Department of Mechanical and Aeronautical Engineering

University of Pretoria

AN ALGEBRAIC MULTIGRID SOLUTION STRATEGY FOR EFFICIENT SOLUTION OF FREE-SURFACE FLOWS

Wilhelm J. van den Bergh

Supervised by Dr. A.G. Malan and Dr. D.N. Wilke

Thesis submitted to the University of Pretoria in candidature for the degree of *M.Eng*

May 2011

Abstract

Free-surface modelling (FSM) is a highly relevant and computationally intensive area of study in modern computational fluid dynamics. The *Elemental* software suite currently under development offers FSM capability, and employs a preconditioned GMRES solver in an attempt to effect fast solution times. In terms of potential solver performance however, multigrid methods can be considered state-of-the-art. This work details the investigation into the use of Algebraic Multigrid (AMG) as a high performance solver tool for use as black box plug-in for *Elemental* FSM. Special attention was given to the development of novel and robust methods of addressing AMG setup costs in addition to transcribing the solver to efficient C++ object-oriented code. This led to the development of the so-called Freeze extension of the basic algebraic multigrid method in an object-oriented C++ programming environment. The newly developed Freeze method reduces setup costs by periodically performing the setup procedure in an automatic and robust manner. The developed technology was evaluated in terms of robustness, stability and speed by applying it to benchmark FSM problems on structured and unstructured meshes of various sizes. This evaluation yielded a number of conclusive findings. First, the developed Freeze method reduced setup times by an order of magnitude. Second, the developed AMG solver offered substantial performance increases over the preconditioned GMRES method. In this way, it is proposed that this work has furthered the state-of-the-art of algebraic multigrid methods applied in the context of free-surface modelling.

Acknowledgements

First and foremost, I would like to express my sincere gratitude to my supervisor, Prof. A.G. Malan, for getting me started on numerics in the first place. This work would not have been possible without your unwavering faith and encouragement. It was a privilege working under your guidance. Thanks also to my co-supervisor Dr. N. Wilke for timely advice and guidance.

To my parents (both sets)... I have no words. The sacrifices you have made for me I can never repay. Faith, love, support, time...all these and more were never in short supply. Mamma, Pappa, Tannie Marié, Oom Manie, ek's verskriklik lief vir julle almal! All my friends, from University and elsewhere, who listened politely as I babbled about data structures and complexity issues, thanks for putting up with me and the sinusoidal moods! For all the coffee and interesting internet tidbits, thanks to the Magnificent Seven. It's not procrastinating, it's team building, right Andrew?

Finally, and always, thank you to my heavenly Father who put me on this path and let me meet all these amazing people. I know You will lead me in future too.

Contents

Nomenclature

Roman Symbols

$\mathbf{A}$	Coefficient matrix
A_{ii}	Diagonal entry of the coefficient matrix
A_{ij}	Off-diagonal entry of the coefficient matrix
AMG	Algebraic multigrid
AMG_e	Element Algebraic multigrid
$BLAS$	Basic Linear Algebra Subprograms
$\mathbf{b}$	Linear equation right-hand side
$\mathbf{C}$	Set of coarse nodes
$\mathbf{C}_{mn}$	Edge coefficient for edge mn
$C-AMG$	Classical Algebraic multigrid
CFD	Computational fluid dynamics
$CICSAM$	Compressive Interface Capturing Scheme for Arbitrary Meshes
$\mathbf{e}$	Error vector
f	Body force
$\mathbf{F}$	Diffusive flux vector
FSM	Free-surface modelling
$\mathbf{G}$	Viscous flux vector
GMG	Geometric multigrid
$GMRES$	Generalised minimum residual
h	Grid spacing
$\mathbf{H}$	Pressure derivative vector
$\mathbf{I}_{2h}^{h}$	Interpolation operator
$\mathbf{I}_{h}^{2h}$	Restriction operator
K	Linear function for pressure correction
l	Edge length (m)
n	Time step/Iteration
$\mathbf{n}$	Unit normal vector
N	Number of unknowns (rows) in a system of linear equations
N_g	Graph depth associated with spatial discretisation
nnz	Number of non-zeros
$O(\bullet)$	Order $\bullet$
p	Pressure (Pa)
$\mathbf{p}$	Pressure vector

$\mathbf{r}$	Residual vector
$\mathcal{S}$	Bounding surface segment
$\mathbf{S}$	Body force vector
S^C	Strongly influencing coarse set
S^F	Strongly influencing fine set
t	Time (s)
$\mathbf{t}$	Unit vector tangent to an edge
u	Velocity (ms^{-1})
$\mathbf{U}$	Temporal flux vector
$\mathcal{V}$	Element volume (m^3)
VOF	Volume-of-fluid
W	Weakly influencing set
x	Spatial direction

Greek Symbols

α	Volume fraction of first fluid
β	Volume fraction of second fluid
ρ	Density
μ	Viscosity
σ	Deviatoric stress
ϕ	Scalar field value
Υ_{mn}	Edge connecting nodes m and n
Ω	Denotes a grid level in multigrid
θ	Algebraic strong influence threshold

Mathematical operators

$\Delta\bullet$	Increment in $\bullet$
δ_{ij}	Kronecker delta: unity if $i = j$, zero if $i \neq j$
$\partial\bullet$	Partial derivative of $\bullet$
$\partial_j^{fv}\bullet$	Finite volume approximation of the spatial partial derivative of $\bullet$ w.r.t. x_j

Superscripts

h	Denotes a quantity on a grid with spacing h
n	Denotes a quantity at time step/iteration n

Subscripts

i	Node index / Cartesian direction
j	Node index / Cartesian direction
k	Node index
m	Node index / Mean value
n	Node index
mn	Denotes an edge mn

On notation

Both vector and index notation are used in this thesis, which is usually clear from context. Vectors and matrices are denoted by lower or upper case letters in bold, while specific components are denoted by plain text with a subscript index. Where index notation features, component subscripts may appear as super or subscripts and are typically denoted by i, j, k and l. Einstein's summation convention is implied in the case of index notation.

List of Figures

Chapter 1

Introduction

1.1 Background

In the field of computational fluid dynamics (CFD), one of the most challenging and relevant areas of study is that of free-surface modelling (FSM). FSM is concerned with the modelling of the dynamic interaction of immiscible fluids [12]. Examples include fuel sloshing in aircraft wings and road transport where fluid dynamics has a considerable impact on vehicle dynamics [1] as well as structural design and internal baffling [6]. In spacecraft, liquid sloshing affects manoeuvres in orbit [44]. Wind-water interactions in the marine environment influence the design of harbours, structures, and ships [36]. Modelling cavitation phenomena accurately is essential in improving marine propeller and hull service lifetimes [48]. Slug flow in pipes (where large bullet-shaped pockets of gas form over the largest part of the pipe cross-section) impact flow characteristics in the oil, natural gas, steam boiler, vaporiser and cooling industries [28]. Molten metal and plastic processes, such as mould filling, stirring, and continuous casting, can all benefit from accurate free-surface flow predictions. The above mentioned flows are typically geometrically complex multi-physics problems, which require fast and accurate FSM tools to simulate. One such tool is the *Elemental* software suite.

Developed as a high resolution, high performance parallel CFD solver, *Elemental* utilises a computationally efficient edge-based vertex-centred finite volume scheme [22] coupled with an artificial compressibility fractional step method [25] to solve the incompressible Navier-Stokes equations. The free-surface interface is described via a volume-of-fluid (VOF) method which is based on the Compressive Interface Capturing Scheme for Arbitrary Meshes (CICSAM) method [42]. Recent successes in simulating fuel sloshing in an aircraft tank [26] has highlighted the efficacy of the code as well as the main current computational bottlenecks, *viz.* the computational effort expended during the solution of the system of equations arising from the so-called pressure correction step. The latter constitutes a large, sparse, asymmetric system of linear equations which is to be solved at every simulation time step. Currently, this is accomplished via a preconditioned GMRES method [24, 21]. As such, the focus of this work was to implement and refine an alternative fast solver for the solution of the pressure equation. Such a solver should

- require optimal ($O(N)$) memory storage
- involve low operation counts and scale as $O(N)$ with problem size

- operate as a robust and automatic ’black box’ plug-in to the existing *Elemental* code.

The need for low memory cost is obvious, but more important is the operation count and scalability of the solver, since memory is not generally the limiting factor in computer hardware. The desirability of a black box tool follows from the requirement that the solver be independent of FSM problem specific factors, and that no changes to the existing code are needed to implement it. The pressure correction step involves a system of *linear* equations which makes the use of a black box type solver possible. The development and implementation of such a solver is the focus of the current work.

From the literature, a number of advanced sparse solvers have been applied to FSM for the purpose of solving the pressure correction equation. These include successive over-relaxation [16], preconditioned Generalised Minimal Residual (GMRES) [18, 27], various conjugate-gradient type methods [15, 37, 35, 2] and geometric multigrid methods [46]. Other attempts at solution acceleration include improved initial condition prediction [20], and algebraic multigrid methods [14]. Each of the aforementioned approaches have inherent strengths and weaknesses. The successive over-relaxation and conjugate-gradient exhibit non-linear growth in solution times as a function of problem size [8]. GMRES is proven to be a very robust solver, but is memory intensive [17], while requiring preconditioning and also not scaling optimally with problem size. Geometric multigrid has the potential for exceptional all-round performance, but can be problematic when grids are highly unstructured or geometries complex, often requiring problem or geometry specific information [8]. Further, the method is more difficult to implement than other approaches, particularly if a black box type solver is sought. Algebraic multigrid (AMG) methods address many of the aforementioned shortcomings while holding the potential of offering optimal memory and computational costs. AMG operates solely on an algebraic system’s coefficient matrices, thus requiring no geometric or problem information. The generation of coarse grids is still fairly complex however, but due to the relative problem specific independence of the method, it makes for a good black box solver, if somewhat less efficient than a tailored geometric method [47]. One other major difficulty faced by algebraic multigrid methods in the context of *Elemental* FSM is that the pressure correction coefficient matrix changes at every simulation time step. This is due to the rapidly evolving fluid interface. Since the solution structure is built on this matrix, continuous re-coarsening is required. This added computational cost is thought to be the prime reason as to why algebraic multigrid is under represented in the context of FSM, even though it is acknowledged as highly applicable to the pressure correction equation [38, 40].

1.2 Thesis Overview

In light of the above, AMG was the solver type selected solver for use with *Elemental* FSM for the purposes of this work. As noted, this was due to AMG’s suitability as a black box type plug-in, its performance (as part of the multigrid family of solution methods), and its proven robustness and speed as a sparse linear system solver. Special attention was given to addressing the performance penalty due to changing coefficient matrices by developing a technique which significantly reduces the CPU time spent on AMG setup. In order to expedite the implementation of the solver, the classical algebraic multigrid method (C-AMG) [34] was chosen due to its relative simplicity, robustness and speed. Further, no additional information is needed by C-AMG beyond the system of linear equations to be solved. Note that more modern AMG algorithms, such as Smoothed Aggregation, require more complex interpolation schemes, or were specifically tailored for finite element methods, such as AMGe [9].

Among the innovations introduced during this study were

- developing a multi-level strategy to reduce the cost of repeated AMG setup, referred to as Freeze-AMG
- transcribing the developed AMG solver to computer code via an object-oriented C++ environment.

The decision to use the C++ language for the solver, as opposed to C or Fortran which are more common in numerical programming, was motivated by several factors. An object-oriented approach offers superior maintainability and reusability of code, and lends itself well to modular programming structures, which may be optimized for computational speed via balanced static and dynamic polymorphism [23]. Further, *Elemental* already utilises C++, and so using this environment would ensure maximum compatibility with existing code. Finally, existing AMG solvers native to C++, such as used by OpenFOAM [13] are unsuitable for investigation with *Elemental*. Other implementations exist in Fortran [38], C [10], Matlab [30] and Python [3].

The developed solver was assessed in terms of speed and robustness by application to benchmark FSM problems. In the interest of rigorous evaluation, structured as well as unstructured meshes of various sizes were employed and CPU times compared to preconditioned GMRES, in addition to evaluating the speed-up offered by Freeze-AMG. These tests yielded promising results, showing that the Freeze-AMG method holds great potential as a competitive sparse solver for use in FSM simulations.

1.3 Thesis Layout

The remainder of this document is structured as follows:

- **Chapter 2** describes the *Elemental* FSM functionality. The employed governing equations, discretisation, and solution methods which give rise to the linear system of equations to be solved are described.
- **Chapter 3** deals with multigrid theory, detailing the components needed for an AMG solver. The Freeze-AMG concept is also introduced and illustrated
- **Chapter 4** details the implementation of the AMG solver in C++. Specific attention is given to the evolution of the solver algorithms from initial to final versions, with commentary on computational performance achieved.
- **Chapter 5** focuses on the rigorous evaluation of the developed solver. Benchmark problems are solved on various meshes and performance compared to a competing solver.
- **Chapter 6** concludes and summarises the document, and tables recommendations for future work.

1.4 Publication List

Publications resulting from this work are as follows:

1.4.1 Conference Papers

- VAN DEN BERGH, W.J., MALAN, A.G., WILKE, D.N. (2011). An Algebraic Multigrid solution strategy for efficient solution of free-surface flows. Second African Conference on Computational Mechanics, AFRICOMP 2011, Cape Town, South Africa.

1.4.2 Journal Papers

- VAN DEN BERGH, W.J., MALAN, A.G., WILKE, D.N. (2011). An Algebraic Multigrid solution strategy for efficient solution of free-surface flows. SUBMITTED FOR REVIEW, International Journal of Numerical Methods in Engineering, 2011.

1.5 Summary

The purpose of this work is to develop a fast and efficient black box AMG solver to be used as plug-in to the *Elemental* FSM code. This is accomplished by development of the so-called Freeze-AMG method, realised in an object-oriented C++ environment.

Chapter 2

Problem Background

2.1 Introduction

The aim of this work is to accelerate the solution of a system of linear equations arising in the course of a free surface modelling simulation. Specifically, the focus is on the interaction of two immiscible, viscous and incompressible fluids modelled with the *Elemental* CFD package. This chapter provides background of the volume-of-fluid (VOF) governing equation set employed by *Elemental*, and details the construction of the set of linear equations, resulting from the pressure correction step, which is to be solved.

Note

For the purposes of this chapter, index notation is used throughout in the interests of brevity.

2.2 Governing Equations

Since the FSM considered in this work involves two fluids, a natural starting point when constructing the governing equation set is to consider the conservation of mass. Assuming that a unique velocity exists in each fluid at every point in space and time, the mass conservation equations may be written in Eulerian form as:

$$\frac{\partial}{\partial t}(\alpha\rho_1) + \frac{\partial}{\partial x_i}(\alpha\rho_1 u_i) = 0 \tag{2.1a}$$

$$\frac{\partial}{\partial t}((1-\alpha)\rho_2) + \frac{\partial}{\partial x_i}((1-\alpha)\rho_2 u_i) = 0, \tag{2.1b}$$

where t and x_i denote time and Cartesian direction i, ρ_1 is the density of Fluid 1, and ρ_2 is the density of Fluid 2. The quantity u_i denotes the velocity field and α constitutes the volume fraction of each fluid as:

$$\alpha = \begin{cases} 1 \text{ for Fluid 1} \\ 0 \text{ for Fluid 2} \end{cases} \tag{2.2}$$

Coupled with the assumption of incompressibility of the two fluids, Equations (2.1a) and (2.1b) may be divided by the respective densities and added to obtain:

$$\frac{\partial u_i}{\partial x_i} = 0, \tag{2.3}$$

which is the well known incompressible flow mass conservation equation. If instead Equations (2.1a) and (2.1b) are *subtracted* after dividing by density, the interface tracking equation used by the VOF method is obtained:

$$\frac{\partial \alpha}{\partial t} + \frac{\partial}{\partial x_i}(\alpha u_i) = 0. \tag{2.4}$$

Similarly to the mass conservation equations, the momentum conservation for each fluid can be considered. These are written as:

$$\frac{\partial}{\partial t}(\alpha\rho_1 u_j) + \frac{\partial}{\partial x_i}(\alpha\rho_1 u_i u_j) + \alpha\frac{\partial p}{\partial x_j} = \frac{\partial}{\partial x_i}\left[\alpha\mu_1\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right)\right] + \alpha\rho_1 f_j \tag{2.5a}$$

$$\frac{\partial}{\partial t}(\beta\rho_2 u_j) + \frac{\partial}{\partial x_i}(\beta\rho_2 u_i u_j) + \beta\frac{\partial p}{\partial x_j} = \frac{\partial}{\partial x_i}\left[\beta\mu_2\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right)\right] + \beta\rho_2 f_j, \tag{2.5b}$$

where $\beta = (1-\alpha)$, p is the pressure, μ_1 and μ_2 are the viscosities of the two fluids, and f_j the body force in a direction j. Adding Equations (2.5a) and (2.5b) yields the mean momentum conservation equation:

$$\frac{\partial}{\partial t}(\rho_m u_j) + \frac{\partial}{\partial x_i}(\rho_m u_i u_j) + \frac{\partial p}{\partial x_j} = \frac{\partial}{\partial x_i}\left[\mu_m\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right)\right] + \rho_m f_j, \tag{2.6}$$

where $\rho_m = \alpha\rho_1 + \beta\rho_2$ is the mean density, and $\mu_m = \alpha\mu_1 + \beta\mu_2$ the mean viscosity.

Combining the above conservation equations and manipulating, the governing equation set may be written as:

$$\frac{\partial \mathbf{U}^j}{\partial t} + \frac{\partial \mathbf{F}^j}{\partial x_j} + \frac{\partial \mathbf{H}^j}{\partial x_j} - \frac{\partial \mathbf{G}^j}{\partial x_j} = \mathbf{S}^j, \tag{2.7}$$

where $\mathbf{S}^j$ is a vector of body forces and

$$\mathbf{U}^j = \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ 0 \\ \alpha \end{pmatrix}, \quad \mathbf{F}^j = \begin{pmatrix} u_1 u_j \\ u_2 u_j \\ u_3 u_j \\ u_j \\ \alpha u_j \end{pmatrix}, \quad \mathbf{H}^j = \begin{pmatrix} p\frac{\delta_{ij}}{\rho_m} \\ p\frac{\delta_{ij}}{\rho_m} \\ p\frac{\delta_{ij}}{\rho_m} \\ 0 \\ 0 \end{pmatrix}, \quad \mathbf{G}^j = \begin{pmatrix} \sigma_{1j} \\ \sigma_{2j} \\ \sigma_{3j} \\ 0 \\ 0 \end{pmatrix}, \tag{2.8a}$$

where δ_{ij} is the Kronecker delta and

$$\sigma_{ij} = \frac{\mu_m}{\rho_m}\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right), \tag{2.8b}$$

with nomenclature as defined previously.

2.3 Solution Procedure

In order to solve the governing equation set accurately and in a fully coupled manner, *Elemental* employs a fractional solution approach. These methods operate on the mass and momentum conservation equations and essentially consist of three steps [31, 25]. Here the velocity u_i and the pressure p at the current time-step n are employed to obtain values at the next time-step $n+1$. In this work, we consider the semi-implicit method which commences by calculating an intermediate velocity variable as:

$$\frac{\Delta U_i^*}{\Delta t} = -\left.\frac{\partial F^{ij}}{\partial x_j}\right|^n + \left.\frac{\partial G^{ij}}{\partial x_j}\right|^n + S_i\Big|^n, \tag{2.9}$$

for $i = 1, 2, 3$, where Δt denotes the physical time-step size, $\Delta t = t^{n+1} - t^n$. Next, the pressure correction equation is constructed as

$$0 = \Delta t \frac{\partial}{\partial x_i}\left(\frac{\Delta U_i^*}{\Delta t} - \frac{1}{\rho_m^n}\frac{\partial p^{n+1}}{\partial x_i}\right) + \left.\frac{\partial u_i}{\partial x_i}\right|^n, \tag{2.10}$$

which may be rewritten as:

$$\frac{\partial}{\partial x_i}\left(\frac{1}{\rho_m^n}\frac{\partial p^{n+1}}{\partial x_i}\right) = \frac{\partial \Delta U_i^*}{\partial x_i} + \left.\frac{1}{\Delta t}\frac{\partial u_i}{\partial x_i}\right|^n, \tag{2.11}$$

for $i = 1, 2, 3$. The pressure correction may now be solved implicitly from this equation, as described in Section 2.4. In the third and final step, the calculated pressure is used to correct the velocity field and calculate the velocity at the next time-step in a semi-implicit manner as:

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} = \frac{\Delta U_i^*}{\Delta t} + \left.\frac{\partial H^{ij}}{\partial x_j}\right|^{n+1}, \tag{2.12}$$

for $i = 1, 2, 3$. In the FSM case the interface position is finally updated using:

$$\frac{\alpha^{n+1} - \alpha^n}{\Delta t} = \left.\frac{\partial \alpha u_i}{\partial x_i}\right|^n, \tag{2.13}$$

where the nomenclature is as defined previously.

2.4 Spatial Discretisation and Equation Construction

Before the pressure correction equation system can be constructed from Equation (2.11), we first describe the discretisation method employed. *Elemental* employs a vertex-centred edge-based finite volume algorithm for the purposes of spatial discretisation, where a compact stencil is employed for second-derivative terms in the interest of both stability and accuracy [22]. This method was selected as it offers natural generic mesh applicability, second-order accuracy and computational efficiency which is greater than element based approaches.

For the purposes of spatial discretisation, a so-called dual mesh is constructed by constructing volumes around each node as in Figure 2.1. The governing equation set is then cast into weak form via integration over an arbitrary control volume $\mathcal{V}$ followed by application of the divergence theorem. The resulting surface integrals are calculated in an edge-wise manner. For this purpose, bounding surface information is similarly stored in an edge-wise manner and termed *edge-coefficients*. The latter for a given internal edge Υ_{mn} connecting nodes m and n, is defined as

$$\mathbf{C}_{mn} = \mathbf{n}^{mn_1}\mathcal{S}_{mn_1} + \mathbf{n}^{mn_2}\mathcal{S}_{mn_2} \tag{2.14}$$

where $\mathcal{S}_{mn_1}$ is a bounding surface-segment intersecting the edge (as in Figure 2.1) and $\mathbf{n}$ is the outward pointing normal unit vector.

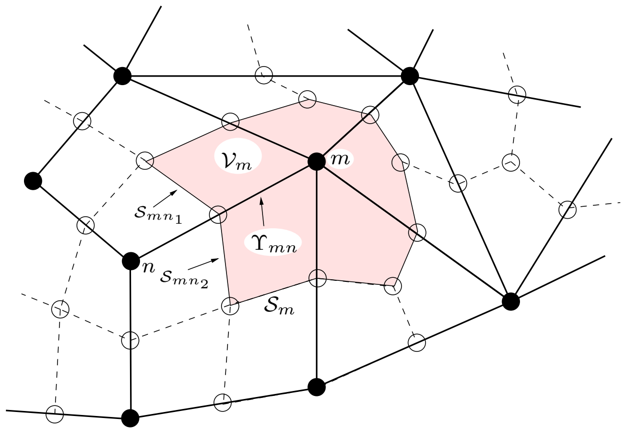


Figure 2.1: Schematic diagram of the construction of the median dual-mesh on hybrid grids. Here, Υ_{mn} depicts the edge connecting nodes m and n.

In order discretise Equation (2.11), it is necessary to apply the above method to obtain discrete forms of first and second derivatives. Here, the first derivative terms are approximated as:

$$\left.\frac{\partial \phi}{\partial x_j}\right|_m \approx \frac{1}{\mathcal{V}_m} \sum_{\Upsilon_{mn}\cap\mathcal{V}_m} \overline{\phi_{mn}} C^j_{mn} = \partial^j_{FV}\phi_m, \tag{2.15}$$

where $\overline{\phi_{mn}}$ denotes the value of a scalar field evaluated at the bounding surface of $\mathcal{V}_m$ and $\partial^j_{FV}\phi_m$ is the finite volume approximation of the first derivative with respect to x_j. Second derivatives are approximated as:

$$\left.\frac{\partial^2 \phi}{\partial x_j^2}\right|_m \approx \frac{1}{\mathcal{V}_m} \sum_{\Upsilon_{mn}\cap\mathcal{V}_m} \left[\frac{\phi_m - \phi_n}{l} t^j + \left.\partial^j_{FV}\phi_m\right|_{norm}\right] C^j_{mn}, \tag{2.16}$$

where t^j now refers to the unit vector tangential to the edge and l is the edge length. We now turn our attention to casting Equation (2.11) into suitable form to be discretised using the above approximations. First we define

$$K(p^{n+1}) = \frac{\partial}{\partial x_i}\left(\frac{1}{\rho^n_m}\frac{\partial p^{n+1}}{\partial x_i}\right) = \left.\frac{\partial \Delta U_i^*}{\partial x_i} + \frac{1}{\Delta t}\frac{\partial u_i}{\partial x_i}\right|^n, \tag{2.17}$$

and read it to mean that some K exists which is linear in p^{n+1}. From this, the following holds:

$$K(p^{n+1}) = K(p^n) + \left.\frac{\partial K}{\partial p}\right|^n (p^{n+1} - p^n). \tag{2.18}$$

Thus, we can state the following:

$$\begin{aligned} K(p^n) + \left.\frac{\partial K}{\partial p}\right|^n \Delta p &= \frac{\partial \Delta U_i^*}{\partial x_i} + \left.\frac{1}{\Delta t}\frac{\partial u_i}{\partial x_i}\right|^n \\ \Rightarrow \left.\frac{\partial K}{\partial p}\right|^n \Delta p &= \frac{\partial \Delta U_i^*}{\partial x_i} + \left.\frac{1}{\Delta t}\frac{\partial u_i}{\partial x_i}\right|^n - K(p^n), \end{aligned} \tag{2.19a}$$

where $\Delta p = (p^{n+1} - p^n)$ is the pressure correction sought. Applying the approximations described in this section, it can be shown that we may now arrive at a system of linear algebraic equations to be solved:

$$\mathbf{A}\Delta\mathbf{p} = \mathbf{b}. \tag{2.20}$$

The matrix **A** has several important characteristics. Firstly, it is extremely sparse owing to the discretisation method's small stencil, which uses only nearest and next-nearest neighbouring nodes. Since every node is only neighboured by a small number of other nodes, each row of **A** necessarily contains only a small number of coefficients. Secondly, **A** is asymmetric, due to the fact that its coefficients are a function of the mean density ρ_m, which varies from node to node, and the unstructured nature of the discretisation mesh. Lastly, **A** changes as a function of time. This follows from the dependence on ρ_m, which will vary at nodes local to the interface as the fluid interface evolves. It is this dynamic nature of **A** that results in repeated costly coarse mesh generation when applying an AMG method to Equation (2.20). This concern is addressed in the next chapter.

2.5 Summary

This chapter described the governing equation set employed by *Elemental* to model free-surface flows, as well as outlining the fractional step solution approach and employed dicretisation method. The spatial discretisation method used was provided *viz.* a vertex-centred edge-based finite volume approach. The pressure correction linear equation system was constructed and shown to be sparse, asymmetric and dynamic.

Chapter 3

Solution Strategy - Algebraic Multigrid

3.1 Introduction

Chapter 2 provided details on the approach *Elemental* uses in solving the incompressible Navier-Stokes equations in the context of free-surface modelling. As shown, the method employed results in a sparse, asymmetric system of linear algebraic equations which is solved to obtain a pressure correction. For the purposes of this chapter, let this problem be represented (as in Chapter 2) by

$$\mathbf{A}\Delta\mathbf{p} = \mathbf{b},$$

where **A** is the coefficient matrix of size $(N \times N)$, $\Delta\mathbf{p}$ is the pressure correction being solved for, and **b** is the right hand side of the equation. This chapter will first provide a general overview of solution methods applied to problems such as Equation (2.20), and justification for choosing Algebraic Multigrid for this work. Details of multigrid methods and the components required for their implementation are provided, before the chapter is ended with a description of the newly developed Freeze-AMG methodology.

3.2 Overview of Solution Strategies

Maturing CFD technology, in conjunction with the rapid advances made in computing power in recent decades, is increasingly being applied to industrially relevant problems. A major challenge, however, remains providing simulation results in a timely enough fashion to be useful. CFD analyses on a relevant scale often involve the solution of sparse algebraic systems containing hundreds of thousands, if not millions, of unknowns. In a competitive industrial environment, a typical design could require hundreds such analyses, which must be performed in a short amount of time. Furthermore, the size and number of analyses demanded is increasing at a greater rate than that at which computing power is. In other words, despite the dramatic improvements made to hardware, and the evolution of parallel computing in the last decade, efficient solution of large, sparse systems of equations remains critical.

The earliest solvers applied to systems of asymmetric linear equations were known as *direct* methods, and included matrix inversion via Gaussian elimination. These methods solve a system of N equations with N unknowns in $O(N^3)$ operations and require memory storage of $O(N^2)$ [45]. Obviously, as problem sizes scale up, computational and memory costs become prohibitive. Simple matrix-free *iterative* solvers, such as Jacobi and Gauss-Seidel, address the memory concerns with a storage cost of $O(N)$. Unfortunately, the number of iterations required to converge to a solution grows rapidly prohibitive as the problem size increases [45].

The development of *Krylov space* methods led to the introduction of improved matrix-free solution strategies such as the generalised minimal residual method (GMRES). Indeed, GMRES has become a widely used method in CFD for solving sparse linear systems of equations [43]. Indeed, a preconditioned GMRES method [21, 27] is currently used by *Elemental* for solution of the pressure correction equation. The minimum complexity for the GMRES algorithm is $O(N_g N)$, where N_g is the graph depth associated with spatial discretisation [19]. An additional concern with the GMRES method is the requirement for preconditioning, and the associated memory costs [21]. In contrast, *multigrid* methods, considered one of the biggest breakthroughs in the area of solving large systems of discretised partial differential equations [39], may require as little as $O(N)$ operations. These solution strategies are therefore called *optimal* methods [9]. As with GMRES, memory costs due to the storage of multiple grid levels' information is a concern, but this was considered a secondary factor to solver speed in this work.

Multigrid techniques can be subdivided into two categories, namely *geometric* and *algebraic*, typically referred to as GMG and AMG respectively. Both methods operate on both high and low frequency errors in a solution by constructing a hierarchy of grids representing the original problem. Obviously, in GMG, this hierarchy consists of a subset of coarser physical grids based on the original discretisation mesh. Obtaining these coarser grids poses a challenge when geometries are complex or grids are anisotropic [8]. Therefore, GMG methods are difficult to apply to arbitrary problems and are thus less attractive as black box type solvers. AMG methods, on the other hand, construct a series of 'grids' based entirely on the coefficient matrix **A** of the equations being solved, making it more favourable as a plug-in solver for the work under consideration.

In light of the above, AMG was chosen as the solution method to be investigated for use with *Elemental* in the context of the pressure correction equation. Since memory usage was not a primary concern, the costs associated with the method were disregarded. The specific AMG method chosen was Classical-AMG (C-AMG), as first detailed in [34]. C-AMG was originally developed for symmetric, positive-definite matrices with positive diagonal and negative off-diagonal entries, such as those arising from the discretisation of elliptic partial differential equations [4]. It has since proven surprisingly effective at solving a wide variety of problems, some of which are quite far from these assumptions [39]. Also, importantly, it requires the bare minimum of information from *Elemental*, namely the pressure correction equation coefficient matrix and residual vector, which are easily available. An overview of multigrid methods, focusing on AMG and the special considerations implemented specifically to address concerns in FSM applications follows.

3.3 Multigrid Methods

As mentioned previously, classic matrix-free iterative solution schemes (also called relaxation schemes), such as Gauss-Seidel and Jacobi, although offering optimal storage characteristics and being widely applicable, suffer from some severe limitations. These arise from the fact that, subsequent to initial fast convergence, residual reduction stalls, requiring a vast number of iterations for complete convergence. Multigrid methods arose from an attempt to remedy this problem. It was found that when an error of high frequency, such as a sinusoidal wave with high wave number, was imposed on a system with a known solution, the error was quickly reduced by Jacobi-like iterative relaxation methods. Conversely, when an error of low frequency was imposed, the iterations required to reduce this error was increased drastically. This gave rise to the concept of regarding an (unknown) error as a Fourier series, and thus the idea of high and low frequency error components.

Taking the Fourier series perspective then, traditional matrix-free iterative methods were only effective at reducing high frequency errors, rapidly 'smoothing out' these components until only low frequency components remain. Multigrid methods were developed to leverage this fact by representing low frequency error components on succesively coarser grids, making them appear to be of higher frequency. A representation of the problem on these coarser grids (using the residual equation) could then be solved to reduce error components directly across a spectrum of frequencies, enabling the improvement of the solution estimate after every iteration. In order to achieve this, all multigrid methods rely on the following basic elements:

- Coarse grid generation - Section 3.3.1
- Intergrid transfer operators, namely interpolation and restriction - Section 3.3.2
- Coarse grid problem representation - Section 3.3.3
- A solution cycle using some iterative method (also called a smoother) - Section 3.3.4.

The above is illustrated conceptually in Figure 3.1. How each component is constructed is explored next, first with a simple geometric example, followed by the algebraic equivalents. Note that sub-sections are based on the more detailed discussion of both GMG and AMG found in [4].

3.3.1 Coarse Grid Generation

Considering the Fourier series solution error representation, it was found that the problematic low frequency modes appear *geometrically smooth*. By representing these functions on a grid with fewer points, the relative frequency of the mode was increased due to the lower number of nodes. In both algebraic and geometric settings coarse grids must be selected based on the requirements that they must contain much fewer nodes than the fine grid and that the resulting interpolation and restriction transfer operators are sufficiently accurate. In a geometric sense then, a fine grid is merely a spatial discretisation using very small grid point spacings, while a coarse grid refers to a spatial discretisation of the same domain using larger spacings. For simple geometries, such as a one-dimensional problem, coarse grids can be constructed in the GMG approach from the fine grid, denoted by the symbol

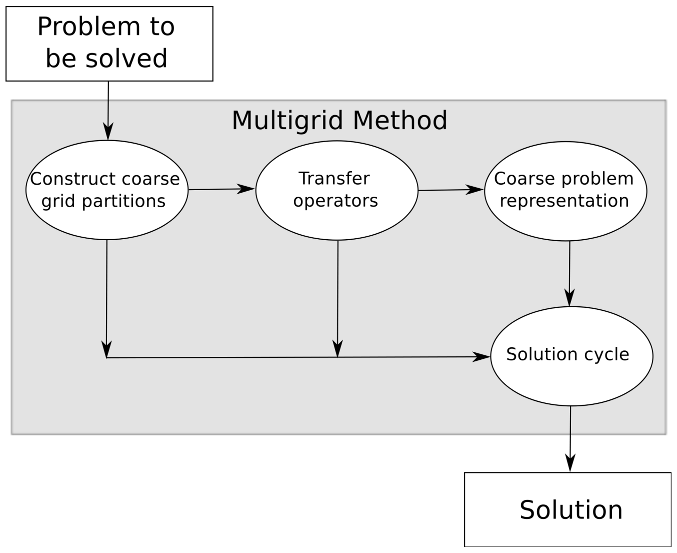


Figure 3.1: Schematic of the components of a multigrid method

Ω, by selecting a subset of the initial grid points such that the grid spacing doubles (in other words, constructing the coarse grid only from the even points of the initial fine grid). This concept is illustrated in Figure 3.2.

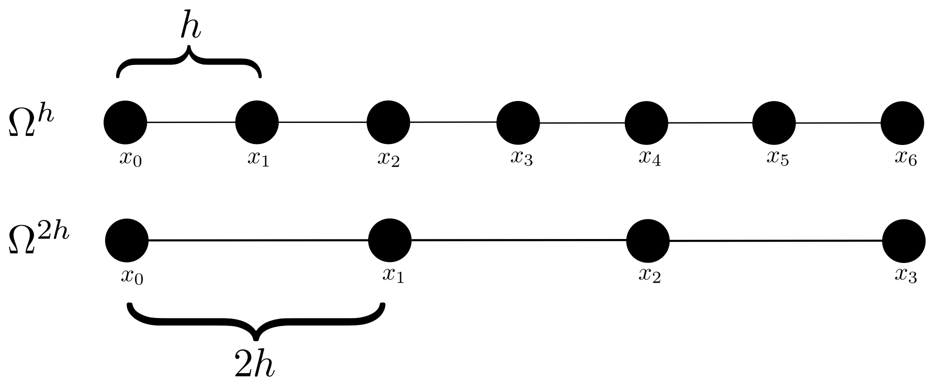


Figure 3.2: Schematic of a one dimensional geometric coarsening

As shown, Ω^h refers to the original fine grid, whereas Ω^{2h} denotes a mesh with double the grid point spacing. For the purpose of further discussion, a super or subscript of h will denote a quantity on the fine grid, and scripts of $2h$, $4h$ and so forth will denote quantities on the second and subsequent coarse grids. Although doubling of grid spacings is simple enough for structured meshes in one or two dimensions, it is obvious that coarse grid generation in a geometric fashion may become problematic when the domain is geometrically complex. In AMG, coarse grid generation is more abstract. Here, the subset of nodes are chosen based on two criteria *viz. algebraic smoothness* and *strong influence*.

Algebraic smoothness is simply a way of describing the errors that are not reduced (smoothed) effectively on a particular mesh by the chosen smoothing/relaxation method, without a geometric representation of the error. In other words, an algebraically smooth error for a node i at a certain iteration n, e_i^n, is one that is not significantly less than the error at the previous iteration, e_i^{n-1}. In the context of a simple iterative scheme such as Gauss-Seidel (most commonly used as a smoother in C-AMG methods), it can be proven that algebraically smooth errors have the characteristic that locally the magnitude of the residual is much less than that of the error [38]. This concept will be expanded in Section 3.3.2, where it will be used to define interpolation operators in the AMG setting. For the purposes of this section, it is sufficient to know that algebraic smoothness guides selection of coarse nodes to ensure accurate interpolation operators.

Strong influence can be seen as a measure of how much a given node affects those around it, and thus how suitable a node is to be chosen for the coarse grid. By examining the magnitudes of the coefficients in a row of the system matrix, a sense of the ’importance’ of a certain node in determining the value of the unknown described by that row’s equation can be gained. We associate the i-th equation of the system matrix with determining the pressure at a specific node i, Δp_i, where Δp is defined as in Chapter 2. Large coefficients in equation i will have a concomitantly large effect on the value of Δp_i. This leads to Definition 3.1 [4]:

Definition 3.1 *Given a threshold value $0 < \theta \leq 1$, the variable Δp_i strongly depends on the variable Δp_j if*

$$-A_{ij} \geq \theta \max_{k \neq i}\{-A_{ik}\}, \quad i \neq j, \{i, j, k\} \in [0,\ N-1], \tag{3.1}$$

where A_{ij} and A_{ik} denote off-diagonal components in row i of the coefficient matrix (j and k refer to column numbers in row i). In other words, the i-th variable depends strongly on the j-th variable if the coefficient A_{ij} is large relative to the largest off diagonal component in row i. Variable j is then said to *strongly influence* variable i. It is clear that, in the context of the pressure correction system, every variable in the system will have a set of nodes that strongly influence it, a set of nodes it strongly influences, and a set of nodes that have weak influence on it. In practice, the choice of θ is not critical [38]. Throughout this work, a value of $\theta = 0.8$ was found to work well.

To effect the selection of a coarse grid, a set of nodes constituting the next grid level, denoted by set **C**, is selected (in C-AMG) based on an initial fine discretisation grid via the use of two heuristics. These use the concepts of algebraic smoothness and strong influence to construct coarse grids which result in sufficiently accurate transfer operators while containing as few nodes as possible. They are:

- **Heuristic 1** : Every node that strongly influences a node i, should either be a coarse node, or depend on another coarse node that strongly influences i.

- **Heuristic 2** : The set of coarse nodes **C** should be a maximal subset of all nodes such that no coarse node depends strongly on another coarse node.

The first heuristic is designed to ensure that interpolation is sufficiently accurate, as shown in Section 3.3.2. The second is intended to limit the size of the coarse grid. It is not always possible to satisfy both heuristics simultaneously, and since the first condition is required for accurate interpolation, it is enforced rigorously. The second condition is then used as a guide in choosing an initial coarsening which is adjusted to satisfy the first condition. With this methodology, we can next step through an example coarse node selection procedure.

Initial Coarse Node Selection Procedure

As stated, the coarsening procedure in C-AMG consists of two passes over the fine grid nodes. The first obtains an initial selection of independent coarse nodes based on the second heuristic. The second ensures that the first heuristic is satisfied. In order to perform the first pass, an initial heuristic is to be assigned to every node i which indicates its suitability as a coarse grid node. This measure, called the *cardinality*, and denoted by λ_i, is defined as

Definition 3.2 *The cardinality of a node i, denoted λ_i, is the number of nodes strongly influenced by i.*

Using this value to choose a series of coarse points is referred to as the *colouring procedure* [4]. In order to clearly illustrate this procedure, an example problem is constructed. Recall Equation (2.11):

$$\frac{\partial^2 p}{\partial x_i^2} = \frac{\partial \Delta U_i^*}{\partial x_i} + \frac{1}{\Delta t}\frac{\partial U_i}{\partial x_i}$$

For the purposes of illustration, assume that the above differential equation is to be solved for the pressure, p, instead of using the pressure correction procedure described in Chapter 2. Using a standard finite difference approach, this equation can be discretised on a grid with 25 interior points, as shown in Figure 3.3. Assuming equidistant spacing h in both dimensions, a second-order finite difference discretisation results in coefficient matrix **A**, of the form

$$\frac{1}{h^2}\begin{bmatrix} 4 & -1 & 0 & 0 & 0 & -1 & & & & & & \\ -1 & 4 & -1 & 0 & 0 & 0 & -1 & & & & & \\ & . & . & . & & & & . & & & & \\ & & & . & . & . & & & & . & & \\ & & & & -1 & 4 & -1 & 0 & 0 & 0 & -1 & \\ & & & & & -1 & 4 & -1 & 0 & 0 & 0 & -1 \end{bmatrix}$$

Considering Equation (3.1), it is clear that for any choice of θ, every off diagonal coefficient in row i will have a strong influence on node i, and be strongly influenced in turn by i. These mutual strong influences can be visualised as the lines between nodes in the grid diagram. Note that strong connections are NOT necessarily mutual as in this example. Accordingly, we can assign cardinalities to every node as in Figure 3.4.

With the cardinality of all nodes known, a node with the highest value of λ is added to the list of coarse nodes. For the example, x_6 is selected, being the first node with a cardinality of four. Since, according to the second heuristic, no coarse node can strongly depend on another coarse node, the nodes that the newly chosen coarse node strongly influences are automatically removed as candidates for coarse nodes. This is shown schematically in Figure 3.5, where x_6 is marked with a solid grid point, and the nodes strongly influenced by x_6 (namely x_1, x_5, x_7 and x_{11}) are depicted by dashed lines. The latter indicates that these nodes may not appear on the coarse grid. Further, in the interest of accurate interpolation, the next coarse node selected should ideally strongly influence one of the removed (dashed)

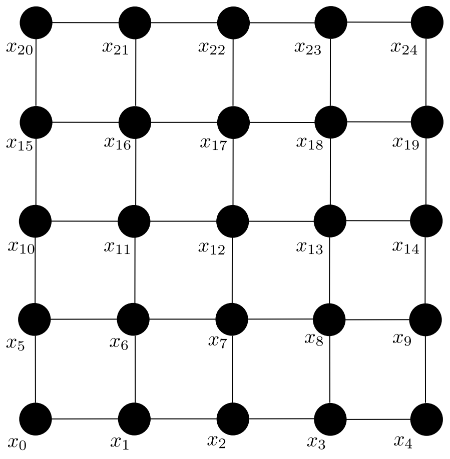


Figure 3.3: Two dimensional unit square

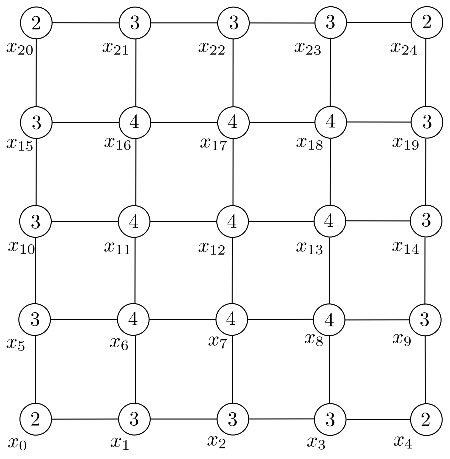


Figure 3.4: The initial cardinalities of each node

nodes. To effect this, the cardinalities of the remaining nodes are incremented according to their influence on the newly marked nodes. These changes are also reflected in the figure.

The coarse node search is now repeated, and continues until all nodes have been either accepted or rejected as a coarse node (see Figure 3.6). With the initial coarsening pass complete, we see that we have complied with both first and second heuristic. Examples where this is not the case, arising when considering asymmetric systems such as Equation (2.20), are explored next.

Secondary Coarsening

In the aforementioned example, both heuristics for the selection of coarse points were fully satisfied after the first coarsening sweep. However, the first heuristic, which requires strict enforcement, may be violated after initial coarsening. This may occur, for instance, when periodic boundary conditions are present, or when matrix coefficients are such that after initial coarsening, certain fine nodes nodes have no coarse nodes which strongly influence

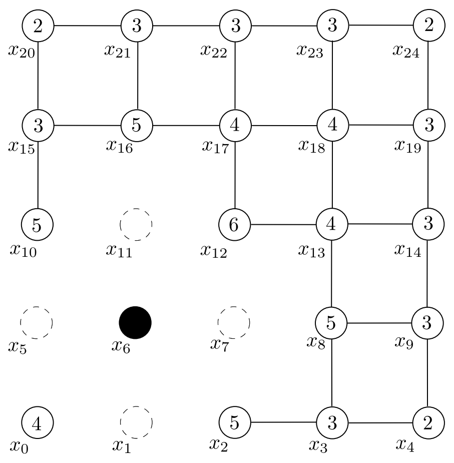


Figure 3.5: The first chosen coarse node and its effects

them. This implies that fine nodes exist that depend strongly only on other fine nodes, with no dependence on a mutual coarse point (as required by the heuristic). To illustrate this, consider the hypothetical situation in Figure 3.7, in which node x_3 strongly depends on nodes x_1 and x_2, but does not strongly influence them (indicated by the mono-directional arrows on the connecting lines). The cardinalities are given to reflect this.

On coarsening, node x_0 is selected as coarse, since it is the first node with highest cardinality. Nodes x_1 and x_2, strongly influenced by x_0, therefore remain fine. This leaves node x_3 strongly influenced by only fine nodes without a common coarse node. Colouring cannot partition node x_3 to be coarse or fine, since its cardinality is zero. To remedy this situation, and comply with the first heuristic, secondary coarsening must be performed. Secondary coarsening, as described in [4], consists of testing each fine node in turn to check for heuristic compliance. In addition, heuristic compliance is attempted by making the *minimum* number of fine nodes into coarse nodes. In short, the procedure is as follows: For each fine node i, investigate whether other fine nodes strongly influence i. If a fine node j is found that strongly influences i, but does NOT itself strongly depend on a coarse node which strongly influences i, j is *tentatively* made into a coarse node. Any remaining fine nodes strongly influencing i are now tested. If these nodes are now all strongly influenced by coarse nodes which influence i, the tentative coarse node is made permanent. If, however, another fine node is found which is not strongly influenced by a coarse node influencing i, node i itself is turned into a coarse node. In this way, it is ensured that all fine nodes strongly influencing each other are also strongly influenced by at least one mutual coarse node. As will be shown, this is critical for meaningful interpolation from coarse grids. The hypothetical situations shown in Figure 3.8 and Figure 3.9 demonstrate the secondary coarsening procedure.

In both figures, node x_3 is the fine node under consideration during secondary coarsening. Figure 3.8 illustrates the situation when only one of the fine nodes strongly influencing x_3 does not depend on a coarse node strongly influencing x_3, namely node x_2. This node is made coarse, which means that the first heuristic is obeyed. However, as seen in the mutual strong influence of nodes x_0 and x_2, the second heuristic (stating that coarse nodes may not strongly depend on each other) is violated. Figure 3.9 illustrates the case where two of

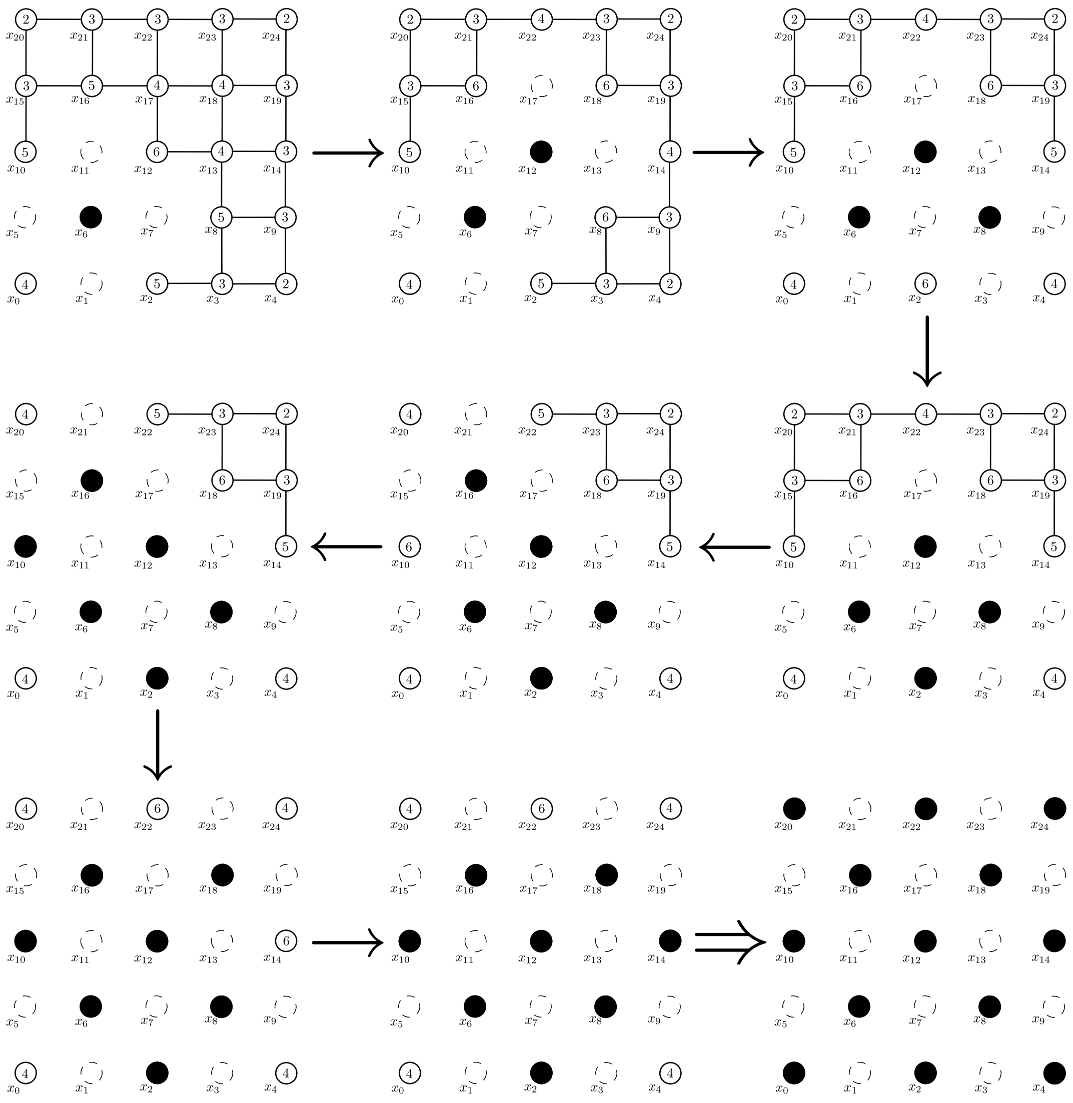


Figure 3.6: Coarse grid generation (colouring), where no secondary coarsening is required

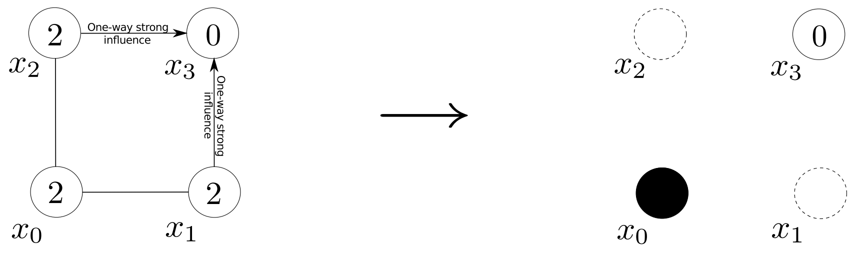


Figure 3.7: Cardinalites which violate the first heuristic after colouring

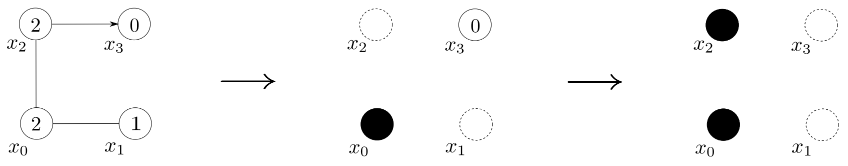


Figure 3.8: Secondary coarsening - Example 1

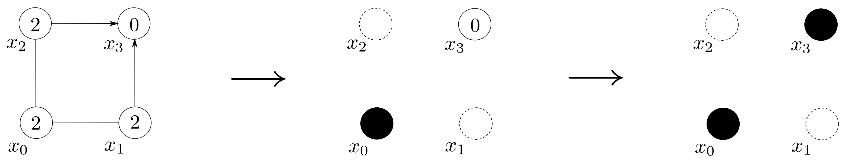


Figure 3.9: Secondary coarsening - Example 2

the fine nodes strongly influencing x_3 are not strongly influenced by coarse nodes strongly influencing x_3. In this case, node x_3 itself is made coarse, enforcing the first heuristic, and in this case, the second as well.

Potential Difficulties

The coarsening procedure used in C-AMG, as described in the preceding section, unfortunately has some limitations. These include computational cost concerns associated with secondary coarsening, and sensitivity to the definition of strength of connection [9]. The first problem is often remedied by defining alternative approaches to interpolation than is described in Section 3.3.2. The second, associated with the definition of strong influence as in Definition 3.1, can result in too many nodes being carried to coarse grids. This happens when a low value of θ classifies many weaker off-diagonal nodes as strong influences, increasing the number of nodes used for interpolation (and thus the computational cost) for little gain in accuracy. Additionally, the coarsening procedure as described does not account for large positive off-diagonal entries [38], since the original assumption on the coefficient matrix was that off-diagonal entries are negative. To remedy this, special attention must be given to these entries in the form of *a posteriori* updates of the coarse/fine splitting. Each of the above concerns have been the subject of numerous research efforts on their own. Since unmodified C-AMG is well suited to the pressure correction equation [38], the above were disregarded for the purposes of the investigation in this work.

3.3.2 Intergrid Transfer Operators

The multigrid method entails operating on the residual equation on successively coarser grids and using the estimated error obtained to improve the solution estimate on the fine mesh. For further discussion, we now formally define the residual and error in terms of Equation (3.2):

$$\mathbf{r} = \mathbf{b} - \mathbf{A}\Delta\mathbf{p}^a = \mathbf{A}\mathbf{e}, \tag{3.2}$$

where $\mathbf{p}^a$ is an approximate solution to the pressure correction equation and $\mathbf{e}$ is the error in the approximation. It follows that a method to transfer the residual $\mathbf{r}$ to the coarse grids, and the error $\mathbf{e}$ back to the fine grids, is needed. The process of transferring the residual from fine to coarse grids is called *restriction*, while the reverse process (transferring the error estimate from the coarse to the fine grids) is called *interpolation* or *prolongation* [4]. As is the case with the construction of coarse geometric spatial grids, these processes are straightforward in the one-dimensional geometric case. Consider an example with $2m$ nodes on the fine grid and m nodes on the coarse grid. Here, restriction is performed by *injection*, whereby a coarse grid node takes on the value of the corresponding node on the fine grid. This is defined as

$$r_m^{2h} = r_{2m}^h,$$

where r_m^{2h} refers to the nodal residual on the coarse grid, and r_{2m}^h to the nodal residual of the even nodes on the fine grid. The restriction procedure is implied by writing

$$\mathbf{I}_h^{2h}\mathbf{r}^h = \mathbf{r}^{2h}, \tag{3.3}$$

where $\mathbf{r}^h$ and $\mathbf{r}^{2h}$ are the residual vectors on the fine and coarse grids respectively, and $\mathbf{I}_h^{2h}$ is some restriction operator of size $(m \times 2m)$.

The most basic interpolation procedure similarly uses injection for the even numbered nodes on the fine grid, with odd numbered nodes being assigned the average of their two adjacent coarse-grid nodes. This is defined as

$$\begin{aligned} e_{2m}^h &= e_m^{2h} \\ e_{2m-1}^h &= \frac{1}{2}(e_{m-1}^{2h} + e_m^{2h}), \end{aligned}$$

where the nomenclature is as defined previously. The interpolation procedure is implied by writing

$$\mathbf{I}_{2h}^h\mathbf{e}^{2h} = \mathbf{e}^h, \tag{3.4}$$

where $\mathbf{e}^{2h}$ and $\mathbf{e}^h$ are the error correction vectors on the coarse and fine grids respectively, and $\mathbf{I}_{2h}^h$ is some interpolation operator of size $(2m \times m)$. For clarity, a graphical representation of these procedures is given in Figure 3.10.

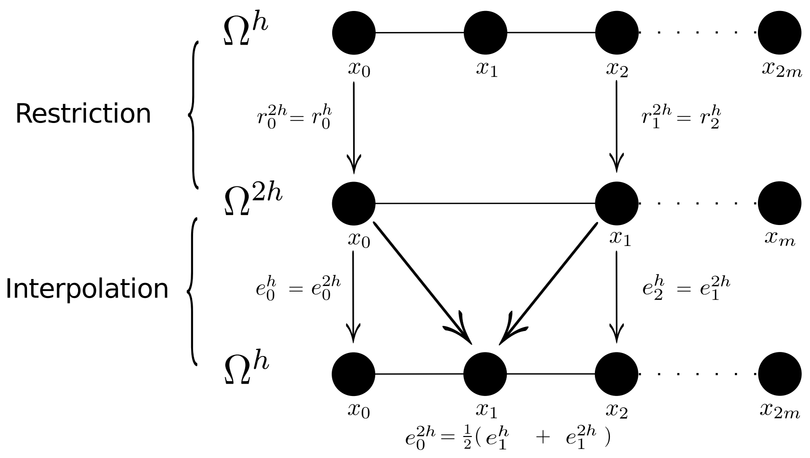


Figure 3.10: Schematic representation of the interpolation and restriction operators

Similarly, in AMG , the task of the interpolation operator is to inject error components from the coarse grid representation to the fine when the node under consideration occurs in both. In the case of the reverse, the error is interpolated to fine grid nodes. As opposed to the simple linear interpolation used in the geometric example, AMG interpolation is based on the concept of a node's *neighbourhood*. The neighbourhood of a node i, denoted N_i, is simply defined as all the off-diagonal coefficients in the row i of **A**. The neighbouring nodes are in addition classified, following Figure 3.11, into:

- Neighbouring nodes that are coarse and influence i strongly, S_i^C
- Neighbouring nodes that are not coarse, but influence i strongly, S_i^F
- Neighbouring nodes that do not influence i strongly, whether coarse or not, W_i.

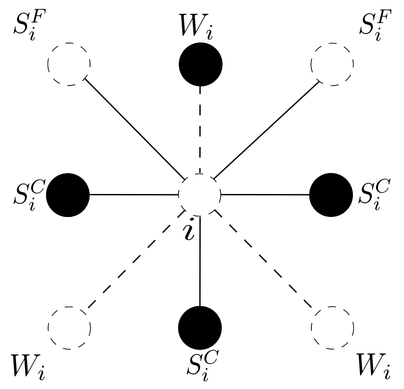


Figure 3.11: Neighbourhood classification of a node

In the figure, the fine and coarse nodes are depicted as previously. Strong influences are again solid lines, while weak influences are dashed lines. Interpolation of the error for a fine node i consists of a weighted sum of each of the coarse nodes strongly influencing it, that is, the nodes S_i^C. The *interpolation weight* ω_{ij} of each coarse node j which strongly influences node i takes into account any strong influences on i which are not from coarse points, as well as any weak influences on i, that is, the nodes S_i^F and W_i. This is done by using the principle of algebraic smoothness alluded to previously. Once these interpolation weights are known, we can define an interpolation operator for every node i on a fine grid h such that

$$(\mathbf{I}_{2h}^{h}\mathbf{e}^{2h})_i = \begin{cases} e_i^{2h} & i \in \mathbf{C} \\ \sum_{j \in S_i^C} \omega_{ij} e_j^{2h} & i \notin \mathbf{C} \end{cases} \tag{3.5}$$

where e_i is the error at node i, ω_{ij} is the interpolation weight of each coarse j strongly influencing i and **C** is the set of coarse nodes.

In order to calculate the values of ω_{ij}, we recall the assumption of algebraic smoothness, that is, for smooth errors, the residual magnitude is much smaller than the error magnitude. We can write this as :

$$|r_i| << A_{ii}|e_i|, \quad i \in [0,\ N-1], \tag{3.6}$$

where A_{ii} refers to the diagonal entry of a row i in the coefficient matrix, and r_i to the residual at node i. Further, N is the size of the matrix **A**, where numbering starts from zero, conforming with C++ programming conventions. If we take the idea that the residual is much smaller than the error to the extreme we can state that:

$$r_i = (\mathbf{Ae})_i \approx 0, \tag{3.7}$$

where **e** now refers to the error vector. Rewriting Equation (3.7) in terms of the coefficients of **A** for a node i, we obtain

$$\begin{aligned} A_{ii}e_i + \sum_{j \in N_i} A_{ij}e_j &\approx 0 \\ \Rightarrow \quad A_{ii}e_i &\approx -\sum_{j \in N_i} A_{ij}e_j \end{aligned}$$

in which the sum over the neighbourhood of i, N_i can be separated into contributions from its constituent nodes S_i^C, S_i^F and W_i as follows

$$A_{ii}e_i \approx -\sum_{j \in S_i^C} A_{ij}e_j - \sum_{j \in S_i^F} A_{ij}e_j - \sum_{j \in W_i} A_{ij}e_j. \tag{3.8}$$

Here, the error at a node i is approximated by the sum of neighbouring nodes' values. It is not possible to calculate the interpolation directly from this equation as only coarse nodal values are directly available. To address this, the last two terms of Equation (3.8) must be written in terms of the error at strongly influencing coarse nodes, S_i^C, as well as the error at the node i itself, e_i. This will produce an interpolation involving only the node i and its strongly influencing coarse nodes. To effect this, we follow [4], in which the contribution of the e_j in the sum over weak neighbours W_i can be replaced by e_i (distributing the weakly connected neighbours to the diagonal). Thus the component sums of interpolation is rewritten as

$$\left(A_{ii} + \sum_{j \in W_i} A_{ij} \right) e_i \approx - \sum_{j \in S_i^C} A_{ij} e_j - \sum_{j \in S_i^F} A_{ij} e_j . \tag{3.9}$$

Next, we must similarly address the S_i^F values. We do this by recalling the first heuristic of coarsening, namely that fine nodes strongly influencing one another must also be strongly influenced by a mutual coarse node. The errors e_j in the sum over S_i^F can therefore be replaced by linear combinations of the error at coarse nodes S_i^C such that

$$e_j \approx \frac{\sum_{k \in S_i^C} A_{jk} e_k}{\sum_{k \in S_i^C} A_{jk}}, \quad j \in S_i^F, \tag{3.10}$$

where e_k is now the error at the strongly influencing coarse nodes S_i^C. Using this approximation, the interpolation weight of a coarse point j, with respect to a fine point i, namely ω_{ij}, can be finally written as

$$\omega_{ij} = - \frac{A_{ij} + \sum_{m \in S_i^F} \left(\frac{A_{im} A_{mj}}{\sum_{k \in S_i^C} A_{mk}} \right)}{A_{ii} + \sum_{n \in W_i} A_{in}}. \tag{3.11}$$

With the interpolation weights defined, the interpolation operator can be constructed. The restriction operator is then simply obtained via the variational property

$$\mathbf{I}_h^{2h} = (\mathbf{I}_{2h}^h)^T \tag{3.12}$$

where the nomenclature is as defined previously.

3.3.3 Coarse Grid Operators

As has been mentioned, the multigrid methods rely on operating on the problem on multiple levels of 'grids'. The above interpolation and restriction operators are used to transfer errors and residuals between grids, but how would the system matrix **A** be represented? For simple geometric problems, one approach is to re-discretise the original problem on the coarser grid in the same fashion as on the fine grid. Since this approach is generally not applicable to algebraic methods, the Galerkin principle is instead used. This is written as

$$\mathbf{A}_{2h} = \mathbf{I}_h^{2h} \mathbf{A}_h \mathbf{I}_{2h}^h \tag{3.13}$$

where, again, $\mathbf{I}_{2h}^h$ refers to the interpolation operator and $\mathbf{I}_h^{2h}$ to the restriction operator. The size of $\mathbf{A}_{2h}$ is $(N_C \times N_C)$, where N_C is the number of coarse nodes on the next grid level.

3.3.4 Solution Cycle

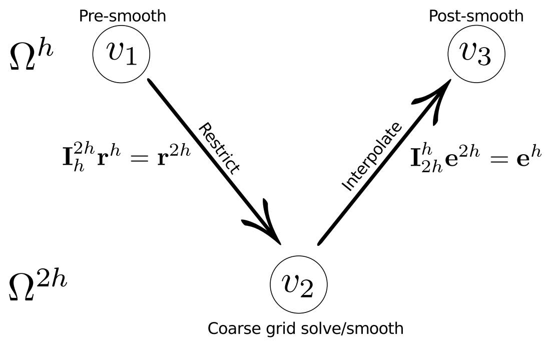


Figure 3.12: Schematic of the two grid V-cycle

As shown in Figure 3.1, the final AMG component to be discussed is the solution cycle. This cycle essentially brings together all components of the method in a manner which effects efficient solution. The simplest possible cycle, shown in Figure 3.12, is the two grid V-cycle. In the context of Equation (2.20) this cycle consists of the following:

1. Relax $\mathbf{A}^h\Delta\mathbf{p} = \mathbf{b}$ for v_1 iterations using Gauss-Seidel on the fine grid with initial estimate $\Delta\mathbf{p}^a$
2. Calculate $\mathbf{r}^h$ using $\mathbf{r}^h = \mathbf{b} - \mathbf{A}^h\Delta\mathbf{p}^a$
3. Restrict $\mathbf{r}^h$ to the coarse grid, using $\mathbf{I}_h^{2h}\mathbf{r}^h = \mathbf{r}^{2h}$
4. Relax $\mathbf{A}^{2h}\mathbf{e}^{2h} = \mathbf{r}^{2h}$ for v_2 Gauss-Seidel iterations on the coarse grid with initial guess $\mathbf{e}^{2h} = \mathbf{0}$
5. Interpolate $\mathbf{e}^{2h}$ to the fine grid using $\mathbf{I}_{2h}^{h}\mathbf{e}^{2h} = \mathbf{e}^h$
6. Correct the estimate $\Delta\mathbf{p}^a$ using $\Delta\mathbf{p}^a = \Delta\mathbf{p}^a + \mathbf{e}^h$
7. Again relax $\mathbf{A}^h\Delta\mathbf{p}^a = \mathbf{b}$ for v_3 Gauss-Seidel iterations on the fine grid with the corrected initial estimate
8. Use the corrected estimate as initial guess for next cycle.

Simply put, the solution cycle commences by relaxing the problem on the finest grid, using some simple iterative smoother such as Jacobi or Gauss-Seidel for v_1 iterations (*pre-smoothing*, followed by calculating and transferring the residual to the coarser grid. On this grid, an error is solved for via either v_2 smoother iterations, or directly. The calculated 'improved' error estimate is interpolated to the fine grid and added to the solution estimate (so-called *correction*). On the fine grid, another v_3 relaxation iterations are performed (*post-smoothing*, completing the V-cycle. The updated solution estimate thus obtained can then be used as the initial estimate for the next V-cycle.

In practice, this process of pre-smoothing, restriction, interpolation and post-smoothing may be extended recursively over multiple levels, but the principles remain the same. Different cycle shapes are also possible, but the V-cycle was deemed adequate for this work.

The optimal values of v_1, v_2 and v_3 are typically problem dependent, and affect the number of V-cycle iterations required to converge. For the purposes of this work, the values used throughout were two, one and four respectively, and were found to perform well on the problems considered in Chapter 5.

3.4 Algebraic Multigrid and FSM

The preceding sections offered an introduction to multigrid methods, in particular providing details of the underlying algorithms of Classical Algebraic Multigrid. Clearly, the algorithm constructs the AMG solution structure from the coefficient matrix **A**. Indeed, this is one of the method's greatest strengths, since this enables it to be an ideal black box type solver. Strictly speaking however, a given algebraic multigrid solution structure is only applicable to the specific coefficient matrix it is constructed from. It stands to reason that if the coefficient matrix changes, this structure must be rebuilt. This is especially true in the FSM problem under consideration, where the matrix coefficients change at every simulation time step due to the evolving fluid interface. Rebuilding the AMG solution structure at every time step would negatively impact performance as the setup phase in C-AMG consumes a comparable amount of time to the solution cycle itself. As such, applying the AMG method in this context requires some special attention in order to remedy this problem. The method developed to effect this is detailed next.

3.5 Freeze-AMG

As mentioned previously, the basic C-AMG method described in this chapter is applicable as-is to the solution of the FSM pressure correction equation, but will be hamstrung by the setup phase. As a result, for the purposes of this work, a method was developed to circumvent this limitation. One approach is to use an alternative problem formulation in which the coefficient matrix of the pressure equation stays unchanged [14]. This allows one AMG setup to be performed, which can then be used for all simulation time-steps. Since it was not feasible to change the formulation used by *Elemental*, as seen in Chapter 2, a different approach was required. In addition, concerns with guaranteeing incompressibility exist in the alternative formulation.

From Chapter 2 it is clear that matrix coefficient changes due to the evolving interface are necessarily *localised* around nodes near the free interface. This being the case, a given AMG solution structure could be expected to converge for coefficient matrices from several consecutive time-steps, provided matrix coefficients did not deviate unacceptably far from their original values [41] (proven as a valid assumption where AMG is used as a preconditioner). In other words, some initial AMG solution structure could be ’frozen’ for many time-steps and used to solve the pressure correction equation. A new solution structure would then be constructed once the coefficient matrix deviated too far from the original matrix. This deviation could be expressed simply as the number of V-cycles required by the AMG solver to converge. For the purposes of this work, a conservative upper limit of 50 V-cycles was set.

From numerical experiments, it was found that the coarsening procedure in Section 3.3.1 is the most CPU intensive phase in the setup procedure, consuming as much time as the rest of the setup phases combined. Therefore, an extension of the above Freeze method, in which only *partial* AMG setup was performed once convergence deteriorated was also investigated. This partial AMG setup would consist of only updating the magnitudes of operator coefficients, maintaining previously selected coarse nodes. These AMG refinements were dubbed *Freeze-AMG* and *Extended Freeze-AMG*, and are illustrated in Figure 3.13.

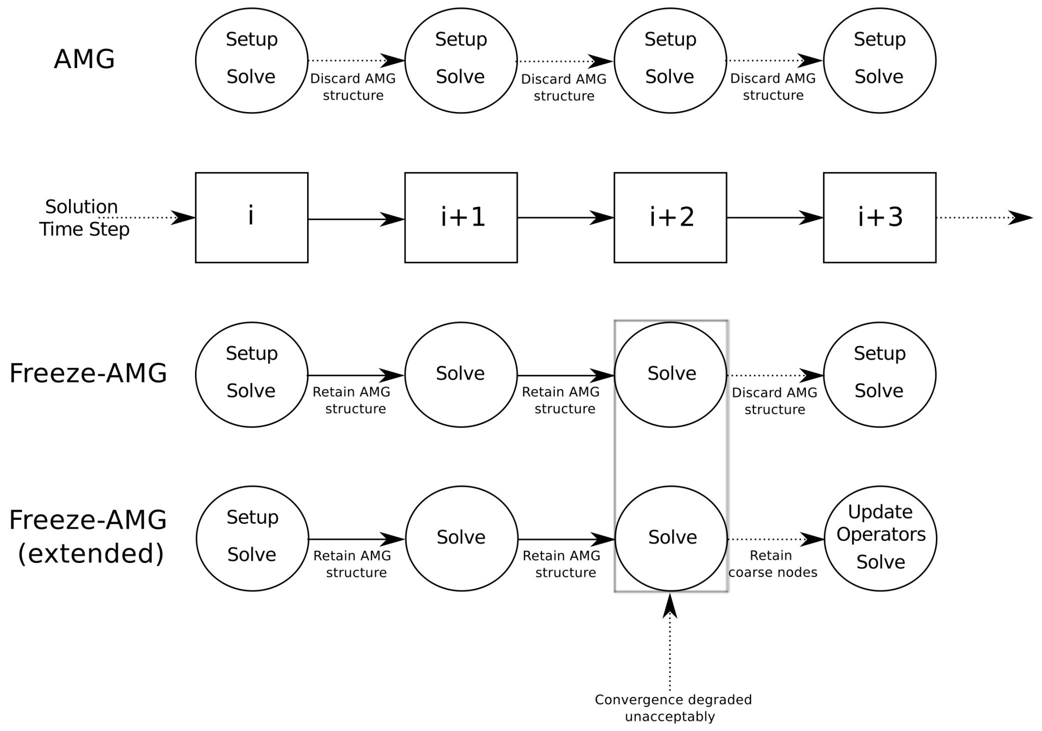


Figure 3.13: Comparison of Freeze-AMG and C-AMG methodologies

3.6 Summary

This chapter provided background to the various solution strategies historically applied in the context of systems of linear equations. Motivation for using the AMG approach, based on its characteristics was provided, with the choice of Classical Algebraic Multigrid for this work justified. In order to provide a foundation on which to describe the implementation of an algebraic multigrid solver, the principles of multigrid were described and illustrated using simple geometric examples. These examples were extended to illustrate the basic AMG theory. Commentary was given on the difficulty faced by the AMG method when solving problems with changing coefficient matrices, such as those occurring in FSM simulations. Finally, the method used to improve on basic AMG in this work, namely Freeze-AMG, was discussed and motivated. The next chapter will discuss in detail how the algorithms constituting the AMG solver were implemented.

Chapter 4

Object-Oriented C++ Implementation

4.1 Introduction

Chapter 3 provided background on the mechanics of the chosen C-AMG solver. The current chapter will detail the actual implementation in an object-oriented C++ environment as a plug-in solver to *Elemental*. To this end, the practicalities of interfacing with *Elemental* and the object-oriented data structures employed are described. For clarity, the remainder of the chapter is then divided into subsections based on the separate AMG components in Chapter 3, as they relate to an object-oriented implementation. Special attention is given to the details of the algorithms, such that the desired computational efficiency is achieved. This is critical in the implementation of an AMG solver, since the theoretical optimal complexity is only possible if *all* individual components achieve this ideal. Finally, the implementation details of the Freeze-AMG method are provided. Before these discussions however, some background and justification for the use of C++ in the context of this AMG implementation is provided.

4.2 C++ as Scientific Programming Language

Historically, the programming language of choice in numerical applications was Fortran 77, which is a procedural type language. Procedural approaches typically become problematic when faced with large codes of high complexity [7], due to issues with code maintainability and reusability. As a result, software design principles have evolved, culminating in the development of *object-oriented programming*. This in turn has resulted in a re-evaluation of the scientific programming paradigm [5]. The principle of *encapsulation*, for example, which is core to the object-oriented approach, greatly enhances code maintainability and reusability, meaning that this approach has gained much momentum in the field of numerical modelling.

As mentioned, Fortran 77, along with its updated version, Fortran 90, are well established in a numerical programming context, offering native support for mathematical arrays. Unfortunately, the Fortran family of languages were not designed with object-oriented approaches in mind, in contrast to the C++ language. In the context of an AMG solver, Fortran was deemed less suitable than C++, since the object-oriented approach lends itself well to such a solver. Specifically, since an AMG solver utilises multiple discreet levels when solving a system of linear equations, with minimal interaction between these levels, the encapsulation offered by the object-oriented approach is naturally desirable. Furthermore, since the solver was to be developed as a plug-in to *Elemental*, the modular nature of object-oriented approaches could be exploited. In light of these factors, C++ was the language chosen for the current work.

4.3 Interface with Elemental

Since the solver would have to operate as a plug-in to an existing program structure, the AMG package was coded as a separate library which could be included into any existing package. In order to take advantage of the object-oriented environment of C++, the solver was created as a self-contained class, in which all the functions needed for the AMG method were defined. This self-contained nature of the AMG solver made it possible to interface with *Elemental* with the minimum amount of changes to the main code. At the start of any FSM simulation in *Elemental* a new AMG solver object would be created, as shown in Figure 4.1. At every simulation timestep where a pressure correction solution is required, the coefficient matrix and right hand side of Equation (2.20) would be passed to the object during the pressure correction step. The object would then construct an AMG solution structure and obtain the pressure correction solution independently before passing the answer back to *Elemental*.

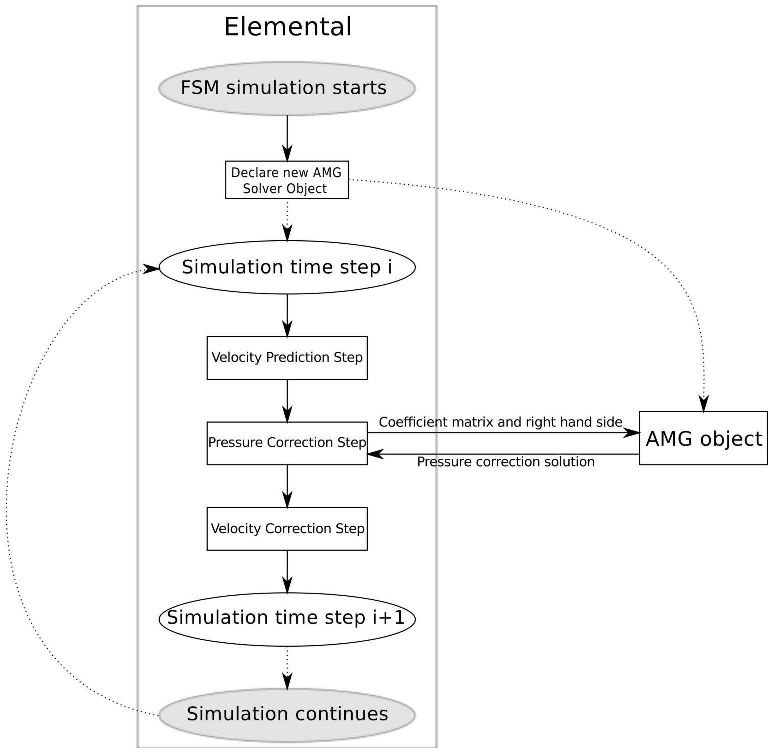


Figure 4.1: Schematic of interaction between the solver and Elemental

4.4 Solver Structure

The structure of the AMG object itself is depicted in Figure 4.2. As shown, the object contains a series of level objects, each of which is self contained. These contain the relevant information for each coarse level of an AMG solution structure, including the coefficient matrices, problem right hand side, transfer operators and the list of nodes selected as coarse. A function within the AMG object creates these levels and performs the necessary operations to obtain the aforementioned information. Finally, a solution cycle is implemented in the AMG object which utilises the level objects in order to obtain the solution to the linear equation set initially passed from *Elemental*. The remainder of this chapter investigates each of the components of the AMG object and the evolution of the implemented algorithms, following the structure of Figure 4.2.

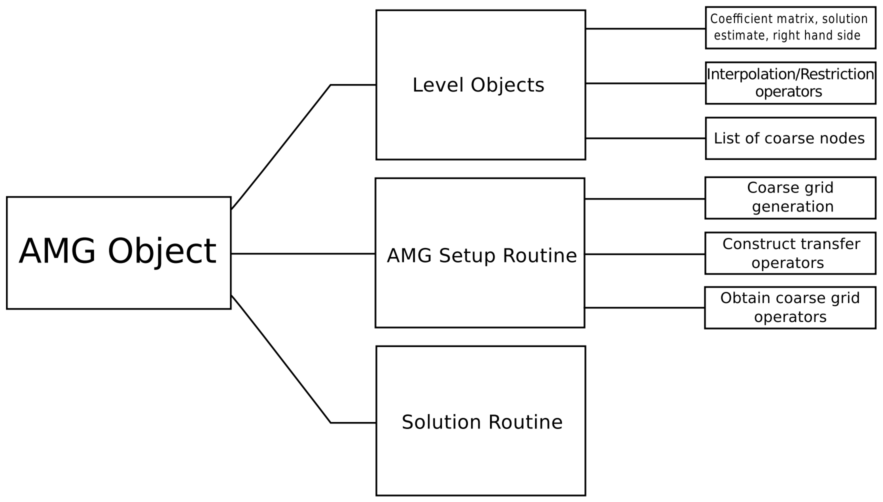

AMG Object
Level Objects
AMG Setup Routine
Solution Routine
Coefficient matrix, solution estimate, right hand side
Interpolation/Restriction operators
List of coarse nodes
Coarse grid generation
Construct transfer operators
Obtain coarse grid operators

Figure 4.2: Structure of the AMG object

4.5 The Level Objects

Contained as members of the AMG object, a number of level objects are stored, one for each discrete coarse level resulting from AMG setup. Defining each level as a separate object enabled the AMG setup function to create new levels as required instead of relying on a predefined number of levels. Each of these objects encapsulates all information specific to a level, independent from other levels. Of particular interest are the level coefficient matrix and transfer operator matrices contained within each object, which are large, sparse matrices. These require a memory efficient storage scheme in order to leverage their sparsity. To this end, the compressed row sparse (CRS) storage scheme already in use in *Elemental* was employed. This scheme utilises three vectors containing coefficients, column numbers and row offsets in order to represent a matrix, storing only non-zero coefficients. This method requires $O(nnz)$ memory storage for each matrix stored, where nnz is the number of non-zero coefficients in the matrix. It however has the disadvantage that specific column elements are not available without a search through an entire row. This limitation necessitates intelligent data handling, which is detailed in sections to follow.

4.6 The AMG Setup Routine

Before a solution estimate could be obtained and passed back to *Elemental* for a given time step, an AMG solution structure has to be constructed according to the principles in Chapter 3. Using an initial coefficient matrix, individual coarse levels have to be defined. This is accomplished by performing several discrete steps, each with its own efficiency challenges. On every level, these steps consist of obtaining a set of coarse nodes which will become the next level, using these nodes to construct transfer operators to and from the next level, and finally constructing the next coarse level's coefficient matrix. These steps are discussed individually in this section, with details of how each algorithm evolved in order to obtain optimal complexity.

4.6.1 Coarse Grid Generation

As shown in Section 3.3.1, the first task in a multigrid method is to define coarse grids based on an initial fine grid. Clearly, this selection can be subdivided into several algorithms which must be applied. Following the aforementioned section, we must first obtain the strong influences of every node, according to Definition 3.1. This data can then be used to perform a colouring procedure, satisfying the second heuristic of C-AMG. Finally, secondary coarsening must be performed in order to satisfy the first heuristic of C-AMG. At this point, a set of coarse nodes which will define the next level will have been obtained, and the construction of transfer operators can commence.

Strong Influence Identification

As stated in Section 3.3.1, each node i in the coefficient matrix **A** possesses a set of nodes that strongly influence it, as well as a set of nodes which it strongly influences, as determined by Definition 3.1. Applying this relation, every off-diagonal coefficient j in a row i must be compared to some threshold value as detailed in Section 3.3.1, and j assigned to the set of nodes which strongly influences i if needed. This procedure is performed for all N rows in the matrix **A**. For the purposes of this section, let this procedure result in every node i having a set S_i^{from}, containing nodes strongly influencing it, and a set S_i^{on}, which contains nodes it strongly influences. Thus, $2N$ sets, each with size depending on the neighbourhood of each node i, are constructed.

Before constructing these sets, a storage mechanism for them was required. Since it is not possible to determine *a priori* how many nodes will be contained in each S_i^{from} and S_i^{on}, it was considered prudent to use the standard vector class for each node i's sets, since objects of this class can be dynamically resized. In order to facilitate easy access, these set vectors were all stored in two other vectors of fixed size N such that every index of these corresponded to a node i. This is shown in Listing 4.1. With the storage scheme implemented, attention could be turned to the algorithm which would construct the sets.

```
AMGSetup::coarsening_function
{
  //Declare vectors containing sets S_i^from and S_i^on
  vector <<vector <int> > strong_influence_from, strong_influence_on;
  //Assign sets
  set_construction_algorithm;
};
```

Listing 4.1: Declaring vectors of sets

Initially, a segregated approach was attempted in which the S_i^{from} and S_i^{on} sets were determined in separate sub-sections of the algorithm, as shown in Listing 4.2. In this approach, the maximum off-diagonal coefficient required by Definition 3.1 for every node i is obtained by inspecting every off-diagonal in that row. Once this maximum is obtained, another pass over the nodes in row i is performed, adding the appropriate nodes to S_i^{from}. These searches over the off-diagonals were limited by the bandwidth of the matrix, which is extremely small compared to the number of nodes N. Constructing a node i's set S_i^{on} consisted of searching for occurrences of i in all other nodes j's sets S_j^{from}.

```
AMGSetup::coarsening_function
{
  //Assign sets
  //Construct strong_influence_from
  for (all nodes on level, i)
  {
    //Determine maximum off-diagonal
    for (off-diagonal nodes, j, in row i of coefficient matrix)
    {
      (obtain maximum off-diagonal)
    }
    //Construct set strong_influence_from
    for (off-diagonal nodes, j, in row i of coefficient matrix)
    {
      (compare nodes to maximum)
      if (coefficient j > maximum*threshold) add j to i's
          strong_influence_from
    }
  }
  //Construct strong_influence_on
  for (all nodes on level, i)
  {
    //Compare to all other nodes on level
    for (all nodes on level, j)
    {
      for (all nodes in j's strong_influence_from)
      {
        if (i occurs in j, add j to i's strong_influence_on)
      }
    }
  }
};
```

Listing 4.2: Segregated set construction

While this implementation works, it is immediately obvious that the separate calculation of S_i^{on} is of $O(N^2)$. This is due to the nested loop in which a search across *all* N sets S_j^{from} is performed for all N nodes i. Clearly, this method is nonsensical, since only the nodes in a specific i's neighbourhood could possibly be added to its S_i^{on} set. As a result, an improved version was created in which both sets are constructed simultaneously, shown in Listing 4.3. This method was based on the obvious fact that if a node i is strongly influenced by a node j, j can be added to S_i^{from} as normal, while i can be added immediately to S_j^{on}.

```
AMGSetup::coarsening_function
{
  //Assign sets
  for (all nodes on level, i)
  {
    //Determine maximum off-diagonal
    for (off-diagonal nodes, j, in row i of coefficient matrix)
    {
      (obtain maximum off-diagonal)
    }
    //Construct strong_influence_from and strong_influence_on
    for (off-diagonal nodes, j, in row i of coefficient matrix)
    {
      (compare nodes to maximum)
      if (coefficient j > maximum)
      {
        (add j to i's strong_influence_from)
        (add i to j's strong_influence_on)
      }
    }
  }
};
```

Listing 4.3: Simultaneous set construction

Using this approach, the influence algorithm was improved to have $O(N)$ complexity, since the outer loop over all nodes is performed only once, while the inner search for the maximum off-diagonal and the comparison to this maximum are limited by the small bandwidth of the matrix, as mentioned before. An evaluation of this algorithm (and all other algorithms in this chapter) was performed by applying it to a single time step of the structured mesh sloshing problem of Section 5.2. Results of this evaluation are shown in Figure 4.3. As shown, various mesh sizes were evaluated. From the curve fit in this figure, it is immediately obvious that the algorithm scales linearly, with problem size.

The Coarsening Procedure : Initial Coarsening

Once the strong influences of all nodes were determined, the initial coarsening phase as described by the procedure in Section 3.3.1 could be implemented. As was the case in the code developed for determining strong influence, multiple methods were implemented before desired performance was achieved. The prototype coarsening algorithm developed and evaluated was based on that in the AMGLab toolkit [30]. This package, developed in Matlab, was intended as a rapid prototyping and expert toolkit for AMG methods as a

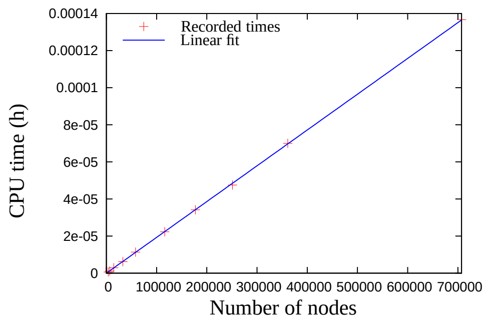


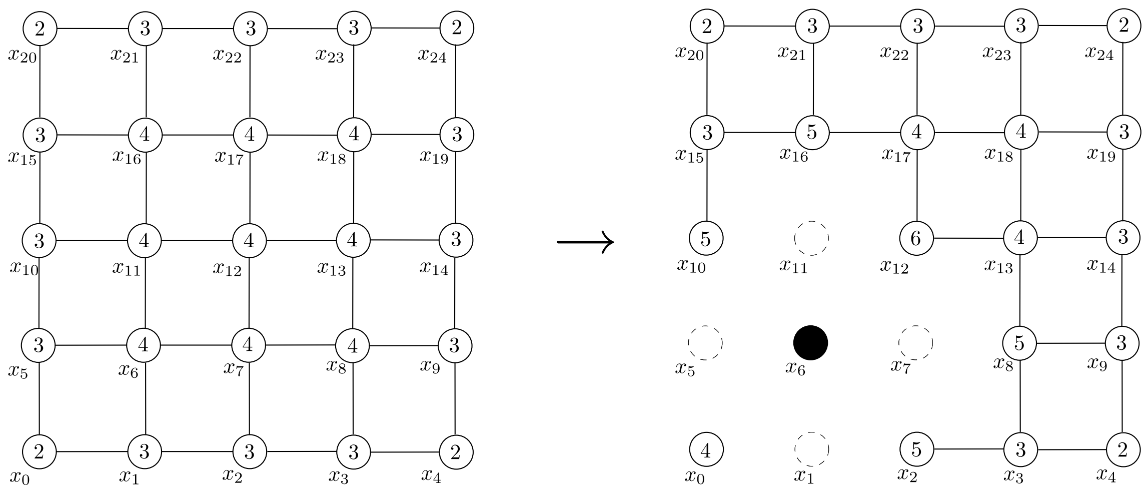
Figure 4.3: Influence algorithm scaling

whole. The rewriting of this algorithm specifically for C++ was required since the AMGLab algorithms were unsuited to larger problems, as will be shown. Additionally, AMGLab does not include secondary coarsening abilities, and the Matlab programming methodology was incompatible with the code used in *Elemental*.

An outline of the AMGLab coarsening approach is provided in Listing 4.4. The basic premise rests on initially assigning all nodes to an unsorted set. In this set, a search for the node with highest cardinality is done, which is then added to the coarse set. Nodes that are strongly influenced are removed from the working set, and marked as remaining fine, while cardinalities are updated to reflect the new coarse selection. This process continues until all unsorted nodes had been dealt with. To clarify the procedure, an example of the AMGLab approach for one coarse selection is shown in Figure 4.4, which recalls the example in Section 3.3.1.

```
{
  //AMGLab coarsening
  unsorted_set = (all nodes on current level);
  coarse_set = (empty set);
  fine_set = (empty set);
  while (unsorted_set not empty)
  {
    (calculate cardinalities of all nodes in unsorted_set)
    (find node in unsorted_set with maximum cardinality)
    (add node to coarse_set, remove from working_set)
    (add coarse node's strong influenced nodes to fine_set, remove from
        working_set)
    (update cardinalities of nodes in unsorted_set)
  }
};
```

Listing 4.4: Details of AMGLab coarsening algorithm



x_{20} x_{21} x_{22} x_{23} x_{24}
x_{15} x_{16} x_{17} x_{18} x_{19}
x_{10} x_{11} x_{12} x_{13} x_{14}
x_5 x_6 x_7 x_8 x_9
x_0 x_1 x_2 x_3 x_4

Unsorted	Coarse	Fine
0		
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		
11		
12		
13		
14		
15		
16		
17		
18		
19		
20		
21		
22		
23		
24		

Unsorted	Coarse	Fine
0	6	1
2		5
3		11
4		7
8		
9		
10		
12		
13		
14		
15		
16		
17		
18		
19		
20		
21		
22		
23		
24		

Figure 4.4: AMGLab coarsening outline

Fundamentally, performing repeated searches over all unsorted nodes to find the next maximum cardinality results in complexity higher than $O(N)$ since approximately $N/2$ searches over a set of nodes initially containing N nodes will have to be done, assuming that $N/2$ coarse points are selected. Indeed, numerical tests performed on a simple 5-point Poisson discretisation matrix of various sizes showed this algorithm to be of greater complexity than $O(N)$, as shown in Figure 4.5. This figure clearly shows scaling of $O(N^2)$.

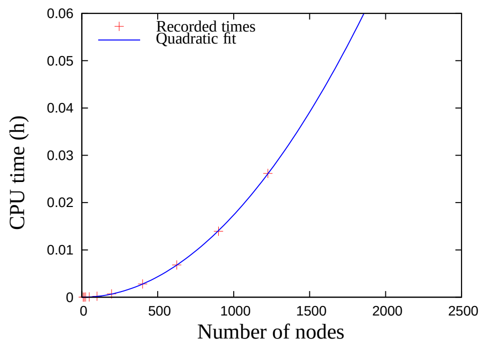


Figure 4.5: Scaling of the AMGLab coarsening algorithm

What exacerbated the complexity of the C++ implementation was the method in which nodes are removed from the working set. In the Matlab environment, newly marked coarse and fine nodes can be removed from the unsorted node list by using built-in high performance tools which can be used to obtain intersects, set differences and unions. In C++, specific nodes to be removed from the working set need to be searched for individually in the unsorted node list.

In an attempt to mitigate these search loops, an intelligent data storage structure was sought, with the underlying idea that the information required for coarsening should, ideally, be updated as coarsening proceeds such that the next node to be made coarse is immediately available when needed. In the first attempt to achieve this, each node is assigned a vector containing its own coarsening information. Individual vectors of information are combined into a standard vector object in C++, in ascending order of node number. The coarsening information of each node could be updated in this vector as coarsening proceeded. Figure 4.6 depicts a vector containing N nodes' information. As shown, the nodal coarsening information includes a node's number, its current cardinality, and whether it still needs to be partitioned into the coarse grid.

During the selection of a coarse node, the vector containing all nodes' information is sorted in some way such that the next node to be made coarse appears first in the vector. Figure 4.7 illustrates the method on the same example shown before, while pseudocode for this process appears in Listing 4.5. Using the standard C++ sort function, this vector is sorted in order of decreasing cardinality, with already sorted nodes last. This results in the next most suitable coarse point being the first in the vector, which is then duly added to the coarse set, and the information of affected nodes updated. This process then repeats until all coarsening is complete. The repeated resorting of the coarsening information causes the performance of this method to be non-optimal, since at best the complexity of a single such

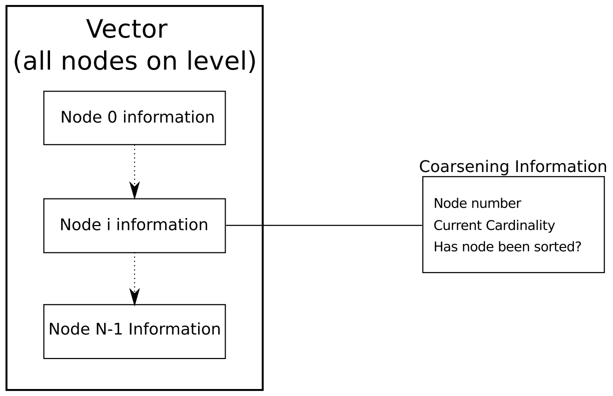
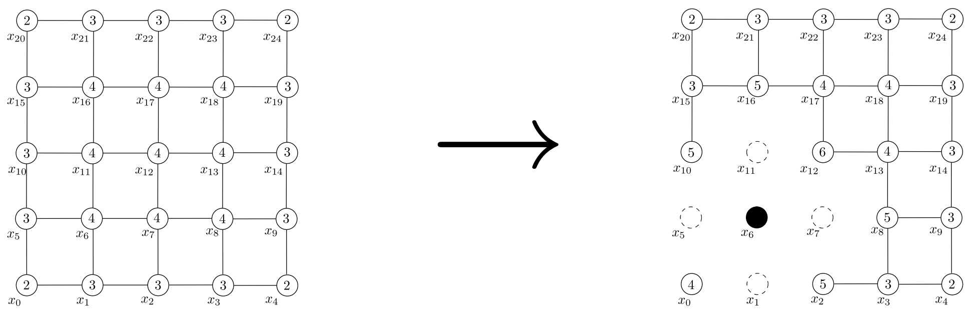


Figure 4.6: The coarsening information vector

sort in C++ is $O(N \log N)$. Instead, a method was needed in which only nodes directly affected by each coarsening step were accessed.

```
{
  //Vector Sort coarsening
  coarse_set = (empty set);
  fine_set = (empty set);
  information_vector;   //Vector containing nodes' coarsening information
  (assign initial cardinalities to information_vector)

  information_vector_clone = information_vector; //Create clone which
      will be sorted
  (sort information_vector_clone by cardinality)

  while (all nodes not sorted)
  {
    (add first node in information_vector_clone to coarse set, mark as
        sorted in information_vector)
    (add coarse node's strong influenced nodes to fine_set, mark as
        sorted in information_vector)
    (update unsorted points' cardinalities in information_vector)

    information_vector_clone = information_vector;
    (sort information_vector_clone by cardinality)
  }

};
```

Listing 4.5: Details of vector sort coarsening algorithm



Coarsening Information Vector

Node Number	Cardinality	Sorted?
0	2	N
1	3	N
2	3	N
3	3	N
4	2	N
5	3	N
6	4	N
7	4	N
8	4	N
9	3	N
10	3	N
11	4	N
12	4	N
13	4	N
14	3	N
15	3	N
16	4	N
17	4	N
18	4	N
19	3	N
20	2	N
21	3	N
22	3	N
23	3	N
24	2	N

Coarsening Information Vector (Sorted)

Node Number	Cardinality	Sorted?
6	4	N
7	4	N
8	4	N
11	4	N
12	4	N
13	4	N
16	4	N
17	4	N
18	4	N
1	3	N
2	3	N
3	3	N
5	3	N
9	3	N
10	3	N
14	3	N
15	3	N
19	3	N
21	3	N
22	3	N
23	3	N
0	2	N
4	2	N
20	2	N
24	2	N

Coarsening Information Vector

Node Number	Cardinality	Sorted?
0	4	N
1	-	Y
2	5	N
3	3	N
4	2	N
5	-	Y
6	-	Y
7	-	Y
8	5	N
9	3	N
10	5	N
11	-	Y
12	6	N
13	4	N
14	3	N
15	3	N
16	5	N
17	4	N
18	4	N
19	3	N
20	2	N
21	3	N
22	3	N
23	3	N
24	2	N

Coarsening Information Vector (Sorted)

Node Number	Cardinality	Sorted?
12	6	N
2	5	N
8	5	N
10	5	N
16	5	N
0	4	N
13	4	N
17	4	N
18	4	N
3	3	N
9	3	N
14	3	N
15	3	N
19	3	N
21	3	N
22	3	N
23	3	N
4	2	N
20	2	N
24	2	N
6	-	Y
1	-	Y
5	-	Y
7	-	Y
11	-	Y

Figure 4.7: Coarsening by sorting an information vector

To effect this, the idea of an intelligent data structure updated with coarsening information as coarsening proceeded was extended. Instead of a vector containing all nodes' information, a different structure was proposed. In this last approach, a vector of discrete tiers of cardinality are constructed, with each containing the corresponding nodes which have that cardinality, as illustrated in Figure 4.9 and Listing 4.6. The first node in the highest tier is immediately available and is chosen as a coarse node. When coarsening assigns specific nodes to coarse and fine sets, these nodes are removed from their tiers. Only nodes affected by changes in cardinality are moved between tiers. This algorithm marks a complete departure from the original AMGLab approach, in that the whole set of unsorted nodes is never searched through. The cost of this increased efficiency is an extra set of coarsening variables that must be stored for each node. The resulting computational performance is of $O(N)$ as shown in Figure 4.8.

```
{
  //Cardinality tier coarsening
  coarse_set = (empty set);
  remain_fine_set = (empty set);
  cardinality_tiers;     //Vector containing each tier and nodes in that
      tier
  (assign nodes their initial position in tiers)
  while (all nodes not sorted)
  {
    (add first node in highest tier of cardinality_tiers to coarse_set)
    (remove node from it tier)
    (add coarse node's strong influenced nodes to remain_fine_set, remove
         from their tiers)
    (move affected nodes to higher tiers in cardinality_tiers)
  }
};
```

Listing 4.6: Details of cardinality tier coarsening algorithm

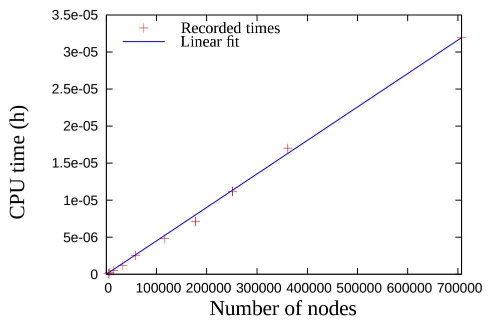
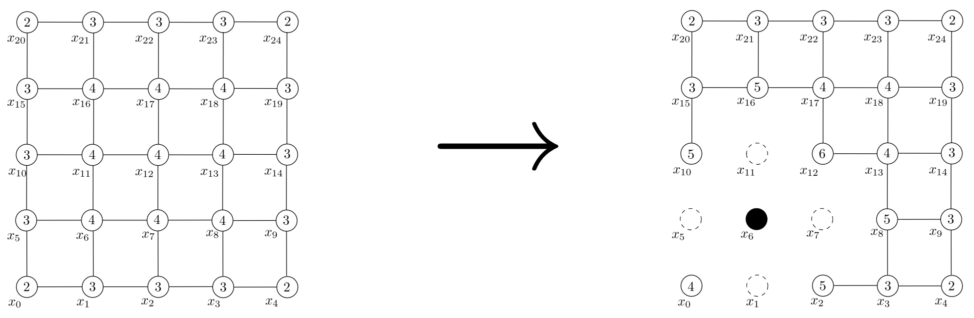


Figure 4.8: Colouring algorithm scaling


2 3 3 3 2
x_{20} x_{21} x_{22} x_{23} x_{24}
3 4 4 4 3
x_{15} x_{16} x_{17} x_{18} x_{19}
3 4 4 4 3
x_{10} x_{11} x_{12} x_{13} x_{14}
3 4 4 4 3
x_5 x_6 x_7 x_8 x_9
2 3 3 3 2
x_0 x_1 x_2 x_3 x_4
2 3 3 3 2
x_{20} x_{21} x_{22} x_{23} x_{24}
3 5 4 4 3
x_{15} x_{16} x_{17} x_{18} x_{19}
5 6 4 3
x_{10} x_{11} x_{12} x_{13} x_{14}
5 3
x_5 x_6 x_7 x_8 x_9
4 5 3 2
x_0 x_1 x_2 x_3 x_4

Cardinality Tiers

Tier	Nodes in tier
2	0, 4, 20, 24
3	1, 2, 3, 5, 9, 10, 14, 15, 19, 21, 22, 23
4	6, 7, 8, 11, 12, 13, 16, 17, 18

Cardinality Tiers

Tier	Nodes in tier
2	4, 20, 24
3	3, 9, 14, 15, 19, 21, 22, 23
4	0, 13, 17, 18
5	2, 8, 10, 16
6	12

Figure 4.9: Coarsening by using cardinality tiers

The Coarsening Procedure : Secondary Coarsening

With the initial coarsening phase complete, attention could be turned to secondary coarsening. Since only nodes strongly influenced by other non-coarse nodes are the focus of secondary coarsening, knowledge of each node's neighbourhood, as defined in Section 3.3.2, was first required. Recalling that the neighbourhood is simply a partitioning of the coefficients in a row i, a minor addition to the algorithm determining the strong influences of nodes was made. This saw the creation of an additional set for each node, namely W_i, in which all nodes *weakly influencing* a node i is recorded. With each node now having the sets S_i^{from}, S_i^{on} and W_i, as well as a list of coarse nodes resulting from the colouring procedure available, it was possible to partition every non-coarse node's neighbourhood.

Using this neighbourhood information, nodes that had strong influences from non-coarse nodes could be examined individually according to the secondary coarsening algorithm described in Section 3.3.1. This involved simply looping through each node's set of strong fine influences twice and comparing it to its set of strong coarse influences. Since the sets are limited to the neighbourhood size at most (in other words the bandwidth of the coefficient matrix), these searches were computationally inexpensive. In order to confirm that this approach was valid, the algorithm was again evaluated over multiple problem sizes, as shown in Figure 4.10, obtaining a clear linear scaling.

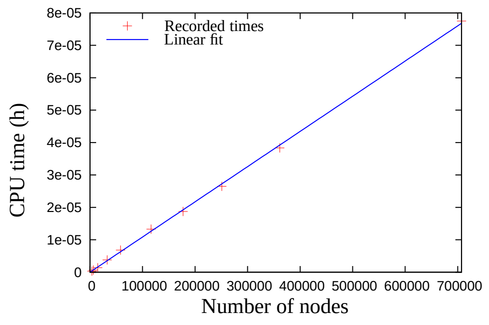


Figure 4.10: Secondary coarsening algorithm scaling

4.6.2 Intergrid Transfer Operators

Once the coarse grid selection algorithm had been completed, the next step was to transcribe the intergrid transfer operators in Section 3.3.2. The construction of the restriction and interpolation operators required the implementation of Equation (3.5) and (3.11). This consists of looping over all nodes in a level and assigning the appropriate coefficient to the interpolation operator. Restriction operators are obtained by a transpose algorithm applied to the interpolation matrix. If a node i was marked as coarse, a coefficient of unity is assigned to the interpolation operator matrix. On the other hand, if i is not coarse, Equation (3.11) must be used to calculate one or more appropriate coefficients.

Cursory inspection of the equation reveals that determining a coefficient ω requires knowledge of the neighbourhood of i. The neighbourhoods of all nodes are also used in the secondary coarsening algorithm described previously, and are thus available for use, suitably updated to reflect the additional coarse nodes. With this information available, constructing the interpolation coefficients of each non-coarse node is then simply a matter of looping over each of these node's neighbourhood sets and applying Equation (3.11). Once the interpolation matrix was completed, its transpose could be calculated and assigned as the restriction operator. The $O(N)$ cost of the intergrid transfer operator construction procedure is confirmed in Figure 4.11.

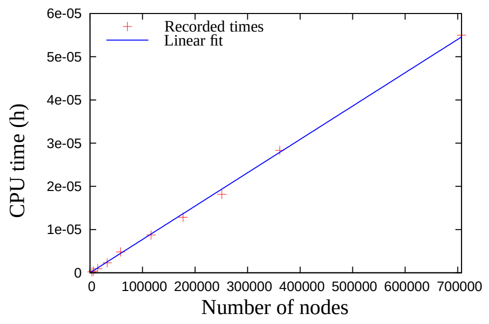


Figure 4.11: Transfer operator construction algorithm scaling

4.6.3 Coarse Grid Operators

The final step during AMG setup is to define the next coarser level's coefficient matrix. As stated in Section 3.3.3, this matrix is obtained using the Galerkin principle, which involves the triple product of Equation (3.13):

$$\mathbf{A}_{2h} = \mathbf{I}_h^{2h}\mathbf{A}_h\mathbf{I}_{2h}^h,$$

with nomenclature as defined previously. For the sake of programming simplicity, this triple product may be split into two separate matrix products to which the same algorithm is applied. First, as in Equation (4.1) an intermediate product is obtained from the system and interpolation matrix multiplication. This product is then multiplied with the restriction matrix to obtain the next level system matrix as in Equation (4.2):

$$\textbf{Intermediate} = \mathbf{A}_h\mathbf{I}_{2h}^h \tag{4.1}$$

$$\mathbf{A}_{2h} = \mathbf{I}_h^{2h}\textbf{Intermediate} \tag{4.2}$$

Since no matrix multiplication tools are available as standard in C++, and due to the compressed row sparse storage used, a specialised algorithm was needed, whether custom coded or obtained from an external library. The latter was considered the simplest option, and so an external C++ library offering linear algebra functions was evaluated. Specifically, the BOOST library which offers basic linear algebra subprograms (BLAS), was investigated. Performance evaluation of this library when using the compressed row sparse matrix

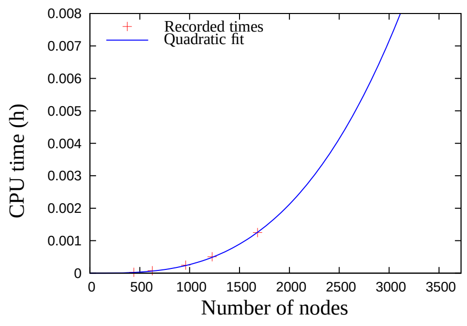


Figure 4.12: BOOST sparse matrix multiplication scaling

storage scheme described previously was not satisfactory however. A performance evaluation of a sparse matrix multiplication in this library is shown in Figure 4.12. In light of this, it was decided to implement a custom high performance sparse multiplication algorithm inside the AMG solver itself, which would receive two sparse matrices **A** and **B** as input and return their product, **AB**. As a starting point, the classic dense matrix multiplication algorithm is recalled. In this algorithm, the resultant product is constructed one coefficient at a time according to

$$AB_{i,j} = \sum_{k=0}^{N-1} A_{i,k} B_{k,j}, \tag{4.3}$$

with nomenclature as defined before. In practice, an implementation of this algorithm is of $O(N^3)$ complexity when A and B are dense square matrices. Applying the same algorithm to an extremely sparse matrix lowers the complexity to $O(nnz)$, where, as before, nnz is the number of non-zero matrix entries in the matrices being multiplied. For illustration purposes, and for comparison to the algorithm developed later in this section, this approach is shown for a simple multiplication, as seen in Figure 4.13.

$$\begin{bmatrix} \textcircled{1} & \textcircled{3} & \textcircled{9} \\ 3 & 5 & 6 \\ 6 & 7 & 2 \end{bmatrix} \times \begin{bmatrix} \textcircled{8} & 3 & 7 \\ \textcircled{5} & 1 & 9 \\ \textcircled{6} & 6 & 2 \end{bmatrix} = \begin{bmatrix} \textcircled{77} & & \\ & & \\ & & \end{bmatrix}$$

Figure 4.13: Traditional dense matrix multiplication

In this figure, the multiplication to obtain the first coefficient in **AB**, $AB_{1,1}$, is diagrammed. As shown, the coloured numbers are the coefficients required to construct the first coefficient of **AB**. Like coloured numbers are multiplied, then added to give the final coefficient in the result. In practice, this necessitates accessing $A_{1,1}$, multiplying it with

$B_{1,1}$, and storing it in $AB_{1,1}$. Access then shifts to $A_{1,2}$, $B_{2,1}$, $A_{1,3}$ and $B_{3,1}$ in turn. This approach takes for granted that all coefficient indices are easily available and accessed, which is *not* the case when a format such as compressed row sparse storage is used. Specifically, compressed row sparse storage contains matrices *row-by-row*, meaning that specific column indices are not immediately available. The continuous access of different rows and columns in the matrices necessitate costly searches for specific indices in this case. Clearly, this is far from ideal.

Instead, a different approach was implemented, which takes advantage of the row-by-row storage format. In a reversal of the traditional implementation of Equation 4.3, where the result's coefficients are constructed one by one, looking up specific rows and columns in the matrices **A** and **B** as needed, the implemented algorithm instead accesses **A** and **B** one row at a time and performs all multiplications involving that row before moving to the next. This has the effect of constructing multiple coefficients in the resultant matrix *simultaneously*, and takes advantage of the immediately available row indices compressed row sparse storage format offers. Figure 4.14 illustrates this multiplication approach schematically.

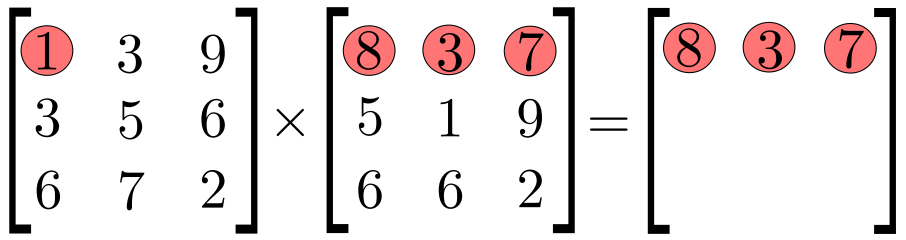


Figure 4.14: Row by row multiplication

As before, like coloured numbers are multiplied. However, as opposed to Figure 4.13, where components across multiple rows are accessed to construct a specific coefficient, all calculations take place within one row, with contributions being added to multiple coefficients in the result. Since only one row is ever being accessed at a time, no searches in the compressed row sparse storage vectors need to be performed. Using this efficient method, a complexity of $O(nnz)$ was achieved, where nnz is the number of non-zero components in the matrices being multiplied. This number is proportional to N and the size of the discretisation stencil used, and as such, the sparse multiplication is of $O(N)$, as shown in Figure 4.15.

4.7 The AMG Solution Routine

As shown in Figure 4.2, obtaining a solution estimate to a given problem is performed in a separate routine. This routine is performed multiple times, depending on the accuracy required from the solution estimate. As seen in Section 3.3.4, the two grid V-cycle in Figure 3.12 can be extended recursively over multiple levels in order to provide a solution cycle. This is the approached adopted in the AMG implementation. A representation of this recursive cycle is given in Figure 4.16, with pseudocode in Listing 4.7. Obviously, the solution cycle implemented has multiple instances of smoothing, restriction and interpolation. These procedures involve only very simple matrix/vector products, and are thus trivial to implement optimally. Scaling of the solution routine is provided in Figure 4.17.

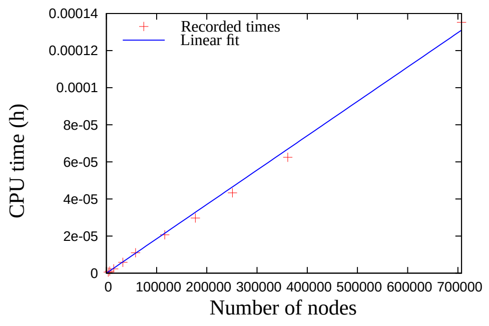


Figure 4.15: Matrix multiplication algorithm scaling

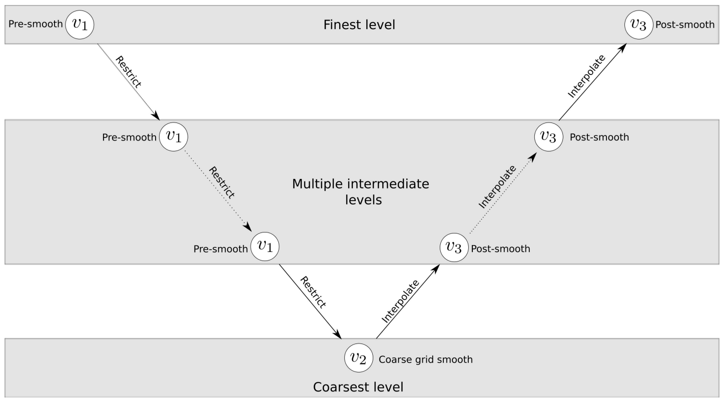


Figure 4.16: Recursive v-cycle

```
{
  //AMG setup
  (create N levels of coarsening)

void v_cycle(current_level) //Recursive cycle function
    int v1, v2, v3; //Number of iterations to use for smoothing

    current_level = 0;     //Start on the finest level

    if (current_level is NOT the coarsest)
    {
      (perform v1 Gauss-Seidel operations on level)
      (restrict solution estimate to current_level+1)
      v_cycle(current_level+1);    //Recursive call

      (interpolate error from current_level+1)
      (correct error estimate on this level)
      (perform v3 Gauss-Seidel operations on this level)
    }
    elseif (current_level IS the coarsest)
    {
      (perform v2 Gauss-Seidel operations on level)
    }
  }

};
```

Listing 4.7: Recursive V-cycle

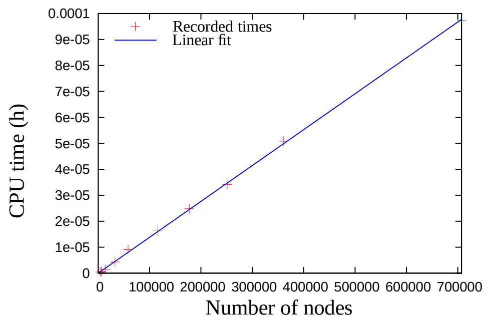


Figure 4.17: V-cycle routine scaling

4.8 Freeze-AMG

As described in Section 3.5, the Freeze-AMG method utilises the AMG setup of a single time step in order to obtain solution estimates for multiple subsequent time steps. Owing to the creation of the separate AMG object in Figure 4.2, a given solution structure would exist in memory and remain accessible until the setup routine was performed again. Since the setup and solution routines independently function using each level's object, it is possible to intelligently perform the setup routine only when needed. Furthermore, it is possible to only perform *parts* of a setup routine, as required by the proposed extension of Freeze-AMG, where only transfer and operator matrices are updated. In this work, a very basic heuristic is used to determine when the setup routine is performed, namely placing an empirical limit on the number of solution cycles allowable. An outline of this process is shown in Figure 4.18. An evaluation of this methodology follows in Chapter 5, in which the characteristics of the extended and basic Freeze-AMG are quantified.

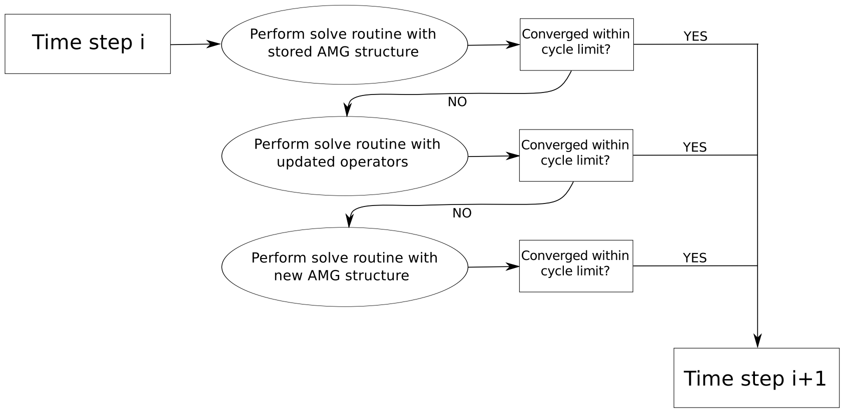

Time step i
Perform solve routine with stored AMG structure
Converged within cycle limit?
YES
NO
Perform solve routine with updated operators
Converged within cycle limit?
YES
NO
Perform solve routine with new AMG structure
Converged within cycle limit?
YES
Time step i+1

Figure 4.18: The Freeze-AMG solution method

4.9 Summary

This chapter started with justification for the C++ object-oriented approach chosen for the AMG solver, followed by details the algorithms required to implement an algebraic multi-grid method, as specified in Chapter 3. Attention was given to the data structures used, and the evolution of algorithms to achieve optimal performance scaling. Specifically the influence, coarsening and operator multiplication algorithms were explored in more detail. Finally, the manner in which the recursive V-cycle was implemented was illustrated, as well as how the Freeze-AMG methodology was used to limit the amount of full coarsenings performed. Implementation of the preceding algorithms are discussed in the next chapter.

Chapter 5

Solver Testing and Evaluation

5.1 Introduction

The previous chapters detailed the development of an algebraic multigrid tool intended to accelerate the solution of the pressure correction equation arising from free surface modelling simulations. This chapter is concerned with verifying the accuracy and multigrid performance of the implemented solver, as well as the performance increases offered over the existing solver on representative problems. In order to evaluate the performance of the solver in terms of efficiency and computational speed improvements, it was applied to two benchmark FSM problems, namely a two-dimensional sloshing tank and dam-break. These were selected as they represent both smooth and violent interface problems, thus serving as a thorough evaluation for the robustness and efficiency of the developed F-AMG method.

In the interests of a rigorous evaluation, each problem was solved on multiple meshes spanning at least an order of magnitude in number of unknowns, and actual CPU times of four solvers compared *viz.* C-AMG, F-AMG, extended F-AMG and preconditioned GMRES. Solution time scaling as a function of mesh size of the four solvers was evaluated and the contribution of the pressure correction solution to the total simulation time compared. Further, both structured and unstructured meshes were employed, with the following being adhered to for all simulations:

- The pressure equation was considered solved only once the scaled residual had been reduced by five orders of magnitude (reducing this to three orders was found to not have a significant effect on CPU times),
- In the interests of automation, the AMG solver was allowed to construct coarse levels until the coarsest level contained one unknown only, with a coarsening threshold (θ) of 0.8,
- The V-cycle (as described in Section 3.3.4) consisted of 2 pre-, 4 post- and 1 coarse-level smoothing iterations,
- The number of V-cycles allowed before a solution structure was deemed inappropriate (ie. re-coarsening was required) was set to fifty,
- All simulations were performed on a Dell Latitude E6510 computer, with dual 2.66GHz Intel Core i7 CPU (4Mb of cache memory each) and 4Gb of 1066Mhz RAM .

In addition to the performance comparison of the implemented solver, it was verified that problems were accurately solved by comparing computed results to experimental or benchmark predictions, and that the expected multigrid convergence behaviour had been obtained. Finally, the performance of the F-AMG methods was evaluated. These topics are the focus of the remainder of this chapter.

5.2 Two-Dimensional Fluid Sloshing

The first test case used to evaluate the solver was the simple side to side slosh in two dimensions of a wave with low amplitude [33]. The computational domain is depicted in Figure 5.1, with a width of 0.1m and a height of 0.065m. The bottom and sides of the domain are slip boundaries, while a fixed pressure condition is applied at the top surface. The initial fluid surface is defined by half a cosine wave with amplitude as shown, and sloshes under the influence of gravity. In order to gauge performance of the F-AMG solver, the motion of the fluid was simulated for ten seconds (twenty-six slosh cycles) using the original preconditioned GMRES solver used by *Elemental*, the basic C-AMG solver, the initial F-AMG implementation and the extended F-AMG method described in Section 3.5. Performance evaluations were performed on both structured and unstructured grids of multiple sizes.

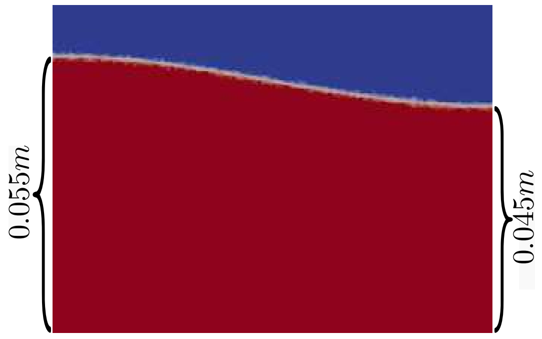


Figure 5.1: Two-dimensional fluid sloshing - Computational domain

To assess correct solution, the implemented AMG solver was applied to a structured mesh containing 17,161 nodes (Figure 5.4). The computed results for the interface height at the left-hand boundary as a function of time compare well to the results of others [32], as shown in Figure 5.2. Next, multigrid speed-up was assessed using the same mesh. The improved convergence achieved via the basic C-AMG solver is depicted in Figure 5.3. As seen, the multigrid solution method achieves a speed-up of multiple orders of magnitude, proving the implementation is sound.

5.2.1 Solver Comparison

In order to rigorously gauge the performance of the implemented solver, multiple simulations of the sloshing problem were performed on differing mesh types using the various matrix free solvers. Examples of these meshes are depicted in Figure 5.4. As shown in Figure 5.5, it was found that solution times scaled linearly with the length of the simulation

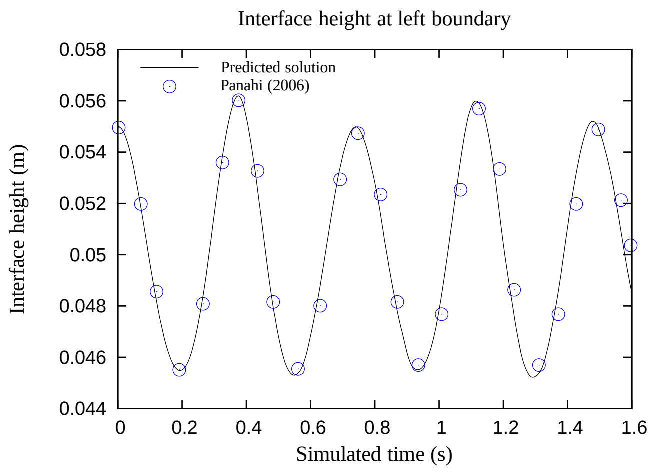


Figure 5.2: Two-dimensional sloshing - Experimental comparison

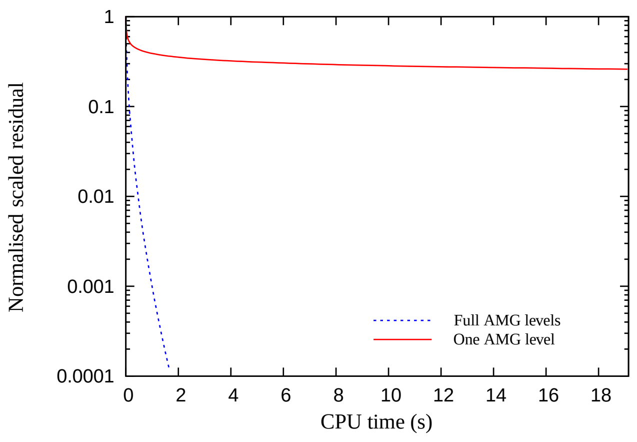


Figure 5.3: Convergence speed-up achieved due to the C-AMG solver

over all mesh sizes for both structured and unstructured meshes. This is thought to be due to the small interface velocity range and smoothness, resulting in uniform real time step sizes throughout.

What is clear from the solver comparison is that the AMG methods outperform the preconditioned GMRES solver, with the F-AMG implementations offering the greatest speed-ups. The latter outperforms GMRES by a large factor on both structured and unstructured problems. The greater amount of time spent on the unstructured problems can be ascribed to the larger discretisation stencil resulting from increased node connectivity. However, it is surprising to note that the two F-AMG methods were almost identical in performance,

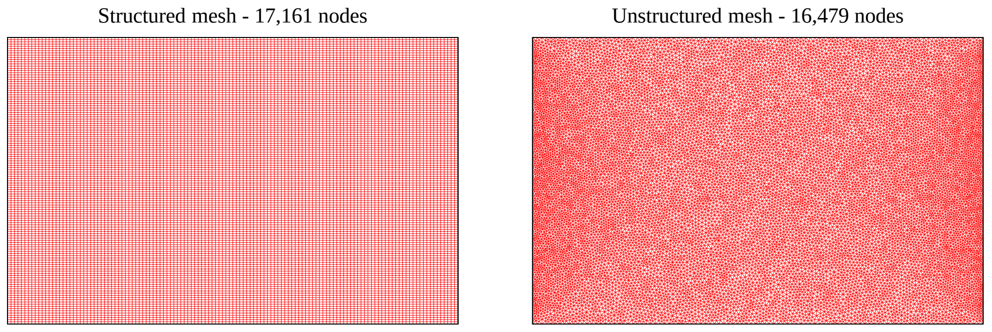

Structured mesh - 17,161 nodes
Unstructured mesh - 16,479 nodes

Figure 5.4: Two-dimensional sloshing - Meshes

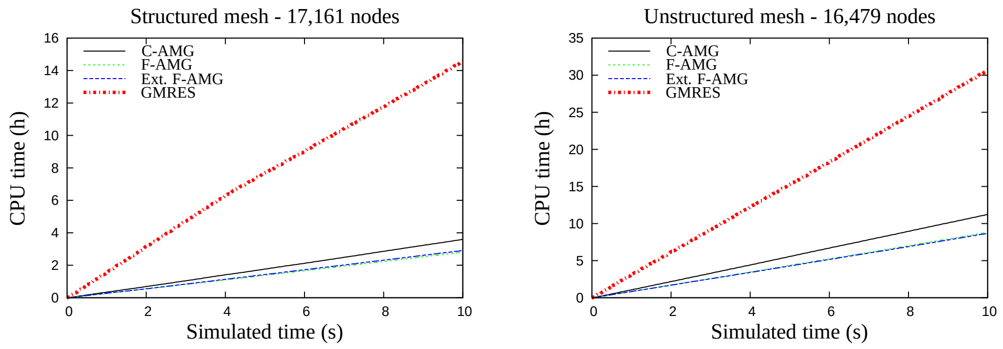


Figure 5.5: Two-dimensional sloshing - Solver comparisons

with no discernible advantage offered by the extended methodology over the basic Freeze method. In addition, the Freeze methods offered only marginal improvements over C-AMG for this problem. Reasons for this are explored in detail in Section 5.4.

Next, solver performance was assessed as a function of mesh size. For this purpose, the four solvers were applied to structured meshes ranging over two orders of magnitude. Since the CPU time scales linearly with the simulation time for this problem, a shortened, one second simulation was performed. The results are summarised in Figures 5.6 and 5.7, with the latter depicting the percentage of overall CPU time used by each pressure solver. As shown, two distinct scaling regimes were observed *viz.* in and out out of cache, with the former offering significantly superior performance.

From the scaling investigation, it is clear that the AMG methods exhibit optimal $O(N)$ scaling with mesh size, with in cache being $1.85^{-5}N$ and out of cache $6.67^{-5}N$. The F-AMG methods continued to be almost identical. This is in contrast to the GMRES method, which exhibits clear non-linear behaviour (this is unsurprising as GMRES scales as $O(N \log N)$). The upshot of this is that as problem size increases, F-AMG specifically outperforms GMRES by a greater and greater amount. Of particular interest is the fact that in the AMG methods, a much lower percentage of CPU time is devoted to the solution of the pressure correction equation, as opposed to the GMRES method in which almost all the time taken was dedicated to this process. This indicates that significant relief of the

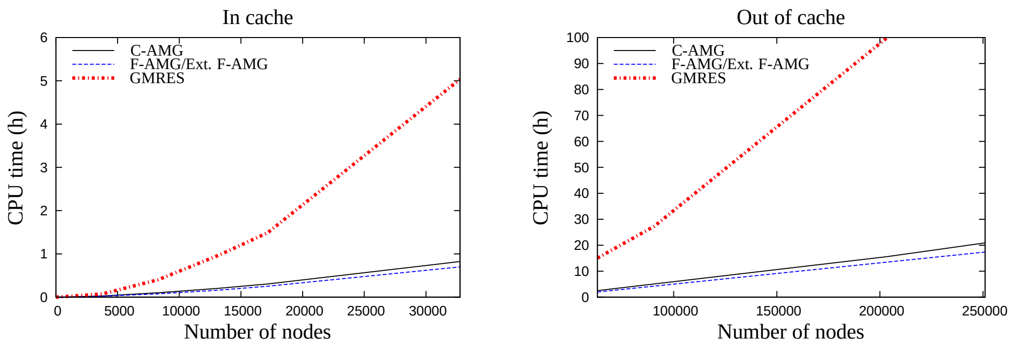


Figure 5.6: Two-dimensional sloshing - Solver scaling

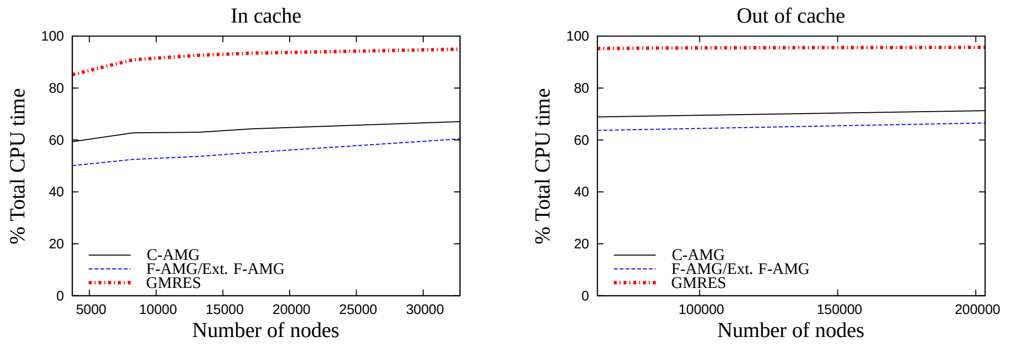


Figure 5.7: Two-dimensional sloshing - Solver time contribution

bottleneck associated with the pressure correction equation has been achieved.

5.3 Two-Dimensional Dam-break

The second problem on which solver performance was evaluated was the canonical two-dimensional dam-break problem [29]. In this problem, a column of water of width $a = 0.146m$ and height $2a$ is allowed to flow under the influence of gravity, as shown in Figure 5.8. All four boundaries are considered no-slip boundaries. This problem exhibits violent interface motion, as seen in the snapshots of the evolving interface, and as such offers the opportunity to test the robustness of the developed AMG setup methodologies. This is of special interest since the matrix coefficients in Equation (2.20) will undergo large changes between time steps as the interface sweeps over the entire domain. Following this, a two second simulation interval was modelled as it contains the time period of the problem characterised by a violent, fast-moving interface. Subsequent to this, the interface settles down into a quasi-static mode which requires little solver effort and is thus of limited interest.

As with the previous test case, verification of the solver was performed by comparing computed results to experimental [29] and published results. These comparisons are shown in Figure 5.9, in which the solution obtained on a structured mesh containing 3,721 nodes

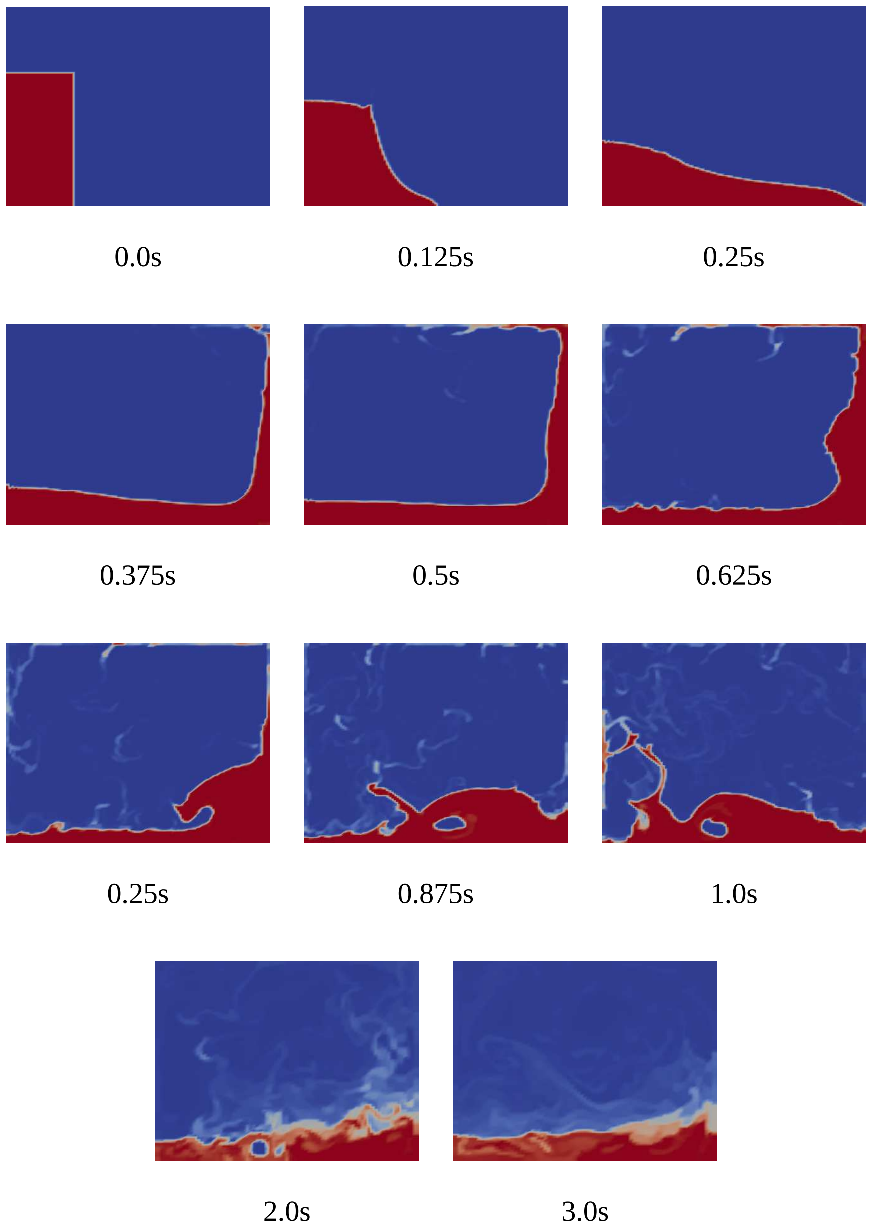


Figure 5.8: Two-dimensional dam-break - Interface evolution

is depicted. As seen, the simulated values show good agreement with experiment and literature, which verifies that the implemented method is solving the problem correctly.

5.3.1 Solver Comparison

As in Section 5.2, multiple simulations across various mesh sizes were performed in order to gauge solver performance. In this case, structured meshes ranging from 3,721 to 32,761

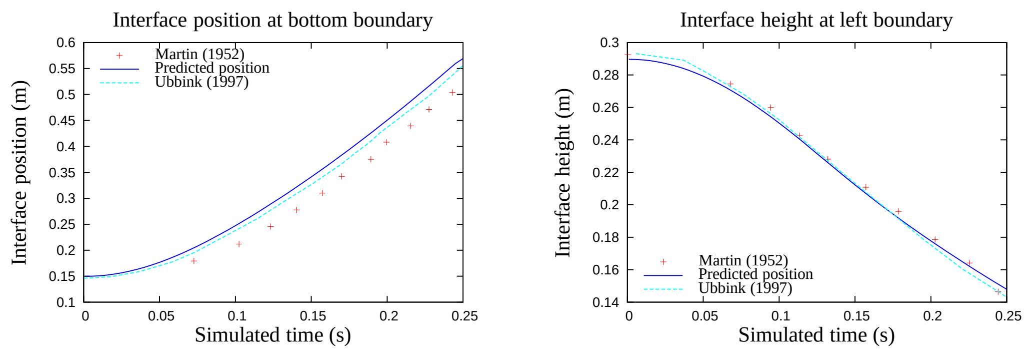


Figure 5.9: Two-dimensional dam-break - Solution verification

nodes were employed, an example of which is depicted in Figure 5.10. The mesh sizes were all chosen to be below the caching limit observed in the previous test case. Solver performance on the meshes investigated showed similar behaviour, as depicted in the representative case in Figure 5.11, in which the solver comparison for a simulation on the finest mesh employed appears.

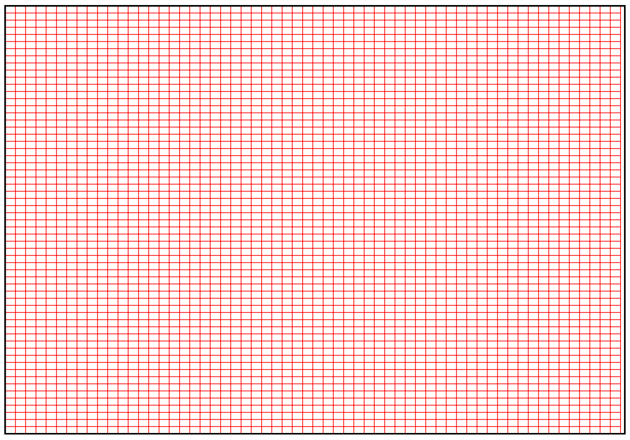


Figure 5.10: Two-dimensional dam-break : Structured mesh

Similar to the results obtained from the sloshing test case, barring the uniform real time step size, the AMG methods again outperform the GMRES solver, with marginal differences between the F-AMG methods. Considering first CPU cost as a function of time simulated, GMRES exhibits drastically slower performance than the AMG methods, which show fairly consistent performance over the entire time period. The aforementioned also suffers from higher CPU cost at 0.3s, which coincides with the interface reaching the opposite wall, as shown in Figure 5.8, where interface motion starts to become much more rapid. While it is unknown why the GMRES method would have difficulty with this situation, it is interesting

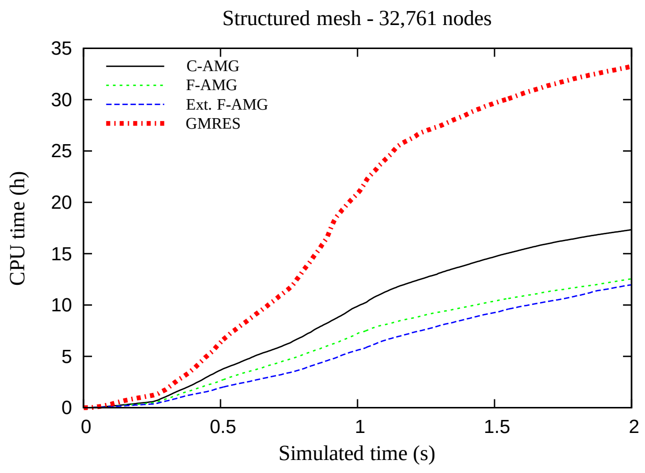


Figure 5.11: Two-dimensional dam-break - Solver comparisons

to note that the F-AMG methods do not exhibit this problem.

Plotting the solver times as the mesh sizes increased yielded the scaling in Figure 5.12. The scaling data again shows the AMG methods exhibiting superior scaling to the GMRES solver, approaching optimal $O(N)$ behaviour, with the latter exhibiting definite non-linear scaling. Thus, as problem sizes increase, the factor by which solution times are reduced increases rapidly.

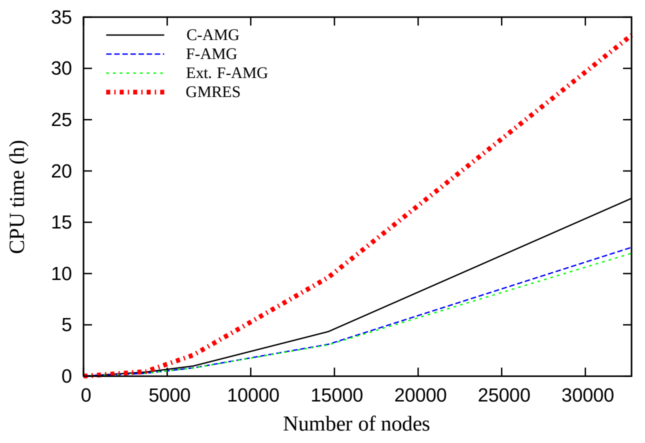


Figure 5.12: Two-dimensional dam-break - Solver scaling

5.4 F-AMG Performance Analysis

As seen in the previous sections, the basic and extended Freeze-AMG methods exhibit almost exactly the same performance on the cases evaluated. Since only updating interpolation operators consumes roughly half as much computational time as a full setup procedure, this ran contrary to expectations. In addition, both F-AMG methods offer lower than expected improvements over the C-AMG solver. In order to investigate possible causes of this behaviour, a detailed breakdown of the time spent during the solution of the pressure correction equation was performed. This data is shown in Table 5.1 for the coarsest meshes on both test cases.

From this information, it becomes clear why the Freeze methods exhibit the observed behaviour. As shown, the Freeze methods perform vastly more V-cycles during the course of a simulation than the basic C-AMG method, while spending much less time on setup. As expected, the extended Freeze method spends the least amount of time on setup, but accordingly more time on V-cycles. The result of this is that the majority of CPU time spent on the solution of the pressure equation becomes devoted to the solution cycle, whereas the setup time consumes much less time. This increase in the total number of solution cycles required is a direct result of the fact that frozen solution structures do not converge in as few cycles as a structure constructed at every time-step, which also accounts for the discrepancy between the extended and basic Freeze methods.

Solution Time Breakdown								
Method	Total CPU time	Pressure eqn.		Sol. cycle			AMG setup	
	(h)	(h)	%	(h)	%	Total cycles	(h)	%
Two-dimensional sloshing, structured								
Ext. F-AMG	0.327	0.124	37.84	0.108	32.96	288392	0.016	4.87
Basic F-AMG	0.315	0.115	36.49	0.097	30.83	259374	0.0178	5.66
C-AMG	0.379	0.174	45.98	0.048	12.68	126378	0.126	33.3
GMRES	0.875	0.675	77.19	n/a	n/a	n/a	n/a	n/a
Two-dimensional sloshing, unstructured								
Ext. F-AMG	0.893	0.384	42.97	0.297	33.24	460789	0.087	9.73
Basic F-AMG	0.839	0.373	44.45	0.287	34.16	446651	0.086	10.29
C-AMG	1.112	0.624	56.14	0.215	19.37	329940	0.410	36.78
GMRES	2.142	1.666	77.77	n/a	n/a	n/a	n/a	n/a
Two-dimensional dam-break, structured								
Ext. F-AMG	0.458	0.241	53.85	0.220	49.11	540635	0.021	4.75
Basic F-AMG	0.448	0.239	53.24	0.218	48.57	540299	0.021	4.67
C-AMG	0.570	0.367	63.99	0.174	29.85	416866	0.194	34.14
GMRES	0.582	0.374	64.29	n/a	n/a	n/a	n/a	n/a

Table 5.1: Breakdown of the pressure equation solution time

It is proposed that the above limited performance improvement due to the freeze mechanism can be ascribed to the extremely basic heuristic used in its implementation, namely a fixed number of allowable V-cycles before re-setup occurs. Since the typical number of

V-cycles required for a solution to converge is problem specific, using a hard limit of fifty cycles may not be ideal. Instead, an adaptive approach which records convergence information as the simulation proceeds could conceivably improve solution times dramatically by finding the optimum balance between setup and solution times. Barring this, it is clear that the additional complexity of the extended F-AMG method offers no real benefits. It is clear though that the solution of the pressure equation consumes a much lower percentage of total CPU time when using the F-AMG methods than the GMRES solver, confirming the behaviour observed in Figure 5.7. This highlights the fact that the bottleneck associated with the solution of the pressure correction equation has been greatly reduced. Additional improvements in solution time can be realised by identifying further bottlenecks in the *Elemental* package, which is beyond the scope of this work.

5.5 Summary

The developed solver tool has been evaluated by applying it to smooth and violent interface test-cases on a range of mesh sizes. Having validated accurate solution via comparing predicted solutions to available benchmark solutions, the performance of the developed Freeze-AMG solvers was evaluated by comparing CPU times for simulations to those of preconditioned GMRES and C-AMG solvers. Additionally, the scaling of the solvers with problem size was investigated. These evaluations showed that the implemented C-AMG and F-AMG were consistently faster than the preconditioned GMRES method, while exhibiting optimal $O(N)$ behaviour. The basic and extended F-AMG methods offered the best solution times, but were almost indistinguishable from one another. Additional investigation into this phenomenon determined that the F-AMG methods spend little time on solver setup (as expected) but that increased CPU time is spent on solution cycles. A proposal to remedy this is to introduce adaptive heuristics for the Freeze methods instead of the hard limit on solution cycles currently imposed. A summary of the contributions of this work, as well as recommendations for further research is provided in the next chapter.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

The aim of the current work was to implement and test a solution acceleration method for solving large, sparse systems of linear equations, which arise during the pressure correction step in free-surface modelling simulations. The chosen acceleration method was to be designed as a plug-in to an existing CFD package *Elemental*, which currently uses a preconditioned GMRES method to solve this equation set. After investigation, the decision was made to examine the suitability of an algebraic multigrid method due to both the optimal performance scaling exhibited by multigrid methods as well as its suitability to be used as a black box plug-in solver. The partitioned solution method employed by *Elemental* results in a changing coefficient matrix for the pressure correction equation at each simulation time step. Due to the fact that AMG methods use this matrix as input in order to construct a solution structure, the setup phase of the AMG method would have to be performed repeatedly. During implementation, specific attention would be given to a method to mitigate these costs.

After developing the governing equation set and the background of the pressure correction equation, an overview of the multigrid method as advanced solver was given, with focus on Classic Algebraic Multigrid. In this way, the algorithmic components required of a multigrid solver were detailed, ready for implementation. Additionally, in anticipation of the aforementioned cost of repeated AMG setup, an augmented methodology was introduced, termed Freeze-AMG. The method relies on maintaining solution structures over multiple simulation time steps until solver convergence decays. Underlying the new method is the observation that frozen solution structures remain adequate approximations to the system of equations due to the fact that only minor changes to the coefficient matrix occurs over several time steps. An extension of this freezing concept was also developed, in which only partial setup of the solution structure was performed in an effort to maintain solver performance for as little computational cost as possible.

The AMG solver in its entirety was developed in object-oriented C++. Special focus was given to obtaining optimal complexity for each individual algorithm. This resulted in multiple implementation refinements. With the solver successfully implemented, a rigorous evaluation was performed. Both smooth and violent interface problems were evaluated on multiple mesh sizes, *viz.* two-dimensional fluid sloshing and dam-breaks.

The developed Freeze-AMG solvers were evaluated by assessing correct solution and computational speed-ups, as well as comparing computational cost to an existing preconditioned GMRES solver and C-AMG. For all problems run and meshes used, the F-AMG solvers proved robust and accurate, while exhibiting the best performance and optimal $O(N)$ scaling. In addition, the Freeze methods reduced the percentage of computational time required to solve the pressure correction equation from 95% for GMRES to 65% while offering an improvement over C-AMG of 5%. Further, the CPU time spent on AMG setup while using the Freeze methods is circa 5 to 10 times lower than the time spent on solution, as opposed to basic C-AMG method, where the time spent on the two are comparable.

The similar CPU cost of the two developed Freeze methods points to the need for an adaptive heuristic which balances the time spent on setup and solution cycles. While beyond the scope of this work, this development may result in a clear distinction between the extended and basic Freeze methods, while offering even greater improvements over the C-AMG solution times. Nevertheless, from the obtained results, it is felt that both Freeze-AMG methods are a promising extension of the basic AMG method in the FSM context, offering a viable alternative as a high-performance linear solver, with great potential for further development.

6.2 Future Work

The scope of this work was limited to a serial implementation of the C-AMG method on limited problem sizes, using only basic heuristics for the F-AMG method. Obviously, in the context of relevant industrial applications, a massively parallel implementation is indispensable. Classical AMG coarsening, which forms the basis of this work, has in the past been considered ill-suited to parallelisation [11], as well as being superceded by more advanced AMG methods [9]. In addition, trends in AMG research have pointed to the use of the method as a preconditioner for GMRES instead of a stand-alone solver as offering the best possible performance. Considering these factors, the following areas are recommended as extensions for this research:

- Implementation and testing of a parallel Freeze-AMG solver
- Automatic adaptive heuristic to determine when to freeze and unfreeze solution structures
- Investigation of more modern AMG algorithms to replace the classical model in use
- Performance comparison of the stand-alone AMG solver and an AMG preconditioned GMRES solver.

References

[1] S. Aliabadi, A. Johnson, and J. Abedi. Comparison of finite element and pendulum models for simulation of sloshing. *Computers & Fluids*, 32(4):535–545, 2003.

[2] R. Aubry, F. Mut, R. Löhner, and J.R. Cebral. Deflated conjugate gradient solvers for the Pressure-Poisson equation. *Journal of Computational Physics*, 227:10196–10208, 2008.

[3] W. N. Bell, L. N. Olson, and J. Schroder. PyAMG: Algebraic multigrid solvers in Python v1.0, 2008. Release 1.0.

[4] W.L. Briggs, V.E. Henson, and S.F. McCormick. *A Multigrid Tutorial*. SIAM, Society for Industrial and Applied Mathematics, 3600, University City Science Center, Philadelphia, PA, Second edition, 2000.

[5] J. R. Cary, S. G. Shasharina, J. C. Cummings, J. V. W. Reynders, and P. J. Hinker. Comparison of C++ and Fortran 90 for object-oriented scientific programming. *Computer Physics Communications*, 105:20–36, 1997.

[6] K. Craig and T. Kingsley. Design optimization of containers for sloshing and impact. *Structural and Multidisciplinary Optimization*, 33(1):71–87, 2007.

[7] J. T. Cross, I. Masters, and R. W. Lewis. Why you should consider object-oriented programming techniques for finite element methods. *International Journal for Numerical Methods in Heat and Fluid Flow*, 9:333–347, 1999.

[8] R.L. Detwiler, S. Mehl, H. Rajaram, and W.W. Cheung. Comparison of an algebraic multigrid algorithm to two iterative solvers used for modeling ground water flow and transport. *Ground Water*, 40(3):267–272, 2002.

[9] R.D. Falgout. An Introduction to Algebraic Multigrid. *Computing in Science and Engineering*, 8(6):24–33, 2006.

[10] R.D. Falgout and U. Yang. *Hypre* : A library of high performance preconditioners. In P. Sloot, A. Hoekstra, C. Tan, and J. Dongarra, editors, *Computational Science ICCS 2002*, volume 2331, pages 632–641. Springer Berlin / Heidelberg, 2002.

[11] V.E. Henson and U.M. Yang. BoomerAMG: A parallel algebraic multigrid solver and preconditioner. *Applied Numerical Mathematics*, 41:155–177, 2002.

[12] C.W. Hirt and B.D. Nichols. Volume of fluid (VOF) method for the dynamics of free boundaries. *Journal of Computational Physics*, 39:201–225, 1981.

[13] H. Jasak, A. Jemcov, and Z. Tuković. OpenFOAM: A C++ library for complex physics simulations. In *International Workshop on Coupled Methods in Numerical Dynamics*, 2007.

[14] D. Kim and P. Moin. Pressure-correction algorithm to solve Poisson systems with constant coefficients for fast two-phase simulations. *Center for Turbulence Research Annual Research Briefs*, 2009.

[15] M.B. Koçyigit, R.A. Falconer, and B. Lin. Three-dimensional numerical modelling of free surface flows with non-hydrostatic pressure. *International Journal for Numberical Methods in Fluids*, 40:1145–1162, 2002.

[16] S. Kurioka and D.R. Dowling. Numerical simulation of free surface flows with the level set method using an extremely high-order accuracy WENO advection scheme. *International Journal of Computational Fluid Dynamics*, 23(3):233–243, 2009.

[17] J.H. Kwon, A. Ecer, J. Periaux, N. Satofuka, and P. Fox. *Parallel Computational Fluid Dynamics : Parallel computing and its applications*. Elsevier, Amsterdam, 2007.

[18] J.W. Lee, M.D. Teubner, J.B. Nixon, and P.M. Gill. Development of a 3D non-hydrostatic pressure model for free surface flows. *Australian and New Zealand Industrial and Applied Mathematics Journal*, 46:C623–C636, 2005.

[19] R. Löhner. *Applied* CFD *Techniques*. John-Wiley and Sons Ltd., Chichester, 2001.

[20] R. Löhner. Projective prediction of pressure increments. *Communications in Numerical Methods in Engineering*, 21:201–207, 2005.

[21] H. Luo, J. D. Baum, and R. Löhner. A fast, matrix-free implicit method for compressible flows on unstructured grids. *Journal of Computational Physics*, (146):664–690, 1998.

[22] A. G. Malan and R. W. Lewis. Modeling coupled heat and mass transfer in drying non-hygroscopic capillary particulate materials. *Communications in Numerical Methods in Engineering*, 19(9):669–677, 2003.

[23] A. G. Malan and R. W. Lewis. On the development of high-performance C++ object-oriented code with application to an explicit edge-based fluid dynamics scheme. *Computers & Fluids*, 33:1291–1304, 2004.

[24] A. G. Malan, J. P. Meyer, and R. W. Lewis. Modelling non-linear heat conduction via a fast matrix-free implicit unstructured-hybrid algorithm. *Computer Methods in Applied Mechanics and Engineering*, 196(45-48):4495–4504, 2007.

[25] A.G. Malan and R.W. Lewis. An artificial compressibility CBS method for modelling heat transfer and fluid flow in heterogeneous porous materials. *International Journal for Numerical Methods in Engineering*, 2010.

[26] A.G. Malan and O.F. Oxtoby. A novel matrix-free hybrid-unstructured parallel free-surface-modelling methodology. *for Airbus UK (Ltd)*, 2009.

[27] A.G. Malan and O.F. Oxtoby. A parallel free-surface-modelling technology for application to aircraft fuel sloshing. In *ECCOMAS CFD - Fifth European Conference on Computational Fluid Dynamics*, 2010.

[28] Z. Mao and A.E. Dukler. An experimental study of gas-liquid slug flow. *Experiments in Fluids*, 8(3):169–182, 1989.

[29] J. C. Martin and W. J. Moyce. An experimental study of the collapse of a liquid column on a rigid horizontal plane. *Philosophical Transactions of the Royal Society of London, Series A*, (244):312–324, 1952.

[30] R. McKenzie. Amglab: An interactive testbench for learning and experimentation with algeraic multigrid methods. Master's thesis, University of Kentucky, 2006.

[31] P. Nithiarasu. An efficient artificial compressibility (AC) scheme based on the characteristic based split (CBS) method for incompressible flow. *International Journal for Numerical Methods in Engineering*, 56(13):1815–1845, 2003.

[32] R. Panahi, E. Jahanbakhsh, and M. Seif. Development of a VOF-fractional step solver for floating body motion simulation. *Applied Ocean Research*, 28:171–181, 2006.

[33] P.E. Raad. The introduction of micro cells to treat pressure in free surface flow problems. *Journal of Fluids Engineering*, 117:683–690, 1995.

[34] J.W. Ruge and K. Stüben. Algebraic multigrid. In S.F. McCormick, editor, *Multigrid Methods*, volume 3 of Frontiers in Applied Mathematics, pages 73–130, Philadelphia, 1987. SIAM.

[35] O. Soto, R. Löhner, and F. Camelli. A linelet preconditioner for incompressible flow solvers. *International Journal of Numerical Methods for Heat & Fluid Flow*, 13(1):133–147, 2003.

[36] J.J. Stoker. *Water Waves*. John Wiley & Sons, New York, 1958.

[37] L. Štrubelj, I. Tiselj, and B. Mavko. Simulations of free surface flows with implementation of surface tension and interface sharpening in the two-fluid model. *International Journal of Heat and Fluid Flow*, 30:741–750, 2009.

[38] K. Stüben. Algebraic multigrid (AMG): An introduction with applications. *GMD Reports*, (70), 1999.

[39] K. Stüben. A review of algebraic multigrid. *Journal of Computational and Applied Mathematics*, 128(1-2):281–309, 2001.

[40] K. Stüben. Multigrid methods and parallel computing. Presented at the Second International FEFLOW User Conference, Potsdam, 2009.

[41] C. Sun. *Parallel Algebraic Multigrid for the Pressure Poisson Equation in a Finite Element Navier-Stokes Solver*. PhD thesis, Rensselaer Polytechnic Institute, 2008.

[42] O. Ubbink and R. I. Issa. A method for capturing sharp fluid interfaces on arbitrary meshes. *Journal of Computational Physics*, 153:26–50, 1999.

[43] H.A. van der Vorst. Efficient and reliable iterative methods for linear systems. *Journal of Computational and Applied Mathematics*, 149:251–265, 2002.

[44] A.E.P. Veldman, J. Gerrits, R. Luppes, J.A. Helder, and J.P.B. Vreeburg. The numerical simulation of liquid sloshing on board spacecraft. *Journal of Computational Physics*, 224(1):82–99, 2007.

[45] H.K. Versteeg and W. Malalasekera. *An Introduction to Computational Fluid Dynamics*. Pearson Education Limited, Edinburgh Gate, 2007.

[46] J. Waltz and R. Löhner. A grid coarsening algorithm for unstructured multigrid applications. *AIAA-00-0925*, 2000.

[47] K. Watanabe, H. Igarashi, and T. Honma. Comparison of geometric and algebraic multigrid methods in edge-based finite element analysis. *IEEE Transactions on Magnetics*, 41(5):1672–1675, 2005.

[48] F.M. White. *Fluid Mechanics*. McGraw-Hill, London, Third edition, 1994.