fluids

MDPI

Review

Linear Solvers in OpenFOAM: A Technical Review and SIMPLE Convergence Study

Mohamed El Abbassi and Cornelis Vuik *

Delft Institute of Applied Mathematics, Delft University of Technology, 2628 CD Delft, The Netherlands; m.elabbassi@tudelft.nl

* Correspondence: c.vuik@tudelft.nl

Abstract

This article reviews the linear solvers available in OpenFOAM and assesses their impact on the convergence behaviour of the SIMPLE algorithm. The discretisation of transport equations in CFD results in large and sparse linear systems, for which the choice of linear solver strongly influences the computational time. Although the solver does not change the final discrete solution, the difference in speed and robustness between the solvers can be more than one order of magnitude. A brief overview is given concerning how the velocity and pressure fields are decoupled in OpenFOAM, followed by a detailed review of the main linear solver families, including direct methods, basic iterative methods, multigrid methods and Krylov subspace methods, with attention to their practical strengths and weaknesses. The performance of the most advanced solvers is evaluated on a full-scale non-reacting kiln case consisting of 2.3 million cells. The pressure-corrector equation is identified as the main bottleneck in the SIMPLE algorithm. The conjugate gradient (CG) solver with a multigrid (MG) preconditioner is found to be the fastest and most stable method, achieving speed-ups of up to a factor of 7 compared to the slower advanced methods. Using MG as a preconditioner also improves the robustness of the Bi-CGStab method.

Keywords: Computational Fluid Dynamics; OpenFOAM; iterative methods; linear solvers; multigrid; Krylov subspace

1. Introduction

Computational Fluid Dynamics (CFD) is a part of fluid mechanics that uses numerical analysis to solve advection–diffusion–reaction-type equations related to a wide range of applications involving fluid flow and heat and mass transfer. Discretisation is a technique in numerical analysis that transforms a continuous problem into a discrete counterpart that can be solved numerically by means of numerical solution methods.

It first requires that the geometry is divided into small, non-overlapping finite elements (or finite volumes), usually called cells. The discretisation method then transforms the partial differential equations into discrete algebraic equations, which are integrated over these discrete cells. After the discretisation of each transport equation, the resulting set of discrete equations needs to be linearised, which leads to a linear system of equations in the form of $\mathbf{Au} = \mathbf{b}$. The resulting systems of equations can be solved numerically using linear solvers.

Although every CFD simulation depends heavily on solving such linear systems, the underlying solution methods typically receive little attention in everyday CFD work and are generally treated as a black-box procedure. Since the choice of linear solver does not

Academic Editor: Michalis Xenos

Received: 30 April 2026
Revised: 31 May 2026
Accepted: 9 June 2026
Published: 11 June 2026

affect the final discrete solution but rather only the speed at which it is reached, the common approach is to rely on default linear solver settings or to compensate for slow convergence with additional computational resources. However, different linear solvers exhibit very different performance characteristics. Understanding these strengths and weaknesses can lead to significant improvements in computational efficiency, often far exceeding what can be achieved by simply increasing computational resources. This is particularly relevant for OpenFOAM, where the linear solver options are very diverse.

This paper provides a detailed review of the linear solver theory in the open-source CFD-toolbox OpenFOAM [1] in order to improve the general understanding and appreciation of its mathematical models, as well as to aid in speeding up its convergence without the need for stronger hardware. Details are provided in Section 2. The theory in this work partially follows Refs. [2–6].

The performance of the solution methods in OpenFOAM is demonstrated for an industrial case in Section 3.

1.1. Decoupling the Velocity and Pressure in CFD

Before discussing the linear solvers used in this work, it is important to briefly revisit how the velocity and pressure fields are decoupled in CFD. Once the velocity field is determined, solving the transport of a particular scalar (such as energy and the variables for turbulence) is relatively easy, as the flow field of that scalar is 'frozen', and the main challenge is usually to determine the source terms. However, calculating the velocity field requires solving the discretised Navier–Stokes equations numerically, which is difficult not necessarily because the velocity field is non-linear, but due to the coupling of the velocity and pressure in the momentum equation. Consider the incompressible steady-state Navier–Stokes equations with neglected gravity

$$\nabla \cdot \boldsymbol{U} = 0, \tag{1}$$

$$\boldsymbol{U} \cdot \nabla \boldsymbol{U} = -\nabla \frac{p}{\rho} + \nabla \cdot \nu \nabla \boldsymbol{U}, \tag{2}$$

where the vector $\boldsymbol{U}$ (bold face) is the velocity, p is the pressure, ρ is the density, and ν is the kinematic viscosity. These equations consist of four equations and four unknowns: U_x, U_y, U_z and p (note that the density here is not a variable but a reference value). This would intuitively mean that these equations are directly solvable, even if there is no explicit equation for p. However, the continuity equation is not really the 'closing' equation, but rather acts as a restriction to the momentum equations in order to satisfy mass conservation.

The Semi-Implicit Method for Pressure-Linked Equations (SIMPLE) algorithm decouples the pressure and velocity fields, allowing their separate calculation, then using one field to correct the other field iteratively until convergence is achieved.

1.2. The SIMPLE Algorithm

The SIMPLE algorithm was developed by Patankar [7] to solve the steady-state incompressible Navier–Stokes equations. The main idea was to use the continuity equation to derive an equation for the pressure after splitting it from the momentum equations. The pressure equation will then act as a corrector for the modified momentum equation. It starts by writing the momentum equations (Equation (2)) in general matrix form as follows:

$$\boldsymbol{A}\boldsymbol{U} = -\nabla p, \tag{3}$$

where $\boldsymbol{A}$ is the coefficient matrix resulting from discretising Equation (2) using the finite volume method. All these coefficients are known. Equation (3) can be solved when

the pressure is known from the previous iteration (starting from an initial guess at the first iteration), of which its gradient acts as the source term here. Also, the non-linearity appearing in the left-hand side (LHS) of Equation (2) is linearised by evaluating the mass fluxes using the velocity field from the previous iteration, resulting in a linear system for the velocity. The LHS of Equation (3) is known in OpenFOAM as the velocity equation. The calculated velocity field does not yet satisfy continuity and is still a guess. This stage is called the momentum predictor.

The derivation of the pressure equation starts from splitting the coefficient matrix $\boldsymbol{A}$ by extracting its diagonal into matrix $\boldsymbol{D}$, as follows:

$$\boldsymbol{AU} = \boldsymbol{DU} - \boldsymbol{H}. \tag{4}$$

The reason for this is because a diagonal matrix can easily be inverted ($\boldsymbol{D}^{-1}$), which is very useful. $\boldsymbol{H}$ is the residual vector containing the contributions of all the neighbour cells, and will be used as a source term for the pressure equation later. Combining Equations (3) and (4) and multiplying on both sides with $\boldsymbol{D}^{-1}$ results in the following equation for the velocity field that is an explicit function of the pressure field:

$$\boldsymbol{U} = \boldsymbol{D}^{-1}\boldsymbol{H} - \boldsymbol{D}^{-1}\nabla p. \tag{5}$$

In the OpenFOAM source code, the diagonal matrix $\boldsymbol{D}$ is defined as `rAU`, and the frequently recurring product $\boldsymbol{D}^{-1}\boldsymbol{H}$ is known as `HbyA`. Now, we can use the continuity equation to derive an explicit equation for the pressure. Substituting Equation (5) into Equation (1) leads to the following Poisson equation, which is known as the pressure-corrector:

$$\nabla \cdot (\boldsymbol{D}^{-1}\nabla p) = \nabla \cdot (\boldsymbol{D}^{-1}\boldsymbol{H}). \tag{6}$$

After solving for p in the above pressure equation, the velocity field (Equation (5)) can be corrected to satisfy continuity. Notice that the velocity is calculated explicitly and this is why this method is called semi-implicit. The problem now is that once the velocity is corrected, the pressure equation is no longer satisfied because the matrix $\boldsymbol{H}$ depends on the velocity, which has been updated. Therefore, this process is iterated until both the velocity and pressure fields no longer change after correction.

The SIMPLE algorithm is commonly extended to compressible flows, which is also considered in the industrial case study of this work. In such cases, the transport equation of energy (or enthalpy) is solved after the momentum predictor step in OpenFOAM, and the turbulence transport equations are solved after the pressure-corrector step. Although the formulation of the pressure equation differs slightly between incompressible and compressible solvers, the underlying structure as a pressure-correction equation remains similar, and the same conclusions regarding solver performance apply.

2. Solution Methods

2.1. Introduction

For each transport equation, the discretisation and linearisation process leads to a system of linear equations, which is written as the following general matrix equation:

$$\mathbf{A}\mathbf{u} = \mathbf{b}, \tag{7}$$

where $\mathbf{A}$ is the coefficient matrix resulting from the linearisation and mesh geometry, $\mathbf{u}$ is the vector that contains the unknown values of the dependent variable at the cell centroids

that needs to be solved for, and $\mathbf{b}$ is the vector containing all the sources, constants and boundary conditions.

Using the momentum predictor in the previous section (Equation (3)), it is easy to see how it is translated into the following general matrix equation:

- $\boldsymbol{A} \equiv \mathbf{A}$,
- $\boldsymbol{U} \equiv \mathbf{u}$,
- $-\nabla p \equiv \mathbf{b}$,

while in the pressure-corrector (Equation (6)), the translation is a bit less apparent:

- $\boldsymbol{D}^* \equiv \mathbf{A}$,
- $p \equiv \mathbf{u}$,
- $\nabla \cdot (\boldsymbol{D}^{-1}\boldsymbol{H}) \equiv \mathbf{b}$,

where $\boldsymbol{D}^*$ is the resulting coefficient matrix after combining $\boldsymbol{D}^{-1}$ with the discretised Laplace operator coefficients of p.

The linear systems are solved by linear solvers, for which different solution methods exist, which are generally grouped into direct and iterative methods, each consisting of many sub-groups. Due to the linearisation process with the finite volume method, the coefficient matrix is very large (many cells required for accuracy), very sparse, and diagonally dominant. Iterative methods therefore have been more popular because they are more suited for this type of application as they typically require lower computational cost per iteration and less memory [2,3]. The (modified) direct methods can, however, be used as a preconditioner, in which they are used to replace the coefficient matrix with a matrix for which the corresponding linear system of equations is easier to solve, while keeping the same solution. The solution methods that are used in OpenFOAM are discussed here.

2.2. Direct Methods

Even though direct methods, as stand-alone solvers, are not efficient for solving sparse systems of linear equations due to their high computational cost, their discussion will lead the way for introducing efficient iterative methods in the next sections.

Direct methods apply Gaussian elimination techniques to solve the above-mentioned system of linear equations. The modern version is **LU factorisation**, which computes the lower and upper triangular matrix $\mathbf{L}$ and $\mathbf{U}$, such that

$$\mathbf{A} = \mathbf{LU}. \tag{8}$$

For diagonally dominant matrices, the upper triangular matrix $\mathbf{U}$ is constructed by performing Gaussian elimination on matrix $\mathbf{A}$, while the elements of $\mathbf{L}$ consist of the Gauss factors by which the rows of $\mathbf{A}$ are multiplied to get $\mathbf{U}$. Furthermore, the diagonal elements of $\mathbf{L}$ are set equal to one, so for a 3×3 matrix, Equation (8) becomes

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ l_{21} & 1 & 0 \\ l_{31} & l_{32} & 1 \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}. \tag{9}$$

When $\mathbf{L}$ and $\mathbf{U}$ are found, the problem $\mathbf{Au} = \mathbf{b}$ is split into two problems that are much easier to solve, namely

$$\mathbf{Ly} = \mathbf{b}, \quad \mathbf{Uu} = \mathbf{y}. \tag{10}$$

To avoid rounding errors, it might be necessary to apply some form of pivoting, i.e., reordering the rows of matrix $\mathbf{A}$. However, for diagonally dominant matrices, this is not

necessary [8]. If **A** is Symmetric Positive Definite (SPD), the LU-factorisation reduces to the **Cholesky factorisation** of **A**.

$$\mathbf{A} = \mathbf{C}\mathbf{C}^{\mathsf{T}}, \tag{11}$$

where **C** is a lower triangular matrix. This results in even greater memory and computational savings as only one triangular matrix needs to be computed.

The efficiency of the direct methods, however, is lost due to the occurrence of fill-in after factorisation, where many zero entries in the sparse matrix become non-zero during the factorisation process. This significantly increases the memory requirements and computational cost, thereby disrupting the advantages of sparse matrix storage, especially for large practical CFD applications. This leaves an opening for iterative methods.

It will be shown that direct methods are very suitable as preconditioners for the Krylov subspace methods where the sparsity pattern is deliberately kept after factorisation, known as **incomplete factorisation**, so that $\mathbf{A} \approx \mathbf{LU}$ or $\mathbf{A} \approx \mathbf{C}\mathbf{C}^{\mathsf{T}}$.

2.3. Basic Iterative Methods

The linear solvers in most CFD codes make use of iterative methods. Iterative methods are techniques created to obtain an approximate solution of linear systems. For the implementation of these methods, successive approximations are used. Starting from an initial guess $\mathbf{u_0}$, a new approximation $\mathbf{u_k}$ is obtained at each iteration k until an approximated solution is found that is close enough to the exact solution **u**.

The error vector is defined as

$$\mathbf{e}_k = \mathbf{u} - \mathbf{u}_k. \tag{12}$$

Solving the error vector is just as difficult as solving the exact solution of the discrete linear system $\mathbf{Au} = \mathbf{b}$. A computable measure of the quality of the approximation is therefore obtained from the residual,

$$\mathbf{r}_k = \mathbf{b} - \mathbf{A}\mathbf{u}_k, \tag{13}$$

which represents the imbalance of the approximation of the conservation laws, with $\mathbf{r}_k = 0$ being the exact solution. A common stopping criterion or tolerance (ϵ) for iterative methods is the relative residual, defined as the 2-norm of the residual of the k-th iteration divided by the 2-norm of the right-hand side,

$$r_k = \frac{||\mathbf{r}_k||_2}{||\mathbf{b}||_2} \leq \epsilon. \tag{14}$$

The idea of an iterative method is that the matrix **A** is decomposed into two matrices, **M** and **N**, such that $\mathbf{A} = \mathbf{M} - \mathbf{N}$, and the original linear system $\mathbf{Au} = \mathbf{b}$ transforms into

$$\mathbf{Au} = (\mathbf{M} - \mathbf{N})\mathbf{u} = \mathbf{b}. \tag{15}$$

Rearranging terms, we obtain

$$\mathbf{Mu} = \mathbf{Nu} + \mathbf{b} = (\mathbf{M} - \mathbf{A})\mathbf{u} + \mathbf{b}. \tag{16}$$

The latter system is used to perform an iterative process, finding at each iteration (k) a more accurate solution. Most of the iterative methods are derived from the following recurrence relation:

$$\mathbf{u}_{k+1} = \mathbf{M}^{-1}(\mathbf{M} - \mathbf{A})\mathbf{u}_k + \mathbf{M}^{-1}\mathbf{b}, \tag{17}$$

Or, after rearranging and using Equation (13),

$$\mathbf{u}_{k+1} = \mathbf{u}_k + \mathbf{M}^{-1}\mathbf{r}_k, \tag{18}$$

where the matrix $\mathbf{M}$ is chosen such that $\mathbf{M}^{-1}$ can be determined easily, so, for example, a diagonal or triangular matrix.

Basic iterative methods (BIMs) are obtained by decomposing the system matrix as $\mathbf{A} = \mathbf{D} - \mathbf{E} - \mathbf{F}$ (see Figure 1), with $\mathbf{D}$ being the diagonal of $\mathbf{A}$, and $\mathbf{E}$ and $\mathbf{F}$ the strictly lower and upper parts. Based on the choice of $\mathbf{M}$ and $\mathbf{N}$, different iterative methods can be obtained. Some of the BIMs are presented in Table 1, of which Jacobi and Gauss–Seidel are the options for OpenFOAM's BIM solver `smoothSolver`.

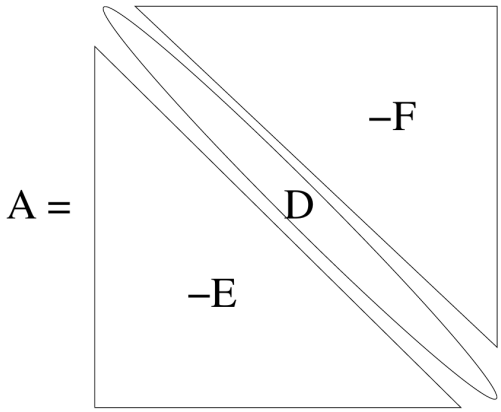

Figure 1. Splitting of coefficient matrix A.

Table 1. Basic iterative methods.

Method	M	N	Iteration
Richardson	$\mathbf{I}$	$\mathbf{I} - \mathbf{A}$	$\mathbf{u}_{k+1} = (\mathbf{I} - \mathbf{A})\mathbf{u}_k + \mathbf{b}$
Jacobi	$\mathbf{D}$	$\mathbf{E} + \mathbf{F}$	$\mathbf{u}_{k+1} = \mathbf{D}^{-1}(\mathbf{E} + \mathbf{F})\mathbf{u}_k + \mathbf{D}^{-1}\mathbf{b}$
Damped Jacobi	$(1/\omega)\mathbf{D}$	$\mathbf{E} + \mathbf{F}$	$\mathbf{u}_{k+1} = \omega\mathbf{D}^{-1}(\mathbf{E} + \mathbf{F})\mathbf{u}_k + \omega\mathbf{D}^{-1}\mathbf{b}$
Gauss–Seidel	$\mathbf{D} - \mathbf{E}$	$\mathbf{F}$	$\mathbf{u}_{k+1} = \mathbf{D}^{-1}(\mathbf{F}\mathbf{u}_k + \mathbf{E}\mathbf{u}_{k+1}) + \mathbf{D}^{-1}\mathbf{b}$
SOR	$\mathbf{D} - \omega\mathbf{E}$	$(1 - \omega)\mathbf{D} + \omega\mathbf{F}$	$\mathbf{u}_{k+1} = \omega\mathbf{D}^{-1}(\mathbf{F}\mathbf{u}_k + \mathbf{E}\mathbf{u}_{k+1}) + (1 - \omega)\mathbf{u}_k + \omega\mathbf{D}^{-1}\mathbf{b}$

Of the presented methods, the Successive Over-Relaxation (SOR) method is significantly faster than the others for optimal damping parameters ω, whose values are not straightforward to find. Nevertheless, BIMs in general are too slow for engineering problems as stand-alone solvers, as they are only effective in eliminating the high-frequency components of the error, and very slow in reducing its low-frequency components (Figure 2). This deteriorates for finer meshes [3].

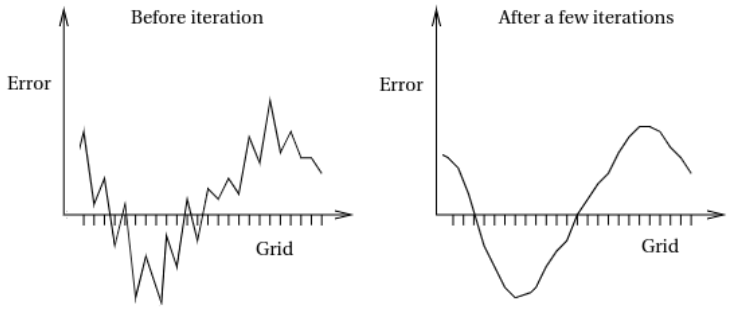

Figure 2. Smoothing process of a BIM.

Nevertheless, it is their ability to eliminate high-frequency error components (smoothing) that allows BIMs to serve as the second set of building blocks, in addition to direct methods, for more efficient iterative methods, such as smoothers for the multigrid method or as preconditioners for the Krylov subspace methods.

2.4. Multigrid Methods

The convergence of a BIM can be accelerated with a multigrid method, which is known to be among the most efficient solvers for elliptic problems [9,10], such as the steady-state transport equation of a particular scalar ϕ

$$\underbrace{\nabla \cdot (\rho \mathbf{U} \phi)}_{\text{advection}} = \underbrace{\nabla \cdot \Gamma \nabla \phi}_{\text{diffusion}} + \underbrace{Q_\phi}_{\text{source}}. \tag{19}$$

The basic idea of a multigrid method is to first smooth the error with a BIM, then transfer the remaining low-frequency error components (recursively) to a coarser grid without losing information until the problem is small enough to solve directly, after which a correction can be transferred back to the fine grid. As a result, the number of iterations required for convergence becomes nearly independent of the mesh element size, and the total computational cost scales approximately linearly with the number of unknowns. This high efficiency explains why multigrid methods are widely used in many CFD codes [4,11,12]. Depending on the size of the 2D or 3D mesh, the speed-up can be one or multiple orders of magnitude as compared to the BIM [3].

The detailed procedure is as follows:

1. **Presmoothing**—Starting from an initial guess $\mathbf{u}_0$, apply a low number ν_1 of iterations with a BIM to eliminate the high-frequency error components (recall from Equation (18)).
$$\mathbf{u}_n^h = \mathbf{u}_{n-1}^h + \mathbf{M}^{-1}\mathbf{r}_{n-1}^h, \tag{20}$$
where $n = 1, \ldots, \nu_1$, and the superscript h denotes the fine grid size.
2. **Restriction**—To transfer the residual to a coarser grid of grid size H, we use the restriction operator I_h^H, such that
$$\mathbf{r}^H = I_h^H \mathbf{r}^h, \tag{21}$$
where the superscript H denotes the coarse grid size.
3. **Solve coarse grid error**—The residual can now be used to determine the coarse grid error by combining Equations (12) and (13) for
$$\mathbf{A}^H \mathbf{e}^H = \mathbf{r}^H. \tag{22}$$
This is known as the residual equation. As mentioned before, solving this equation is just as hard as solving the original system of linear equations. However, at the current grid level, the problem is small enough to solve it directly, and the obtained error vector can be used as an approximation at the fine grid in the following step.
4. **Prolongation**—The error e^H is projected into the fine grid using the prolongation operator I_H^h (inverse of I_h^H), and the vector $u_{\nu_1}^h$ is corrected with the approximated error.
$$\mathbf{u}_{\nu_1}^h = \mathbf{u}_{\nu_1}^h + I_H^h e^H \tag{23}$$
5. **Postsmoothing**—Apply a low number of postsmoothing ν_2 iterations as described in step 1 to obtain $\mathbf{u}_1 =: \mathbf{u}_{\nu_1+\nu_2}^h$

In the above procedure, the sequence described by steps 2 to 4 is known as defect correction, or coarse grid correction (CGC), and can be summarised via the following equation:

$$\mathbf{u}_n^h = \left(\mathbf{I} - I_H^h (\mathbf{A}^H)^{-1} I_h^H \mathbf{A}\right) \mathbf{u}_{n-1}^h. \tag{24}$$

Although the BIM and the CGC are individually not efficient, the combination of the two as applied above is very efficient as it effectively reduces all the frequency components of the error. The described procedure above is essentially a two-grid cycle when going through the steps once. When steps 1–2 (hence, also steps 4–5) are completed recursively within one loop, we arrive at the multigrid method. Most of the time, this is necessary as the long wavelengths of the error modes become shorter in the coarser mesh and therefore presmoothing is needed to eliminate them quickly. Applying a certain number of iterations of the restriction step will lead to a negligible cost for solving the residual equation compared to a smoothing sweep at the finest grid.

Applying step 3 only once is called a V-cycle, and applying it multiple times can result in a W-cycle or an F-cycle, which are more accurate but require also more work. A schematic is shown in Figure 3. There are many strategies and choices to tune the multigrid method, such as the amount of smoothing sweeps, grid levels, and cycles, with varying trade-offs between the speed of solving a single iteration and the overall rate of convergence.

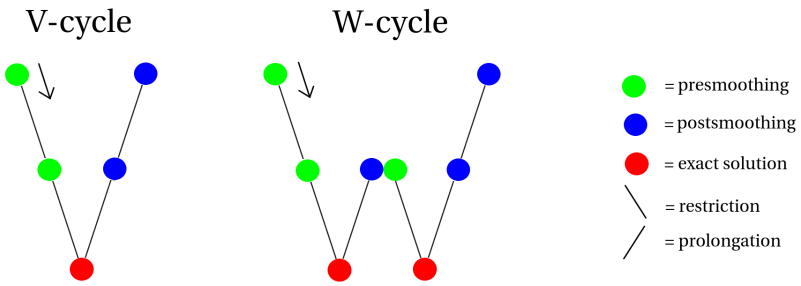

Figure 3. Different multigrid cycles starting from the finest grid at the top to the coarsest grid at the bottom and back.

One of the approaches to constructing the coarser grid is the application of the restriction operator directly to the system matrix, which is known as the **algebraic multigrid** (AMG). This method is relatively easy to implement, but works as a black-box solver as it does not have any geometric interpretation of the mesh. The **geometric multigrid** (GMG), on the other hand, obtains the coarser grid by clustering (or agglomerating) the cells of the mesh.

OpenFOAM works with the **generalised method of the geometric–algebraic multigrid** (GAMG), which by default operates as a GMG solver, but also gives the option to enable the older AMG implementation (of its predecessor) instead [13]. The GMG solver agglomerates the cells by pairing each cell with an unpaired neighbour cell with the largest shared face area. This is repeated for every coarse grid level. An example is shown in Figure 4. If a cell does not have a match, it will be added to the neighbouring group. The system matrix is adjusted accordingly.

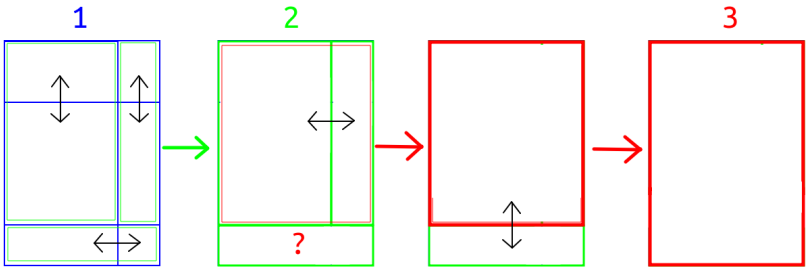

Figure 4. Geometric agglomeration using face pairs in three coarse grid levels: 1, 2 and 3.

The GAMG is a V-cycle method where the user can select a smoother (Gauss–Seidel being most favoured) and adjust certain parameters, such as the mesh size at the coarsest

level and the amount of presmoothing and postsmoothing sweeps. The solver automatically calculates the number of intermediate coarse grid levels. A practical example of how the GAMG solver is set up is shown in Section 3.

Next to the multigrid methods, another class of solvers called Krylov subspace methods can be adopted.

2.5. Krylov Subspace Methods

Having discussed basic iterative and multigrid methods, this section focuses on the Krylov subspace methods used in OpenFOAM [2]. These methods are based on projection processes onto Krylov subspaces $\mathcal{K}_i$, that is,

$$\mathcal{K}_i(\mathbf{A}, \mathbf{r}_0) = span\{\mathbf{r}_0, \mathbf{A}\mathbf{r}_0, \mathbf{A}^2\mathbf{r}_0, \ldots, \mathbf{A}^{i-1}\mathbf{r}_0\}, \tag{25}$$

where $\mathcal{K}$ is called the Krylov space of dimension i corresponding to matrix $\mathbf{A}$ and initial residual $\mathbf{r}_0$ (Equation (13)). The basic idea of Krylov subspace methods is that for large and sparse matrices, it is more efficient to use $\mathbf{A}$ only in matrix–vector products. In this way, the approximate solution is constructed as a polynomial in $\mathbf{A}$ applied to the initial residual, without explicitly forming or inverting the matrix [14].

The **Conjugate Gradient** (CG) method is a Krylov subspace method that is developed to minimise the following quadratic equation:

$$F(\mathbf{u}) = \frac{1}{2}\mathbf{u}^\mathsf{T}\mathbf{A}\mathbf{u} - \mathbf{b}^\mathsf{T}\mathbf{u}, \tag{26}$$

where $\mathbf{A}$ is an SPD matrix. Solving this minimisation problem (which is finding $\mathbf{u}^*$ such that $\nabla F(\mathbf{u}) = 0$) is equivalent to solving $\mathbf{A}\mathbf{u} = \mathbf{b}$, and falls under this type of gradient method (see Figure 5).

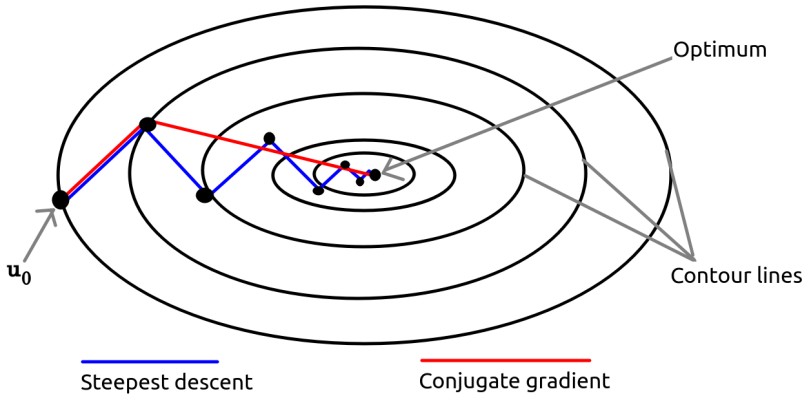

Figure 5. Comparison of the convergence between the steepest descent and CG method for an $n \times n$ matrix of size 2. In theory, the CG method should converge after n steps.

The solution $\mathbf{u}^*$ is approached recursively via the relation

$$\mathbf{u}_{k+1} = \mathbf{u}_k + \alpha_k \mathbf{d}_k, \tag{27}$$

where $\mathbf{d}_k$ is the search direction for the next iteration. One way is to choose the residual vector as the search direction, which is known as the steepest descent method. However, this can lead to the same search directions being computed more than once and oscillations occurring around the local minima, leading to very slow convergence. Therefore, in the CG method, every search direction should be in a unique direction. This is accomplished by

selecting a set of search directions that are **A**-orthogonal (or **A**-conjugate) to the previous directions. This means that they satisfy the condition $\mathbf{d}_k^T \mathbf{A} \mathbf{d}_j = 0$ for $k \neq j$.

The CG method starts with the residual vector being chosen as the first search direction,

$$\mathbf{d}_0 = \mathbf{r}_0 = \mathbf{b} - \mathbf{A}\mathbf{u}_0. \tag{28}$$

The factor α_k in Equation (27) ensures that the minimum point is found along the current search direction. The following expression can thus be derived:

$$\alpha_k = \frac{\mathbf{d}_k^T \mathbf{r}_k}{\mathbf{d}_k^T \mathbf{A} \mathbf{d}_k}. \tag{29}$$

After obtaining the new value for **u** (Equation (27)), the new residual is calculated using

$$\mathbf{r}_{k+1} = \mathbf{r}_k - \alpha_k \mathbf{A} \mathbf{d}_k. \tag{30}$$

To make sure that the next search direction is **A**-conjugate to the previous one, the following coefficient is used and simplified as

$$\beta_k = \frac{\mathbf{r}_{k+1}^T \mathbf{r}_{k+1}}{\mathbf{r}_k^T \mathbf{r}_k}, \tag{31}$$

so that finally, the next search direction can be calculated, which completes the algorithm as follows:

$$\mathbf{d}_{k+1} = \mathbf{r}_{k+1} + \beta_k \mathbf{d}_k. \tag{32}$$

If **A** is the SPD and the denominators of Equations (29) and (31) are equal to zero, it means that the CG method breaks down when the problem is already solved ($\mathbf{r}_k = 0$), which makes the method robust. The rate of convergence of the CG methods depends on the spectral properties (i.e., the eigenvalue distribution) of matrix **A** and is related to the condition number of matrix **A**, as follows:

$$\kappa_2(\mathbf{A}) = \frac{\lambda_{\max}(\mathbf{A})}{\lambda_{\min}(\mathbf{A})}, \tag{33}$$

where λ_{max} and λ_{min} are the largest and smallest eigenvalues of **A**, respectively. The more clustered the eigenvalues are, the smaller the value of $\kappa_2(\mathbf{A})$, and hence, faster convergence occurs. For CFD applications, the condition number is about the square of the maximum number of grid points; hence, the standard CG method is slow [3].

2.5.1. Preconditioners

Preconditioners are used to accelerate the convergence by transforming the problem into a similar one, but with a more clustered eigenvalue distribution. This is done by multiplying the original system of equations by the inverse of the preconditioned matrix **P** as follows:

$$\mathbf{P}^{-1}\mathbf{A}\mathbf{u} = \mathbf{P}^{-1}\mathbf{b}. \tag{34}$$

In order for the preconditioned matrix **P** to be effective, it must fulfill the following requirements. **P** should approximate **A**, the computation of $\mathbf{P}^{-1}$ should be cheap, and the condition number of the transformed system should be smaller than that of the original system of equations, as follows:

$$\kappa_2(\mathbf{P}^{-1}\mathbf{A}) = \frac{\lambda_{\max}(\mathbf{P}^{-1}\mathbf{A})}{\lambda_{\min}(\mathbf{P}^{-1}\mathbf{A})} << \kappa_2(\mathbf{A}). \tag{35}$$

For the CG method, $\mathbf{P}$ must also be an SPD matrix. Therefore, a good and common choice is Cholesky factorisation (Equation (11)), which yields $\mathbf{P} = \mathbf{C}\mathbf{C}^\mathsf{T}$. But, due to fill-in, it takes large amounts of work and memory to construct $\mathbf{C}$. Hence, the non-zero fill-in elements are disregarded, leading to the **incomplete Cholesky factorisation** of $\mathbf{P}$.

The algorithm of the CG method (Equations (27)–(32)) can be transformed into that of the preconditioned CG method by replacing $\mathbf{r}_k$ with $\mathbf{z}_k = \mathbf{P}^{-1}\mathbf{r}_k$. The modification of the CG method is summarised in Algorithm 1.

Algorithm 1 Preconditioned Conjugate Gradient (PCG) method

$\mathbf{r}_0 = \mathbf{b} - \mathbf{A}\mathbf{u}_0$ and $\mathbf{d}_0 = \mathbf{P}^{-1}\mathbf{r}_0$ ▷ Choose starting direction
for $k = 0, 1, \ldots,$ until convergence **do**
  $\mathbf{z}_k = \mathbf{P}^{-1}\mathbf{r}_k$
  $\alpha_k = \frac{\mathbf{d}_k^\mathsf{T}\mathbf{z}_k}{\mathbf{d}_k^\mathsf{T}\mathbf{A}\mathbf{d}_k}$
  $\mathbf{u}_{k+1} = \mathbf{u}_k + \alpha_k\mathbf{d}_k$
  $\mathbf{r}_{k+1} = \mathbf{r}_k - \alpha_k\mathbf{A}\mathbf{d}_k$
  $\beta_k = \frac{\mathbf{r}_{k+1}^\mathsf{T}\mathbf{z}_{k+1}}{\mathbf{r}_k^\mathsf{T}\mathbf{z}_k}$
  $\mathbf{d}_{k+1} = \mathbf{z}_{k+1} + \beta_k\mathbf{d}_k$
end for

Due to its superlinear convergence and being robust, the preconditioned CG (PCG) method is one of the best methods for solving the heat equation and the pressure-corrector equation (Equation (6)), as the coefficient matrices $\mathbf{A}$ that result from discretising these diffusion equations on the mesh are SPD.

For the general transport equations that contain the advection term, the coefficient matrix $\mathbf{A}$ is not symmetric. Therefore, the PCG method is not applicable. A significant amount of Krylov subspace methods are developed for general non-SPD matrices $\mathbf{A}$, but none possess all of the following three key properties that the CG method has for SPD matrices:

1. Being a Krylov subspace method: $\mathbf{u}_k \in \mathcal{K}_k(\mathbf{A}, \mathbf{r}_0)$.
2. Optimality: The minimiser of the quadratic function (Equation (26)) exists; hence, convergence is guaranteed.
3. Short recurrences, thus requiring decreased computing work and storage.

2.5.2. General Matrices

The **bi-conjugate gradient** (BiCG) method is a Krylov subspace method for general matrices that effectively reformulates the non-symmetric system into a form that can be treated using CG-like techniques. This is done by rewriting $\mathbf{A}\mathbf{u} = \mathbf{b}$ as

$$\begin{bmatrix} 0 & \mathbf{A} \\ \mathbf{A}^\mathsf{T} & 0 \end{bmatrix} \begin{bmatrix} \hat{\mathbf{u}} \\ \mathbf{u} \end{bmatrix} = \begin{bmatrix} \mathbf{b} \\ 0 \end{bmatrix}, \tag{36}$$

where $\hat{\mathbf{u}}$ is a dummy variable that is not solved for, but is rather only used to convert the system. The update relations in the CG method for the residuals and search directions are augmented by their shadow equivalents, which are based on $\mathbf{A}^\mathsf{T}$,

$$\hat{\mathbf{r}}_{k+1} = \hat{\mathbf{r}}_k - \alpha_k\mathbf{A}^\mathsf{T}\hat{\mathbf{d}}_k \tag{37}$$

and

$$\hat{\mathbf{d}}_{k+1} = \hat{\mathbf{r}}_{k+1} + \beta_k\hat{\mathbf{d}}_k. \tag{38}$$

The orthogonality of the residual and search direction with their shadow counterparts is ensured via the relation

$$\hat{\mathbf{r}}_i^{\mathsf{T}} \mathbf{r}_j = \hat{\mathbf{d}}_i^{\mathsf{T}} \mathbf{A} \mathbf{d}_j = 0, \quad i \neq j, \tag{39}$$

hence the name bi-orthogonal or bi-conjugate. The BiCG method is very close to the CG method and generates the same solution as the CG method if A is SPD. It does, however, require two matrix-vector multiplications (with $\mathbf{A}$ and with $\mathbf{A}^{\mathsf{T}}$) instead of one, such that every iteration is nearly twice as expensive as compared to the CG method.

Although the BiCG method shares two of the key properties of the CG method, there is no optimality for general matrices $\mathbf{A}$; hence, convergence is not guaranteed. Moreover, irregular convergence can lead to serious breakdown due to large rounding errors. The **Biconjugate Gradient STABilised** (Bi-CGSTAB) method [15] is a more robust variant where the recurrence relations $\tilde{\mathbf{r}}_k = P_k(\mathbf{A})\mathbf{r}_0$ are modified to the form

$$\tilde{\mathbf{r}}_k = Q_k(\mathbf{A}) P_k(\mathbf{A}) \mathbf{r}_0 \tag{40}$$

such that multiplication with $\mathbf{A}^{\mathsf{T}}$ is avoided. For this recurrence, a k-th degree polynomial is taken of the form

$$Q_k(\mathbf{A}) = (\mathbf{I} - \omega_1 \mathbf{A})(\mathbf{I} - \omega_2 \mathbf{A}) \dots (\mathbf{I} - \omega_k \mathbf{A}) \tag{41}$$

with suitable constants ω_k in the k-th iteration that minimise residual r_k with respect to ω_k, giving the method a semi-optimality property and smoother convergence behaviour as compared to the BiCG method. The Bi-CGSTAB method is currently the most recent and advanced Krylov subspace method that is implemented in the standard library of OpenFOAM, and it can be preconditioned using, for example, **incomplete LU factorisation**, which is appropriate for general matrices. It is much faster than its predecessor, but sudden breakdown due to rounding errors still occur for the more difficult problems in this paper.

2.5.3. Other Krylov Subspace Methods

Although not used in this work, it is worth mentioning the **Generalised Minimal RESidual** (GMRES) method [2], which falls under another class of Krylov subspace methods for solving general systems of linear equations. It is a stable method with respect to rounding errors, and has an optimality property that guarantees convergence. While the CG method aims at minimising the A-norm of the error $\mathbf{u} - \mathbf{u}_k$ (only possible for SPD matrices), the GMRES method finds orthogonal vectors (search directions) that minimise the Euclidean norm of the residual $\mathbf{r}_k$, leading to a comparable superlinear convergence behaviour to the CG method. The main drawback, however, is that the whole sequence of search directions from the previous iterates have to be stored and multiplied with, such that the computational and memory requirements quickly become prohibitive if the method does not converge after a few iterations. Restarting the method after a certain number of iterations overcomes this limitation of long recurrences, but comes at the cost of losing its nice convergence properties due to throwing away all search information [16]. The generalised conjugate residual (GCR) method is similar to the GMRES method but with somewhat more floating-point operations per iteration. However, the GCR method gives the ability to truncate to the last few search directions that are saved for the next iteration, which in return performs better than restarting the GMRES method.

Hybrid methods provide a balance between the short recurrences of the BiCG-type methods on one side and the optimality property of the GMRES-type methods on the other. First on the list is the **GMRES Recursive** (GMRESR) method [16], which consists of an inner and outer loop. The solution is first approximated in an inner-loop with the restarted GMRES method, after which the found search directions are condensed to the

outer-loop, where they are used to approximate the solution with the truncated GCR method. The GMRESR method generally speeds up the convergence rate of the restarted GMRES method. Later improvements in the GMRESR-type methods include **GCRO**-type methods, where the inner iteration loop takes place in a Krylov subspace orthogonal to the subspace of the outer loop's (so-called subspace recycling), yielding further acceleration of the convergence rate [17,18].

The final hybrid method worth mentioning, which leans more towards the BiCG-type methods, is the **Induced Dimension Reduction** (IDR) method. This method was already proposed before the Bi-CGSTAB method but has been revived to a more generalised variant IDR(s) [19]. The IDR(s) method is closely related to the Bi-CGSTAB method in the sense that the matrix-vector multiplication with $\mathbf{A}^{\mathsf{T}}$ is avoided via a minimising polynomial. The main difference is that the generated residuals are forced to be in subspaces of decreasing order until an s-dimensional space is reached. For the right value of s (typically $s \leq 10$), this robust and efficient method is at least as fast as the Bi-CGSTAB method when $s = 1$ (original IDR method) and significantly faster for increased values of s, especially for more difficult problems. When s is large enough, the rate of convergence is nearly as good as full GMRES [20]. Hybrid methods such as the **IDR(s)** method and the **GCRO**-type methods are good candidates for the next generation of linear solvers for general matrices to implement in OpenFOAM.

3. Impact of Linear Solvers on Convergence of the SIMPLE Algorithm

The speed of convergence towards a solution depends not only on the type of linear solvers used, but also on the type of problem that is being solved. This section evaluates the performance of the advanced linear solvers in OpenFOAM by conducting several numerical experiments on different cases.

3.1. Relative Tolerance

The solver algorithms of OpenFOAM are constructed in 'inner' and 'outer' loops. In the inner loops, the linear systems of equations are solved sequentially, and this is where the linear solvers do the work, which are referred to as 'inner' iterations. The algorithm proceeds to the next linear system of equations in the inner loop once the current relative tolerance target is reached. In the outer loop, the flow develops to the next (pseudo) time step, which may or may not require a time integration, depending on whether the SIMPLE algorithm is used for steady-state problems or the PISO algorithm for transient problems, or combinations thereof (PIMPLE). As the algorithms are iterative themselves, the linear solvers do not need to iterate to the smallest possible error in each loop before the algorithm progresses.

To illustrate, the 2D steady-state heat equation is a problem that does not require time integration or a flow to be developed. As such, this problem is solved within one outer iteration, and even up to one inner iteration with the most efficient solver, as is shown in Figures 6 and 7.

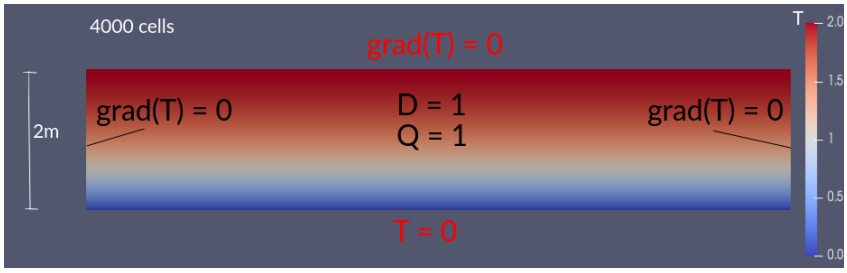

Figure 6. Example 2D steady-state heat equation problem on a rectangular surface with 4000 equally divided cells, along with its boundary conditions.

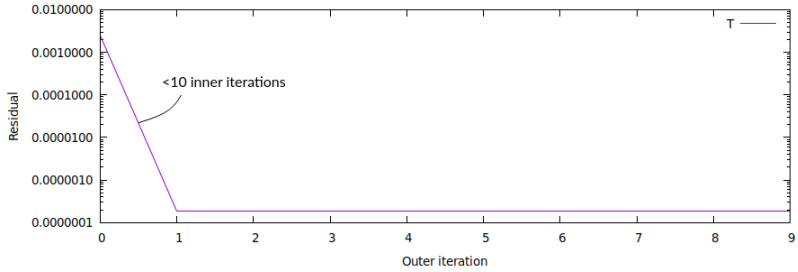

Figure 7. Residual plot of the 2D steady-state heat equation problem.

In a fluid-related problem, however, the flow physically develops on each outer iteration. Aiming for a low relative tolerance per inner loop can be more stable, but it also makes it much more time-consuming. The famous Pitz and Daily's wind tunnel tutorial case [6] demonstrates that using a relative tolerance of 10^{-1} instead of 10^{-3} for all equations will lead to about twice as fast convergence, while a relative tolerance of 10^{-3} leads to smoother convergence with less oscillations, as is shown in Figure 8.

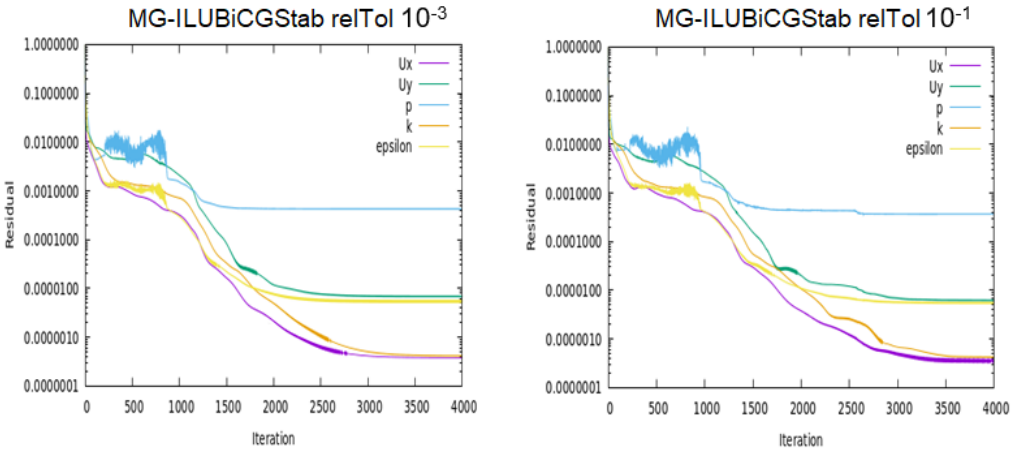

Figure 8. Comparison between two simulations of the Pitz and Daily wind tunnel tutorial case using 10^{-3} and 10^{-1} as relative tolerances, respectively. The pressure equation is solved with the GAMG method, while the other equations are solved with the Bi-CGSTAB method and ILU preconditioner.

3.2. Numerical Experiment: Non-Reacting Flow

This section evaluates the performance of the linear solvers on a three-dimensional, full-scale, non-reacting and non-rotating kiln. The used solver is `rhoSimpleFOAM`, which is the standard solver for turbulent non-reacting steady-state compressible flows.

The geometry is shown in Figures 9 and 10, where the air flow enters the domain at atmospheric pressure through the primary and secondary inlets with 0.15 kg/s and 1.6 kg/s, respectively, while the fuel inlet flow is excluded.

For this kiln, the Cartesian mesh is employed with about 2.3 million cells, as shown in Figures 11 and 12. It consists of predominantly hexahedrons (96%) with polyhedrons in the transitions of the nine refinement levels, as well as prisms to connect with the surface boundary layers that are not aligned with the grid. The mesh is highly orthogonal yet unstructured, which disconnects the burner inlets from the rest of the domain and allows local refinement without deforming the cells too much. The mesh is generated with cfMesh, which is a cross-platform library for automatic volume mesh generation based on OpenFOAM.

The resulting linear systems from the finite volume discretisation are large and sparse; the pressure equation leads to (near) SPD systems, while the momentum and scalar transport equations give rise to non-symmetric systems.

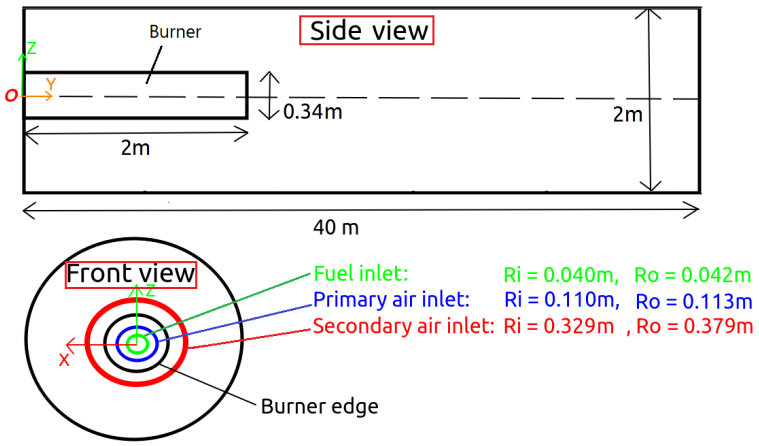

Figure 9. Global dimensions of the kiln, with the inner and outer radii of the inlets.

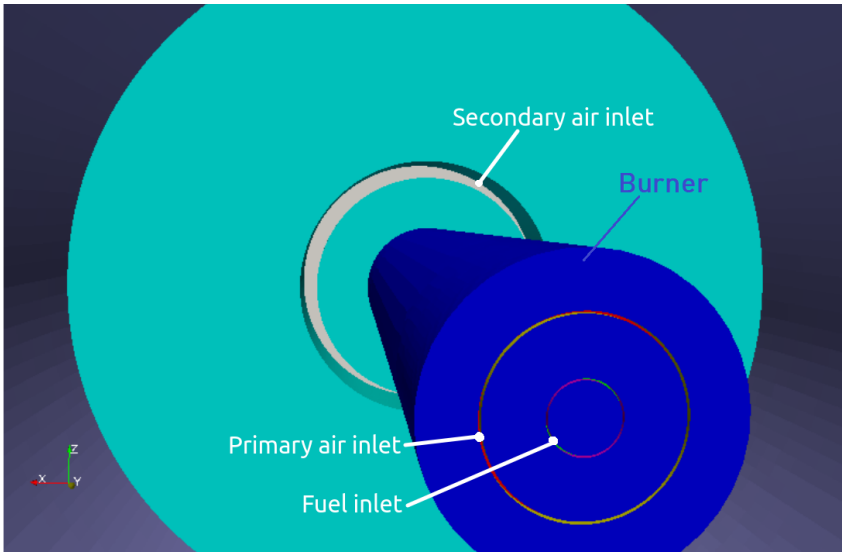

Figure 10. Inside view of the burner side of the kiln with the three inlets.

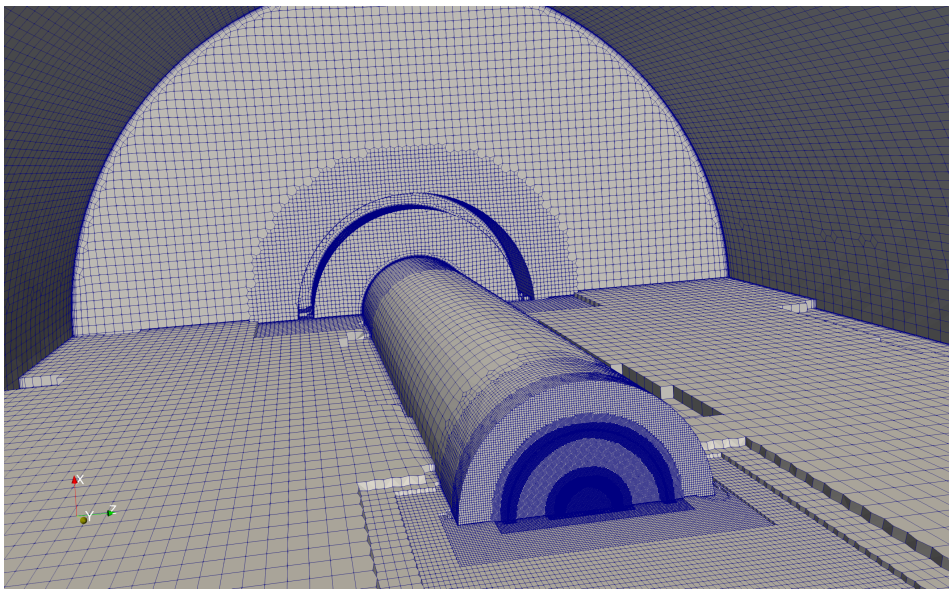

Figure 11. Inside view of the mesh at the burner side of the kiln.

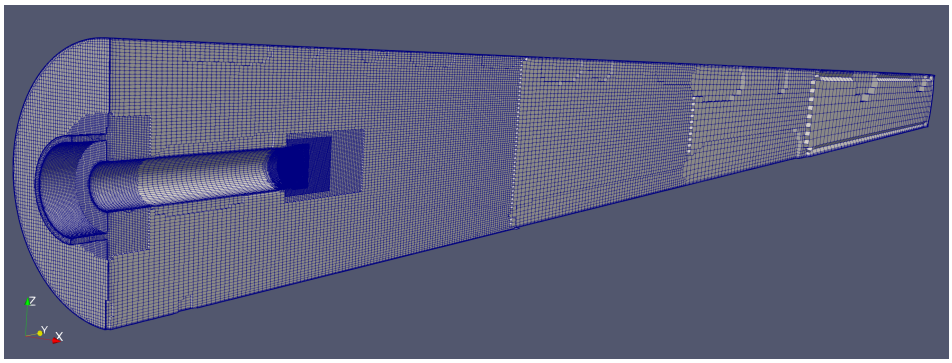

Figure 12. Longitudinal cross section of the mesh.

Further boundary conditions are shown in Table 2. All simulations are run for 5000 outer iterations on 20 cores in a single node.

Table 2. Boundary conditions for the kiln model.

Variable	Primary Air	Secondary Air	Walls	Outlet
T [K]	293	773	zG *	zG
Y_{CH_4} [-]	0	0	zG	-
Y_{O_2} [-]	0.23	0.23	zG	-
Y_{N_2} [-]	0.77	0.77	zG	-
$\dot{m}$ [kg/s]	0.15	1.6	-	-

* zG stands for the Neumann boundary condition zeroGradient.

The simulations are run in steady-state. The central differencing scheme is used for the gradient and Laplacian terms. The advection terms for velocity and enthalpy are discretised using the second-order upwind scheme, while for the kinetic energy and energy dissipation, this is done with the first-order upwind scheme to prevent OpenFOAM from crashing.

Figure 13 shows the basic stream pattern in the longitudinal cross-section, where the annular-shaped recirculation zone surrounding the burner is spotted.

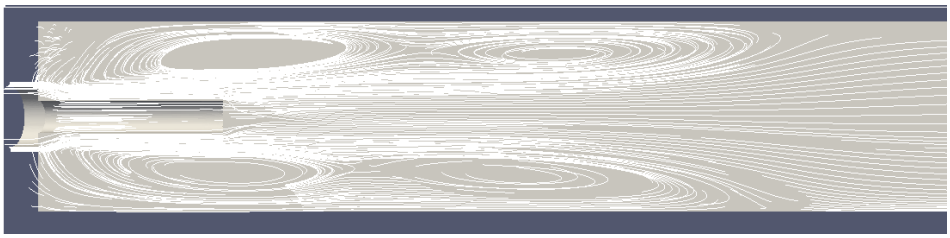

Figure 13. Stream pattern inside the kiln for the studied case in this section.

3.2.1. Solver Combinations

The most advanced standard linear solvers of OpenFOAM are tested on different applications in different combinations, as shown in Table 3. When a Krylov subspace method is used, the solver name is preceded with a preconditioner. Instead of the IC and ILU factorisations, OpenFOAM implements simplified diagonal-based variants as a preconditioner, respectively, the DIC and DILU, where off-diagonal terms are dropped. The reciprocal of the preconditioned diagonal is calculated and stored. The MG solver can also be used as a preconditioner. The settings for OpenFOAM's MG solver (GAMG) are shown in Appendix A, and for these type of problems with complex shape and flows, the geometric variant of the GAMG method (agglomerator: faceAreaPair) is preferred, which is about twice as fast as the algebraic variant for this problem. Finally, OpenFOAM's BIM solver (`smoothSolver`) is also tested using the Gauss–Seidel (GS) smoother.

Table 3. Tested solver combinations. The first solver of each method is used for the pressure equation, while the second solver is used for all other independent variables.

Method	p Eq.	Other Variables
MG-DILUBiCGSTAB	Solver: MG	Solver: Bi-CGSTAB, Prec: DILU
MG-MG	Solver: MG	Solver: MG
DICCG-MG	Solver: CG, Prec: DIC	Solver: MG
DICCG-DILUBiCGSTAB	Solver: CG, Prec: DIC	Solver: Bi-CGSTAB, Prec: DILU
MGCG-DILUBiCGSTAB	Solver: CG, Prec: MG	Solver: Bi-CGSTAB, Prec: DILU
MGCG-MGBiCGSTAB	Solver: CG, Prec: MG	Solver: Bi-CGSTAB, Prec: MG
MGCG-GSBIM	Solver: CG, Prec: MG	Solver: BIM, Smoother: GS

3.2.2. Results

Before discussing the linear solver performance, it is observed that using the advanced solvers of OpenFOAM usually leads to one or two inner iterations per outer iteration for all variables, except with the pressure equation, which usually requires one or more orders of magnitude of inner iterations before moving to the next outer iteration. Therefore, the pressure equation is the major bottleneck that delays the solution from converging quickly. This occurs for most non-reacting flow cases. Figure 14 shows that the BIM solver for the pressure equation will not converge within a thousand inner iterations throughout the simulation. This is because the pressure-corrector convergence is usually dominated by low-frequency error, which is why the BIM performs poorly and should not be used for pressure. Out of the advanced methods, the CG method with the DIC preconditioner is the least efficient method for the pressure equation, requiring two orders of magnitude of inner iterations on average to reach the relative target of 10^{-3}. The MG method performs significantly better, and the most efficient method is the combination of the former two: the CG method with the MG as a preconditioner.

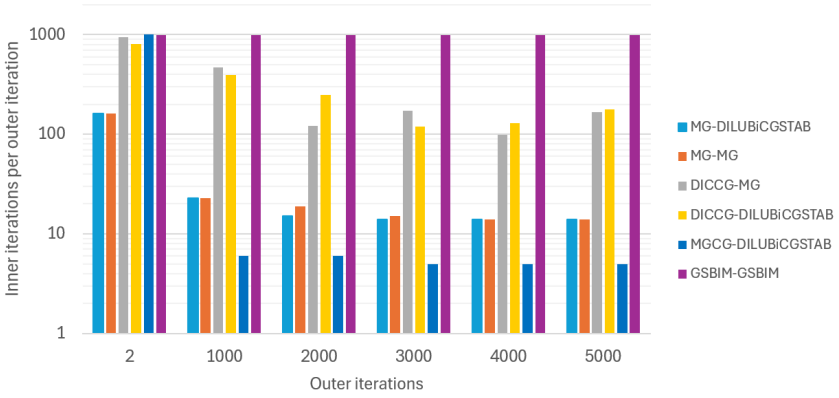

Figure 14. Number of inner iterations of the pressure equation per outer iteration, using a relative tolerance of 10^{-3}.

Figure 15 and Table 4 show the overall performance of the different linear solver combinations. Three criteria are used to determine the speed of convergence:

- The time (in hours) to reach 5000 outer iterations;
- The number of outer iterations to reach convergence;
- The time (in hours) to reach convergence.

The time needed to reach 5000 iterations with the different combinations is in line with the amount of required inner iterations for the pressure equation, as is discussed above, since the pressure equation is the bottleneck.

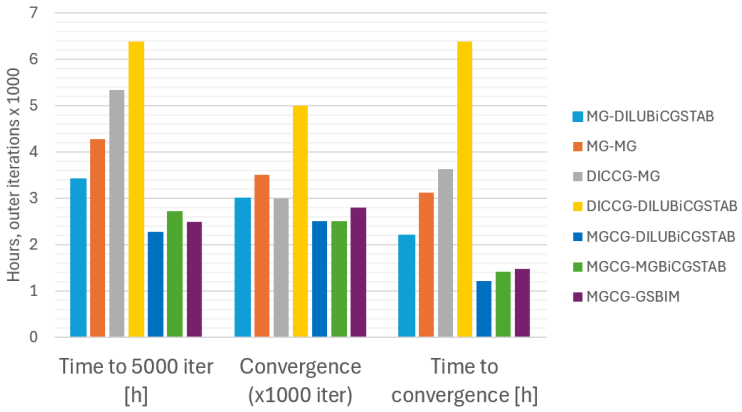

Figure 15. Simulation durations of the different methods w.r.t. time and outer iterations, using a relative tolerance of 10^{-3}.

When looking at the number of outer iterations required to converge, the CG method with an MG preconditioner also requires the least amount of iterations. This further accelerates the convergence time, which is a factor two to three times faster than the other solver combinations. However, if the solver is forced to a strictly compressible flow environment by setting the transonic option to on, the pressure equation becomes non-symmetric and the MG method will not work.

With regard to the other variables, the fastest solver is the Bi-CGSTAB method with the DILU preconditioner. The MG method, as solver or as preconditioner, is a bit slower because the other variables only require one or two inner iterations to reach the relative target of 10^{-3}, and one MG operation is computationally more expensive than one Bi-CGSTAB operation and much more expensive than one DILU operation.

In contrast to the pressure equation, the BIM method works well for the other equations and reaches convergence just as fast as the Bi-CGSTAB method. The BIM requires significantly more inner iterations to reach the relative target per outer iteration, but it is a simple and computationally cheap method.

Table 4. Simulation time of the different methods w.r.t. time and outer iterations, using a relative tolerance of 10^{-3}.

Method	Time to 5000 Iter	Convergence	Time to Convergence
MG-DILUBiCGSTAB	3.4 h	3000 iter	2.2 h
MG-MG	4.3 h	3500 iter	3.1 h
DICCG-MG	5.3 h	3000 iter	3.6 h
DICCG-DILUBiCGSTAB	6.4 h	5000 iter	6.4 h
MGCG-DILUBiCGSTAB	2.3 h	2500 iter	1.2 h
MGCG-MGBiCGSTAB	2.7 h	2500 iter	1.4 h
MGCG-GSBIM	2.5 h	2800 iter	1.5 h

The simulation can be accelerated further by relaxing the relative target from 10^{-3} to 10^{-1}, as is shown in Figure 16 and Table 5. The use of the highly orthogonal Cartesian mesh justifies the loose relative target of 10^{-1}, as the global continuity error reaches 10^{-11} when converged, which is effectively zero, indicating that no net mass imbalance is present. However, this relative target may lead to less stability. In fact, when using the DICCG-DILUBiCGSTAB or MGCG-GSBIM method, the solution will even quickly diverge, as can be seen in Figure 17, while the MGCG-DILUBiCGSTAB method remains stable and the convergence is accelerated with 25 % as compared with the relative tolerance of 10^{-3}. Comparing these two extremes, with on one side the slowest of the selected methods, which is restricted to a relative target of 10^{-3}, and on the other side, the fastest method,

which is stable at a relative target of 10^{-1}, a speed-up of nearly a factor of 7 is reached. This speed-up is even higher if compared with the case where the `smooth-Solver` is applied for the pressure, but this should always be avoided.

Table 5. Simulation time of the different methods w.r.t. time and outer iterations, using a relative tolerance of 10^{-1}.

Method	Time to 5000 Iter	Convergence	Time to Convergence
MG-DILUBiCGSTAB	2.0 h	3000 iter	1.2 h
MG-MG	2.5 h	4000 iter	2.0 h
DICCG-MG	2.7 h	3000 iter	1.8 h
DICCG-DILUBiCGSTAB	-	Diverged	-
MGCG-DILUBiCGSTAB	1.5 h	3000 iter	0.9 h
MGCG-GSBIM	-	Diverged	-

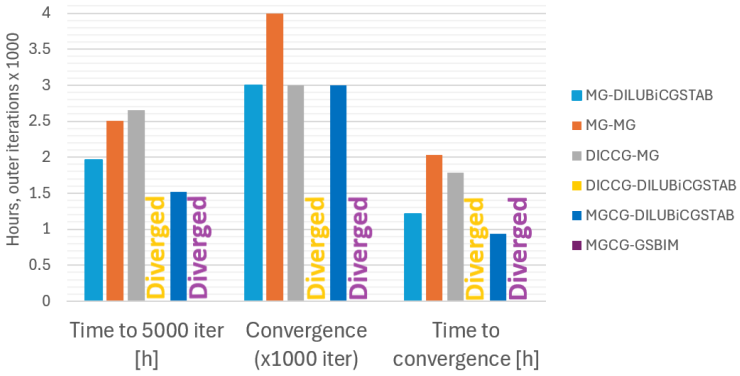

Figure 16. Simulation durations of the different methods w.r.t. time and outer iterations, using a relative tolerance of 10^{-1}.

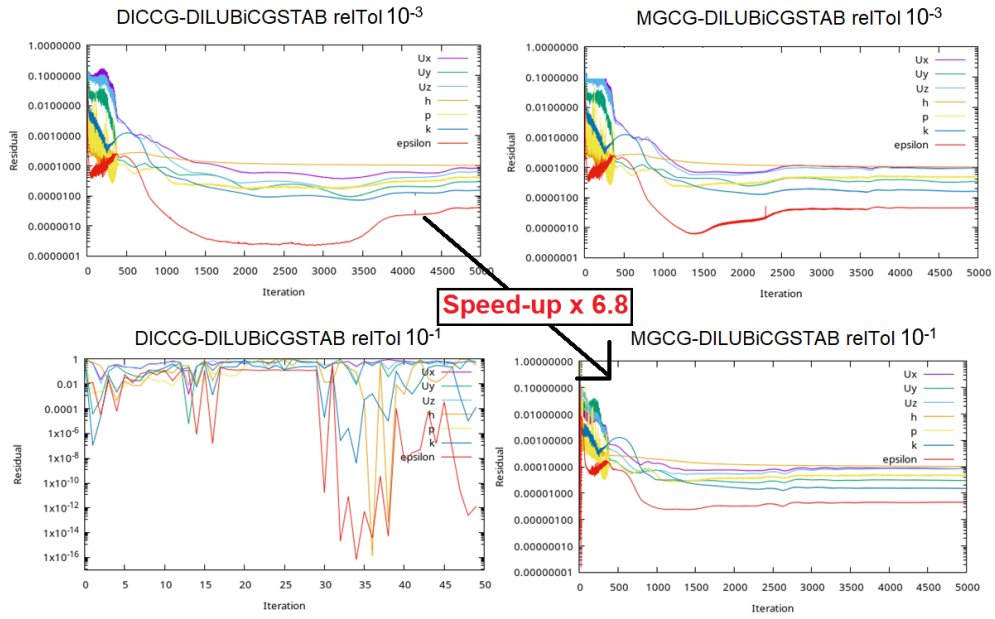

Figure 17. Residuals of the studied case for the least efficient and most efficient solver combination.

3.2.3. Further Advancements

As discussed at the end of Section 2, faster linear solvers than the ones provided by OpenFOAM do exist. The state-of-the-art linear solvers (such as IDR and GCRO) and preconditioners can be found in the open-source library Portable, Extensible Toolkit for Scientific computation (PETSc), and a plug-in for OpenFOAM was developed in [21]. Their

work also discusses the very poor scalability of the MG preconditioner of OpenFOAM for massively parallel clusters (10^3 cores), whereas the DIC preconditioner is shown to have outstanding superlinear scalability. It is reported that the DIC preconditioner overtakes the MG preconditioner in convergence time when using more than 2000 cores for a 3D laminar lid-driven cavity flow problem consisting of 64 million cells. The PETSc counterpart of OpenFOAM's MG preconditioner solves the scalability issue.

4. Conclusions

This work demonstrated the large differences in speed and robustness between the linear solvers available in OpenFOAM, with convergence times varying by more than one order of magnitude. The pressure-corrector equation was the major bottleneck that delayed the solution from converging quickly. Several of the more advanced standard linear solvers were tested with a 20-core high-performance computer on a 3D full-scale kiln consisting of 2.3 million cells.

The most efficient and stable solution method turned out to be the conjugate gradient (CG) solver, combined with the Generalised Geometric–Algebraic MultiGrid (GAMG) as a preconditioner. A speed-up factor of 7 was reached as compared with the slowest of the advanced methods. Even higher speed-ups were reached compared with the basic linear solvers of OpenFOAM. Also, the stability was enhanced when the GAMG was used as a preconditioner for the Bi-CG stabilised (Bi-CGSTAB) method instead of a simple diagonal-based preconditioner.

Finally, due to the iterative nature of the SIMPLE algorithm, a faster linear solver than the MG-CG combination was redundant for this case as each time step required one or two iterations. It should be noted, however, that the results presented are based on a fixed parallel configuration, and the relative performance of the solvers may differ for large-scale parallel computations, where alternative preconditioners have been shown to exhibit better scalability.

Author Contributions: Conceptualisation, M.E.A.; methodology, M.E.A.; software, M.E.A.; validation, M.E.A. and C.V.; formal analysis, M.E.A.; investigation, M.E.A.; resources, C.V.; data curation, M.E.A.; writing—original draft preparation, M.E.A.; writing—review and editing, C.V.; visualisation, M.E.A.; supervision, C.V.; project administration, C.V.; funding acquisition, C.V. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflicts of interest.

Appendix A

The settings for OpenFOAM's MG solver (GAMG) are shown below:

```
solver                  GAMG;
tolerance               1e-9;
relTol                  e-3; //e-1;
smoother                GaussSeidel;
cacheAgglomeration      true;
nCellsInCoarsestLevel   1500;
agglomerator            faceAreaPair; //algebraicPair;
mergeLevels             1;
```

```
nPreSweeps          1; //1 for p, 0 for all other
nPostSweeps         2;
nFinestSweeps       2;
maxIter             800;
directSolveCoarsest false;
```

where the number of cells at the coarsest level is set to 1000, which is roughly the square root of the total amount of cells.

References

1. Website of the OpenFOAM Foundation. Available online: https://openfoam.org/ (accessed on 5 March 2026).
2. Saad, Y. *Iterative Methods for Sparse Linear Systems*; SIAM: Philadelphia, PA, USA, 2003. [CrossRef]
3. Ferziger, J.H.; Peri'c, M. *Computational Methods for Fluid Dynamics*, 3rd ed.; Springer: Berlin/Heidelberg, Germany, 2002.
4. Versteeg, H.K.; Malalasekera, W. *An Introduction to Computational Fluid Dynamics: The Finite Volume Method*, 2nd ed.; Pearson Education: Harlow, UK, 2007.
5. Moukalled, F.; Mangani, L.; Darwish, M. *The Finite Volume Method in Computational Fluid Dynamics: An Advanced Introduction with OpenFOAM and Matlab*; Springer: Cham, Switzerland, 2016. [CrossRef]
6. Greenshields, C. *OpenFOAM User Guide*; OpenFOAM Foundation Ltd.: London, UK, 2017.
7. Patankar, S.V. *Numerical Heat Transfer and Fluid Flow*; Hemisphere Publishing: London, UK; McGraw-Hill: Columbus, OH, USA, 1980. [CrossRef]
8. Golub, G.H.; Loan, C.F.V. *Matrix Computations*, 4th ed.; The Johns Hopkins University Press: Baltimore, MD, USA, 2013.
9. Fulton, S.; Ciesielski, P.; Schubert, W. Multigrid Methods for Elliptic Problems: A Review. *Mon. Weather Rev.* **1986**, *114*, 943. [CrossRef]
10. Tielen, R.; Möller, M.; Göddeke, D.; Vuik, C. p-multigrid methods and their comparison to h-multigrid methods within Isogeometric Analysis. *Comput. Methods Appl. Mech. Eng.* **2020**, *372*, 113347. [CrossRef]
11. Trottenberg, U.; Oosterlee, C.; Schüller, A. *Multigrid*, 1st ed.; Academic Press: London, UK, 2001.
12. Wesseling, P.; Oosterlee, C. Geometric multigrid with applications to computational fluid dynamics. *J. Comput. Appl. Math.* **2001**, *128*, 311–334. [CrossRef]
13. Behrens, T. *OpenFOAM's Basic Solvers for Linear Systems of Equations*; Technical Report; Technical University of Denmark: Lundtofte, Denmark, 2009.
14. Wesseling, P. *Principles of Computational Fluid Dynamics*, 1st ed.; Springer: Berlin/Heidelberg, Germany, 2001. [CrossRef]
15. van der Vorst, H.A. Bi-CGSTAB: A Fast and Smoothly Converging Variant of Bi-CGfor the Solution of Nonsymmetric Linear Systems. *SIAM J. Sci. Stat. Comput.* **1992**, *13*, 631–644. [CrossRef]
16. van der Vorst, H.A.; Vuik, C. GMRESR: A family of nested GMRES methods. *Numer. Linear Algebra Appl.* **1994**, *1*, 369–386. [CrossRef]
17. Amritkar, A.; de Sturler, E.; Świrydowicz, K.; Tafti, D.; Ahuja, K. Recycling Krylov subspaces for CFD applications and a new hybrid recycling solver. *J. Comput. Phys.* **2015**, *303*, 222–237. [CrossRef]
18. Thomas, S.; Baker, A.; Gaudreaulte, S. Augmented MGS-CGS Block-Arnoldi Recycling Solvers. *SIAM J. Sci. Comput.* **2025**, *47*, A1458–A1485. [CrossRef]
19. Sonneveld, P.; van Gijzen, M. IDR(s): A Family of Simple and Fast Algorithms for Solving Large Nonsymmetric Systems of Linear Equations. *SIAM J. Sci. Comput.* **2009**, *31*, 1035–1062. [CrossRef]
20. Sonneveld, P. *On the Convergence Behaviour of IDR (s)*; Reports of the Department of Applied Mathematical Analysis 10-08; Department of Applied Mathematical Analysis: Delft, The Netherlands, 2010.
21. Bnà, S.; Spisso, I.; Olesen, M.; Rossi, G. PETSc4FOAM: A Library to Plug-In PETSc into the OpenFOAM Framework. PRACE White Paper. 2020. Available online: https://zenodo.org/records/3923780?preview_file=WP294.pdf (accessed on 18 February 2026).