are very low in a heteroviscous system using no, or a non-ideal, preconditioner holds.

Low(er) condition numbers for A in a system such as Eq. 37 can result in higher convergence rates. This is directly observed in Figures 30, 31 and 32. In contrast to the penalty method, where CG and PCG were not able to converge in the case of the circular inclusion test, the Schur complement method is able to converge for the three different cases. Once again, CG and Jacobi preconditioned CG are equal in an isoviscous system, and only PCG is visualized (Figure 30) for non-redundancy. The residual decrease plots are more dense compared to the penalty method as every inner (CG) solve is plotted. For higher resolutions, this can result in hundreds of lines. For improved observation and interpretation of results, supplementary plots are provided which only show three lines per resolution; one for the first, middle and last outer iteration.

The isoviscous system converges relatively quickly, in terms of iteration count, and the residual decrease pattern is a straight line (when using a log-scale on the y-axis) (Figure 30). Non-preconditioned CG for the circular inclusion experiences an up-and-down convergence pattern but reaches the tolerance in a relatively straight line. PCG for the circular inclusion shows iteration counts similar to the isoviscous system. Iteration counts are approximately 10 times lower compared to CG for the same problem. The convergence pattern, however, is not a straight line. Rather, the residual shows a periodical increase and decrease a couple of times.

As discussed in the section about CG convergence, the number of iterations required for convergence is predictable. A resolution increase of a factor of 2 would result in twice as many iterations (for well-conditioned problems). We can observe this behaviour when using the Schur complement method (Figures 30, 31 and 32).

The number of iterations is an indicator for the convergence rate. However, when the individual iterations are costly, one might seek a method with a slightly worse convergence rate but faster computation time. To determine which solving method is the fastest, the two tests (isoviscous and circular inclusion boxes) are run for resolution of 32×32 to 128×128 elements. As BLAS and LAPACK (libraries used by SciPy and NumPy) are inherently parallel on MacBooks, the tests are run on the Linux workstation. The results are shown in Figure 33.

For both tests, the penalty method with a direct solver performs the best. The non-convergence of penalty CG/PCG for the isoviscous case at resolution of higher than 64×64 and non-convergence of the circular inclusion case for CG and PCG observed in Figure 27, 28, 29 is the reason for exclusion of these methods in Figure 33. Measuring computation time for non-convergent solutions is meaningless.

It has already been stated that the condition numbers of CG and PCG are equal for an isoviscous case. Consequently, they require the same amount of iterations to reach convergence. However, because the PCG method requires more matrix-vector and vector-vector multiplications than CG, PCG performs slightly worse in terms of solve time. The direct solve in combination with the penalty method results in the lowest computation times for both the isoviscous test and circular-inclusion-test. In case of the circular inclusions, CG and PCG are only able to converge when used as an inner solve in one of the Uzawa methods. Excluding the penalty method, PCG performs the best followed by the direct solve and CG.

Another point of attention is that in the above mentioned tests, all matrix multiplications were carried out using the sparse CSR format of the matrices. On account of code simplicity, the matrices are built up in their dense form. Conversion from dense to sparse is carried out by SciPy:

```
A_sparse=scipy.sparse.csr_matrix(A)
```

Sparse matrix multiplications can the be carried out through:

```
C=sparse.csr_matrix.dot(A,B)
```

where A is a sparse matrix and B can either be a sparse matrix or a vector. The time-save derived from the numerous, now sparse, matrix multiplications is sufficiently large compared to the computation time of conversions of A and P from dense to sparse to include it in the algorithms.

5.3 Discussion and conclusions

Not every solver is applicable to every problem that it is exposed to. The penalty method can perform well with the help of a direct solver. Using iterative solvers with a penalty method is not a good choice. The high condition numbers of penalty method matrices (i.e. ill-conditioned matrices) are not suitable and result in very low convergence rates, up to the point where it is unusable even for lower resolutions. The condition number of the matrices can be lowered in the case of the circular inclusion test by applying the diagonal preconditioner. Although, because of the inherently poor convergence rate of CG and Jacobi PCG with penalty methods, the high performance lags behind.

The Schur complement method opens the door to experimenting with preconditioners and solvers. Low condition numbers result in higher convergence rates without over-complicating the code. Here, the effect of applying a diagonal preconditioner results in a significant speed-up of the convergence rate (one order of magnitude for the circular inclusion test). Surprisingly, for the circular inclusion test, using a sparse direct solver as inner solve is vastly slower than PCG. The speed of the direct solver could be improved by, for example, specifying that the matrix is SPD. Because `scipy.sparse.linalg.spsolve` does not support this format, no further optimization of the direct solve is possible at this point in time, leaving PCG as the best inner solve for the Schur complement method.

Further improvements to solving the system could be made by also integrating other iterative solvers such as GMRES, Flexible GMRES and BiCGSTAB. Additional preconditioners can also be setup by students in the coming years as I have made `compute_preconditioner()` function in which the user formulates the preconditioning matrix. Examples of preconditioners which might perform well are ILU, ICC and SSOR.

The tools that Python provides prove to be useful in the assessment of code performance. Integrated functions for calculating eigenvalues, for example, allow us to compute condition numbers of matrices in a simple way. Conversion of dense to CSR storage (or many other sparse formats) is also a powerful and convenient tool to have access to in many cases when working with large numerical calculations.

By translating the code from Fortran to Python, the first step to finite element modelling as well as a basic understanding of solvers and preconditioners becomes more accessible to students. For them it might suffice to only use the penalty method with a direct solver, which could be useful when carrying out simple simulations.

Others might find a use in the Schur complement method using iterative solvers and are able to write their own solvers and preconditioner functions in Python and add them to the project. This opens up the path for simPyFEM to become a powerful, yet easy to understand educational tool for the understanding of geodynamic behaviour coded in a modern programming language which is easy to adapt.

References

Amestoy, P. R., Duff, I. S., L'Excellent, J.-Y., and Koster, J. (2001). *MUMPS: A General Purpose Distributed Memory Sparse Solver*, pages 121–130. Springer Berlin Heidelberg, Berlin, Heidelberg.

Anaconda (2018). Anaconda, stable release 5.0.2. https://www.anaconda.com/.

Balay, S., Abhyankar, S., Adams, M. F., Brown, J., Brune, P., Buschelman, K., Dalcin, L., Eijkhout, V., Gropp, W. D., Kaushik, D., Knepley, M. G., May, D. A., McInnes, L. C., Rupp, K., Sanan, P., Smith, B. F., Zampini, S., Zhang, H., and Zhang, H. (2017a). PETSc users manual. Technical Report ANL-95/11 - Revision 3.8, Argonne National Laboratory.

Balay, S., Abhyankar, S., Adams, M. F., Brown, J., Brune, P., Buschelman, K., Dalcin, L., Eijkhout, V., Gropp, W. D., Kaushik, D., Knepley, M. G., May, D. A., McInnes, L. C., Rupp, K., Smith, B. F., Zampini, S., Zhang, H., and Zhang, H. (2017b). PETSc Web page. `http://www.mcs.anl.gov/petsc`.

Balay, S., Gropp, W. D., McInnes, L. C., and Smith, B. F. (1997). Efficient management of parallelism in object oriented numerical software libraries. In Arge, E., Bruaset, A. M., and Langtangen, H. P., editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press.

Bangerth, W. (2013). `http://www.math.colostate.edu/~bangerth/videos.html`.

Bangerth, W., Burstedde, C., Heister, T., and Kronbichler, M. (2012). Algorithms and data structures for massively parallel generic adaptive finite element codes. *ACM Trans. Math. Softw.*, 38(2):14:1–14:28.

Bijwaard, H., Spakman, W., and Engdahl, E. R. (1998). Closing the gap between regional and global travel time tomography. *Journal of Geophysical Research: Solid Earth*, 103(B12):30055–30078.

Blankenbach, B., Busse, F., Christensen, U., Cserepes, L., Gunkel, D., Hansen, U., Harder, H., Jarvis, G., Koch, M., Marquart, G., Moore, D., Olson, P., Schmeling, H., and Schnaubelt, T. (1989). A benchmark comparison for mantle convection codes. *Geophysical Journal International*, 98(1):23–38.

Borzi, A. and Schulz, V. (2009). Multigrid methods for PDE optimization. *SIAM Review*, 51(2):361–395.

Braess, D. (2007). *Finite Elements: Theory, Fast Solvers, and Applications in Solid Mechanics*. Cambridge University Press, 3 edition.

Bryan, K. and Shibberu, Y. (2005). Penalty functions and constrained optimization. Lecture notes from Rose-Hulman Institute of Technology.

Calo, V. M., Collier, N. O., Pardo, D., and Paszyński, M. R. (2011). Computational complexity and memory usage for multi-frontal direct solvers used in p finite element analysis. *Procedia Computer Science*, 4:1854 – 1861. Proceedings of the International Conference on Computational Science, ICCS 2011.

COMSOL (2014). `https://www.comsol.com/blogs/much-memory-needed-solve-large-comsol-models/`.

Courant, R., Friedrichs, K., and Lewy, H. (1928). Über die partiellen Differenzengleichungen der mathematischen Physik. *Mathematische Annalen*, 100:32–74.

Davies, D. R., Wilson, C. R., and Kramer, S. C. (2011). Fluidity: A fully unstructured anisotropic adaptive mesh computational modeling framework for geodynamics. *Geochemistry, Geophysics, Geosystems*, 12(6):n/a–n/a.

Davis, T. A. (2004). Algorithm 832: Umfpack v4.3—an unsymmetric-pattern multifrontal method. *ACM Trans. Math. Softw.*, 30(2):196–199.

de Groot, L., Pimentel, A., and Di Chiara, A. (2016). The multimethod palaeointensity approach applied to volcanics from terceira: Full-vector geomagnetic data for the past 50 kyr. *Geophysical Journal International*, 206(1):590–604.

Dongarra, J. and Sullivan, F. (2000). Guest editors' introduction: The top 10 algorithms. *Computing in Science and Engg.*, 2(1):22–23.

Duretz, T., May, D. A., Gerya, T. V., and Tackley, P. J. (2011). Discretization errors and free surface stabilization in the finite difference and marker-in-cell method for applied geodynamics: A numerical study. *Geochemistry, Geophysics, Geosystems*, 12(7):n/a–n/a. Q07004.

Dziewonski, A. and Haddon, R. (1974). The radius of the core-mantle boundary inferred from travel time and free oscillation data; a critical review. *Physics of the Earth and Planetary Interiors*, 9(1):28 – 35.

Falgout, R. D. (2006). An introduction to algebraic multigrid. *Computing in Science and Engg.*, 8(6):24–33.

Fedorenko, R. (1964). The speed of convergence of one iterative process. *USSR Computational Mathematics and Mathematical Physics*, 4(3):227 – 235.

Floricich, M., Large, S., Jones, D., Helmi, A., and Lörtzer, G. (2011). Conditioning reservoir model and aiding well planning using probabilistic seismic inversion in a central north sea field. In *Conditioning reservoir model and aiding well planning using probabilistic seismic inversion in a Central North Sea Field*, volume 2, pages 1059–1063.

Fullsack, P. (1995). An arbitrary lagrangian-eulerian formulation for creeping flows and its application in tectonic models. *Geophysical Journal International*, 120(1):1–23.

Gerya, T. (2010). *Introduction to Numerical Geodynamic Modelling*. Cambridge University Press.

Gerya, T. V. and Yuen, D. A. (2003). Characteristics-based marker-in-cell method with conservative finite-differences schemes for modeling geological flows with strongly variable transport properties. *Physics of the Earth and Planetary Interiors*, 140(4):293 – 318.

GNU (2014). `https://people.sc.fsu.edu/~jburkardt/py_src/fem2d_bvp_linear/fem2d_bvp_linear.html`.

Guerri, M., Cammarano, F., and Tackley, P. (2016). Modelling earth's surface topography: Decomposition of the static and dynamic components. *Physics of the Earth and Planetary Interiors*, 261:172 – 186.

Harlow, F. H. and Welch, J. E. (1965). Numerical calculation of time-dependent viscous incompressible flow of fluid with free surface. *The Physics of Fluids*, 8(12):2182–2189.

Heinrich, H. (1965). S. g. mikhlin, variational methods in mathematical physics. (international series of monographs in pure and applied mathematics. *ZAMM - Journal of Applied Mathematics and Mechanics*, 45(4):270–270.

Hestenes, M. R. and Stiefel, E. (1952). Methods of Conjugate Gradients for Solving Linear Systems. *Journal of Research of the National Bureau of Standards*, 49(6):409–436.

Ho, N., Olson, S. D., and Walker, H. F. (2015). Accelerating the Uzawa Algorithm. *ArXiv e-prints*.

Horn, R. A. and Zhang, F. (2005). Basic properties of the schur complement. In Zhang, F., editor, *The Schur complement and its applications*, chapter 2, pages 17–46. Springer, New York.

Ismail-Zadeh, A. and Tackley, P. (2010). *Computational Methods for Geodynamics*. Cambridge University Press.

J N Reddy, D. (2005). *An Introduction to the Finite Element Method*. McGraw-Hill Education.

Jacobi, C. G. J. (1845). Uber eine neue auflosungsart der bei der methode der kleinsten quadrate vorkommenden linearen gleichungen. *Astronomische Nachrichten*, 22:297–306.

Karato, S.-i. (2008). *Deformation of Earth Materials: An Introduction to the Rheology of Solid Earth*. Cambridge University Press.

Kronbichler, M., Heister, T., and Bangerth, W. (2012). High accuracy mantle convection simulation through modern numerical methods. *Geophysical Journal International*, 191(1):12–29.

Leng, W. and Zhong, S. (2008). Viscous heating, adiabatic heating and energetic consistency in compressible mantle convection. *Geophysical Journal International*, 173(2):693–702.

Li, J. and Widlund, O. B. (2007). On the use of inexact subdomain solvers for bddc algorithms. *Computer Methods in Applied Mechanics and Engineering*, 196(8):1415 – 1428. Domain Decomposition Methods: recent advances and new challenges in engineering.

Li, X. S. (2005). An overview of superlu: Algorithms, implementation, and user interface. *ACM Trans. Math. Softw.*, 31(3):302–325.

Liu, W. and Vinter, B. (2015). Speculative segmented sum for sparse matrix-vector multiplication on heterogeneous processors. *Parallel Computing*, 49:179 – 193.

Lowrie, W. (2007). *Fundamentals of Geophysics*. Cambridge University Press.

Maffione, M., Thieulot, C., van Hinsbergen, D. J. J., Morris, A., Plümper, O., and Spakman, W. (2015). Dynamics of intraoceanic subduction initiation: 1. oceanic detachment fault inversion and the formation of supra-subduction zone ophiolites. *Geochemistry, Geophysics, Geosystems*, 16(6):1753–1770.

Mancktelow, N. S. (2002). Finite-element modelling of shear zone development in viscoelastic materials and its implications for localisation of partial melting. *Journal of Structural Geology*, 24(6):1045 – 1053. Micro structural Processes: A Special Issue in Honor of the Career Contributions of R.H. Vernon.

May, D., Brown, J., and Pourhiet, L. L. (2015). A scalable, matrix-free multigrid preconditioner for finite element discretizations of heterogeneous stokes flow. *Computer Methods in Applied Mechanics and Engineering*, 290:496 – 523.

May, D., Sanan, P., Rupp, K., Knepley, M., and Smith, B. (2016). Extreme-scale multigrid components within petsc. In *Extreme-scale multigrid components within PETSc*.

May, D., Schellart, W., and Moresi, L. (2013). Overview of adaptive finite element analysis in computational geodynamics. *Journal of Geodynamics*, 70(Supplement C):1 – 20.

May, D. A., Brown, J., and Pourhiet, L. L. (2014). ptatin3d: High-performance methods for long-term lithospheric dynamics. In *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 274–284.

Meurant, G. (1999). *Computer Solution of Large Linear Systems*. Studies in Mathematics and Its. North-Holland.

Mikhlin, S. (1971). *The numerical performance of variational methods*. Wolters-Noordhoff Series of Monographs and Textbooks on Pure and Applied Mathematics. Wolters-Noordhoff Publishing.

Murakami, M., Hirose, K., Kawamura, K., Sata, N., and Ohishi, Y. (2004). Post-perovskite phase transition in mgsio3. *Science*, 304(5672):855–858.

Pinchover, Y. and Rubinstein, J. (2005). *An Introduction to Partial Differential Equations*. Number v. 10 in An introduction to partial differential equations. Cambridge University Press.

Quinquis, M. E., Buiter, S. J., and Ellis, S. (2011). The role of boundary conditions in numerical models of subduction zone dynamics. *Tectonophysics*, 497(1):57 – 70.

Reddy, J. (1986). *Applied functional analysis and variational methods in engineering*. McGraw-Hill Ryerson, Limited.

Reddy, J. (2002). *Energy Principles and Variational Methods in Applied Mechanics*. Wiley.

Reddy, J. (2004). *An Introduction to Nonlinear Finite Element Analysis*. OUP Oxford.

Reddy, J. and Gartling, D. (2010). *The Finite Element Method in Heat Transfer and Fluid Dynamics, Third Edition*. Computational Mechanics and Applied Analysis. Taylor & Francis.

Saad, Y. (2003). *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2nd edition.

Saad, Y. and Schultz, M. H. (1986). Gmres: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 7(3):856–869.

Schellart, W. P. and Moresi, L. (2013). A new driving mechanism for backarc extension and backarc shortening through slab sinking induced toroidal and poloidal mantle flow: Results from dynamic subduction models with an overriding plate. *Journal of Geophysical Research: Solid Earth*, 118(6):3221–3248.

Schenk, O. and Gartner, K. (2006). On fast factorization pivoting methods for sparse symmetric indefinite systems. *ETNA. Electronic Transactions on Numerical Analysis [electronic only]*, 23:158–179.

Schmid, D. W. (2002). *Finite and infinite heterogeneities under pure and simple shear*. PhD thesis, ETH Zurich.

Schmid, D. W. and Podladchikov, Y. Y. (2003). Analytical solutions for deformable elliptical inclusions in general shear. *Geophysical Journal International*, 155(1):269–288.

Schubert, G., Turcotte, D., and Olson, P. (2001). *Mantle Convection in the Earth and Planets*. Mantle Convection in the Earth and Planets. Cambridge University Press.

Seidel, L. (1874). Uber ein verfahren die gleichungen, auf welche die methode der kleinsten quadrate fuhrt, sowie lineare gleichungen uberhaupt, durch successive annaherung aufzulosen. *Bayer. Akad. Wiss. Math. Phys. Kl. Abh.*, 11:81–108.

Tackley, P. J. (2008). Modelling compressible mantle convection with large viscosity contrasts in a three-dimensional spherical shell using the yin-yang grid. *Physics of the Earth and Planetary Interiors*, 171(1):7 – 18. Recent Advances in Computational Geodynamics: Theory, Numerics and Applications.

Thieulot, C. (2011). FANTOM: Two- and three-dimensional numerical modelling of creeping flows for the solution of geological problems. *Physics of the Earth and Planetary Interiors*, 188:47–68.

Thieulot, C. (2014). Elefant: a user-friendly multipurpose geodynamics code. *Solid Earth Discussions*, 6:1949–2096.

Thieulot, C. (2017). Analytical solution for viscous incompressible stokes flow in a spherical shell. *Solid Earth*, 8(6):1181–1191.

Thieulot, C. (2018). Ghost: Geoscientific hollow sphere tesselation. *Solid Earth Discussions*, 2018:1–14.

Thornton, C. (2000). Numerical simulations of deviatoric shear deformation of granular media. *Géotechnique*, 50(1):43–53.

Tosi, N., Stein, C., Noack, L., Hüttig, C., Maierová, P., Samuel, H., Davies, D. R., Wilson, C. R., Kramer, S. C., Thieulot, C., Glerum, A., Fraters, M., Spakman, W., Rozel, A., and Tackley, P. J. (2015). A community benchmark for viscoplastic thermal convection in a 2-d square box. *Geochemistry, Geophysics, Geosystems*, 16(7):2175–2196.

Uzawa, H. and Arrow, K. J. (1989). *Iterative methods for concave programming*, pages 135–148. Cambridge University Press.

van der Giessen, E. and Aref, H. (2003). *Advances in Applied Mechanics*. Number v. 39 in Advances in Applied Mechanics. Academic Press.

van der Vorst, H. A. (1992). Bi-cgstab: A fast and smoothly converging variant of bi-cg for the solution of nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 13(2):631–644.

van der Walt, S., Colbert, S. C., and Varoquaux, G. (2011). The numpy array: A structure for efficient numerical computation. *Computing in Science Engineering*, 13(2):22–30.

Van Hinsbergen, D., Lippert, P., Dupont-Nivet, G., McQuarrie, N., Doubrovine, P., Spakman, W., and Torsvik, T. (2012). Greater india basin hypothesis and a two-stage cenozoic collision between india and asia. *Proceedings of the National Academy of Sciences of the United States of America*, 109(20):7659–7664.

Wesseling, P. (2004). *An Introduction to Multigrid Methods*. An Introduction to Multigrid Methods. R.T. Edwards.

Young, D. (1954). Iterative methods for solving partial difference equations of elliptic type. *Transactions of the American Mathematical Society*, 76(1):92–111.

Zhong, S., Yuen, D., and Moresi, L. (2007). *Numerical Methods for Mantle Convection*, volume 7. Elsevier.

Zhong, S., Zuber, M. T., Moresi, L., and Gurnis, M. (2000). Role of temperature-dependent viscosity and surface plates in spherical shell models of mantle convection. *Journal of Geophysical Research: Solid Earth (1978–2012)*, 105(B5):11063–11082.