AMERICAN UNIVERSITY OF BEIRUT

THE PERFORMANCE OF ILU (0), SPAI, AND AINV AS SMOOTHERS IN AN ALGEBRAIC MULTIGRID SOLVER

by
SOBHI MOHAMMAD TAKKOUSH

A thesis
submitted in partial fulfillment of the requirements
for the degree of Master of Engineering
to the Department of Mechanical Engineering
of the Faculty of Engineering and Architecture
at the American University of Beirut

Beirut, Lebanon
April 2016

AMERICAN UNIVERSITY OF BEIRUT

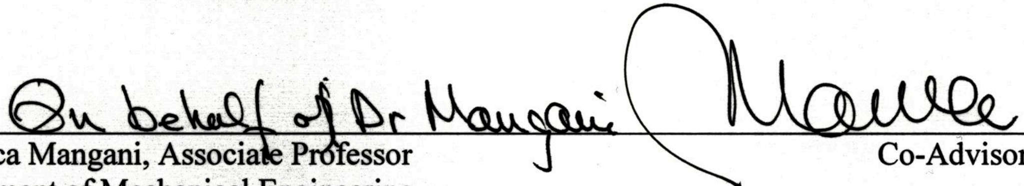
THE PERFORMANCE OF ILU (0), SPAI, AND AINV AS SMOOTHERS IN AN ALGEBRAIC MULTIGRID SOLVER

by

SOBHI MOHAMMAD TAKKOUSH

Approved by:

Dr. Marwan Darwish, Professor — Advisor
Department of Mechanical Engineering



Dr. Luca Mangani, Associate Professor — Co-Advisor
Department of Mechanical Engineering
Lucerne University of Applied Sciences and Arts, Switzerland

Dr. Fadl Moukalled, Associate Dean & Professor — Member of Committee
Department of Mechanical Engineering

Dr. Kamel Aboughali, Chairperson & Professor — Member of Committee
Department of Mechanical Engineering

Date of thesis defense: April 22, 2016

AMERICAN UNIVERSITY OF BEIRUT

THESIS, DISSERTATION, PROJECT RELEASE FORM

Student Name: Takkoush Sobhi Mohammad
Last First Middle

☒ Master's Thesis ○ Master's Project ○ Doctoral Dissertation

☑ I authorize the American University of Beirut to: (a) reproduce hard or electronic copies of my thesis, dissertation, or project; (b) include such copies in the archives and digital repositories of the University; and (c) make freely available such copies to third parties for research or educational purposes.

☐ I authorize the American University of Beirut, **three years after the date of submitting my thesis, dissertation, or project,** to: (a) reproduce hard or electronic copies of it; (b) include such copies in the archives and digital repositories of the University; and (c) make freely available such copies to third parties for research or educational purposes.

[signature] May 5, 2016

Signature Date

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my advisor Professor Marwan Darwish for his continuous support, guidance, patience and motivation throughout my graduate studies. Very special thanks to Professor Luca Mangani from Lucerne University of Applied Sciences and Arts for his guidance, feedback and valuable time in contributing to this work. I would like also to thank the rest of the committee members Professors Fadl Moukalled and Kamel Aboughali for their assessment of this work and their constructive comments.

A “smooth” thank you for my colleagues in the CFD group for their moral support at all times.

Last but not least, a more than thank you to my beloved parents, wife and family for supporting me in every aspect of my life, encouraging me, and standing by me in hard and good times.

AN ABSTRACT OF THE THESIS OF

Sobhi Mohammad Takkoush for Master of Engineering
Major: Mechanical Engineering

Title: The Performance of ILU (0), SPAI, and AINV as Smoothers in an Algebraic Multigrid Solver

Incomplete LU decomposition with no fill-in ILU (0) has been used as a standard smoother with algebraic multigrid solvers in many applications. With recent developments new techniques have emerged following the sparse approximate inverse approach. Methods like Sparse Approximate inverse (SPAI) and AINV have been implemented as preconditioners but rarely as smoothers. In this work, the ILU (0), SPAI and AINV will be implemented as smoothers in both scalar and block versions within an algebraic multigrid solver. A comparative assessment of the performance of these techniques as smoothers in an algebraic multigrid solver will be performed in the context of a finite volume discretization method. The smoothers are implemented in uFVM, an in-house MATLAB® based CFD code, then in an open source CFD toolbox OpenFOAM®. The results of SPAI in uFVM show how it is computationally expensive and not robust, thus it was not considered in OpenFOAM®. Using OpenFOAM®, two turbulent fluid flow test cases with high aspect ratio are used to compare the smoothers. Residual convergence rates, number of iterations as well as CPU time are used to evaluate the performance. For segregated flow solver, AINV and ILU (0) show robustness having same convergence rate, number of iterations and CPU time with slight difference; however, ILU (0) outperforms AINV in every aspect for coupled flow solver.

CONTENTS

ACKNOWLEDGMENTS.. v

ABSTRACT.. vi

LIST OF ILLUSTRATIONS.. ix

LIST OF TABLES.. xi

Chapter

I. INTRODUCTION.. 1

A. Background.. 2

1. General Transport Equation.. 2
2. Systems of Algebraic Equations.. 3

B. Solutions of Algebraic Linear Systems.. 5

1. Direct Methods.. 5
2. Iterative Methods.. 6
3. Preconditioning and Convergence of Iterative Methods.. 7
4. Multigrid Methods.. 11

II. LITERATURE REVIEW.. 15

A. Methods based on Frobenius Norm Minimization.. 16

B. Factorized Sparse Approximate Inverses.. 17

C. Inverse ILU Techniques.. 18

D. Incomplete Factorization Methods.. 19

III. THE FINITE VOLUME METHOD.. 21

A. Introduction …………………………………………………………… 21

B. The Discretization Process ………………………………………………… 21

1. Domain Discretization (Geometric Discretization)………………. 22
2. Equation Discretization……………………………………… 24

IV. ILU (0), SPAI AND AINV ITERATIVE SOLVERS………. 28

A. Smoothers Algorithms ……………………………………………… 28

1. ILU (0) (Incomplete LU Factorization) Method………………… 28
2. SPAI (Sparse Approximate Inverse) Method……………………. 30
3. AINV (Approximate Inverse) Method……………………….. 37

B. ORTHOMIN Method ……………………………………………… 41

C. Block Methods…………………………………………………….. 42

1. Block ILU (0) (Incomplete LU Factorization) Method………… 43
2. Block AINV Method…………………………………………… 44

V. TEST CASES AND RESUTLS…………………………………. 45

A. Flow over a Stator Blade (Compressible-Segregated)…………………… 45

B. 90°Pipe Bend (Incompressible-Coupled)………………………………… 51

VI. CONCLUSION……………………………………………………... 57

Appendix

I. SOFTWARE DESCRIPTION.. 58

BIBLIOGRAPHY... 68

ILLUSTRATIONS

Figure Page

1. Error Frequency modes in 1D Grid………………………………………. 12

2. A schematic grid systems hierarchy with MG approach………………... 13

3. (a) Discretized domain into non-overlapping elements, (b) Details of an element……………………………………………………………... 22

4. Unstructured Mesh………………………………………………………. 23

5. Mesh Terminology for 2D………………………………………………. 23

6. Mesh Terminology for 3D………………………………………………. 24

7. Typical Element Shapes…………………………………………………. 24

8. One dimensional mesh…………………………………………………... 25

9. ILU (0) Factorization flowchart…………………………………………. 29

10. SPAI algorithm flowchart……………………………………………….. 36

11. The outer-product form of AINV algorithm flowchart………………….. 40

12. Computational domain and boundary conditions……………………….. 46

13. Computational grid for the flow over a blade problem………………….. 47

14. Final Residual vs Time for Smoothers…………………………………... 48

15. Initial & Final Residual using ILU (0)…………………………………... 48

16. Initial & Final Residual using AINV……………………………………. 49

17. Number of Iterations vs Time of ILU (0)……………………………….. 50

18. Number of Iterations vs Time of AINV…………………………………. 50

19. Computational domain and boundary conditions……………………….. 52

20. Top View of Mesh on the Pipe Symmetry Plane………………………... 53

21. Pressure Residual vs Time for Block Smoothers………………………... 54

22. Velocity Residual vs Time for Block Smoothers………………………... 54

23. Number of Iterations vs Time Step of Block ILU (0)…………………… 55

24. Number of Iterations vs Time Step of Block AINV…………………….. 55

25. OpenFOAM Case……………………………………………………….. 59

26. uFVM Domain…………………………………………………………... 61

27. uFVM Coefficients Storage Arrays……………………………………... 62

28. Overview of OpenFOAM Structure……………………………………... 63

29. Matrix-Vector Multiplication in uFVM…………………………………. 65

30. Matrix-Vector Multiplication in OpenFOAM…………………………... 66

31. Modified Matrix-Vector Multiplication in OpenFOAM………………... 67

TABLES

Table Page

1. Stator Vane Data 46
2. Smoothers Comparison............ 51
3. Initial and Boundary Conditions............ 52
4. Block Smoothers Comparison............ 56

CHAPTER I

INTRODUCTION

Systems of linear algebraic equations naturally occur in many fields in Engineering such as in dynamics, electric circuits and structural analysis. Advancements, improvements and innovations in computers made it possible to solve very large systems of algebraic equations in quick and accurate manners. This progress has not only allowed engineers and scientists to solve and handle complex problems, but has also encouraged them to use linear systems to solve problems in fields where they do not naturally occur as in fluid dynamics, chemical processes, heat transfer, stress-strain function, thermodynamics and a lot more. Even nonlinear systems can be treated as linear systems and solved iteratively which is considered a useful technique in analyzing complex systems. Among others, the numerical solution of the conservation equations governing fluid flow and transfer phenomena problems, using the finite volume method, is reduced to solving large systems of algebraic equations. Indeed most of the computational power required to solve these problems is spent on solving the arising algebraic systems. Therefore the development of efficient techniques for solving these systems is critical to the size of problems that can be tackled and to the time needed for solutions to be obtained with these methods.

Techniques for solving algebraic systems of equations are grouped under two categories denoted by direct and iterative methods. Direct methods are not suitable for very large systems that arise in CFD applications due to their prohibitive storage and computational power requirement and the use of iterative algebraic solvers is the norm.

While requiring low storage, the rate of convergence of iterative methods drastically deteriorates as the size of the algebraic system increases. This problem is more critical in highly compressible systems such as turbo machinery, supersonic flow, anisotropic diffusion and others where an ill-conditioned and weakly diagonally dominant matrix A arises which leads to large Eigen values and breakdown in convergence rate of the solution. This has constituted a severe limitation for iterative solvers. The development of multigrid methods remedied this weakness. The idea behind multigrid methods is to solve the system of equations on a hierarchy of grids. Since iterative solvers are capable of removing oscillatory errors but not the smooth components of the error, solving on successively coarser grid errors that are smooth on a fine grid will look more oscillatory on a coarser one and thus are easier to be removed by the iterative solver. This is why these iterative solution methods are denoted by smoothers in the context of multigrid methods.

A. Background

This section includes a brief overview of linear algebraic systems of equations, discretization process, preconditioning and solutions of algebraic equations. This background is important for understanding the context of this research.

1. General Transport Equation

The governing equations of interest in this work are the ones representing conservation of mass, momentum, energy, and related transfer phenomena. For a scalar variable, the general transport equation can be written as

$$\frac{\partial(\rho\emptyset)}{\partial t} + \nabla.(\rho\mathbf{v}\emptyset) = \nabla.\left(\Gamma^{\emptyset}\nabla\emptyset\right) + Q^{\emptyset} \quad (1.1)$$

where ρ is the density, **v** is the velocity vector, $\Gamma^{\emptyset}$ is the diffusion coefficient, and $Q^{\emptyset}$ is the source term. The meanings of $\Gamma^{\emptyset}$ and $Q^{\emptyset}$ are specific to the modeled $\emptyset$ equation. Replacing $\emptyset$ by a certain variable to be solved yields different differential equations which describe certain physical or flow property.

The Continuity Equation:

$$\frac{\partial\rho}{\partial t} + \nabla.(\rho\boldsymbol{v}) = 0 \quad (1.2)$$

The Momentum Equation:

$$\frac{\partial(\rho\mathbf{v})}{\partial t} + \nabla.(\rho\mathbf{v}\mathbf{v}) = \nabla.\tau - \nabla p + \mathbf{B} \quad (1.3)$$

where $\tau = -\mu(\nabla\mathbf{v} + \nabla\mathbf{v}^{T}) + \frac{2}{3}\mu\mathbf{I}\nabla.\mathbf{v}$

The Energy Equation:

$$\frac{\partial(\rho h)}{\partial t} + \nabla.(\rho v h) = \nabla.(k\nabla T) + S_h \quad (1.4)$$

where h is the specific enthalpy, k the thermal conductivity and T is the temperature.

The Species Equation:

$$\frac{\partial(\rho \mathrm{Y_i})}{\partial t} + \nabla.(\rho\mathbf{v}\mathrm{Y}_i) = \nabla.(\Gamma_i\nabla\mathrm{Y}_i) + R_i \quad (1.5)$$

where ρ is the density Y_i is the mass of species i per mass of mixture, Γ_i the diffusion coefficient for Y_i in the mixture and R_i is the rate of formation of Y_i through chemical reactions.

2. Systems of Algebraic Equations

A linear system of equations is a collection or set of two or more equations with the same set of variables. A system of *n* algebraic equations with *n* unknowns is expressed in a general form as

$$\begin{array}{l} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \cdots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + \cdots + a_{2n}x_n = b_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + \cdots + a_{3n}x_n = b_3 \\ \vdots \qquad \vdots \qquad \vdots \qquad \ldots \qquad \vdots \qquad \vdots \\ a_{n1}x_1 + a_{n2}x_2 + a_{n3}x_3 + \cdots + a_{nn}x_n = b_n \end{array} \tag{1.6}$$

System of linear equations can be represented either by vector form or matrix form

Vector form

$$x_1 \begin{bmatrix} a_{11} \\ a_{21} \\ a_{31} \\ \vdots \\ a_{n1} \end{bmatrix} + x_2 \begin{bmatrix} a_{12} \\ a_{22} \\ a_{32} \\ \vdots \\ a_{n2} \end{bmatrix} + x_3 \begin{bmatrix} a_{13} \\ a_{23} \\ a_{33} \\ \vdots \\ a_{n3} \end{bmatrix} + \ldots + x_n \begin{bmatrix} a_{1n} \\ a_{2n} \\ a_{3n} \\ \vdots \\ a_{nn} \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_n \end{bmatrix}$$

Matrix form

$$\mathbf{Ax} = \mathbf{b} \tag{1.7}$$

where

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nn} \end{bmatrix}, \quad \boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_n \end{bmatrix} \tag{1.8}$$

with **A**: *n x n* coefficient matrix, ***x***: *n* vector of unknowns, **b**: *n* vector of constant terms

Each system of linear equations has three properties [1]:

- Consistency (A linear system is inconsistent if it has no solution, otherwise it is consistent).
- The equations are independent if none of them can be derived from others.
- Two linear systems are equivalent if equations in the second system can be derived from equations of first system.

B. Solutions of Algebraic Linear Systems

As mentioned above solution techniques to solve linear algebraic systems of equations [Eq. (1.7)] are generally classified as direct and iterative methods. Since for non-linear systems the coefficients need to be updated, direct methods are not attractive. Moreover iterative methods are more appropriate and suitable in terms of computational cost per iteration and memory requirements. An overview of these techniques is given next.

1. Direct Methods

Direct methods solve the above system [Eq. (1.7)] in one step. This is done by inverting $\mathbf{A}$ to compute $\mathbf{x}$ as

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{b} \tag{1.9}$$

Some of the famous direct methods include: Gauss Elimination, LU decomposition, TriDiagonal Matrix Algorithm (TDMA) and PentaDiagonal Matrix Algorithm (PDMA). The last two are especially suitable for sparse, banded matrices. The resultant matrix $\mathbf{A}$ using the FVM is sparse with many zero entries, with its inverse being very expensive computationally requiring large memory. Thus the use of direct

methods in CFD is impractical and almost never employed because A is large and its elements (i.e., the coefficients) depend on the solution necessitating iterative updates. Moreover nowadays industrial CFD problems include hundreds of thousands to millions of cells/elements with more than 4 unknowns per cell; making **A** very large.

2. Iterative Methods

System of equations result from CFD analysis has two important and distinguishing characteristics: **A** is very sparse and the system is approximate. Direct methods do not take advantage of these two characteristics, that's why they are computationally expensive and require a lot of memory especially when dealing with non-linear system where update is required by outer iterations, thus they are not used in CFD applications. Even if sparse direct methods tend to work well for 2D PDE discretization, they scale more poorly for 3D problems [2]. On the other hand, Iterative methods are easily formulated to take advantage of the coefficient matrix sparsity.

Iterative methods are widely used in CFD applications. These methods follow guess-and-correct methodology which gradually improves the guessed initial solution by forming a correction equation based on an estimate of the residual. First type of these methods are the Stationary iterative methods which include the Jacobi, Gauss-Seidel; Successive over-relaxation (SOR), and ILU which are widely used, simpler, easy to implement but not as effective as non-stationary iterative methods which are the other type. The other types are non-stationary iterative methods which involve data that changes at each and every iteration due to inner-products with residuals. They are based on the idea of orthogonalisation of vectors and subspace projections. These methods are known as Krylov iterative methods [3] which include Conjugate Gradient, Generalized

minimal residual method (GMRES), biconjugate gradient method, Orthomin and many others.

The rate of convergence of iterative methods significantly decreases when size of algebraic system increases (coarse to fine mesh). Moreover, it will also decrease after removing initial errors (high frequency errors) leaving smooth errors on which these iterative methods are not very effective and the convergence rate stalls. This will require a large number of iterations for the solution to converge mainly because the location of Eigen values of the matrix is spread apart from each other [4]. Two important techniques are widely used to improve the performance and convergence of the iterative solvers which are pre-conditioning and Multigrid. The use of these techniques will cause the Eigen values to clump and cluster around (1, 0), thus accelerating the convergence rate.

3. Preconditioning and Convergence of Iterative Methods

To accelerate convergence of iterative methods, preconditioning is usually used whereby the original system is replaced by an equivalent system having the same solution but an improved condition number thus improving the condition of the matrix making it is easier to converge.

Let's discuss preconditioning and how it affects convergence rate. The condition number (A) measures the sensitivity of the system with small change δ which is expressed mathematically as

$$\|A + \delta A\|_2 \|x + \delta x\|_2 = \|b + \delta b\|_2 \tag{1.10}$$

A system of equations is considered to be well-conditioned if a small change in the coefficient matrix **A** or a small change in RHS vector of sources **b** results in a small change in the solution vector **x** [5] (i.e. system has low condition number)

$$\|\delta A\|_2, \|\delta b\|_2 \geq \|\delta x\|_2 \quad (1.11)$$

A system of equations is considered to be ill-conditioned if a small change in the coefficient matrix **A** or a small change in RHS vector of sources **b** results in a large change in the solution vector **x**.(i.e. system has high condition number)

$$\|\delta A\|_2, \|\delta b\|_2 < \|\delta x\|_2 \quad (1.12)$$

In Ill-conditioned system, the solution will diverge. One way to prevent divergence is under relaxation of solution

$$x_{k+1}^{new,used} = \alpha x_{k+1}^{new,predicted} + (1-\alpha)x_k \quad (1.13)$$

where α is relaxation factor [0,1]

Another complicated but effective way is based on the condition number of coefficient matrix (A) is preconditioning.

Usually the coefficient matrix **A** has high condition number, so in order to have low condition number and ensure convergence, the system is transformed into a preconditioned one which has the same solution as original but with better condition number and spectral properties [6].

$$(M^{-1} A) < (A) \quad (1.14)$$

where M is non-singular preconditioning matrix of A

To solve $\mathbf{Ax} = \mathbf{b}$

Decompose **A** = **L** + **D** + **U**

$= \mathbf{M} + \mathbf{S}$

where **L,U** are the lower and upper triangular sub-matrices of **A, M** is "large" accounting for most of **A** and **S** is "small".

Let k be the iteration number

In iterative methods, at first iteration where k =1:

The variable at first iteration is $\mathrm{x}^{\mathbf{1}} = \mathbf{M}^{-1}\mathbf{b}$

With residual $\mathrm{r}^{1} = \mathbf{b} - \mathbf{A}\mathrm{x}^{1}$

Then for each iteration:

$$\mathrm{x}^{(\boldsymbol{k}+\mathbf{1})} = \mathrm{x}^{(\boldsymbol{k})} + \mathbf{M}^{-1}\mathrm{r}^{(\boldsymbol{k})}$$

$$\mathrm{r}^{(k+1)} = \mathbf{b} - \mathbf{A}\mathrm{x}^{(k+1)}$$

To derive general form of preconditioned system:

$$\mathrm{x}^{(\boldsymbol{k}+\mathbf{1})} = \mathrm{x}^{(\boldsymbol{k})} + \mathbf{M}^{-1}\mathrm{r}^{(\boldsymbol{k})}$$

$$= \mathrm{x}^{(\boldsymbol{k})} + \mathbf{M}^{-1}(\mathbf{b} - \mathbf{A}\mathrm{x}^{(k)})$$

$$= \mathrm{x}^{(\boldsymbol{k})} + \mathbf{M}^{-1}\mathbf{b} - \mathbf{M}^{-1}\mathbf{M}\mathrm{x}^{(k)} - \mathbf{M}^{-1}\mathbf{S}\mathrm{x}^{(k)}$$

$$= \mathrm{x}^{(\boldsymbol{k})} + \mathbf{M}^{-1}\mathbf{b} - \mathrm{x}^{(k)} - \mathbf{M}^{-1}\mathbf{S}\mathrm{x}^{(k)}$$

$$= -\mathbf{M}^{-1}\mathbf{S}\mathrm{x}^{(k)} + \mathbf{M}^{-1}\mathbf{b}$$

$$\mathrm{x}^{(\boldsymbol{k}+\mathbf{1})} = \mathbf{P}\mathrm{x}^{(k)} + \mathbf{z} \tag{1.15}$$

where $\mathbf{P} = \mathbf{D}^{-1}(\mathbf{L} + \mathbf{U})$ $\quad Jacobi$ and $\mathbf{P} = (\mathbf{D} - \mathbf{L})^{-1}\mathbf{U}$ $\quad Gauss - Seidel$

Define the error/correction at step k to be

$\mathrm{e}^{(k)} = \mathrm{x}^{(\boldsymbol{k})} - \mathrm{x}$ (x is the exact solution)

$\mathrm{e}^{(k+1)} = \mathrm{x}^{(\boldsymbol{k+1})} - \mathrm{x}$

$= \mathbf{P}\mathrm{x}^{(k)} + \mathbf{z} - \mathrm{x}$

$= \mathbf{P}(\mathrm{e}^{(k)} + \mathrm{x}) + \mathbf{z} - \mathrm{x}$

$= \mathbf{P}\mathrm{e}^{(k)} + \mathbf{P}\mathrm{x} + \mathbf{z} - \mathbf{P}\mathrm{x} - \mathbf{z}$

$$\mathrm{e}^{(k+1)} = \mathbf{P}\mathrm{e}^{(k)} \quad (1.16)$$

In linear Algebra, a matrix **A** has set of eigenvalues and eigenvectors such that

$$\mathbf{A}\boldsymbol{\upsilon} = \lambda\boldsymbol{\upsilon} \quad (1.17)$$

where λ: eigen value, $\boldsymbol{\upsilon}$: eigen vector

Eq. (22 & 23) can be written in terms of eigenvalues of preconditioning matrix **P**, since any variable $\emptyset$ or e is a linear combination of eigenvalues of matrix **P**

$$\mathrm{x} = \sum_{i=1}^{N} c_i \boldsymbol{v}_i \quad (1.18)$$

where c = constant, v= eigenvector and N = size of square matrix

Since error is the difference between variables at iteration k and exact

$$e = \sum_{i=1}^{N} c_i \boldsymbol{v}_i \quad (1.19)$$

with $\mathrm{e}^{(k)} = \mathbf{P}^k \mathrm{e}^0$

but $e^{(k)} = \mathbf{P}^k \sum_{i=1}^{N} c_i \boldsymbol{v}_i$

we get $e^{(k)} = \sum_{i=1}^{N} c_i \mathbf{P}^k \boldsymbol{v}_i$

but $\mathbf{P}\boldsymbol{\upsilon} = \lambda\boldsymbol{\upsilon}$

$$e^{(k)} = \sum_{i=1}^{N} c_i \lambda_i^k \, \boldsymbol{v}_i \tag{1.20}$$

For convergence:

$$\lim_{k\to\infty} e^{(k)} = 0$$

$$\lim_{k\to\infty} e^{(k)} = \lim_{k\to\infty} \sum_{i=1}^{N} c_i \lambda_i^k \, \boldsymbol{v}_i = 0$$

$$e^{(k)} \to 0 \;\leftrightarrow\; \lambda^k \to 0 \;\leftrightarrow\; \rho\,(\mathbf{P}) < 1$$

A sufficient condition for convergence is when norm$\|\mathbf{P}\| < 1$. More precisely, convergence will occur when spectral radius $\rho\,(\mathbf{P}) = \max|\,\lambda\,(\mathbf{P})\,|$ is less than one [6].

4. Multigrid Methods

Multigrid methods were independently introduced by Fedorenko [7] and Poussin [8] in the 1960s, and then it gained popularity with the work of Settari and Azziz [9], and later in the 1970s with the theoretical work of Brandt [10] who showed that iterative solvers are efficient at eliminating high frequency errors but inefficient at removing the low frequency or smooth component of the error. Iterative solvers are known as smoothers when used in multigrid algorithms due to their smoothing property.

As can be seen in Figure 1, the error frequency varies from high frequency (short wave length λ_1) to low frequency (long wave length λ_5). It can be noticed how the high frequency error looks oscillatory over an element and can be easily detected and removed by the smoother, while the low frequency error is smoothed or spread over the entire domain and cannot be removed easily [6].

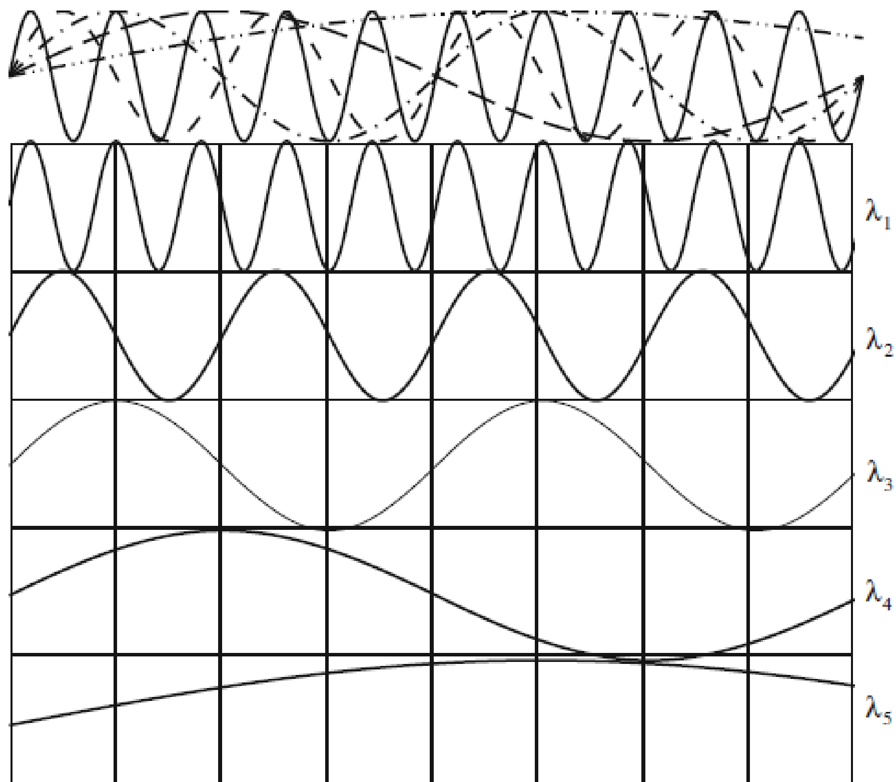


Figure 1: Error Frequency modes in 1D Grid [6]

The purpose of multigrid methods is to improve the convergence rate of iterative solvers by transferring the remaining low frequency errors (smooth errors) from a fine grid to a coarser grid where the low frequency error will appear as high frequency; making it easier for the smoother to detect the error and eliminate it, thus increasing the rate of convergence [11, 12, 13]. This is done through a hierarchy of coarse grids (Figure 2) where the smooth error is restricted to the coarse grid and then smoothed and transferred to a coarser grid and so on until the coarsest grid is reached. Then this correction is prolonged back to fine grids also with application of smoothers, until reaching the original finest grid.

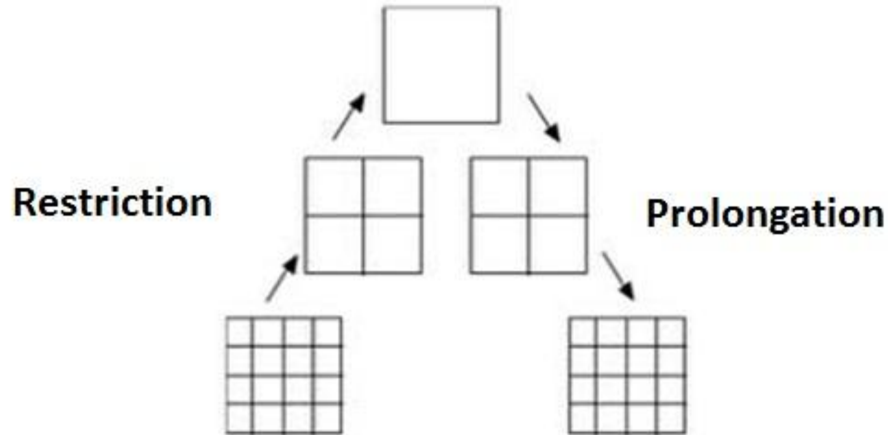


Figure 2: A schematic grid systems hierarchy with MG approach [6]

Multigrid methods are classified as Geometric Multigrid [7] and Algebraic Multigrid [8]. In geometric multigrid, cells of fine mesh are agglomerated to form a coarse grid. This approach is complex on unstructured grids since irregular shaped elements are difficult to agglomerate. In this work the Algebraic multigrid approach will be followed where the agglomeration process is purely algebraic based on the coefficients with no need for any geometric information.

Let j be the level at which the solution is sought and G_I the set of cells i at fine level j that agglomerate to form cell I at coarse level (j+1).

The main steps of Multigrid (Geometric or Algebraic) are divided into two parts [6].

From fine to coarse grid:

- Restriction: Error is transferred or restricted from fine to coarse grid where the residual at level (j+1) on agglomerated cell I becomes the summation of residuals of G_I at level j that agglomerate to form the coarser one.

$$r_I^{(j+1)} = \sum_{i \in G_I} r_i^{(j)} \tag{1.21}$$

- System of equations updated for coarse level
- Number of smoother iterations is applied

From coarse to fine grid

- Prolongation: Correction is transferred or prolonged from coarse to fine grid where the error at level (j+1) on agglomerated cell I is inherited to set G_I at level j.

$$e^{(j)} = I_{j+1}^{j} e^{(j+1)} \tag{1.22}$$

where I_{j+1}^{j} is prolongation operator or interpolation matrix

- Solution variables are updated or corrected

$$x^{(j)} = x^{(j)} + e^{(j)} \tag{1.23}$$

- Number of smoother iterations is applied

Traditionally the Gauss-Siedel and ILU(0) [14,15] (Incomplete LU decomposition with no fill-in) iterative methods have been used as smoothers in the context of algebraic multigrid methods. The SPAI [16-22] and AINV [23,24,25,26,27,28] Sparse Approximate Inverse iterative methods have never been used for that purpose. The aim of the proposed work is to improve the performance of iterative solvers with preconditioning and Multigrid by implementing the ILU(0), SPAI, and AINV solvers as smoothers in an algebraic multigrid environment, an acceleration framework, in the context of the Finite Volume Method (FVM), and to evaluate and compare their performance in terms of CPU time by solving a number of three-dimensional fluid flow problems using open source CFD software OpenFOAM.

CHAPTER II

LITERATURE REVIEW

This section gives an overview of popular smoothers with special focus on ILU (0), SPAI and AINV. It also includes the importance for using Sparse Approximate inverse methods as smoothers.

Sparse approximate inverse methods are based on approximating the inverse matrix directly similar to polynomial preconditioning; however, the approximate inverse in polynomial preconditioning is available implicitly in the form of polynomial in the coefficient matrix **A**, while with sparse approximate inverse methods, the approximate inverse matrix $\boldsymbol{M} \approx \boldsymbol{A}^{-1}$ is explicitly computed and stored [29]. The preconditioning operation of these methods reduces to a matrix-vector product. These methods were first proposed in the early 1970s by Benson [30] and Frederickson [31], but they received little attention due to the lack of effective strategies for automatically determining nonzero pattern for the sparse approximate inverse. With new developments in these strategies, interest has been renewed.

Two main reasons or motivations led to the development of these methods. First reason was the parallel processing. The second reason was that incomplete factorization techniques can fail in strongly non-symmetric and indefinite matrices due to instability in the factorization process itself or in the back substitution phase as shown by Chow & Saad [32]. This drawback has been eliminated in most approximate inverse methods even on serial processing.

A. Methods based on Frobenius Norm Minimization

These methods were historically the first to be introduced among all categories. They are based on Frobenius norm minimization. Benson & Frederickson [33, 34] were the first to propose a sparse approximate inverse preconditioner in a static way by computing the norm of (**AM-I**) for prescribed priori chosen sparsity pattern for **M**. A common choice was that **M** has the same sparsity pattern as matrix **A**, but this has been proven not to be robust for general sparse problems and has high computational cost and storage.

A more robust approach, for general sparse matrices is to start with a simple initial guess for the nonzero pattern of **M** (e.g. diagonal matrix) then successively augment or fill-in this pattern until a criterion of certain residual is reached. This approach was first proposed by Cosgrove et al. [16] where they suggested the initial structure of **M** to be diagonal. Same approach but with different augmentation strategies were also followed by Grote and Huckle [17] who introduced the SPAI preconditioner. A similar algorithm was reported by Gould and Scott [18] who introduced the same algorithm of SPAI but the optimal reduction of the residual is determined for the full minimization problem instead of the 1D minimization in SPAI making their algorithm more accurate but expensive. The serial cost of computing the preconditioner in the SPAI algorithm can be very high. To lower the cost, Chow and Saad [19] used an iterative method to reduce the residuals of each column of the approximate inverse by applying a dropping strategy whereby excessive fill-in in **M** during the augmentation process is removed; their algorithm is named Minimal Residual (MR). Grote and Bernard introduced in [20] a block version of SPAI.

Zhang [21] introduced an iterative form of SPAI where a thin M is derived in each step, making the Least Squares problems very cheap. Holland et al. [22] generalized this SPAI approach by allowing a sparse target matrix other than Identity matrix, which is useful for two-level preconditioning. Huckle and Kallishcko [35] generalized SPAI with the target approach which was developed into MSPAI (modified SPAI) [36] of an improved preconditioner by combining SPAI with the probing method of Chan and Mathew [37].

B. Factorized Sparse Approximate Inverses

There are several approaches for calculating the factorized approximate inverse. One approach is to construct it directly from **A** without the need for any information about the triangular factors of **A.** This approach yields a class of methods like FSAI preconditioner which was introduced by Kolotilina and Yeremin [38] where they assumed **A** is SPD (Symmetric Positive Definite) matrix with sparsity pattern of nonzeros only in positions corresponding to nonzeros in the lower triangular matrix of **A.** Kaporin [39] followed the same method in [38] but considered the sparsity pattern of the lower triangular of $\mathbf{A}^{\mathbf{k}}$ where k is an integer. This method is more sophisticated but more costly. FSAI can be made to solve nonsymmetric cases; but with no guarantee for the solvability of the local linear systems and the non-singularity of the approximate inverse. The advantage of FSAI is that it can be implemented in parallel [40, 41]. Its main disadvantage is the need for prescribed sparsity pattern of the approximate inverse in advance.

Another method is based on incomplete bi-conjugation which was first proposed by Benzi [42]. This method was introduced for symmetric matrices [23] then

for non-symmetric ones [24] and was denoted as the AINV method. Then Block version of AINV was introduced [25] which proved to be more robust than scalar one. Unlike FSAI, this mehod (AINV) does not require the sparsity pattern to be known in advance and can be applied to general sparse matrices. Bridson and Tang [26] introduced the outer product version of AINV making it more robust.

The third method to compute sparse approximate inverse preconditioner directly from coefficient matrix **A** is based on bordering. It was proposed by Saad [43] referred to as AIB. It differs from AINV in the computations for the inverse factors where they are tightly coupled. This method is perfectly suitable for symmetric cases, while it is not accurate for non-symmetric cases. Bru et al. [44, 45] proposed to compute the sparse approximate factors using Inverse of Sherman-Morrison (ISM).

The main advantages of all factorized sparse approximate inverse methods is their lower cost and lower number of user defined parameters as compared to methods based on Frobenius norm minimization.

C. Inverse ILU Techniques

This class is considered among factorized sparse approximate inverse but it is based on two-stage process: first incomplete LU factorization (ILU) of **A** is computed using standard methods/techniques, and then ILU are approximately inverted [14, 15]. This class shares some advantages of the class in the previous section (factorized); however, it has several disadvantages like it assumes that ILU factorization has already been computed, and it won’t work if the ILU factorization was unstable which is sometimes the case for highly non-symmetric, indefinite problems [32, 46]. For this

reason, it cannot be parallelized or in other words it limits the parallel efficiency of this class.

D. Incomplete Factorization Methods

Incomplete factorization methods were introduced in the 60's by Buleev [27] and Oliphant [28, 47]. The relation of incomplete factorization methods with matrix splitting was considered by Varga [48, 49] who provided a convergence analysis for M-matrices. The first who introduced the term preconditioning was Evans [50] who also considered the use of sparse LU factors as preconditioner. These methods were of particular interest in the field of oil reservoir simulation, e.g. Stone [51] and Dupont-Kendall-Rachford [52] who proposed approximate factorization method for elliptic problems. The major breakthrough was in mid 1970s by Meijerink and van der Vorst [53] who established, recognized and proved the existence of incomplete factorization for M-matirces and showed that preconditioning Conjugate Gradient, by using Incomplete factorizations, can result in efficient results. In their paper, they also introduced the ILU (0) which is the most basic form of ILU preconditioner. This article had a very important role in capturing the attention of researchers about the importance of this preconditioning technique. Moreover Kershaw [54] popularized this approach.

There are several ILU methods that can be classified under two types [55]: a type that changes only the non-zero entry values in the preconditioning matrix obtained by ILU (0) which includes shifted ILU by Manteuffel [56] and modified ILU (MILU) by Gustafsson [57] based on the levels of fill-in concept for finite difference discretizations. The other type changes both non-zero entry structure as well as their value. The second type includes the work of Watts [58] who introduced ILU with k

extra diagonals like in [57] and generalized it using the definition of fill-in of high order ILU for unstructured matrices. This type also includes ILUT (Threshold) by Saad [59] and Crout ILU by Li, Saad and Chow [60] which allowed fill-ins only within certain ranges.

Recently there has been a shift to use approximate inverse methods as smoothers in multigrid methods. Studies showed that they can be effective and robust smoothers for both algebraic and geometric multigrid. Broker et al. [61, 62] obtained a flexible and parallel AMG with SPAI as a smoother. Broker used finite-element discretization scheme in his work for convection-diffusion problems. Tang and Wang [63] also proved the effectiveness of SAI as a smoother in multigrid methods and compared results with Gauss-Seidel smoother using both finite element and finite difference discretization schemes. Meurant [64] extended AINV to multilevel preconditioner with PCG for solving SPD problems (a finite difference discretization scheme was used).

CHAPTER III

THE FINITE VOLUME METHOD

A. Introduction

In fluid mechanics and heat transfer problems, FVM is widely used due to its conservative property. Finite volume method or sometimes called control volume method divides the geometrical domain into a finite number of discrete cells/elements or control volumes, over which on the integral of PDE with conservation of $\emptyset$ will be applied in a discrete sense [65]. Thus FVM is considered as cell-based schemes since primary values are stored at each cell centroid. The volume integrals are then converted to surface integral by Gauss's divergence theorem (known as Green's theorem for 2D). In this proposal, FVM will be used, and an illustration example that will lead us to the set of discrete algebraic equation will be shown below.

B. The Discretization Process

The conservation equation given by Eq. (2.7) is discretized using the Finite Volume method. In this method the solution domain is decomposed into a set of non-overlapping elements, as shown Figure 3a. Furthermore a cell-centered approach is adopted whereby variables are stored at the centroid of each cell. The governing equations are then integrated over each element (Figure 3b) to yield a system of algebraic equations that relate the value of ϕ at the centroid of an element to its neighboring values, with the solution of this system of equations yielding the solution of the original problem.

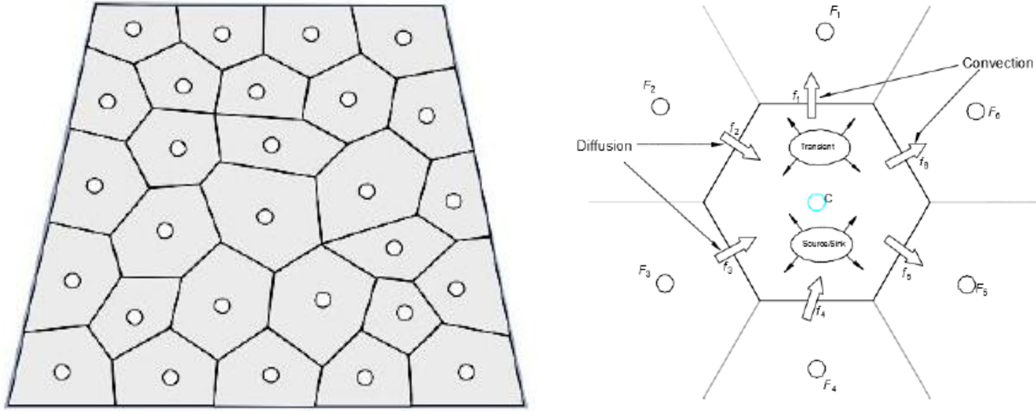


Figure 3: (a) Discretized domain into non-overlapping elements [6], (b) Details of an element [6]

The final algebraic equation is written as

$$a_C \emptyset_C + \sum_{F=NB(C)} a_F \emptyset_F \ = b_C \tag{3.1}$$

An equation similar to Eq. (3.1) is obtained at the centroid of every element in the domain. The collection of these equations forms a system, which in matrix form can be written as

$$\mathbf{A}\emptyset = \mathbf{b} \tag{3.2}$$

As explained above, the method used to solve the resulting system is the main concern of this work.

1. Domain Discretization (Geometric Discretization)

After Domain modeling, the domain needs to be discretized. This is done by mesh generation (Figure 4) which divides the domain into elements or cells or control volumes, then associates or assigns with each elements one or more discrete value of $\emptyset$ [66].

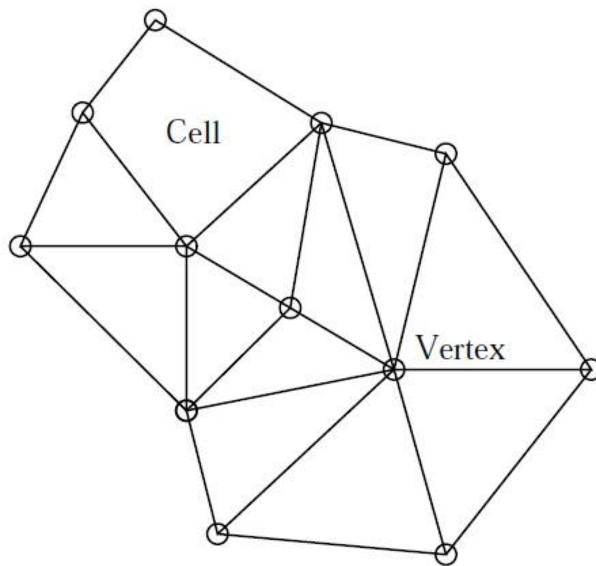


Figure 4: Unstructured Mesh [6]

The components of a mesh are cells or elements. Each cell has its own cell centroid. Moreover each cell is surrounded by faces which meet at nodes or vertices as shown in figure 5 & 6.

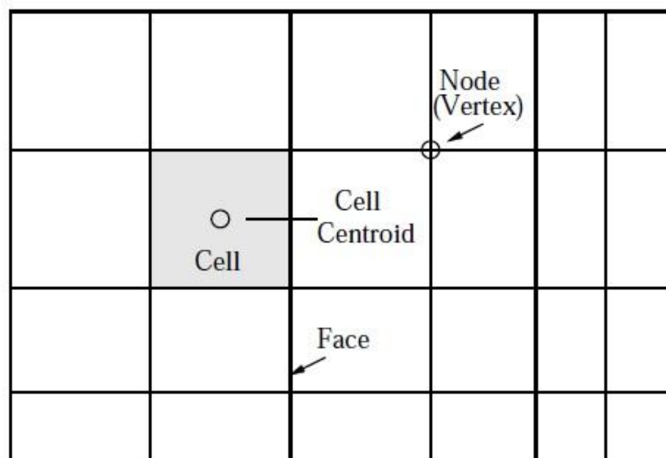


Figure 5: Mesh Terminology for 2D [6]

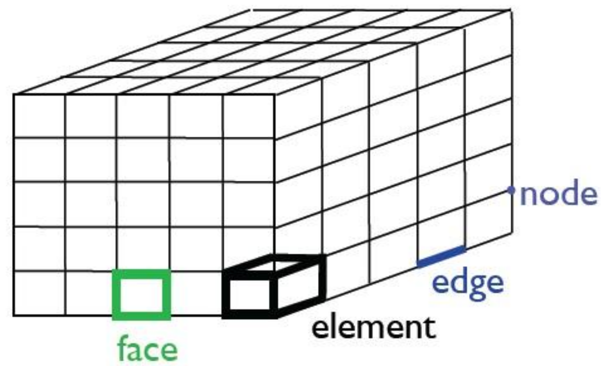


Figure 6: Mesh Terminology for 3D [6]

All elements have the same shape to be set at meshing stage. Below figure shows typical element shape used.

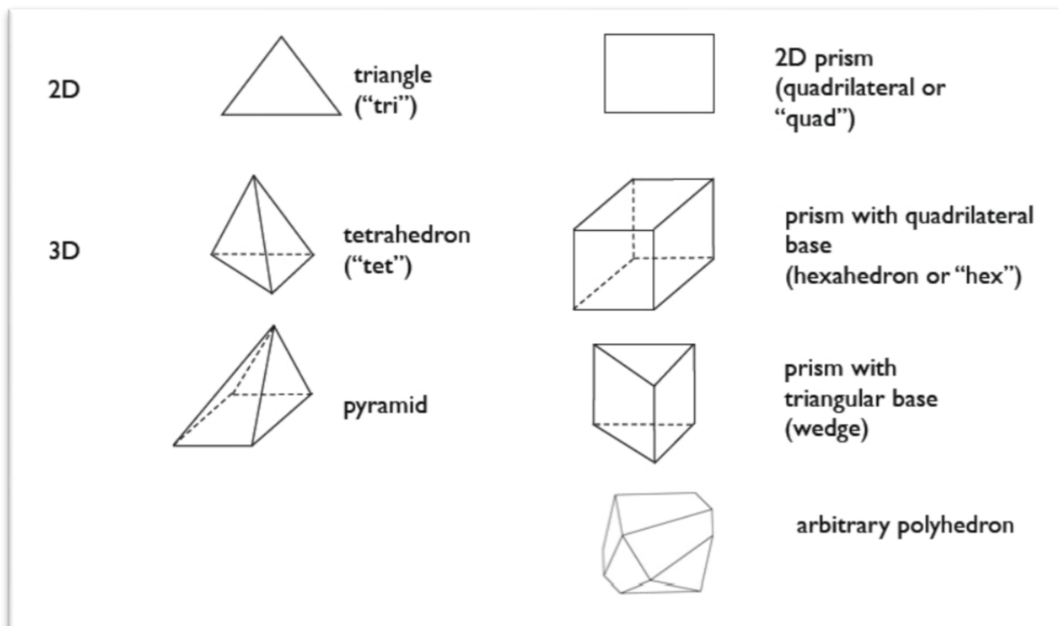


Figure 7: Typical Element Shapes [6]

2. Equation Discretization

After the physical modeling and setting up the partial differential equations (mathematical modeling) that describes the process, we need to convert the PDE to a set

of discrete algebraic equations using any of methods described below. This process is called Equation Discretization. Below is an illustration example that will lead us to the set of discrete algebraic equations.

Consider one-dimensional diffusion with a source term:

$$\frac{\mathrm{d}}{dx}\left(\Gamma\frac{\mathrm{d}\emptyset}{dx}\right)+ S = 0 \tag{3.3}$$

In below figure is one-dimensional mesh, with cells with centroids W,P and E. According to FVM method, the discrete values of $\emptyset$ will be stored there. The cell faces are w and e.

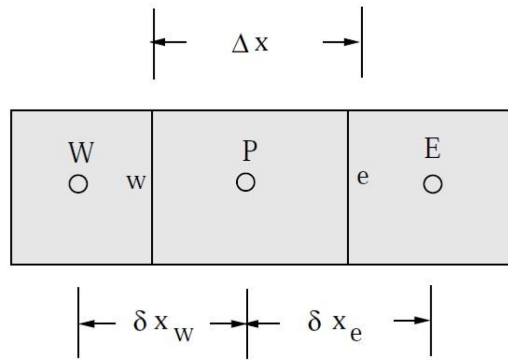


Figure 8: One dimensional mesh

Integrating the above differential equation at cell P:

$$\int_w^e \frac{\mathrm{d}}{dx}\left(\Gamma\frac{\mathrm{d}\emptyset}{dx}\right) dx + \int_w^e S dx = 0 \tag{3.4}$$

Then

$$\left(\Gamma\frac{\mathrm{d}\emptyset}{dx}\right)_e - \left(\Gamma\frac{\mathrm{d}\emptyset}{dx}\right)_w + \int_w^e S dx = 0 \tag{3.5}$$

Assume that $\emptyset$ varies linearly between cells (profile assumption)

Thus

$$\frac{\Gamma(\emptyset_E-\emptyset_P)}{\delta x_e} - \frac{\Gamma(\emptyset_P-\emptyset_W)}{\delta x_w} + \bar{S}\Delta x = 0 \tag{3.6}$$

Collecting terms, we got

$$a_P \emptyset_P = a_E \emptyset_E + a_W \emptyset_W + b \tag{3.7}$$

with

- $a_E = \frac{\Gamma}{\delta x_e}$
- $a_W = \frac{\Gamma}{\delta x_w}$
- $a_P = \sum a_{NB} = a_E + a_W$

$$b = \bar{S} \Delta x \tag{3.8}$$

Similar can be done and derived for the rest of cells (W and E), yielding a set of algebraic equations. Note that the integration of the equations over each element is referred to as local assembly; however, the construction of overall system of algebraic equation from these contribution is referred to as global assembly.

Although Finite Element Methods can be applied to fluid problems by careful treatment to be conservative, it is more stable than FVM approach [67]; however, it requires more storage memory and has slower solution times than the FVM [68].

Applying the discretization process (using FVM) on General Scalar Transport Equation:

$$\frac{\partial(\rho\emptyset)}{\partial t} + \nabla.(\rho \mathbf{v} \emptyset) = \nabla.(\Gamma \nabla \emptyset) + S \tag{3.9}$$

By Divergence theorem

$$\iint_V \frac{\partial(\rho\emptyset)}{\partial t} dV + \oint_{\partial V} (\rho \mathbf{v} \emptyset). d\mathbf{S} = \oint_{\partial V} (\Gamma \nabla \emptyset). d\mathbf{S} + \iint_V S dV \tag{3.10}$$

$$\iint_V \frac{\partial(\rho\emptyset)}{\partial t} dV + \sum_{f=faces(V)} \int_f (\rho \mathbf{v}\emptyset).d\mathbf{S} = \sum_{f=faces(V)} \int_f (\Gamma\nabla\emptyset).d\mathbf{S} + \iint_V S dV \qquad (3.11)$$

Applying above equation on each control volume will lead to discrete system of algebraic equations:

$$a_C\emptyset_C + \sum_{F=NB(C)} a_F\emptyset_F = b_C \qquad (3.12)$$

Or in Matrix form:

$$\mathbf{A}\emptyset = \mathbf{b} \qquad (3.13)$$

CHAPTER IV

ILU (0), SPAI AND AINV ITERATIVE SOLVERS

In this section, we explain and describe the algorithms of the smoothers being used and implemented in this thesis which are ILU(0) , SPAI and AINV. In addition the algorithm of the accelerator ORTHOMIN is also explained. Note that smoothers are implemented and tested first on uFVM MATLAB then transferred to OpenFOAM.

A. Smoothers Algorithms

1. ILU (0) (Incomplete LU Factorization) Method

A standard LU factorization of matrix A will lead to A=LU where L is lower matrix and U is upper matrix, however this has very high computation and storage cost. A more simple approach is an incomplete factorization of coefficient matrix A which will yield A=LU+r where r is the residual of factorization. In this section, algorithm of ILU (0) [53,69,70] will be described. In ILU (0) factorization can be performed using Gaussian elimination but the LU factors have the same nonzero patterns as in matrix A, so any new non-zero element arising in the process is dropped if it appears at location where zero element appear in A.

After ILU (0) factorization of A, we can solve the linear system for the error at each iteration k:

- Compute Residual $r^{(k)} = b - Ax^{(k)}$
- Solve $y^{(k)} = L^{-1}r^{(k)}$ by forward substitution
- Solve $d^{(k)} = U^{-1}y^{(k)}$ by backward substitution

- Update solution Solve $x^{(k+1)} = x^{(k)} + d^{(k)}$
- Repeat until $\|r^{(k)}\|_2 \leq \varepsilon$ where ε is tolerance

Below flowchart describes the process of ILU (0) algorithm

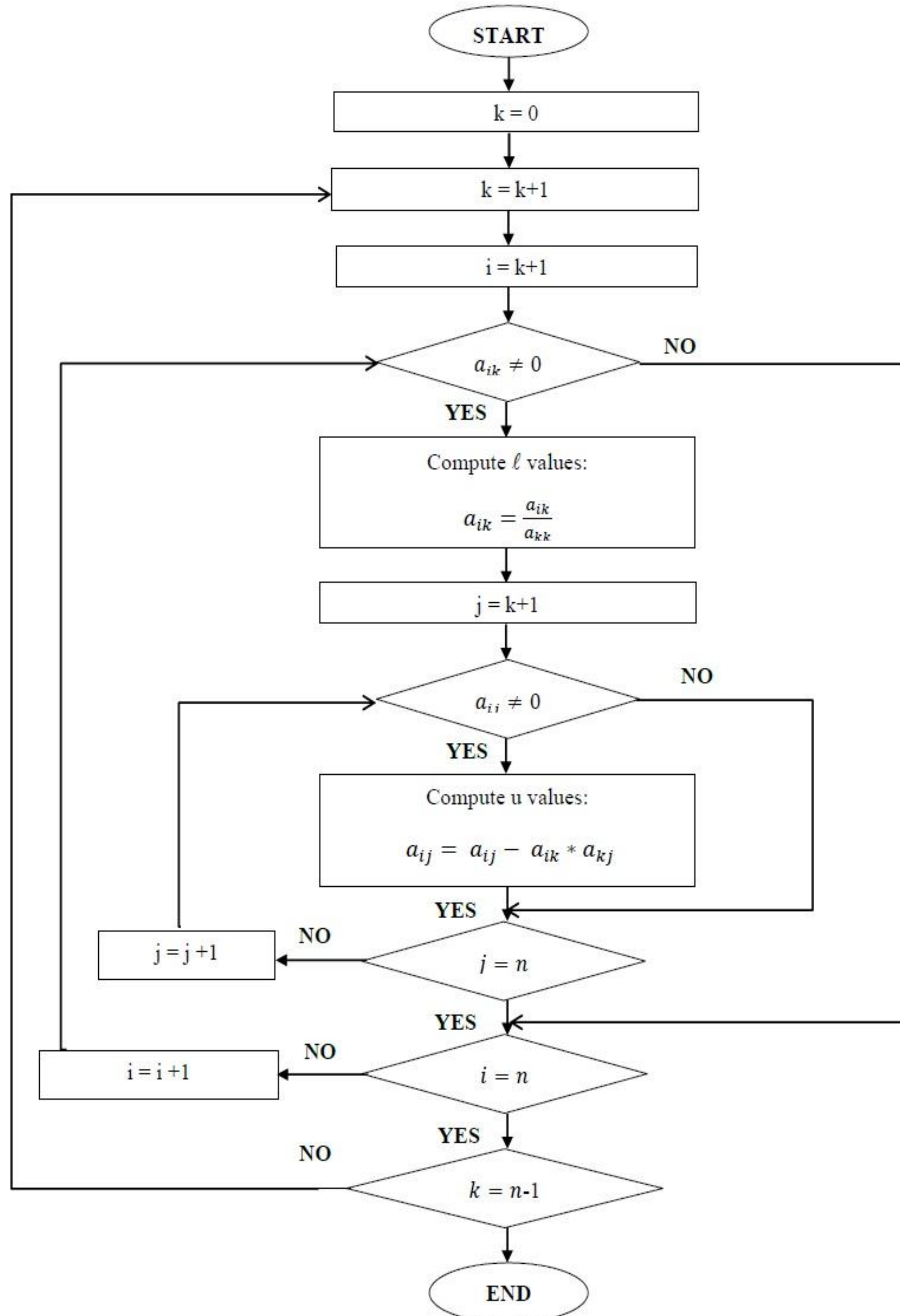


Figure 9: ILU (0) Factorization flowchart

2. SPAI (Sparse Approximate Inverse) Method

The SPAI algorithm used in this thesis is based on the one by Grote and Huckle [17]. Given sparse matrix $A \in \mathbb{R}^{nxn}$, the main idea of SPAI is to construct a sparse approximate inverse matrix $\boldsymbol{M \approx A^{-1}}$ which is the solution of Frobenius norm minimization:

$$\min_{M} \|AM - I\|_F^2 \tag{4.1}$$

The main advantage of SPAI is its inherent parallelism since the Frobenius norm can be split into sum of Euclidean norm:

$$\|AM - I\|_F^2 = \sum_{k=1}^{n} \|(AM - I)e_k\|_2^2 = \sum_{k=1}^{n} \|Am_k - e_k\|_2^2 \tag{4.2}$$

where m_k and e_k are the k-th columns of M and I respectively.

The solution of (4.2) separates into n independent least squares problems

$$\min_{m_k} \|Am_k - e_k\|_2 \; , k = 1, \ldots, n \tag{4.3}$$

The main difficulty is determining the sparsity pattern/structure of **M**, or else the iterations will become expensive. The SPAI algorithm starts with fixed initial sparsity pattern $\mathbf{J}_k$, e.g., diagonal sparsity pattern, then dynamically start augmenting the sparsity pattern of M (the approximate inverse) in an adaptive procedure to further reduce the residual $r_k = \|Am_k - e_k\|_2$. As discussed before, start with initial sparsity pattern $\mathbf{J} = \{ \mathrm{j} : \mathrm{m_k}(\mathrm{j}) \neq 0, \; \mathrm{j} = 1, \ldots, \mathrm{n}\}$ which contains set of indices which contains non-zero entries in column $\mathrm{m_k}(\mathrm{j})$ such that m_k is reduced into $\widehat{m}_k = m_k(\mathrm{J}) \; \in \mathbb{R}^{\boldsymbol{qx1}}$**.**

Let $\mathbf{I} = \{ \mathrm{i} : \sum_{j \epsilon J} |a_{ij}| \neq 0, \; \mathrm{i} = 1, \ldots, \mathrm{n}\}$ be set of indices which contains non-zero rows of A (: , **J**)) (not identically zero). **I** is also referred to as the shadow of **J.**

Remove zero entries from A,m and e to get

- $\widehat{\mathrm{A}} = \mathrm{A}(\mathrm{I}\,,\mathrm{J}\,) \in \mathbb{R}^{\boldsymbol{pxq}}$
- $\widehat{m}_k = m_k(\,\mathrm{J}\,) \in \mathbb{R}^{\boldsymbol{qx1}}$
- $\hat{e}_k = e_k(\,\mathrm{I}\,) \in \mathbb{R}^{\boldsymbol{px1}}$

Equation (4.3) is now reduced to

$$\min_{\widehat{m}_k} \left\|\hat{A}\widehat{m}_k - \hat{e}_k\right\|_2 \,, k = 1, \dots, n \tag{4.4}$$

In the original algorithm of SPAI, problem (4) is solved by QR decomposition, but since the reduced submatrix are all very small, normal equations will be used to solve the problem.

$$\widehat{m}_k = \left(\widehat{\mathrm{A}}(:, I)\right)^T (\,\widehat{\mathrm{A}}^T\widehat{\mathrm{A}}\,)^{-1} \tag{4.5}$$

After computing $\widehat{m}_k$, compute residual $\hat{\mathrm{r}}_\mathrm{k} = \hat{A}\widehat{m}_k - \hat{e}_k$ and its norm $\|\hat{\mathrm{r}}_\mathrm{k}\|_2 = \left\|\hat{A}\widehat{m}_k - \hat{e}_k\right\|_2$, then check if it is less than or equal to epsilon ϵ_{SPAI} which is the tolerance set by the user which controls level of fill in and quality of M.

If the residual norm is not less than or equal to epsilon, then we need to further decrease the approximate inverse residual by augmenting the sparsity pattern with new entries. This reduction in the residual is based on two steps:

- Identification set of new candidates that are not in the current sparsity pattern
- Selection of most profitable entries that will cause largest reduction in the residual

To do this, set $\mathbf{L_k}$ which contains set of indices ℓ for which $r_k(l) \neq 0$. In all cases $\mathrm{L_k} = \mathrm{I_k} \cup \{\mathrm{k}\}$ For each $\ell \in \boldsymbol{L_k}$, find an index Set N_ℓ which contains indices j of

non-zero elements of row A(ℓ,☺ that are not in **J** yet. The potential new candidates to be added in sparsity pattern **J** are grouped into

$$\bar{J}_k = \cup N_l \qquad (4.6)$$

From $\bar{J}_k$, we should select most profitable indices that will cause largest reduction in the residual norm. This selection is done by considering the univariate minimization problem:

$$\min_{m_{jk}} \left\| A\left(M_k + m_{jk}e_j\right) - e_k \right\|_2 = \min_{m_{jk}} \left\| r_k + m_{jk}Ae_j \right\|_2 \qquad (4.7)$$

With $Ae_j = A_j$, the solution of (4.7) is by derivate of (4.7) with respect to m_{jk}

$$\frac{d}{dm_{jk}} \left\| r_k + m_{jk}A_j \right\|_2^2 = 2r_k^T A_j + 2m_{jk} \left\| A_j \right\|_2^2 = 0$$

This will give

$$m_{jk} = -\frac{r_k^T A_j}{\|A_j\|_2^2} \qquad (4.8)$$

The second derivative of $\frac{d^2}{dm^2{}_{jk}} \left\| r_k + m_{jk}Ae_j \right\|_2^2 = 2\left\| A_j \right\|_2^2$ which is strictly positive, thus m_{jk} will minimize the new residual norm.

The norm ρ_j of the new residual obtained by adding new entry j in the sparsity pattern J_k becomes:

$$\rho_j{}^2 = \left\| r_k + m_{jk}A_j \right\|_2^2 = r_k^T r_k + 2m_{jk} r_k^T A_j + m_{jk}^2 A_j^T A_j$$

$$= \|r_k\|_2^2 - 2\frac{(r_k^T A_j)^2}{\|A_j\|_2^2} + \frac{(r_k^T A_j)^2}{\|A_j\|_2^4} \left\| A_j \right\|_2^2$$

$$= \|r_k\|_2^2 - \frac{(r_k^T A_j)^2}{\|A_j\|_2^2} \qquad (4.9)$$

For each $j \in \bar{\mathrm{J}}_\mathrm{k}$, calculate the norm ρ_j of the new residual. After calculating al residual norms, calculate their averages then delete from $\bar{\mathrm{J}}_\mathrm{k}$ all but most profitable indices based on $\rho_\mathrm{j} < \bar{\rho}_\mathrm{k}$ where $\bar{\rho}_\mathrm{k}$ is the mean of norms. To prevent dense inverse, from the reduced $\bar{\mathrm{J}}_\mathrm{k}$, keep only β number of indices which gives the largest reduction in residual.

Next we update J_k with new indices in $\bar{\mathrm{J}}_\mathrm{k}$ such that $\mathrm{J}_\mathrm{k} = \mathrm{J}_\mathrm{k} \cup \bar{\mathrm{J}}_\mathrm{k}$, then calculate $\widehat{m}_k$ again along with its residual until reaching required tolerance epsilon or maximum number of update iterations α is reached. Note that $\mathrm{J}_\mathrm{k} \cap \bar{J}_k = \emptyset$.

After applying SPAI algorithm and getting each column of approximate inverse M independently, apply smoothing step

$$x^{(k+1)} = x^{(k)} - M(Ax^{(k)} - b) \qquad (4.10)$$

Note that above algorithm of SPAI yields right preconditioner. To get left preconditioner we shall use row version of SPAI where (1) becomes

$$\min_{\mathrm{M}} \|MA - I\|_F^2 \qquad (4.11)$$

In this thesis, row version of SPAI is implemented, and the resulted m_k is the kth row of approximate inverse M instead of column.

The above algorithm of SPAI is called SPAI(ϵ); however, Broker et al.[61,62] introduced two simplified forms of SPAI with fixed sparsity patterns in their paper :which are SPAI-0 and SPAI-1:

- **SPAI-0**: M=diag ($\mathrm{m_{kk}}$) is diagonal with $m_{kk} = \frac{a_{kk}}{\|\mathrm{a_k}\|_2^2}$, $1 \le k \le n$
- **SPAI-1**: The sparsity pattern J (M) = J (A)

Note that no pattern update is needed for SPAI-0 and SPAI-1; thus their implementation is simpler than SPAI(ϵ); however, they can be used as initial guess for SPAI(ϵ)

Below are the steps for SPAI algorithm (for every column m_k of M):

- Input user defined parameters such as initial sparsity pattern $\mathbf{J_k}$ (It's more efficient to start with diagonal sparsity pattern), tolerance ϵ_{SPAI},α, and β.
- Determine row indices $\mathbf{I_k}$ that correspond to non-zero rows in submatrix A(:, $\mathbf{J_k}$)
- Construct submatrix $\widehat{A} = A(\mathbf{I}_k, \mathbf{J}_k)$, $\widehat{m}_k = m_k(\mathbf{J}_k)$ and $\hat{e}_k = e_k(\mathbf{I}_k)$ such that Least squares problem is reduced to $\min_{\widehat{m}_k} \left\|\hat{A}\widehat{m}_k - \hat{e}_k\right\|_2$, $k = 1, \ldots, n$
- Calculate $\widehat{m}_k$ by normal equations $\widehat{m}_k = \left(\widehat{A}(:, I)\right)^T (\widehat{A}^T\widehat{A})^{-1}$
- Compute residual $\hat{r}_k = \hat{A}\widehat{m}_k - \hat{e}_k$ and its norm $\|\hat{r}_k\|_2 = \left\|\hat{A}\widehat{m}_k - \hat{e}_k\right\|_2$
- If $\|\hat{r}_k\|_2 < \epsilon_{SPAI}$ then break, else for iteration 1 to α :
- Set $\mathbf{L_k}$ which contains set of indices ℓ for which $r_k(l) \neq 0$ ($L_k = I_k \cup \{k\}$)
- For each $\ell \in \mathbf{L}_k$: find an index Set N_ℓ which contains indices j of non-zero elements of A(ℓ,:) that are not in $\mathbf{J_k}$ yet.
- Set $\bar{J}_k = \cup N_l$
- For each $j \in \bar{J}_k$, calculate the norm ρ_j of the new residual :
- $\rho_j = (\|\hat{r}_k\|_2^2 - \frac{[\hat{r}_k^T A_j(I_k)]^2}{\|A_j(I_k)\|_2^2})^{1/2}$

- Delete from $\bar{J}_k$ all but most profitable indices based on $\rho_j < \bar{\rho}_k$ where $\bar{\rho}_k$ is the mean of norms

- From the reduced $\bar{J}_k$, keep only β number of indices which gives the largest reduction in residual.

- Update J_k with new indices in $\bar{J}_k$ $\quad (J_k = J_k \cup \bar{J}_k)$

- Go back to step 2 and repeat until either max number of iterations α is reached or $\|\hat{r}_k\|_2 < \epsilon_{SPAI}$

- Assemble column $\boldsymbol{m_k}$ *from* $\boldsymbol{\hat{m}_k}$

- Apply smoothing step $\quad x^{(k+1)} = x^{(k)} - M(Ax^{(k)} - b)$

Where ϵ_{SPAI} is the tolerance set by the user which controls level of fill in and quality of M (between 0 & 1), $\alpha \geq 0$: maximum number of pattern update iterations to limit the fill-in per column, and $\beta \geq 0$: maximum number of indices j to be added to **J.**

Note that we have implemented all versions of SPAI (SPAI-0, SPAI-1 and SPAI(ϵ) in uFVM MATLAB and run many test cases to find out it is really demanding and requires high computational cost. This is mainly due to the search of new candidates then the computation of reduction in the residual that each candidate will cause. Moreover SPAI algorithm needs to solve highly dense least square problems which make these two steps very expensive and require a lot of iterations to search and select, so we did not consider SPAI in OpenFOAM.

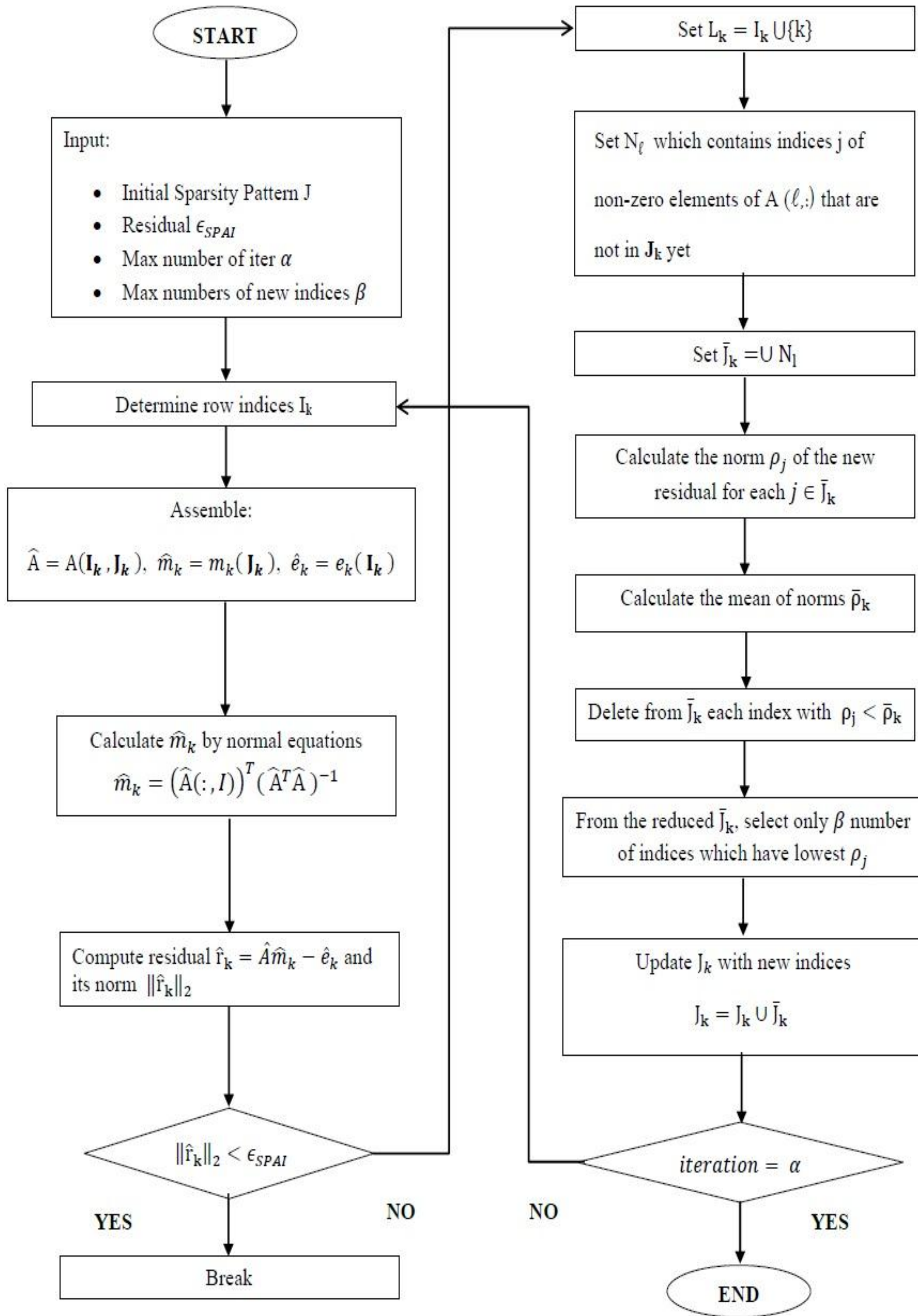


Figure 10: SPAI algorithm flowchart

3. AINV (Approximate Inverse) Method

First let us discuss the original basic inner-product version of AINV method developed by Benzi and Tuma [24]. Given sparse matrix A $\in \mathbb{R}^{nxn}$ **,** this method is based on incomplete inverse factorizations which mean incomplete factorizations of A^{-1}. If the factorization of A=LDU where L is unit lower triangular matrix, D is diagonal matrix, and U is unit upper triangular matrix, then $A^{-1} = U^{-1}D^{-1}L^{-1} = ZD^{-1}W^T$ where $Z = U^{-1}$ and $W = L^{-T}$ are unit upper triangular matrices.

The approach of Benzi and Tuma does not require any information of triangular factors of A (no factorization of A) and the AINV is constructed directly from A and no need for sparsity pattern to be known in advance.

The inverse factors Z & W will be dense, so in order to preserve sparsity pattern of A, factorized sparse approximate inverse $\boldsymbol{M \approx A^{-1}}$ will be constructed instead. If $\bar{Z} \approx Z$ *and* $\bar{W} \approx W$ the resulting factorized approximate inverse M would be $\boldsymbol{M = \bar{Z}\bar{D}^{-1}\bar{W}^T \approx A^{-1}}$ **.**

AINV preconditioner is based on an algorithm which computes two sets of vectors$\{z_i\}_{i=1}^n$, $\{w_i\}_{i=1}^n$, which are A-biconjugate such that $w_i^T A z_j = 0$ iff i ≠ j.

If Z= [z_1, z_2….z_n] is matrix whose ith column z_i and W= [w_1, w_2….w_n] is matrix whose ith column w_i then

$$\mathbf{W^TAZ = D} = \begin{bmatrix} p_1 & 0 & \cdots & 0 \\ 0 & p_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & p_n \end{bmatrix} \tag{4.12}$$

Where

$$p_i = w_i^T A z_j \neq 0 \tag{4.13}$$

This gives

$$M \approx A^{-1} = ZD^{-1}W^T = \sum_{i=1}^{n} \frac{z_i w_i^T}{p_i} \qquad (4.14)$$

The approximate inverse of A can be known if two incomplete sets of A-biconjugate vectors are known, but there are infinitely many such sets. Matrices W and Z will be explicitly computed by bi-conjugation process applied to columns of two non-singular matrices $W^{(0)}$ and $Z^{(0)}$.So a computationally convenient choice is to let $W^{(0)}$= $Z^{(0}$=I.

To avoid dense inverse and preserve the sparsity of A, incomplete bi-conjugation process will be applied based on drop tolerance where new fill-in entries are dropped if their absolute magnitude in less than prescribed tolerance Tol (0<Tol<1) or approximate inverse will have pre-specified sparsity pattern.

The inner product version of AINV algorithm is as below:

- Set W & Z as Identity matrix I
- For i=1,2,…n :
- For j= i, i+1…n:
- Compute $p_j = a_i^T z_j \;\; and \;\; q_j = c_i^T w_j$ where

$a_i^T \; and \; c_i^T \;\; are \; the \; ith \; row \; of \; A$ & A^T

- End for
- If i = n go to step 10
- Orthogonalize z & w by subtracting multiple columns of z from ith

column of z (same for w) $\boldsymbol{z_j = z_j - \frac{p_j}{p_i} z_i}$; $\boldsymbol{w_j = w_j - \frac{q_j}{q_i} w_i}$

- Apply dropping to preserve sparsity pattern
- End for

- Set Z= [z_1, z_2….z_n] and W= [w_1, w_2….w_n] that are calculated in above steps
- Compute $x \approx M^{-1}b = ZD^{-1}W^Tb = \sum_{i=1}^{n}(\frac{w_i^T b}{p_i})z_i$

Since the coefficient matrix is sparse (contains too many zeros), the inner product at step 4 of the above algorithm can be computationally expensive. These inner products are often zero. To avoid this, we shift to outer-product form developed by Bridson and Tang [26]. Their algorithm has same concept as that of Benzi but with switching the order of the loops. Let consider the vectors *l* and *u* the j'th column of row *LD* and *DU* respectively. Thus the algorithm of outer product form of AINV would become as below.

The outer product version of AINV algorithm is as below:

- Take as input the Coefficient Matrix A and drop Tolerance δ
- Set W & Z as Identity matrix I
- For i=1,2,…n :
- Compute $l = AZ_i$
- Compute $u = W_i^T A$
- Compute $D_{ii} = uZ_i$ **or** $D_{ii} = W_i^T l$
- For j > i
- Orthogonalize z & w by subtracting multiple columns of z from ith column of z (same for w) $\boldsymbol{Z_j = Z_j - drop(\frac{u_j}{D_{ii}}Z_i, \delta); W_j = W_j - drop(\frac{l_j}{D_{ii}}W_i, \delta)}$
- Apply dropping to preserve sparsity pattern with magnitude below δ are dropped.
- End for

- End for
- Set Z= [z_1, z_2….z_n] and W= [w_1, w_2….w_n] that are calculated in above steps.
- Compute $x \approx M^{-1}b = ZD^{-1}W^T b$

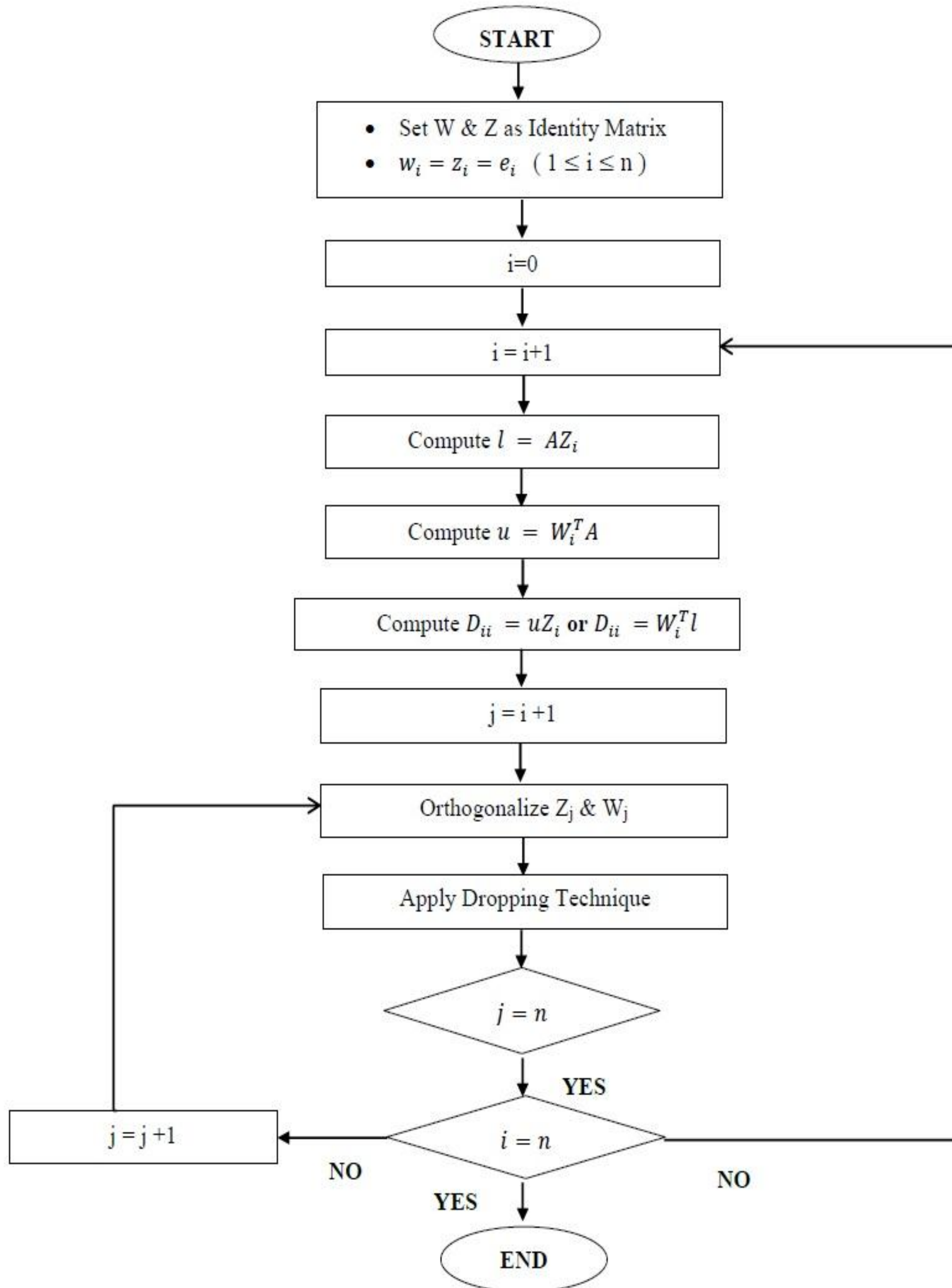


Figure 11: The outer-product form of AINV algorithm flowchart

In this work, we didn't apply the dropping techniques strategy; however, we computed the approximate inverse with sparsity pattern same as that of coefficient matrix A. In this case, while comparing performance of ILU0 and AINV, both smoothers would have same sparsity pattern to that of A.

B. ORTHOMIN Method

In this work, we are going to adopt ORTHOMIN algorithm developed by Vinsome [71] to accelerate the convergence of the smoothers. Two important techniques are used, within this method, which are minimization and orthogonalisation. ILU and AINV use fixed point iteration where a solution is improved by calculating the correction using previous iteration results. However the correction vector is calculated in certain directions and not in others. So in order to accelerate the convergence after applying the smoother, an orthogolnization technique embedded in ORTHOMIN is needed to eliminate or subtract the spaces of the correction vectors which has already been computed and increasing the chance of producing correction vectors in directions that these smoothers cannot produce. Orthomin will use "k" previous orthogonal vectors p to construct new vector such that Ap is orthogonal to all previous k Ap vectors.

ORTHOMIN(k) algorithm steps are shown below:

- Compute residual $r_0 = b - Ax_0$
- Set $p_0 = M^{-1} r_0$ (where M is preconditioned matrix of A)
- For i=1,2,…n :
- Compute minimization parameter $\boldsymbol{\alpha_j} = \frac{(\boldsymbol{r_j}, \boldsymbol{Ap_j})}{(\boldsymbol{Ap_j}, \boldsymbol{Ap_j})}$
- Compute $\boldsymbol{x_{j+1}} = \boldsymbol{x_j} + \boldsymbol{\alpha_j}\, \boldsymbol{p_j}$
- Compute $\boldsymbol{r_{j+1}} = \boldsymbol{r_j} - \boldsymbol{\alpha_j}\, \boldsymbol{Ap_j}$

- Compute orthogonality coefficients $\boldsymbol{\beta_{ij}} = -\frac{(\boldsymbol{AM^{-1}r_{j+1}}, \boldsymbol{Ap_i})}{(\boldsymbol{Ap_i}, \boldsymbol{Ap_i})}$, for

$\boldsymbol{i = j - k + 1, .. j}$ where k is number of previous vectors

- Compute $\boldsymbol{p_{j+1}} = \boldsymbol{M^{-1}r_{j+1}} + \sum_{\boldsymbol{i=j-k+1}}^{\boldsymbol{j}} \boldsymbol{\beta_{ij} p_i}$
- End for

C. Block Methods

In 3D fluid flow problems, the variables need to be solved are mainly the velocity fields in x,y and z directions as well as the pressure field by discretizing and solving the momentum and continuity equations. The solution procedure can be divided into segregated and coupled. In segregated solver, the continuity and momentum equations are decoupled and solved sequentially [72]. The individual momentum equations are solved for each velocity component then continuity equation is solved for pressure. Although segregated solvers require less memory, they have low convergence rate and need more iterations. In contrast, the coupled or block algorithms have high rate of convergence on the expense of high memory since all velocity components and pressure fields are solved at the same time or simultaneously where all discretized equations are solved in one system [73] .As shown in below equations where the system of algebraic equation at centroid of one cell between segregated and coupled are compared.

$$a_C \emptyset_C + \sum_{NB} a_F \emptyset_F = b_C \tag{4.15}$$

where $\emptyset$ is the variable to be solved (u,v,w or p)

While for coupled system [74], the system of algebraic equations at centroid of one cell would be:

$$\begin{bmatrix} a_C^{uu} & a_C^{uv} & a_C^{uw} & a_C^{up} \\ a_C^{vu} & a_C^{vv} & a_C^{vw} & a_C^{vp} \\ a_C^{wu} & a_C^{wv} & a_C^{ww} & a_C^{wp} \\ a_C^{pu} & a_C^{pv} & a_C^{pw} & a_C^{pp} \end{bmatrix} \cdot \begin{bmatrix} u_C \\ v_C \\ w_C \\ p_C \end{bmatrix} + \sum_{NB} \begin{bmatrix} a_{NB}^{uu} & a_{NB}^{uv} & a_{NB}^{uw} & a_{NB}^{up} \\ a_{NB}^{vu} & a_{NB}^{vv} & a_{NB}^{vw} & a_{NB}^{vp} \\ a_{NB}^{wu} & a_{NB}^{wv} & a_{NB}^{ww} & a_{NB}^{wp} \\ a_{NB}^{pu} & a_{NB}^{pv} & a_{NB}^{pw} & a_{NB}^{pp} \end{bmatrix} \cdot \begin{bmatrix} u_{NB} \\ v_{NB} \\ w_{NB} \\ p_{NB} \end{bmatrix} = \begin{bmatrix} b_C^u \\ b_C^v \\ b_C^w \\ b_C^p \end{bmatrix} \quad (4.16)$$

As can be seen from Eq. (4.16) the coefficient matrix coefficients a_c and a_{NB} are now tensors in coupled system instead of scalars as in segregated systems. As such we will present how ILU(0) and AINV block (coupled) version would become.

1. Block ILU (0) (Incomplete LU Factorization) Method

The block version is the same as segregated; however, the main differences are that scalars become tensors and division would be replaced by inverse as shown below:

- For k=1,2,…n:
- Compute Residual $R^{(k)} = b - Ax^{(k)}$
- For i=k+1,…n:
- Compute ℓ values $A_{ik}A_{kk}^{-1}$
- Solve $y^{(k)} = L^{-1}r^{(k)}$ by forward substitution
- For j=k+1,…n:
- Compute *u* values $A_{ij} = A_{ij} - A_{ik}A_{kj}$
- Solve $d^{(k)} = U^{-1}y^{(k)}$ by backward substitution
- Update solution Solve $x^{(k+1)} = x^{(k)} + d^{(k)}$
- Repeat until $\|r^{(k)}\|_2 \le \varepsilon$ where ε is tolerance

2. Block AINV Method

For block version of AINV, tensors *L* and *U* the j'th block column of row *LD* and *DU* respectively. The algorithm would become as:

- Take as input the Coefficient Matrix A and drop Tolerance δ
- Set W & Z as Identity matrix I
- For i=1,2,…n :
- Compute block $L = AZ_i$
- Compute block $U = W_i^T A$
- Compute block $D_{ii} = UZ_i$ **or** $D_{ii} = W_i^T L$
- For j > i
- Orthogonalize Z & W by subtracting multiple block columns of z from

ith column of z (same for w) $\boldsymbol{Z_j = Z_j - Z_i\,(D_{ii}^{-1}U_j)}$ **;** $\boldsymbol{W_j = W_j - W_i\,(L_j D_{ii}^{-1})}$

- Apply dropping to preserve sparsity pattern with norm of $\boldsymbol{Z_j}$ or $\boldsymbol{W_j}$ below

δ are dropped

- End for
- End for
- Set Z= [z_1, z_2….z_n] and W= [w_1, w_2….w_n] that are calculated in above

steps

- Compute $x \approx M^{-1}b = ZD^{-1}W^T b$

CHAPTER V

TEST CASES AND RESULTS

In this section, we will present and describe the test cases used to evaluate and compare the performance of each smoother. The test cases setup and run were conducted on OpenFOAM using segregated and coupled fluid flow solver. We run the two test cases on OpenFOAM using segregated solver for first test case and coupled one for the second. For each test case, we compare the efficiency and robustness of each smoother by showing the residual convergence rate for each smoothers and calculating how much the residual is decreased at each time step, the number of iterations needed by each smoother to achieve this decrease. In addition we will present the CPU convergence time needed to achieve convergence. We focus on the pressure equation since it is of an elliptic type and is critical for the solution and convergence of whole system.

A. Flow over a Stator Blade (Compressible-Segregated)

The first test case is based on the experimental setup of Hylton et al. [75]. In their study, they investigated two aero-thermodynamic linear cascades. The NASA-C3X cascade experiment, made up of three linear cascade vanes, was chosen as representative of the first stator stage of a gas turbine.

The shape of the blade of C3X in the original setup is prismatic which allow us to use periodic and symmetry boundary conditions. Furthermore, a reduction is applied

to quasi 3D of the domain, shown in Figure 12, in which the 2D section is discretized with only one cell in the span direction of the blade.

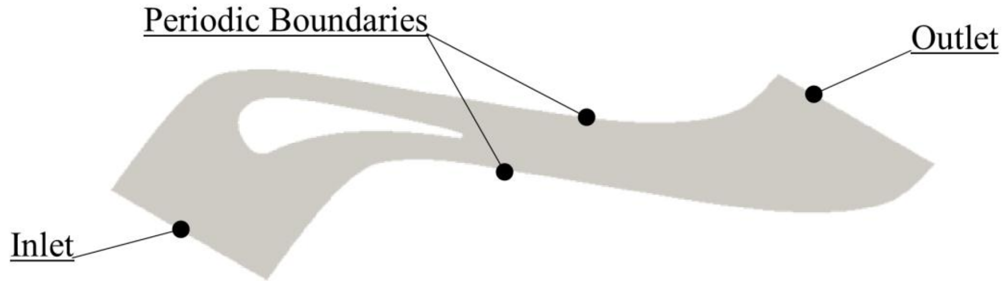


Figure 12: Computational domain and boundary conditions [76]

Table 1 summarizes the reference boundary conditions which are based on a particular operating point reported by Hylton et al. (code n° 4422, run n° 112), with an isentropic exit Mach number of 0.9.

Table 1: Stator Vane Data [76]

Inlet Total Pressure	P_0	321800	*Pa*
Inlet Total Temperature	T_0	783	*K*
Dissipation Length	L_d	0.001	*m*
Turbulence intensity	Tu	4%	[-]
Outlet Static Pressure	P_{out}	192500	*Pa*

As shown in Figure 13, the computational grid is made of multi-blocks O-type grid, resulting in a mesh with size of 14,500 hexahedral elements. In addition, Figure 13 also shows that the grid fully resolves the boundary layer close to the walls of the blade and the wake. This test case, despite of its moderate mesh size, is still very demanding due to the very high anisotropic mesh used which yields elements of aspect ratios reaching a maximum of 30,000, and resulting in a very stiff system of equations [76].

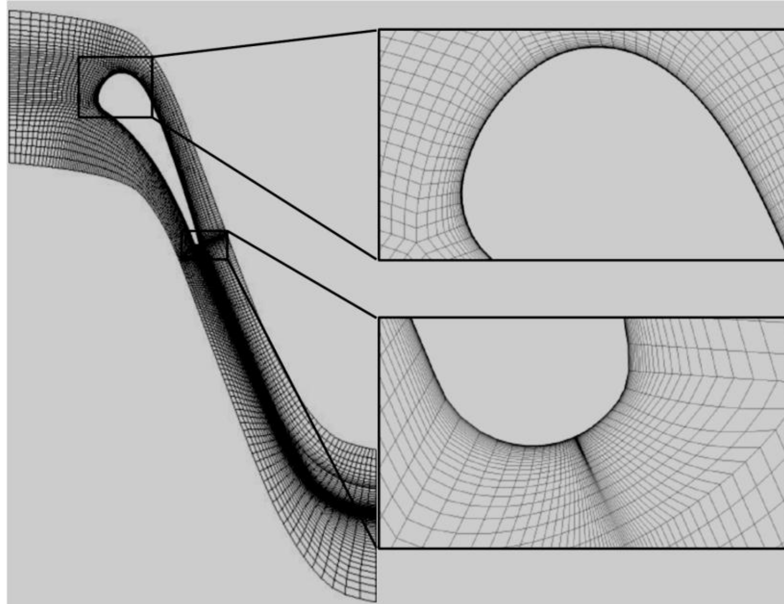


Figure 13: Computational grid for the flow over a blade problem [76]

Segregated compressible flow solver was used to solve this problem. The conservation equations discretized in this test case are the continuity [Eq. (1.2)], momentum [Eq. (1.3)] and energy [Eq. (1.3)] equations. In addition, the turbulence model used is modified k-ω Shear Stress Transport (SST) [77] model with automatic wall treatment.

Figure 14 shows the convergence rate residual with each smoother versus time. It can be noticed the high agreement between ILU (0) and AINV in the convergence rate and time/iterations needed. In the next graphs, the Residual of pressure before and after applying each smoother is presented. In Figure 15, it can be seen how the residual is consistently decreasing with time. After applying the ILU (0) smoother, the residual is locally decreasing with an average of 27 times the initial residual with standard deviation of 5. This shows how ILU (0) is stable. By comparing this residual decrease to that of AINV in Figure 16, the pressure residual is decreasing with a mean value of 25

times with standard deviation of 4. This shows how AINV is quite effective as ILU (0) in reaching convergence over time with nearly same robustness and stability.

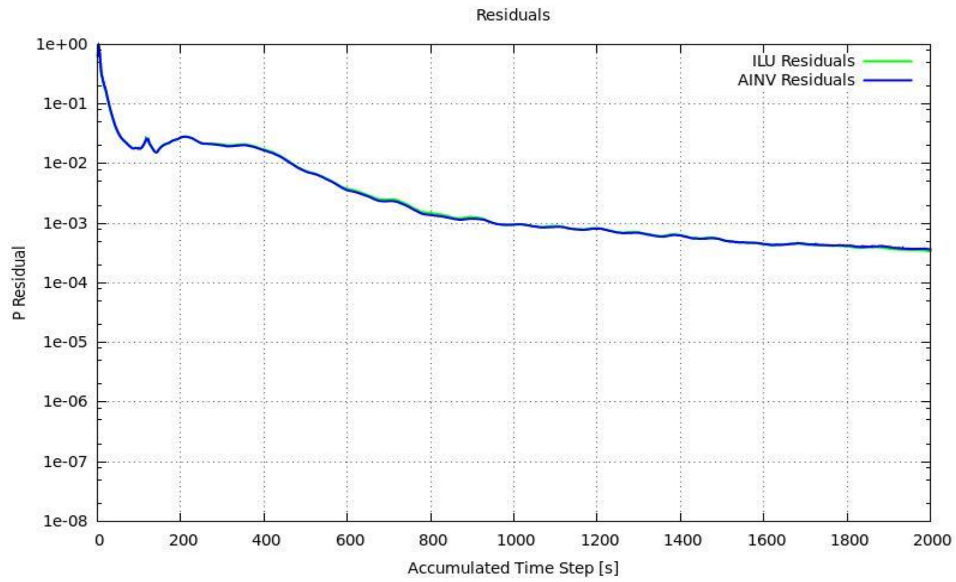


Figure 14: Final Residual vs Time for Smoothers

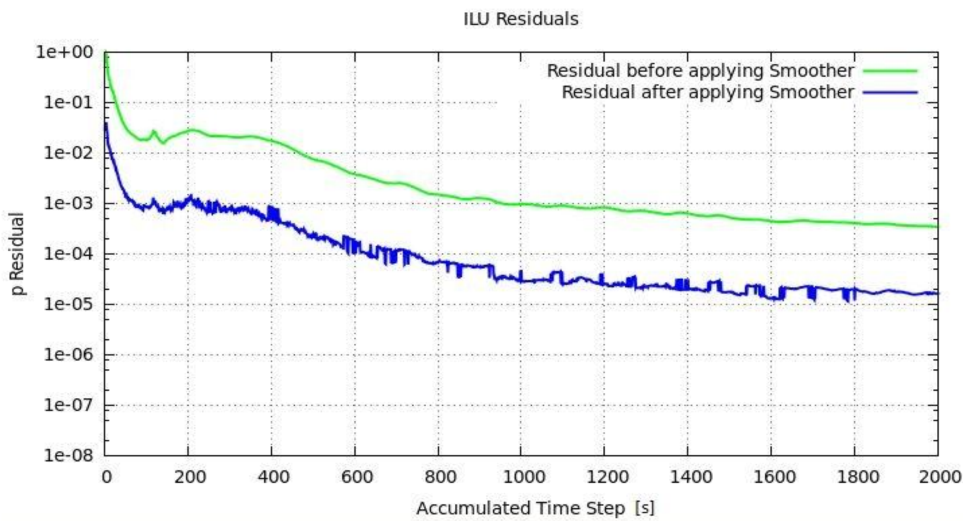


Figure 15: Initial & Final Residual using ILU (0)

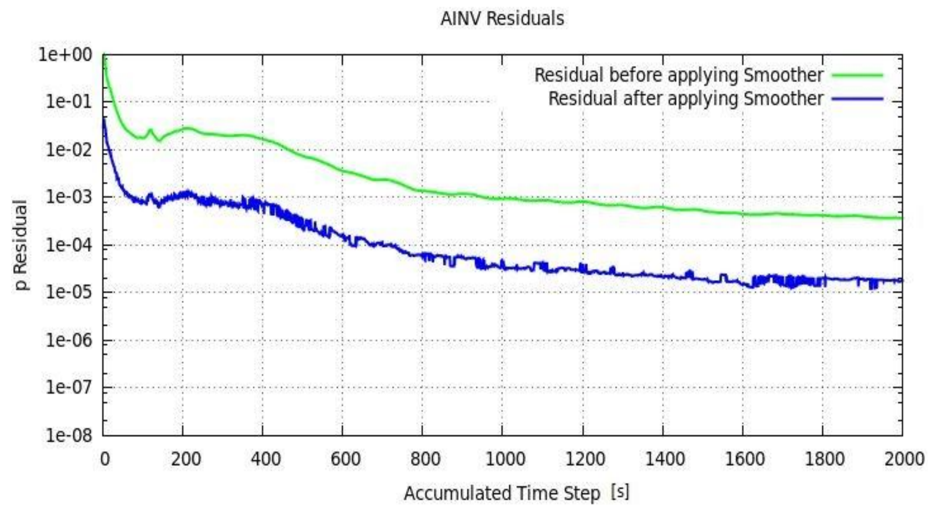


Figure 16: Initial & Final Residual using AINV

In addition to how much the residual is decreasing with time, it is also important to monitor the number of iterations needed by each smoother to reach the final residual. As shown in Figures 17 and 18, ILU (0) and AINV start with high number of iterations in each time step then decrease to almost reach a steady number of iterations. ILU (0) requires minimum 4 iterations and maximum of 10 with a mean value of 5 iterations in each time step to lower the residual by order of 27 as mentioned earlier, while the AINV smoother requires a little more to reach minimum of 5 and maximum of 14 iterations with mean value of 6 iterations. This clearly shows how robust these two smoothers are with performance of AINV almost reaching that of ILU (0).

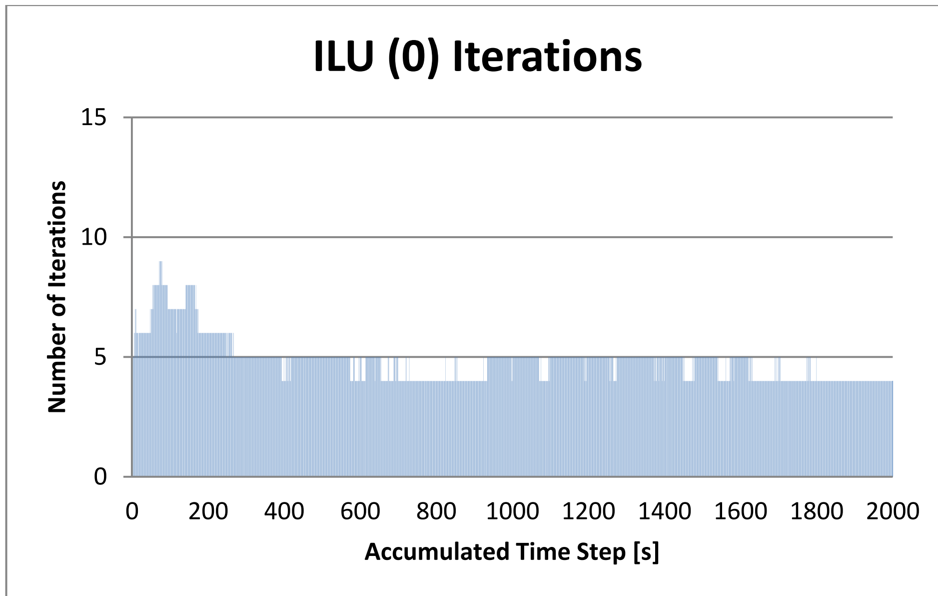


Figure 17: Number of Iterations vs Time of ILU (0)

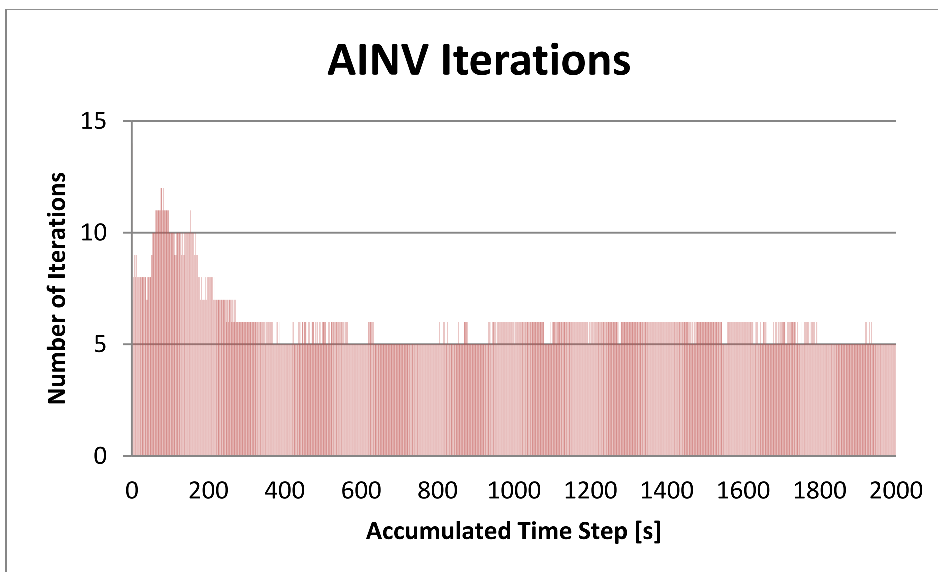


Figure 18: Number of Iterations vs Time of AINV

Table 2 summarizes the residual decrease after applying each smoother, number of iterations needed as well as CPU time. It is clearly shown how the

performance of the two smoothers in this test case, which has high aspect ratio and causes an ill-conditioned matrix, is nearly identical with slight difference.

Table 2: Smoothers Comparison

Smoother	Avg. of Res. Decrease	Standard Dev. Of Res Decrease	Avg. # of iterations	CPU Convergence Time (sec)
ILU (0)	27	5	5	216.96
AINV	25	4	6	238.03
Error (%)	7.4	-	-	9.71

B. 90° Pipe Bend (Incompressible-Coupled)

The second test case is based on an interesting field, which is turbulent fluid flow through curved pipes and channels, has been investigated theoretically and experimentally for decades [78, 79]. In addition, with high computing power, numerical investigations have been developed recently to include unsteady techniques [80].

Turbulent flow through 90° pipe bend is of great importance since it tackles important features in turbulence modeling. Some of these features are geometry induced pressure gradients, longitudinal streamline curvatures and many others [81].

The test case in interest is a 90° pipe bend with circular cross section as shown in Figure 19. The boundary conditions are of one water inlet and one outlet with no slip condition on the walls and symmetry on the bottom wall for computational efficiency.

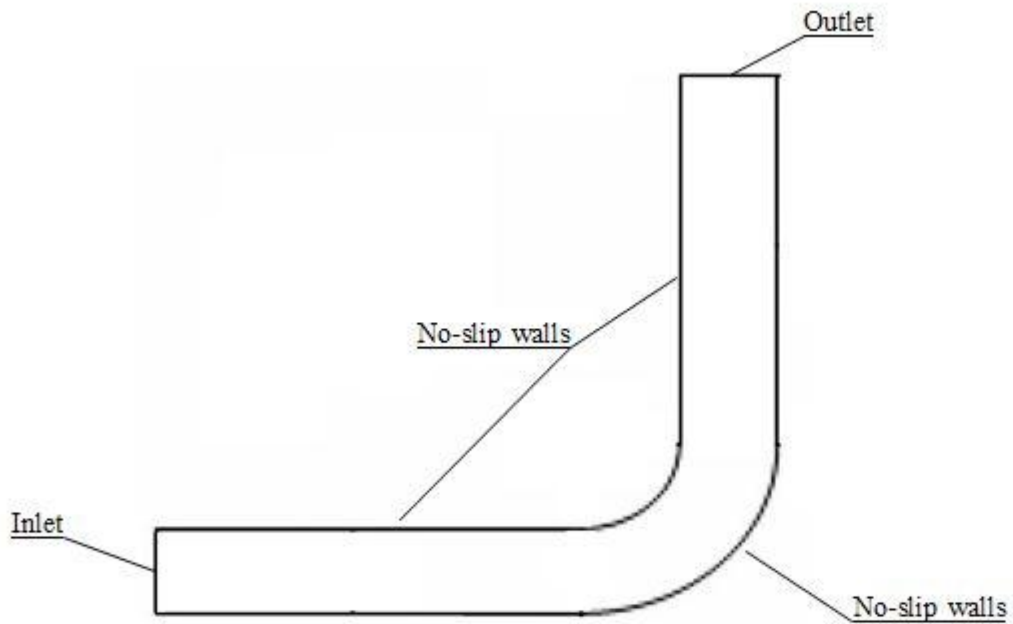


Figure 19: Computational domain and boundary conditions

Table 3 summarizes the initial and boundary conditions of this test case.

Table 3: Initial and Boundary Conditions

Inlet Pressure	P_{inlet}	Zero Gradient	*Pa*
Inlet Velocity	V_{inlet}	1.67	*m/sec*
Outlet Pressure	P_{outlet}	10000	*Pa*
Outlet Velocity	V_{outlet}	Zero Gradient	*m/sec*

As shown in Figure 20, the computational grid is made of a mesh with size of 1.5 million hexahedral elements. This test case is also of high importance due to high anisotropic mesh which results in elements of very high aspect ratio reaching maximum of 478174 which is considered very large and will definitely yields a stiff system of equations with an ill-conditioned matrix.

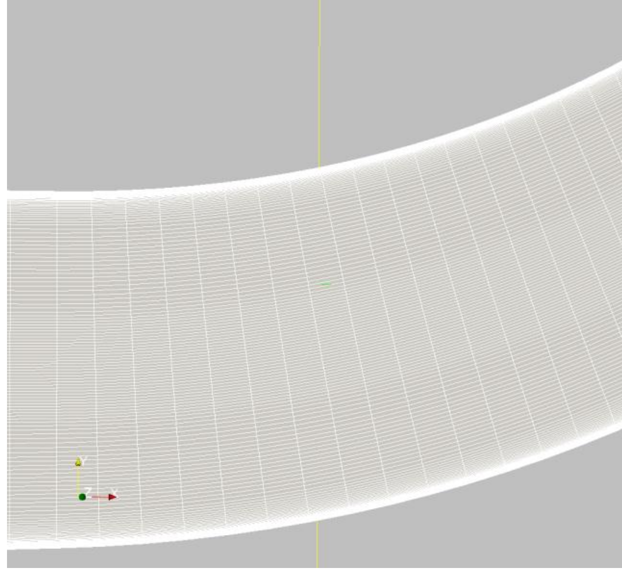


Figure 20: Top View of Mesh on the Pipe Symmetry Plane

Coupled incompressible flow solver was used to solve this problem. The conservation equations discretized in this test case are the continuity [Eq. (1.2)], momentum [Eq. (1.3)] and energy [Eq. (1.3)] equations. In addition, the turbulence model used is k-ω .

Since Coupled Solver will solve the variables of momentum and continuity equations which are velocity components and pressure simultaneously, we present the residual convergence rate of each variable versus time step for both smoothers. Figure 21 shows the residual convergence rate of pressure versus time step for both smoothers. It can be noticed how Block ILU (0) is consistently converging and decreasing the residual without any fluctuations in a stable manner; whereas, Block AINV causes the residual to decrease at first at a higher rate than Block ILU (0) with some fluctuation then after some time steps the residual increases causing the solution to diverge. Figure 22 clearly shows how Block ILU (0) is steadily decreasing the Velocity residuals for convergence while Block AINV is decreasing the velocity residuals slowly with a lot of time and iterations and result in divergence.

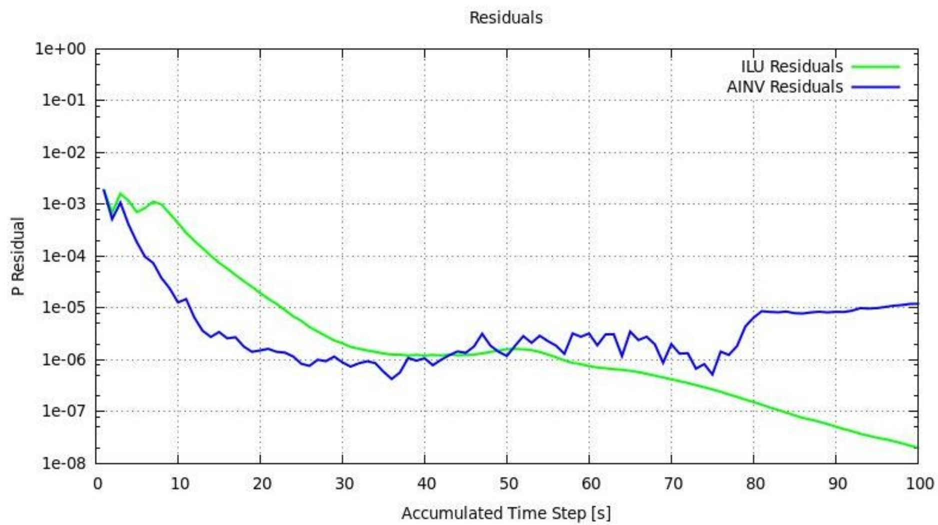


Figure 21: Pressure Residual vs Time for Block Smoothers

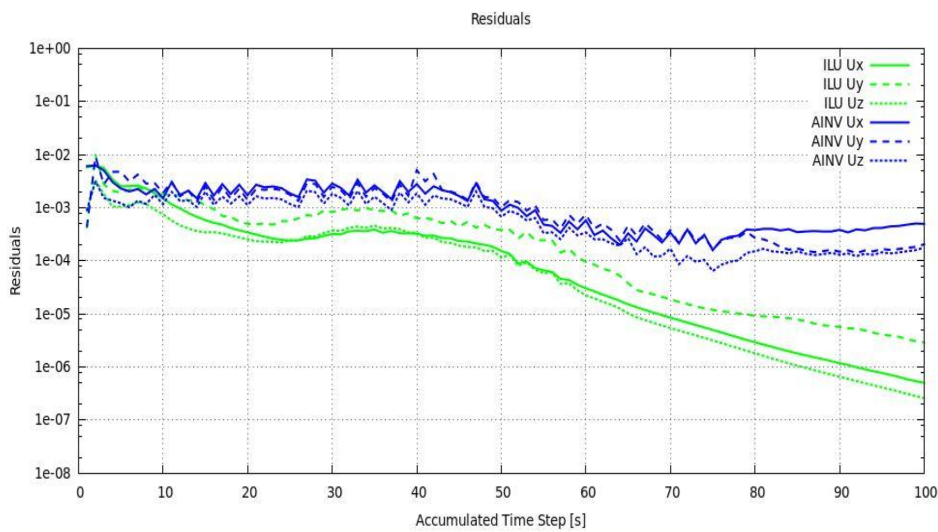


Figure 22: Velocity Residual vs Time for Block Smoothers

By calculating the number of iterations each smoother required at each time step do decrease the residual by a certain tolerance, Figure 23 shows how the Block ILU (0) almost keep the same number of iterations in each time step with minimum of 3 and maximum of 5 iterations with mean value of 4 iterations, while Block AINV in Figure 24 requires a lot more in each time step with minimum of 3 and maximum of 17

iterations with mean value of 7 iterations and with a lot of fluctuations. This clearly reflects the divergence that occurred with Block AINV Smoother.

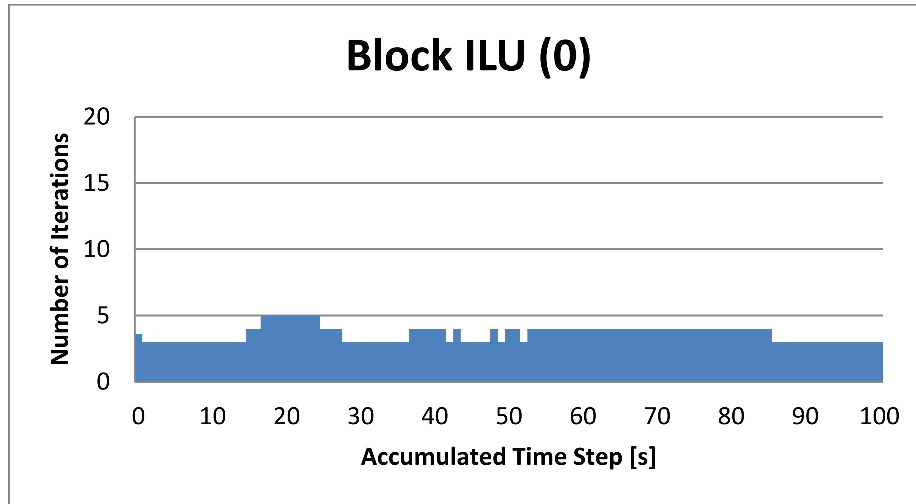


Figure 23: Number of Iterations vs Time Step of Block ILU (0)

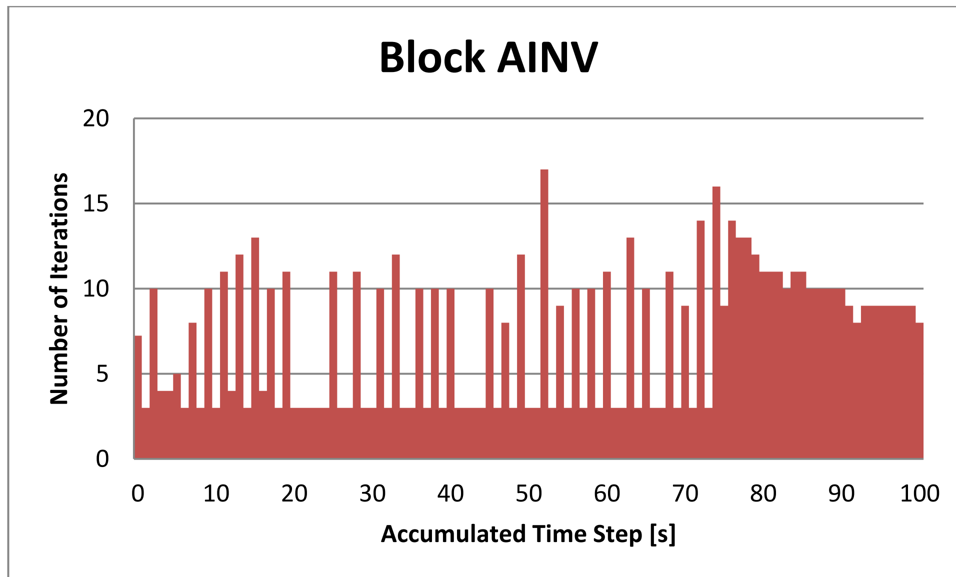


Figure 24: Number of Iterations vs Time Step of Block AINV

Table 4 summarizes the number of iterations needed by each smoother as well as CPU time. It is clearly shown how the Block ILU(0) smoother in this test case

outperforms Block AINV requiring less iterations and CPU time and being more robust, while Block AINV diverges requiring more iterations and almost twice the CPU time.

Table 4: Block Smoothers Comparison

Smoother	Avg. # of iterations	CPU Convergence Time (sec)
ILU (0)	4	2530
AINV	7	4500
Error (%)	-	77.86

CHAPTER VI

CONCLUSION

Two algebraic Multigrid smoothers were presented and compared. AINV in scalar and block version was implemented in OpenFOAM and compared to already existing ILU (0) smoother. The two smoothers were tested for two demanding test cases with turbulent flow having anisotropic mesh resulting in elements with very high aspect ratio and yielding a stiff system of equations.

Results clearly show that AINV has same performance as ILU (0) in segregated solver, whereas it does not perform well compared to Block ILU (0) in coupled solver as shown by Bridson and Tang [26] where they used Block AINV as preconditioner and failed to converge in many cases. This may due to many inverse operations required by Block AINV. In addition no drop tolerance was used. The implementation of drop tolerance might result in a better smoother.

Improvements adopted for future work include drop tolerance implementation for Scalar and Block AINV thus increasing its robustness.

APPENDIX

A. Software Description

Algorithms were first developed and tested on a MATLAB based program developed at the American University of Beirut denoted by “uFVM”, then coded into an open source code called OpenFOAM. Both will be described in following subsections.

1. uFVM (MATLAB)

uFVM is a computational fluid dynamics code implemented in MATLAB that uses finite volume method over unstructured as well as structured grids [6]. It is written in a way that allows the user to easily develop and add their own algorithms and functions to the already coded ones.

Regarding the Geometry and Mesh, uFVM reads the test case from an OpenFOAM test case directory. The OpenFOAM test case contains at least three basic folders as shown in figure 25.

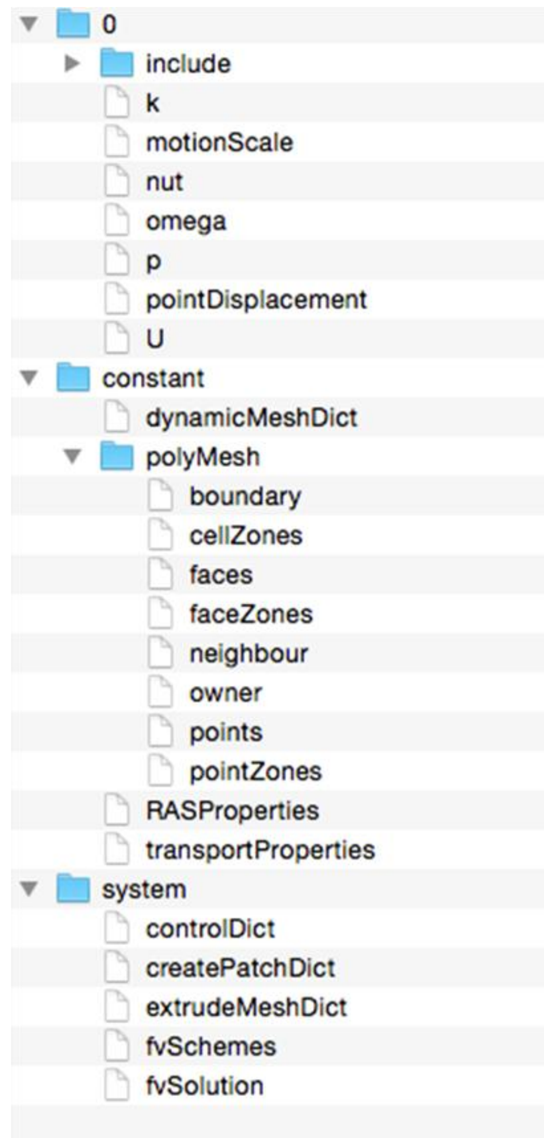


Figure 25: OpenFOAM Case

The “0” directory contains initial and boundary conditions of each field, property and variable used in the test case at time zero. The “constant” directory contains information about physical properties, and “polyMesh” subdirectory which contains description about the test case geometry as well as its mesh. The “system” directory contains at least three files that define case setup: *controlDict* defines general control parameters of the test case; *fvSchemes* defines the discretization schemes used

and *fvSolution* contains information about the solution methods and relaxation used [82].

As can be noticed from figure 25, the ployMesh folder contains several files that describe the mesh. To make things more clear, we describe some of the files mentioned. The “points” file contains a list of vectors that describe cell vertices sorted from vertex 0, 1 etc. with its corresponding x, y, and z coordinates. The “faces” file contains list of faces, each of which is list of indices to vertices in points file. Each face has an owner and a neighbor. The “owner” file contains list of owner cell labels such that each entry relates to the index of the owned face. For example the first entry is the owner label for face 0 then the second entry is the owner of face 1 and so on. It is important to note that the number of owners is equal to the sum of interior faces plus boundary faces. The “neighbour” file contains list of neighbor cell labels same as owner but for neighbors. “Boundary” file contains list of patches that include dictionary for each patch with its corresponding patch name [82].

As mentioned earlier, uFVM reads and converts all geometrical data from OpenFOAM test case folders using certain function that will read all the files in polyMesh folder and convert them into MATLAB structure arrays as shown in figure 26. As noticed, the “points” file is stored into structure array called *nodes* array, “faces” to *faces* array that list each face with its corresponding nodes that form it, surface area, owner and neighbor of each face, while the *elements* structure array contains all elements in the mesh with its corresponding neighbors, faces owned, nodes that form each element, volume of the elements as well its centroid.

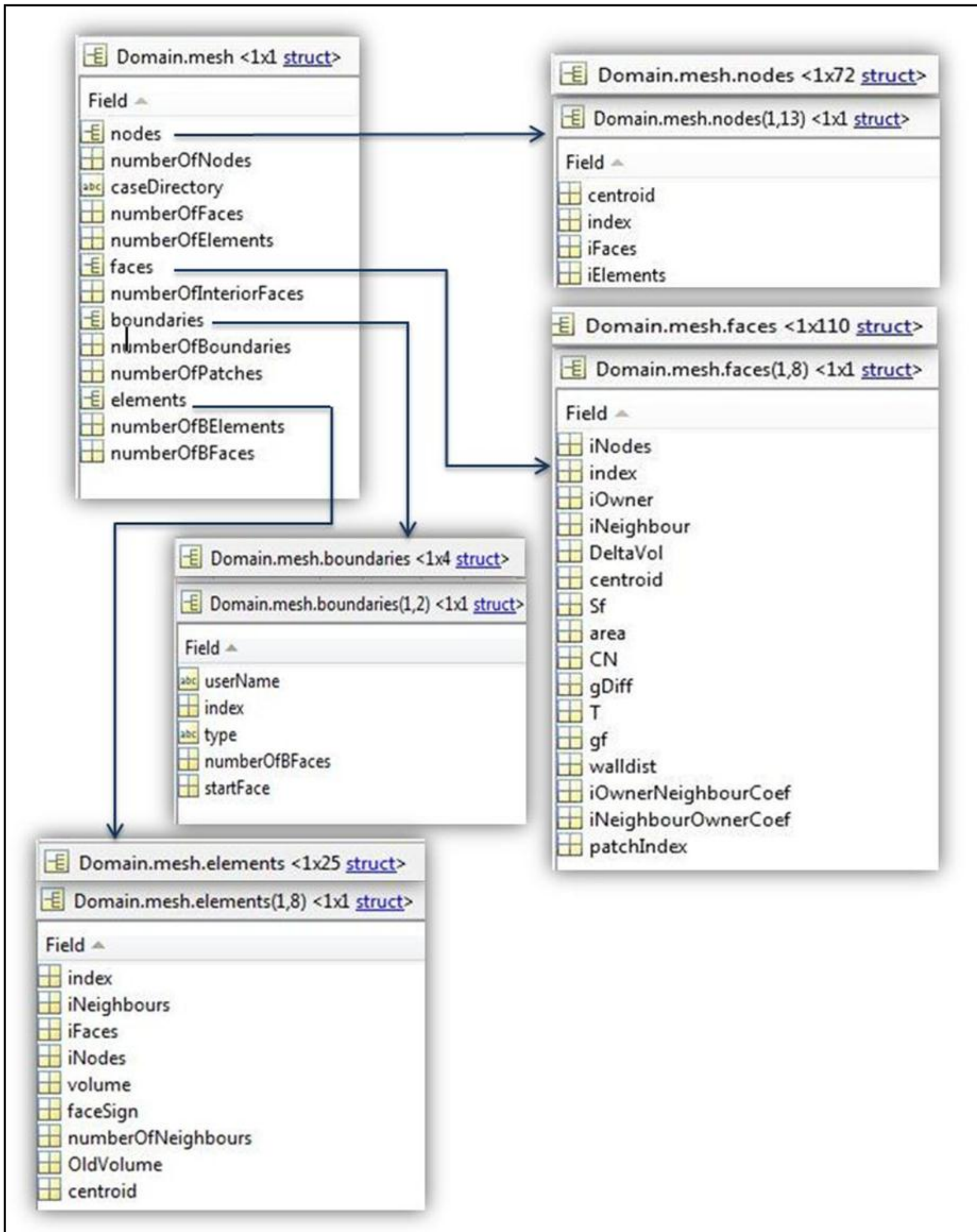


Figure 26: uFVM Domain

Regarding the Storage of Coefficients, we will focus now on how matrix coefficients are stored in uFVM. After setting up the test case geometry as well as the governing equations, initializing the variable fields, it is time to discretize and solve the equations over the elements. Discretization of the equations involves the computation of the coefficients for the resulting system of algebraic equations. In uFVM the coefficients are stored in *coefficients* structure array which contains a_c, a_{nb},b_c and other arrays such as *cconn* that consists of neighboring elements and *csize* which contains the number of neighbors for each element as shown in below figure 27.

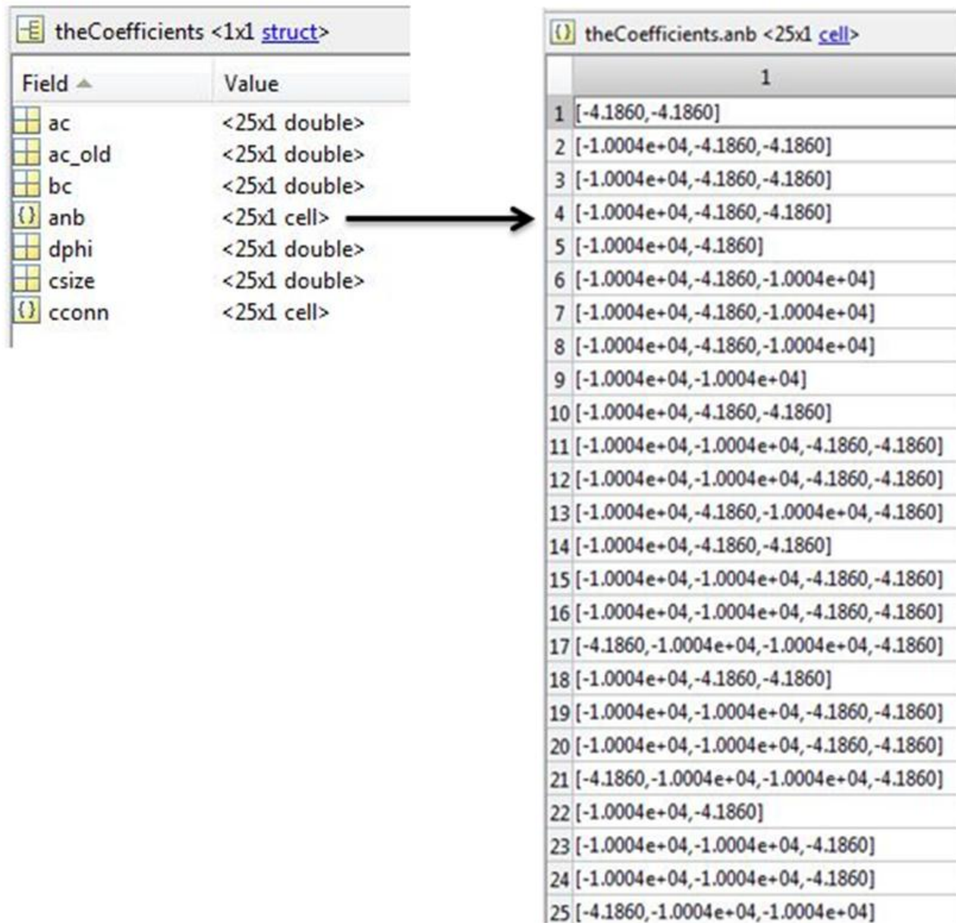


Figure 27: uFVM Coefficients Storage Arrays

2. OpenFOAM

OpenFOAM or Open Field Operation and Manipulation, is an open source C++ library that uses finite volume method in continuum mechanics to develop a variety of applications like solvers and utilities [82]. Solvers are designed to solve complex problems in many fields like chemical reactions, laminar and turbulent flows, energy and heat transfers, dynamic, solid dynamics, fluid-structure interface and many others. In addition to built-in solvers, OpenFOAM allow users with coding background to develop and add new solvers and utilities. It works on a number of operating systems and supports parallel interface. It was developed at Imperial College in 1989, and then in 1996 first commercial version was launched. First it was sold by Nabla company then was released as open source in 2004 [82]. OpenFOAM supports, besides its meshing and solving ability, pre and post processing tools as shown in figure 28.

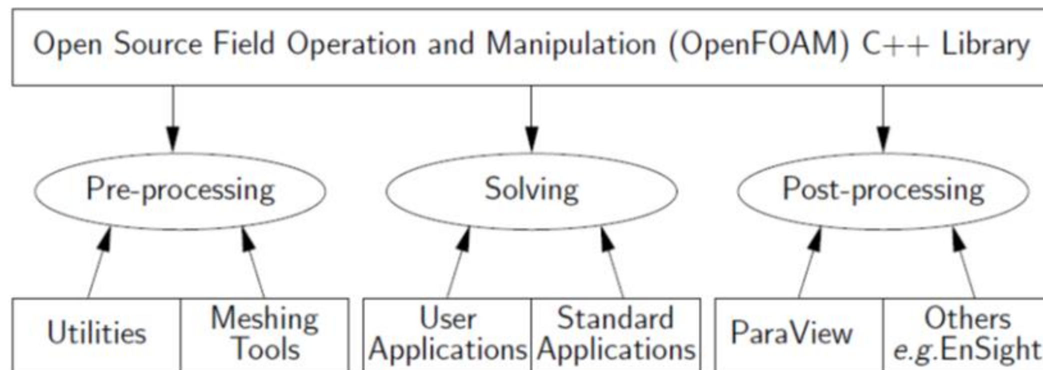
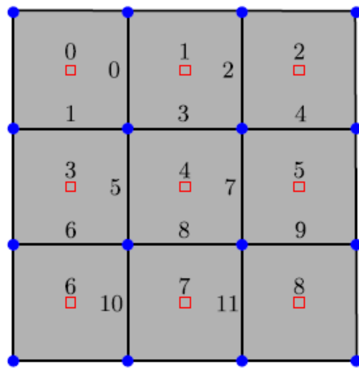


Figure 28: Overview of OpenFOAM Structure [82]

The matrix coefficients in OpenFOAM are stored using column, diagonal and row storage or what is known as an arrow storage format. The matrix is known as LDU Matrix which is a square matrix with sparse addressing. The coefficients are stored into 3 main scalar field arrays: the diagonal which contains the main diagonal coefficients, and two for off-diagonal coefficients which are the upper and lower triangle arrays [83].

These arrays contain all non-zero entries in the coefficient matrix. The addressing for these arrays is stored in addressing array that is called lduAddressing array. The diagonal array or list is indexed by cell index, while the off-diagonal upper and lower triangles are indexed or referenced by face index where only adjacent cells influence one another [82]. So for each internal face, there will be two coefficients, one for upper triangle and other for lower triangle. They are referenced by two additional arrays called upperAddr and lowerAddr [84].

Consider the below matrix that is resulted from fvm discretization of below square domain which consists of 9 elements. The following storage arrays along with referenced arrays are shown.



$$\begin{bmatrix} 10 & 2 & 0 & 3 & 0 & 0 & 0 & 0 & 0 \\ 1 & 7 & 3 & 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 2 & 4 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 9 & 2 & 0 & 4 & 0 & 0 \\ 0 & 2 & 0 & 5 & 12 & 1 & 0 & 3 & 0 \\ 0 & 0 & 3 & 0 & 1 & 11 & 0 & 0 & 3 \\ 0 & 0 & 0 & 5 & 0 & 0 & 15 & 1 & 0 \\ 0 & 0 & 0 & 0 & 2 & 0 & 2 & 18 & 5 \\ 0 & 0 & 0 & 0 & 0 & 6 & 0 & 2 & 14 \end{bmatrix}$$

diagonal = [10, 7, 4, 9, 12, 11, 15, 18, 14]

upper = [2, 3, 3, 2, 1, 2, 4, 1, 3, 3, 1, 5]

lower = [1, 1, 2, 2, 3, 5, 5, 1, 2, 6, 2, 2]

upperAddr= [1, 3, 2, 4, 5, 4, 6, 5, 7, 8, 7, 8]

lowerAddr= [0, 0, 1, 1, 2, 3, 3, 4, 4, 5, 6, 7]

Note that for upper array, the lowerAddr acts as row coordinates and upperAddr as column coordinates, while for lower array, the upperAddr acts as row coordinates and lowerAddr as column coordinates. The size of diagonal array is equal to the number of elements, while the sized of upper, lower, upperAddr and lowerAddr arrays are equal to the number of internal faces. The upper array contains entries which belong to owners of the face shared while the lower contains the entries which belongs to the neighbor cell of same face [84].

Storage format and addressing are important especially for many operations used in discretization and solving procedure. One of which is sparse matrix-vector product which plays a major role in almost all cfd code. We will show and compare how this product is done in uFVM versus OpenFOAM as below.

Below are algorithms for matrix-vector product for uFVM and OpeFOAM respectively, thus A.x=b would be as follow:

```
for (register label iElem=0; iElem<nCells; iELem++)
{
     cconn = cconn [iElem];
    nLocalNeighbours = length(cconn);
            for (register label iNB=0; iNB<nLocalNeighbours;iNB++)
                  {
                          ilocalnb = cconn[iElem][iNB];
                           b[iElem]+=A[iElem][iNB]*x[ilocalnb];
                  }
}
```

Figure 29: Matrix-Vector Multiplication in uFVM

```
for (register label iElem=0; iElem<nCells; iELem++)
{
    b[iElem] = Diagonal[iElem]*x[iElem];
}
for (register label iFace=0; iElem<nFaces; iFace++)
{
   iOwner = owner [iFace];
  iNeigh = neighbor[iFace];
  b[iOwner]+= upper[iFace]*x[iNeigh];
  b[iNeigh]+= lower[iFace]*x[iOwner];
}
```

Figure 30: Matrix-Vector Multiplication in OpenFOAM

As shown in the above algorithms, the matrix-vector product in uFVM is row wise where it passes over each row to calculate the corresponding product, while in OpenFOAM it is face wise where it passes over each face and calculate the product of corresponding owner and neighbor coefficients of each face with corresponding variable x. Moreover the loops in uFVM are nested while in OpenFOAM are not. This difference plays major role in speed, robustness and parallelism.

In the coming smoothers, a lot of row-column products is needed, so to accelerate calculations more, we are going to introduce additional three more arrays that belongs to lduAddressing which are "*ownerStartAddr*", "*losortStartAddr*" and "*losortAddr*".

For the same matrix above, below are the following arrays:

ownerStartAddr = [0, 2, 4, 5, 7, 9, 10, 11, 12, 12]

losortStartAddr = [0, 0, 1, 2, 3, 5, 5, 7, 8, 10, 12]

losortAddr = [0, 2, 1, 3, 5, 4 ,7 ,6, 8, 10, 9, 11]

Using these addressing the Matrix-Vector multiplication becomes as follow:

```
for (register label iElem=0; iElem<nCells; iELem++)
{
    b[iElem] = Diagonal[iElem]*x[iElem];
}
for (register label iElem=0; iElem<nCells; iELem++)
{
  for (register label j = ownerStartAddr[iElem]; j < ownerStartAddr[iElem+1]; j++)
    {
       b[iElem]+= upper [j]*x[upperAddr [j]];
     }
  for(register label j = losortStartAddr[iElem]; j < losortStartAddr[iElem+1]; j++)
    {
       b[iElem]+= lower [losortAddr[j]]*x[lowerAddr [losortAddr[j]];
     }
}
```

Figure 31: Modified Matrix-Vector Multiplication in OpenFOAM

Notice how *ownerStartAddr'* are upper cell faces offset, and *losortStartAddr* are lower cell faces offset while *losortAddr* are lower faces assembled row wise instead of typical column wise. In this way we combined the robustness of loops used in OpenFOAM with the row-column multiplication using vectorization which can lead to parallelized algorithms with high speed for future use [85].

BIBLIOGRAPHY

[1] H. Anton and C. Rorres, “Systems of Linear Equations and Matrices,” in *Elementary Linear Algebra with Supplemental Applications*, 11th Edition International Student Version, Wiley, 2014, ch. 1

[2] A. Bondeson and T. Rylander, *Computational Electromagnetics,* Springer Science & Business Media, 2005

[3] S. MA and A. Chronopoulos, “Implementation of iterative methods for large sparse nonsymmetric linear systems of a parallel vector machine”, *International Journal of High Performance Computing Application.*, vol. 4, pp. 9-24, 1990.

[4] J. Jin and D.J. Riley, “ Finite Element Formulation,” *in Finite Element Analysis of Antennas and Arrays,* Wiley-IEEE Press,2009,ch. 2,sec. 4, pp. 49-50.

[5] W. E and T. Li, ‘Iterative methods for solving linear system’, Princeton University, 2007.

[6] F. Moukalled et al., *The Finite Volume Method in Computational Fluid Dynamics*. Springer, 2015.

[7] P. Fedorenko, “A relaxation method for solving elliptic difference equations”, *USSR Computational Mathematics and Mathematical Physics 1*, pp. 1092–1096, 1962.

[8] F.V. Poussin, “An accelerated relaxation algorithm for iterative solution of elliptic equations”, *SIAM Journal of Numerical Analysis 5*, pp. 340-351, 1968.

[9] A. Settari and K. Aziz, “A Generalization of the Additive Correction Methods for the Iterative Solution of Matrix Equations”, *SIAM Journal of Numerical Analysis*, vol.10, no. 3, pp. 506–521, 1973.

[10] A. Brandt, “Multi-Level Adaptive Solutions to Boundary-Value Problems”, *Mathematics of Computations,* vol. 31, no. 138, pp. 333–390, 1977.

[11] W. Hackbusch, *Multi-Grid Methods and Applications*, Berlin, Germany: Springer, 1985.

[12] W. Hackbusch, *Iterative Solution of Large Sparse Systems of Equations*, New York: Springer, 1994.

[13] P. Wesseling, *An Introduction to Multigrid Methods*, Chichester, UK: Wiley, 1992.

[14] R. Suda, *Large scale circuit analysis by preconditioned relaxation methods*, Keio University, Japan, 1994.

[15] A. C. Van Duin, “Scalable parallel preconditioning with the sparse approximate inverse of triangular matrices”, *SIAM Journal on Matrix Analysis and Applications*, vol. 20, pp. 987-1006, 1999.

[16] J.D.F. Cosgrove *et al.,* “Approximate inverse preconditioning for sparse linear systems”, *International Journal of Computer Mathematics*, vol. 44, pp. 91-110, 1992.

[17] M. Grote and T. Huckle, “Parallel preconditioning with sparse approximate inverses”, *SIAM Journal on Scientific Computing*, vol.18, pp. 838-853, 1997.

[18] N.I.M. Gould and J.A. Scott, “Sparse approximate-inverse preconditioners using norm-minimization techniques”, *SIAM Journal on Scientific Computing*, vol. 19, pp. 605-625, 1998.

[19] E. Chow and Y. Saad, “Approximate inverse preconditioners via sparse-sparse iterations”, *SIAM Journal on Scientific Computing*, vol.19, pp. 995-1023, 1998.

[20] S. Barnard and M. Grote, "A block version of the SPAI Preconditioner", in *Proc. of the 9th SIAM conference on Parallel Processing for Scientific Computing*, San Antonio, TX, 1999.

[21] J. Zhang, "A Sparse Approximate Inverse Technique for Parallel Preconditioning of General Sparse Matrices", *Applied Mathematics and Computation*, vol. 130, pp. 63-85, 2002.

[22] R. M. Holland et al, "Sparse Approximate Inverses and Target Matrices", SIAM Journal on Scientific Computing, vol. 26, pp. 1000-1011, 2005.

[23] M. Benzi *et al.,* "A sparse approximate inverse preconditioner for the conjugate gradient method", *SIAM Journal on Scientific Computing*, vol. 17, pp. 1135-1149, 1996.

[24] M. Benzi and M. Tuma, "A sparse approximate inverse preconditioner for nonsymmetric linear systems", *SIAM Journal on Scientific Computing*, vol. 19, no. 3, pp. 968-994, 1998.

[25] M. Benzi *et al.,* "Stabilized and block approximate inverse preconditioners for problems in solid and structural mechanics", *Computer Methods in Applied Mechanics and Engineering*, vol. 190, pp. 6533-6554, 2001.

[26] R. Bridson and W.P. Tang, "Refining an approximate inverse", *J. Comput. Appl. Math.*, vol. 123, pp. 293-306, 2000.

[27] N. I. Buleev, "A numerical method for the solution of two-dimensional and three dimensional equations of diffusion", *Math. Sb, 51*, pp. 227-238, 1960.

[28] T. A. Oliphant, "An implicit numerical method for solving two-dimensional time-dependent diffusion problems", *Quarterly of Applied Mathematics,* vol. 19, pp. 221-229, 1961.

[29] M. Benzi and M. Tuma, “A comparative study of sparse approximate inverse precondition-ers”, *Applied Numerical Mathematics*, vol. 30, pp. 305-340, 1999.

[30] M.W. Benson, “Iterative solution of large scale linear systems”, M.S. Thesis, Lakehead Univ., ThunderBay, Canada, 1973.

[31] P.O. Frederickson, “Fast approximate inversion of large sparse linear systems”, Math. Report 7, Lakehead Univ., Thunder Bay, Canada, 1975.

[32] E. Chow and Y. Saad, “Experimental study of ILU preconditioners for indefinite matrices”, *Journal of Computational and Applied Mathematics*, vol.86, pp. 387-414, 1997.

[33] M.W. Benson and P.O. Frederickson, “Iterative solution of large sparse linear systems arising in certain multidimensional approximation problems”, *Utilitas Mathematica*, vol.22, pp. 127-140, 1982.

[34] M. Benson *et al.*, “Parallel algorithms for the solution of certain large sparse linear systems”, *International Journal of Computer Mathematics*, vol. 16, pp. 245-260, 1984.

[35] T. Huckle and A. Kallischko, “Frobenius Norm Minimization and Probing for Preconditioning”, *International Journal of Computer Mathematics*, vol. 84, no.8, pp. 1225-1248, 2007.

[36] A. Kallischko, “Modified sparse approximate inverses (MSPAI) for parallel preconditioning”, Ph.D. thesis, Technische Universität München, 2008.

[37] T.F.C. Chan and T.P. Mathew, “ The Interface Probing Technique in Domain Decomposition”, *SIAM Journal on Matrix Analysis and Applications*, vol.13, no.1, pp. 212-238, 1992.

[38] L.Yu. Kolotilina and A.Yu. Yeremin, "Factorized sparse approximate inverse preconditioning I. Theory", *SIAM Journal on Matrix Analysis and Applications*, vol. 14, pp. 45-58, 1993.

[39] I.E. Kaporin, "New convergence results and preconditioning strategies for the conjugate gradient method", *Numerical Linear Algebra with Applications*, vol.1, pp. 179-210, 1994.

[40] M.R. Field, "An efficient parallel preconditioner for the conjugate gradient algorithm", Hitachi Dublin Laboratory Technical Report HDL-TR-97-175, Dublin, Ireland, 1997.

[41] L.Yu. Kolotilina and A.Yu. Yeremin, "Factorized sparse approximate inverse preconditioning II:Solution of 3D FE systems on massively parallel computers", *International Journal of High Speed Computing*, vol. 7, pp. 191-215, 1995.

[42] M. Benzi, "A direct row-projection method for sparse linear systems", Ph.D. Thesis, Dept. Math.,North Carolina State Univ., Raleigh, NC, 1993.

[43] Y. Saad, *Iterative Methods for Sparse Linear Systems,* Boston:PWS, 1996, p. 308.

[44] J. Mas *et al.*, "Balanced incomplete factorization", *SIAM Journal on Scientific Computing*, vol. 30, pp. 2302-2318, 2008.

[45] R. Bru *et al.,* "Improved balanced incomplete factorization", *SIAM Journal on Matrix Analysis and Applications*, vol.31, pp. 2431-2452, 2010.

[46] H.C. Elman, "A stability analysis of incomplete LU factorizations", *Journal of Computational Mathematics*, vol. 47, pp. 191-217, 1986.

[47] T. A. Oliphant, "An extrapolation process for solving linear systems", *Quarterly of Applied Mathematics,* vol. 20, pp. 257-267, 1962.

[48] R. S. Varga, “Factorizations and normalized iterative methods” in *Boundary Problems in Differential Equations*, Madison, WI: University of Wisconsin Press, 1960, pp. 121–142.

[49] R. S. Varga, *Matrix Iterative Analysis*, Englewood Cliffs, NJ: Prentice Hall, 1962.

[50] D. J. Evans, “The Use of Preconditioning in Iterative Methods for Solving Linear Equations with Symmetric Positive Definite Matrices”, *Journal of the Institute of Mathematics and its Applications,* vol. 4, pp. 295-314, 1968.

[51] H. Stone*, Iterative Solution of Implicit Approximations of Multidimensional* Partial Differential. Equations”, *SIAM Journal on Numerical Analysis*, vol. 5, pp. 530-558, 1968.

[52] T. Dupont et al., “An approximate factorization procedure for solving self-adjoint elliptic difference equations”, *SIAM Journal on Numerical Analysis*, vol. 5, pp. 559-573, 1968.

[53] J. A. Meijerink and H. A. van der Vorst, “An iterative solution method for linear systems of which the coefficient matrix is a symmetric M-matrix”, *Mathematics of Computation*, vol. 31, pp. 148-162, 1977.

[54] D. S. Kershaw, “The incomplete Cholesky conjugate gradient method for the iterative solution of systems of linear equations”, *Journal of Computational Physics*, vol.26, pp. 43-65, 1978.

[55] Y. Miki and T. Washizawa, “A^2ILU: Auto-accelerated ILU Preconditioner for Sparse Linear Systems “, *SIAM Journal on Scientific Computing* , vol. 35, no. 2, pp. 1212-1232, 2013.

[56] T. A. Manteuffel, “An incomplete factorization technique for positive definite linear systems”, *Mathematics of Computation*, vol. 34, pp. 473-497, 1980.

[57] I. Gustafsson, “A class of first order factorization methods”, *BIT*, vol. 18, pp. 142-156, 1978.

[58] J. W. Watts III, “A conjugate gradient truncated direct method for the iterative solution of the reservoir simulation pressure equation”, *Society of Petroleum Engineers Journal*, pp. 345-353, 1981.

[59] Y. Saad, “ILUT: A dual threshold incomplete LU factorization”, *Numerical Linear Algebra with Applications*, vol. 1, pp. 387-402, 1994.

[60] N. Li *et al.*,” Crout versions of ILU for general sparse matrices”, *SIAM Journal on Scientific Computing*., vol. 25, pp. 716-728, 2003.

[61] O. Broker et al., “Robust parallel smoothing for multigrid via sparse approximate inverses”, *SIAM Journal on Scientific Computing*, vol. 23, pp. 1396-1417, 2001.

[62] O. Broker and M. J. Grote, “Sparse approximate inverse smoothers for geometric and algebraic multigrid”, *Applied Numerical Mathematics*, vol. 41, pp. 61-80, 2002.

[63] W. Tang and W. Wan, “Sparse approximate inverse smoother for multigrid”, *SIAM Journal on Matrix Analysis and Applications,* vol. 21, pp. 1236–1252, 2000.

[64] G.A. Meurant, “A multilevel AINV preconditioner”, *Numerical Algorithms,* vol. 29, pp. 107-129, 2002.

[65] Patankar, SV 1980. *Numerical Heat Transfer and Fluid Flow*. McGraw-Hill, New York.

[66] J.Y. Murthy and S.R. Mathur, *Numerical Methods in Heat, Mass, and Momentum Transfer,* 2002.

[67] K.S. Surana *et al.*, “k-Version of Finite Element Method in Gas Dynamics: Higher-Order Global Differentiability Numerical Solutions”, *International Journal for Numerical Methods in Engineering*, 69(6), pp. 1109-1157.

[68] K.H. Huebner *et al*., *The Finite Element Method for Engineers,* 3rd Edition, New York: Wiley InterScience.

[69] H. A. van der Vorst, “Iterative solution methods for certain sparse linear systems with a nonsymmetric matrix arising from PDE Problems”, *Journal of Computational Physics*, vol. 44, pp. 1-19, 1981.

[70] R. Beauwens and L. Quenon, “Existence criteria for partial matrix factorizations in iterative methods”, *SIAM Journal on Numerical Analysis*, vol. 13, pp. 615-643, 1976.

[71] P.K.W. Vinsome, “Orthomin, An iterative method for solving sparse sets of simultaneous linear equations”,in*: Proceedings of the Fourth Symposium on Reservoir Simulation*, Society of Petroleum Engineers of AIME, SPE 5729, 1976.

[72] M. Darwish et al., “A Coupled Finite Volume Solver for the Simulation of Disperse Multiphase Flows”, presented at the Fifth European Conference on Computational Fluid Dynamics ECCOMAS CFD 2010, Lisbon, Portugal, 2010.

[73] U. Falk and M. Schafer, “A Fully Coupled Finite Volume Solver for the Solution of Incompressible Flows on Locally Refined Non-Matching Block-Structured Grids”, presented at the Sixth International Conference on Adaptive Modeling and Simulation ADMOS, 2013.

[74] L. Mangnai et al., “Development of a Novel Fully Coupled Solver in OpenFOAM: Steady-State Incompressible Turbulent Flows”, *Numerical Heat Transfer, Part B,*vol. 66, pp. 1-20, 2014.

[75] L.D. Hylton et al., *Analytical and experimental evaluation of the heat transfer distribution over the surfaces of turbine vanes,* Tech Rep. 182133, NASA, Lewis Research Center, Cleveland Ohio.

[76] L. Mangnai et al., “Development of a Pressure-Based Coupled CFD Solver for Turbulent and Compressible Flows in Turbomachinery Applications”, in *Proc. Of ASME TURBO EXPO 2014: Power for Land, Sea & Air*, Dusseldorf, Germany, June 16-20, 2014.

[77] L. Mangnai and A. Andreini, “Application of an object-oriented CFD code to heat transfer analysis”, *ASME Paper (GT2008-5118).*

[78] S.A. Berger et al., “Flow in Curved Pipes”, *Annual Review of Fluid Mechanics,* vol. 15, pp. 461-512, 1983.

[79] F.T. Smith, “Fluid Flow into a Curved Pipe”, in*: Proceedings of the Royal Society A,* vol. 351, pp. 71-87.

[80] F. Rutten et al., “Large-eddy simulation of low frequency oscillations of the Dean vortices in turbulent pipe bend flows”, *Physics of Fluids,* vol. 17, 2005.

[81] R. Rohrig et al., “Comparative computational study of turbulent flow in a 90 ° pipe elbow”, *International Journal of Heat and Fluid Flow,* vol. 55, pp. 120-131, 2015.

[82] OpenFOAM Foundation. (2004, December 10). OpenFOAM Foundation. Retrieved from http:// www.openfoam.org.

[83] H. Jasak and H. Rusche, ‘Five Basic Classes in OpenFOAM’, Penn State University, 2011.

[84] M. Taouil, “ A Hardware Accelerator for the OpenFoam Sparse Matrix-Vector Product”, M.S. Thesis, Dept Elect Eng., Delft Univ. of Technology, Delft, Netherlands, 2009.

[85] M. Culpo.(2013). *Current Bottlenecks in the Scalability of OpenFOAM on Massively Parallel Clusters.* Partnership for Advanced Computing in Europe, Brussel, Italy [Online]. Available: http://www.prace-

ri.eu/IMG/pdf/Current_Bottlenecks_in_the_Scalability_of_OpenFOAM_on_Massi
vely_Parallel_Clusters-2.pdf