Delft University of Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft Institute of Applied Mathematics

Parallelization of An Experimental Multiphase Flow Algorithm

Master Thesis Literature Review
submitted to the
Delft Institute of Applied Mathematics

in partial fulfillment of the requirements
for the degree

MASTER OF SCIENCE
in
APPLIED MATHEMATICS

by

ANKIT MITTAL
Delft, the Netherlands
March 2016

Table of Contents

1. INTRODUCTION

Multiphase flows are one of the most widely occurring flows in nature. Such flows either have two or more immiscible fluids separated by an interface or one or more fluids, again separated by an interface, which are in different phases. Such flows normally occur in atmosphere (for example bubbly flows), chemical reactors, turbo-machines, fuel injectors, pipe flows, etc. Multiphase flows are also of particular interest to the petroleum industry where such flows often occur in wells and pipelines during oil and gas production.

In multiphase flows, since the interface also gets advected with the flow, accurate calculation of the interface location is necessary for accurate prediction of the flow field. In order to calculate the interface at each time step one can use either the Level Set (LS) or the Volume of Fluids (VOF) method. The LS method, though computationally cheap, is rather inaccurate, while the VOF method though more accurate, is computationally very expensive. In [1] van der Pijl studied bubbly flows, and to calculate the interface an approach based on the combination of the level set and volume of fluid methods (MCLS) was adopted. This approach gave more accurate results (as compared to the Level set method) without incurring prohibitively large computational costs.

A similar project titled “CFD to study instabilities in 3d flows” is currently being carried out at TU Delft jointly with TNO Netherlands, Shell, and Deltares. The aim of the project is to study multiphase flows in pipes, where those are initially filled with oil and water is pumped to flush the oil out or vice-versa. A somewhat similar approach as [1] is employed here to predict the interface and a working code is available. The code extensively uses the Krylov subspace methods to iteratively solve the obtained linear system of equations. The aim of this thesis is to improve the speedup of the available code.

In multiphase flows, there is a jump in viscosity and density across the interface due to the difference in properties of the two fluids. This jump slows down the convergence of the iterative solvers. The present code as well is plagued by the slow convergence and hence proper preconditioners to improve the convergence are analyzed in this endeavor. Also, we consider how to improve the performance of the given code by changing its structure, and by using less computation and memory intensive solvers.

As size of the system grows, even the best solvers with the most suitable preconditioners take a prohibitively large amount of time to generate the numerical results on a single processor. The way-out is to use parallel programming and to split the whole big system into smaller more manageable pieces, and distribute them amongst the available processors. But parallelization is far from straightforward, as explained later, if the number of processors increase the convergence behavior of the preconditioner deteriorates. Hence techniques like deflation have to be used to restore the performance of the preconditioner. Therefore we study the feasibility of parallelization with the deflation technique. To suppress the communication cost, communication overlapping solvers are also studied.

Furthermore, we acknowledge that the decoupled time integration method used in the available code is not kinetic energy conserving, which might be of interest in turbulent flow cases. To preserve the kinetic energy, one has to integrate the flow field in a coupled manner. The coupled system suffers from a very poor convergence, therefore we also study a few preconditioners to improve the convergence of the coupled solvers.

In this report, section (2) explores in detail the physics of incompressible multiphase flows and in section (3) the structure of the code is discussed. Further, sections (4) and (5) deal with suitable solvers and preconditioners, while section (6) covers the parallelization study for the current system. Later in section (7), some already obtained speed-up results are presented.

2. GOVERNING EQUATIONS & SOLVING TECHNIQUES

In the present study we deal with a multiphase flow between the two fluids which are separated by a sharp interface. The flow is characterized by the velocity $\mathbf{v}(x,y,z,t)$ and the pressure $p(x,y,z,t)$ (where bold indicates a vector). Due to the assumption of incompressibility, the fluids on either side have different but constant densities and viscosities, also we assume the flow to be isothermal and Newtonian.

In this section we present a very brief overview of the governing equations based on [1], for a more detailed discussion one is referred to [1, 2]. The physics of our problem is split into two distinct parts, the flow and the interface. In our approach they are treated separately, hence we present them one after the other. First we shall discuss the flow part.

2.1. FLOW

The flow is governed by the 3-d unsteady incompressible Navier-Stokes equations. In this study we are not interested in the thermal energy, and since in incompressible flows the energy equation is decoupled from the momentum and continuity equations we do not solve it. Also, since we are essentially dealing with pipe flows, the cylindrical coordinates system is an obvious choice for the coordinate system in which the Navier-Stokes equations are solved. The N-S equations in cylindrical coordinates are,

Continuity equation,

$$\frac{1}{r}\frac{\partial r u_r}{\partial r}+\frac{1}{r}\frac{\partial u_\theta}{\partial \theta}+\frac{\partial u_z}{\partial z}=0$$

Radial momentum equation,

$$\frac{\partial u_r}{\partial t}+\frac{\partial (u_r u_r)}{\partial r}+\frac{1}{r}\frac{\partial (u_r u_\theta)}{\partial u_\theta}+\frac{\partial (u_r u_z)}{\partial z}+\frac{(u_r u_r - u_\theta u_\theta)}{r} = -\frac{1}{\rho}\frac{\partial p}{\partial r}+\rho g_r+\mu\left[\frac{\partial \tau_{rr}}{\partial r}+\frac{1}{r}\frac{\partial \tau_{\theta r}}{\partial \theta}+\frac{\partial \tau_{zr}}{\partial z}+\frac{(\tau_{rr}-\tau_{\theta\theta})}{r}\right]+f_r$$

Angular momentum equation,

$$\frac{\partial u_\theta}{\partial t}+\frac{\partial (u_\theta u_r)}{\partial r}+\frac{1}{r}\frac{\partial (u_\theta u_\theta)}{\partial \theta}+2\frac{u_r u_\theta}{r}+\frac{\partial (u_\theta u_z)}{\partial z}=-\frac{1}{\rho r}\frac{\partial p}{\partial \theta}+\rho g_\theta+\left[\frac{\partial \tau_{r\theta}}{\partial r}+\frac{1}{r}\frac{\partial \tau_{\theta\theta}}{\partial \theta}+2\frac{\tau_{r\theta}}{r}+\frac{\partial \tau_{z\theta}}{\partial z}\right]+f_\theta$$

Axial momentum equation,

$$\frac{\partial u_z}{\partial t}+\frac{\partial (u_r u_z)}{\partial r}+\frac{1}{r}\frac{\partial (u_\theta u_z)}{\partial \theta}+\frac{\partial u_z u_z}{\partial z}+\frac{(u_r u_z)}{r}=\frac{-1}{\rho}\frac{\partial P}{\partial z}+\rho g_z+\left[\frac{\partial \tau_{rz}}{\partial r}+\frac{1}{r}\frac{\partial \tau_{\theta z}}{\partial \theta}+\frac{\partial \tau_{zz}}{\partial z}+\frac{\tau_{rz}}{r}\right]+f_z$$

where ρ is the density, p is the pressure, g is the gravitational constant, r,θ,z are the space variables, u_r,u_θ,u_z are the velocity components and f_r,f_θ,f_z are the external force components in respective r,θ,z directions. If we mark the two fluids as 0 and 1, then to separate the two fluid regimes, we introduce a so-called color function χ defined as

$$\chi(\boldsymbol{x})=\begin{cases}0, & \boldsymbol{x}\in fluid\,0\\ 1, & \boldsymbol{x}\in fluid\,1\end{cases}$$, where $\boldsymbol{x}$ is the position vector

then the density and viscosity can be expressed as

$$\rho=\rho_0+(\rho_1-\rho_0)\Psi, \qquad \mu=\mu_0+(\mu_1-\mu_0)\chi ,$$

where subscripts 0 and 1 indicate the respective fluids, and Ψ is the VOF function (discussed in the next subsection). To get a smooth pressure and gradient of velocity across the interface we regularize the color function, i.e., smear it out over a small but finite distance.

2.2. INTERFACE

For the interface convection we follow the Eulerian approach and only look at a fixed space with a fixed grid. For numerical treatment of the interface we use the Volume Tracking methods. In these methods we assign a color to each of the fluid regimes and the region where the color function changes implicitly define the interface. The benefit of such a method is that the changes in the interface topology and coalescence are automatically taken care of.

The volume tracking methods can be sub-categorized into two subcategories viz. the Level Set (LS) method and the Volume of Fluid (VOF) method. In both LS and VOF methods the fluid interface is identified by some coloring function and the function is advected in an Eulerian way as,

$$\frac{\partial \Phi}{\partial t} + \boldsymbol{u} . \nabla(\Phi) = 0 \ .$$

We further discuss each of the above two techniques in some detail.

2.2.1. LEVEL SET METHOD

In the Level Set method the interface is defined by the marker function Φ . The marker function is defined to be positive in one fluid and negative in the second fluid. Hence, the locations where the marker function is zero marks the location of the interface, i.e.,

$$Interface = \{\boldsymbol{x} \mid \Phi(\boldsymbol{x}, t) = 0\} \ .$$

The signed distance function $d(\boldsymbol{x}, t)$ is a well suited marker function. The level set function is advected according to

$$\frac{\partial \Phi}{\partial t} + \boldsymbol{u} . \nabla(\Phi) = 0 \ .$$

The level set function Φ is a smooth function that, unlike VOF, allows for a straightforward calculation of the interface curvature. The main disadvantage is that if the interface is advected using this approach the mass is not preserved due to the viscous and convective smoothing. Another disadvantage of LS is that when it is advected through a non-uniform flow it may no longer correspond to a distance function. To remedy this one has to reinitialize the level set function after, some or every numerical integration step depending on some criterion.

2.2.2. VOLUME OF FLUID METHOD

In the VOF method, we use the volume of fluid function Ψ to implicitly define the location of the interface. This function measures the fractional volume of a certain fluid in a computational cell. Ψ can be 0 or 1 or somewhere in between if the computational cell is filled with both the fluids, i.e., it contains an interface. The cells which contain the interface are called mixed cells. We mathematically define Ψ for each cell as

$$\Psi(\boldsymbol{x}_k)=\frac{1}{\Omega_k}\int_{\Omega_k}\chi \, d\Omega \ ,$$

where χ is the color function (defined for each cell) which is 0 in one fluid and 1 in the second fluid, $\boldsymbol{x}_k$ is the node and Ω_k is the volume of the corresponding computational cell k . The interface is advected using

$$\frac{\partial\chi}{\partial t}+\boldsymbol{u}.\nabla(\chi)=0 \ .$$

The advantage of the VOF technique is that it is mass conserving, unlike the LS method. The main disadvantage is that the evaluation of normals and curvatures, and the interface reconstruction are much more tedious and computationally expensive.

Because of the advantages and disadvantages of both the methods, there is no clear choice amongst them. In the given code a hybrid of both is used which, while being mass conserving, is relatively easy to evaluate. The details of this hybrid model are not discussed here and one is referred to [1] for more information.

2.3. DISCRETIZATION AND LINEARIZATION

In the present study we use the Finite Difference Method (FDM) to discretise the continuous Navier-Stokes equations, and since the collocated storage of variables give rise to an odd-even decoupling, which introduces spurious oscillations into the solution called the Checkerboard modes, the variables are stored in the Arakawa C grid [3], also known as the staggered grid. In the

Arakawa C grid, the pressure is stored at the center while various velocity components are stored at the respective cell faces. Illustration 1 shows the locations where the variables are defined for each cell in a 2d domain.

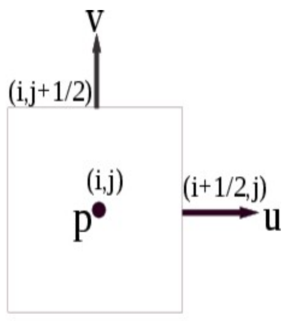

Illustration 1: Arrangement of variables in Arakawa C grid

In such a description we solve the continuity equation at the center of each cell, while the momentum equation is solved at the cell boundaries.

The methodology for the space discretization is similar to the one used by Morinishi et al. [2]. For the viscous terms discretization is done using the standard second order scheme employing a three point stencil (in 1-d), similarly for the convective terms a second order central scheme is employed. If we require variables at locations where they are not defined, we approximate them by averaging the variables at the known locations, for example, if we require $u_{i,j}$ we approximate it as

$$u_{i,j} = \frac{u_{i-\frac{1}{2},j} + u_{i+\frac{1}{2},j}}{2} .$$

For the time integration of the momentum equation, a 2nd order midpoint implicit method is used, while for the interface advection a 1st order explicit method is employed. One important aspect of the implicit time integration is the linearization of the convective terms. In the present scheme Newton linearization [4] is used, which can be described as follows.

Let u and v be any two variables which are function of some variable t (not necessarily time). Then by Tailor series

$$(uv)^{n+1}=(uv)^n+\left(\frac{\partial uv}{\partial t}\right)^n \Delta t+O(\Delta t^2) = (uv)^n+\left(u\frac{\partial v}{\partial t}+v\frac{\partial u}{\partial t}\right)^n \Delta t+O(\Delta t^2) ,$$

which can be further expressed as

$$(uv)^{n+1}= (uv)^n+\left(v^n\frac{(u^{n+1}-u^n)}{\Delta t}+u^n\frac{(v^{n+1}-v^n)}{\Delta t}\right)\Delta t+O(\Delta t^2) \ .$$

Neglecting $O(\Delta t^2)$ terms gives the Newton linearization

$$(uv)^{n+1}= -(uv)^n+v^n u^{n+1}+u^n v^{n+1} \ .$$

After the space-time discretization we get a system of nonlinear equations i.e., the discrete continuity and momentum equations which are defined as follows:

<u>Discrete continuity equation</u>

$$B\boldsymbol{u}^{\boldsymbol{n+1}}=\boldsymbol{g} \ ,$$

where $\boldsymbol{g}$ contains the discrete velocity boundary conditions and B is defined as

$$B=\begin{bmatrix}B_r & B_\theta & B_z\end{bmatrix} \ ,$$

where B_r, B_θ, B_z are the discrete divergence operators corresponding to the discrete velocities in the radial, angular and axial directions.

<u>Discrete momentum equation</u>

$$\frac{\boldsymbol{u}^{n+1}-\boldsymbol{u}^n}{\Delta t}+\bar{\hat{F}}(\boldsymbol{u}^{n+1})=-\frac{1}{\rho^{n+\frac{1}{2}}}G\boldsymbol{p}^{n+\frac{1}{2}}+\frac{1}{\rho}\boldsymbol{f}_s^{n+\frac{1}{2}}+\boldsymbol{h}^{n+\frac{1}{2}} \tag{2.1}$$

where $\bar{\hat{F}}(\boldsymbol{u}^{n+1})$ contains only the diffusion and nonlinear convection terms.

After linearization we get

$$\frac{\boldsymbol{u}^{n+1}-\boldsymbol{u}^n}{\Delta t}+\hat{F}\boldsymbol{u}^{n+1}=-\frac{1}{\rho^{n+\frac{1}{2}}}G\boldsymbol{p}^{n+\frac{1}{2}}+\boldsymbol{\tau}^n+\frac{1}{\rho}\boldsymbol{f}_s^{n+\frac{1}{2}}+\boldsymbol{h}^{n+\frac{1}{2}} \ . \tag{2.2}$$

The matrix $\hat{F}$ is of the form

$$\hat{F}=\begin{bmatrix}\hat{F}_r & 0 & 0\\ 0 & \hat{F}_\theta & 0\\ 0 & 0 & \hat{F}_z\end{bmatrix} \ ,$$

where $\hat{F}_r$, $\hat{F}_\theta$, $\hat{F}_z$ are derived from the linearized implicit time discretization of the radial, angular and axial momentum equations respectively and contain only the convective and diffusive terms. G is the discrete gradient operator of the form

$$G = \begin{bmatrix} G_r \\ G_\theta \\ G_z \end{bmatrix} .$$

As above G_r, G_θ, G_z are the discrete gradient operators obtained from the discretization of the radial, angular and axial momentum equations respectively. Finally, $\rho^{n+\frac{1}{2}}, \boldsymbol{f}_s^{n+\frac{1}{2}}, \boldsymbol{h}^{n+\frac{1}{2}}$ in (2.1) and (2.2) are the discrete density, surface tension, and body force respectively calculated based on the known location of the interface at $n+\frac{1}{2}$ time level, and $\boldsymbol{\tau}^n$ in (2.2) contains all the terms which appear due to linearization. Moreover $\boldsymbol{p}^{n+\frac{1}{2}} = \boldsymbol{p}^{n+1}$, the reason for writing pressure at $n+\frac{1}{2}$ will be clear in the following subsection. Further, it is to be noted that for our discretization the discrete gradient operator G is equal to the transpose of the discrete divergence operator B , hence we can replace G by B^T in the above equations.

Clubbing the above mentioned linearized discrete momentum and discrete continuity equations together we get

$$A\,\boldsymbol{x} = \begin{bmatrix} F & B^T \\ B & 0 \end{bmatrix} \begin{bmatrix} \boldsymbol{u} \\ \boldsymbol{p} \end{bmatrix} = \begin{bmatrix} \boldsymbol{f} \\ \boldsymbol{g} \end{bmatrix} = \boldsymbol{b} \ , \qquad (2.3)$$

where F contains the contributions from $\hat{F}$ and $\frac{1}{\Delta t} I$. We must solve the above system (2.3) at each time step to obtain a time dependent solution.

2.4. TIME INTEGRATION METHODOLOGY

We now discuss the methodology for the time integration which until now we assumed to be given. The time integration is split into two decoupled parts, the flow integration and the interface advection. The calculations of flow and interface are staggered in time, i.e., we first calculate the flow field at the new time step based on the current interface position and then calculate the new position of the interface based on the current flow field; the next section discusses this in detail.

What is to be stressed at this junction is the time integration technique for the flow field used in the available code. Although an algorithm is presented later, perhaps it is prudent to present the mathematical motivation, disadvantages and remedies of the used integration methodology here. As discussed above, the interface is assumed to be given for the flow field time integration. To obtain the velocity and pressure at the new time step one has to solve the discrete system (2.3). The system can be solved either in a decoupled manner (solve for the velocity and pressure separately) or in a coupled fashion. We further discuss the decoupled (pressure-correction/projection) scheme implemented in the available code.

In the decoupled solver we have to solve for velocity from (2.2)

$$\frac{\boldsymbol{u}^{n+1}-\boldsymbol{u}^{n}}{\Delta t}+\hat{F}\boldsymbol{u}^{n+1}=-\frac{1}{\rho^{n+\frac{1}{2}}}B^{T}\boldsymbol{p}^{n+\frac{1}{2}}+\boldsymbol{\tau}^{n}+\frac{1}{\rho}\boldsymbol{f}_{s}^{n+\frac{1}{2}}+\boldsymbol{h}^{n+\frac{1}{2}}\ .$$

We can not solve the above system directly since $\boldsymbol{p}^{n+\frac{1}{2}}$ is not known. Hence the following scheme is adopted:

Solve

$$\frac{\hat{\boldsymbol{u}}-\boldsymbol{u}^{n}}{\Delta t}+\hat{F}\hat{\boldsymbol{u}}=-\frac{1}{\rho^{n-\frac{1}{2}}}B^{T}\boldsymbol{p}^{n-\frac{1}{2}}+\boldsymbol{\tau}^{n}+\left(\frac{1}{\rho}\boldsymbol{f}_{s}\right)^{n-\frac{1}{2}}+\boldsymbol{h}^{n-\frac{1}{2}}\ . \tag{2.4}$$

Since the above equation is solved with pressure at previous time step $\hat{\boldsymbol{u}}$ is not solenoidal. But we want $\boldsymbol{u}^{n+1}$ to be divergence-free, hence we subtract (2.2) from equation (2.4) to get

$$\frac{\boldsymbol{u}^{n+1}-\hat{\boldsymbol{u}}}{\Delta t}=\left(-\frac{1}{\rho^{n+\frac{1}{2}}}B^{T}\boldsymbol{p}^{n+\frac{1}{2}}+\left(\frac{1}{\rho}\boldsymbol{f}_{s}\right)^{n+\frac{1}{2}}+\frac{1}{\rho^{n-\frac{1}{2}}}B^{T}\boldsymbol{p}^{n-\frac{1}{2}}-\left(\frac{1}{\rho}\boldsymbol{f}_{s}\right)^{n-\frac{1}{2}}\right)\ . \tag{2.5}$$

It can be shown [5] that if a 2nd order time integration method is used, neglecting $\hat{F}\boldsymbol{u}^{n+1}-\hat{F}\hat{\boldsymbol{u}}$ in the above equation will still give a 2nd order approximation if we use a good enough approximation of pressure in equation (2.4).

Now, all that remains is to find the pressure which will give a solenoidal velocity. From the continuity equation we have $B\boldsymbol{u}^{n+1}=\boldsymbol{g}$, hence we can derive an equation for the pressure by taking the discrete divergence of equation (2.5):

$$-B\frac{1}{\rho^{n+\frac{1}{2}}}B^T\boldsymbol{p}^{n+\frac{1}{2}}=B\left(-\frac{1}{\Delta t}\hat{\boldsymbol{u}}-\left(\frac{1}{\rho}\boldsymbol{f}_s\right)^{n+1/2}-\frac{1}{\rho^{n-\frac{1}{2}}}B^T\boldsymbol{p}^{n-\frac{1}{2}}+\left(\frac{1}{\rho}\boldsymbol{f}_s\right)^{n-\frac{1}{2}}+\frac{1}{\Delta t}\boldsymbol{g}\right)\ . \quad (2.6)$$

The equations (2.4), (2.6) and (2.5) are known as the predictor, Poisson and corrector equations respectively. From the above discussion we get a time integration technique, i.e., first predict the velocity based on the pressure at the previous time step, then solve the Poisson equation to find the pressure at the current time step which will give a divergence-free velocity, and then finally update the velocity at the current time step by using the corrector equation. The reason for writing $\boldsymbol{p}^{n+\frac{1}{2}}$ is simply because the pressure is calculated *in between* the velocity updates at n^{th} and $(n+1)^{th}$ time step. This integration technique is known as the projection scheme or the pressure correction scheme, the algorithm used in the current code is presented in section 3.

The current procedure of solving, though is robust, computationally *cheap* and widely used, does not preserve kinetic energy of the fluid. Conservation of kinetic energy of the fluid becomes a topic of interest in the turbulent flows where energy is transferred between different eddies. In the turbulent case if the kinetic energy is not conserved, the size of eddies will not be accurately predicted giving an overall inaccurate flow.

To preserve the kinetic energy one has to solve the nonlinear system (2.1) in a coupled manner, i.e., simultaneously solve for both the velocity and pressure. Solving such a nonlinear system is very expensive, and hence in this endeavor we solve the linearized system ((2.3)/(2.2)) in a coupled manner. This, though not kinetic energy conserving, is a step towards it and should give a more accurate result. System (2.3) gives rise to a saddle point problem being solved:

$$A\,\boldsymbol{x}=\begin{bmatrix}F & B^T\\ B & 0\end{bmatrix}\begin{bmatrix}\boldsymbol{u}\\ \boldsymbol{p}\end{bmatrix}=\begin{bmatrix}\boldsymbol{f}\\ \boldsymbol{g}\end{bmatrix}=\boldsymbol{b}\ .$$

The matrix A has a zero block on the diagonal and hence is a saddle point matrix. Solving such a coupled system is much more expensive than solving the pressure correction scheme. Therefore in the last century, the computational power severely limited the applications for which a coupled system could be solved. The present day computers however are much faster, making the solution of the coupled systems more practical to obtain.

To solve a coupled system we can either use the direct solvers or the iterative solvers. The direct solvers are very expensive for large systems and hence will not be discussed any further. The iterative solvers can be further classified into the segregated (not to be confused with the earlier discussed decoupled solver) and the coupled methods. In the segregated methods, velocity and pressure are solved separately one after the other, the order in which they are solved differs amongst different solvers. The idea is to solve two smaller problems one for each velocity and pressure. Coupled methods, on the other hand, solve the complete system simultaneously.

Iterative methods to solve the saddle point problem, especially the Krylov subspace methods, without a suitable preconditioner suffer from a terribly slow convergence due to the presence of a zero diagonal block in matrix A which makes it highly indefinite. Thus proper preconditioning must be used to get a higher efficiency from such solvers. The preconditioners for a saddle point problem tend to blur the distinction between the segregated and coupled methods since the preconditioners for the coupled methods are often based on the segregated methods. We shall discuss the appropriate preconditioners in the later sections.

3. DESCRIPTION OF CODE

The FDM scheme is used in this approach with variables stored in the Arakawa C grid as discussed in the previous section. Second order accurate schemes are used for the space discretization, and a second order implicit time integration method with Newton's linearization is used in the predictor step. The code is written in Fortran 90, and uses various LAPACK subroutines.

3.1. OVERALL ALGORITHM

In the present code, as pointed out in the previous section, the calculation of flow and interface variables are fully decoupled. The flow at new time step $(n+1)$ is calculated based on the interface position at the previous time step $(n+1/2)$, and then the interface is advected based on the just calculated flow variables. The values of density and viscosity only depend on the interface and hence are fixed for a given interface position. The overall flow of algorithm is shown in Illustration 2.

3.2. FLOW SOLVER

The flow solver part is broadly divided into three steps, as mentioned in section 2, viz. the predictor step for a first approximation of velocity based on the pressure value at the previous time step, the Poisson step to update the pressure based on the velocity calculated in the predictor step, and the corrector step to obtain a divergence-free velocity. Restarted GMRES is used to solve the implicit predictor equation, while for the Poisson equation the Preconditioned Conjugate Gradient method (PCG) is used. A brief flowchart of flow algorithm is given in Illustration 3.

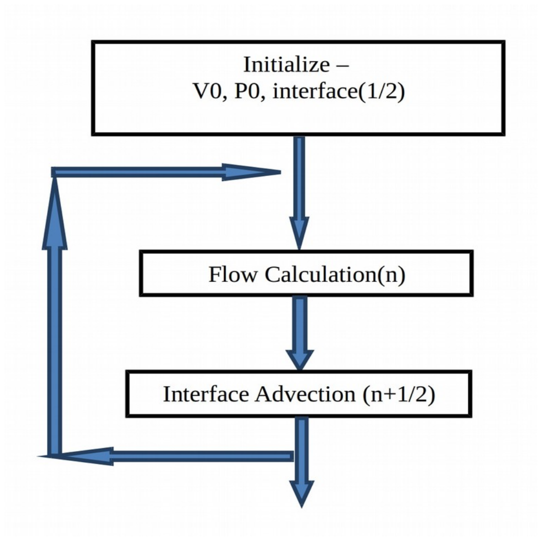

Illustration 2: Overall flow chart of the algorithm

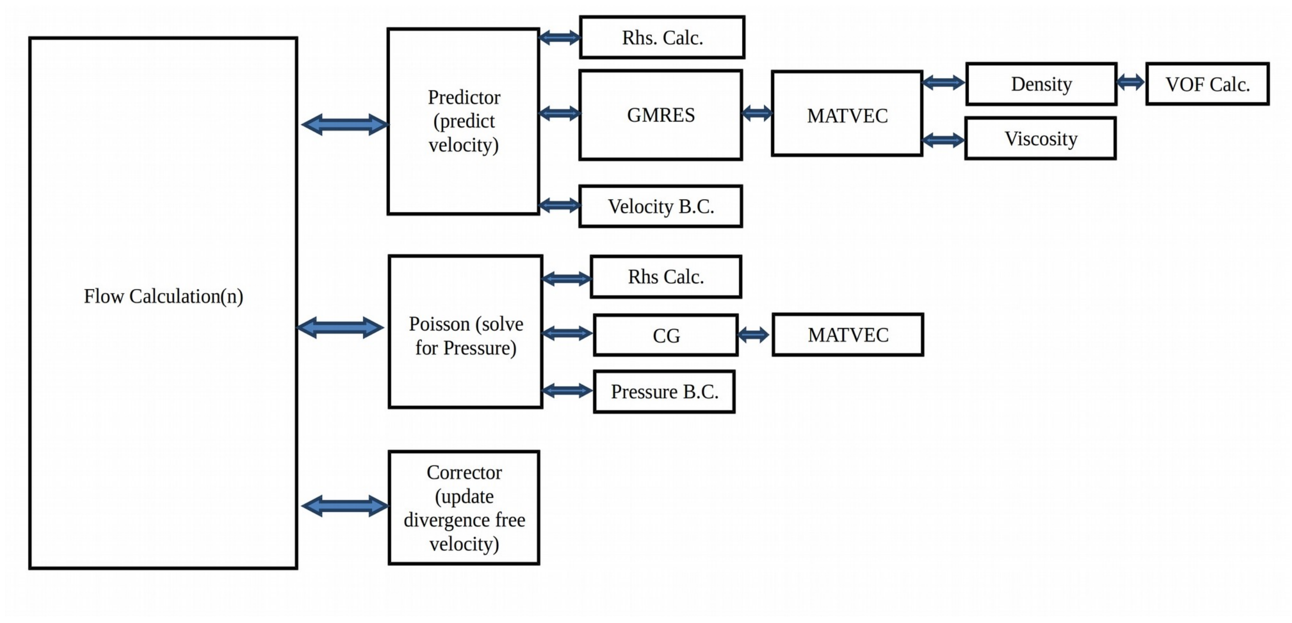

Illustration 3: Brief flowchart of flow algorithm

3.3. INTERFACE SOLVER

The interface is advected as described in Section 2. For a stable advection of the interface the Courant number should be less than a half. It may so happen that the Courant number in flow calculation gets higher than one half, in which case the time step for the interface advection is reduced (so as to have a Courant number which is less than a half) and multiple time step integrations are performed to increase the overall time step by the same amount as that in the flow calculation. How many steps need to be performed is calculated in Subcycling module.

Illustration 4 gives a brief flowchart of Interface advection algorithm.

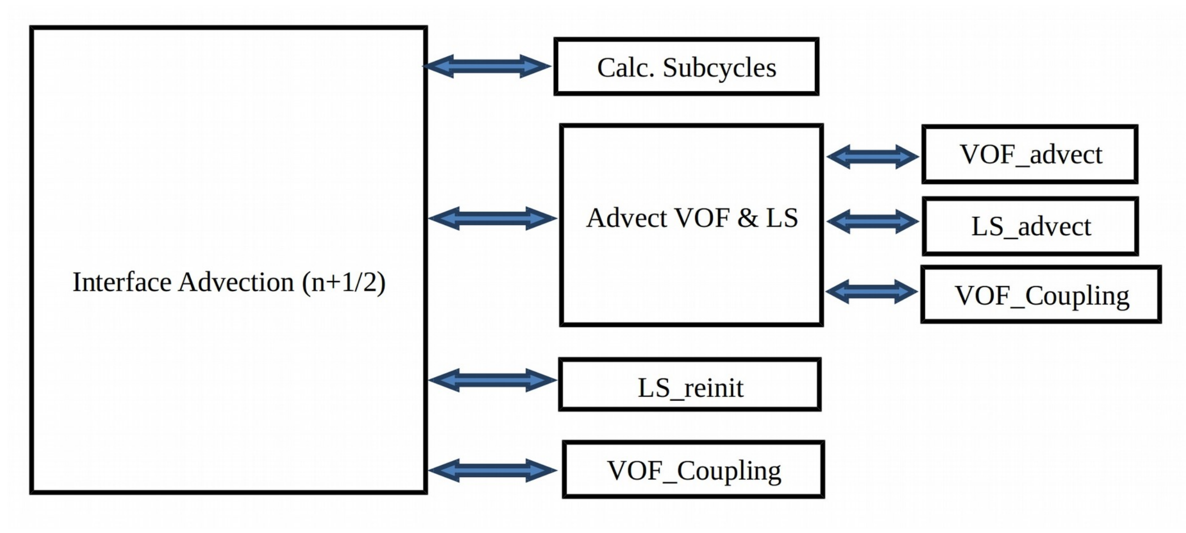

Illustration 4: Brief flowchart of interface advection algorithm

4. SOLVERS

The choice of the solver plays a vital role in achieving the desired speed up. Improving the performance of the code is not just about parallel programming but also about using smart solvers that are more suitable for the problem in hand, i.e., a solver which converges quickly with relatively small computational and storage complexities. To solve a problem of the form $A\boldsymbol{x}=\boldsymbol{b}$, a whole plethora of solvers is available ranging from direct to iterative. Direct solvers generally decompose the matrix A into product of matrices which are easier to invert, for example, the Gaussian Elimination [6] method splits the matrix A into a product of lower (L) and upper (U) triangular matrices. These matrices are relatively easy to invert using the forward and backward substitution. The main drawback however is in the time complexity involved in forming these lower and upper triangular matrices, a matrix of size $N\times N$ requires $O(N^3)$ floating point operations (flops). Although there exists solvers which work more efficiently than Gaussian Elimination, they still are too expensive for large N .

An alternative to the direct methods are the so-called iterative methods. In these methods, unlike the direct methods, one does not compute the solution exactly, rather the solution is updated iteratively and if the current solution satisfies the convergence criterion the iterations are stopped [6]. The definiteness of a matrix, plays a vital role in the performance of the iterative methods [6]. For the systems having highly indefinite matrices the convergence of an iterative method deteriorates drastically. We can improve the convergence by preconditioning the matrix [7], which will be discussed later.

Krylov subspace methods [6] are probably the most used subclass of iterative solvers. In these methods we look for the solution of $A\boldsymbol{x}=\boldsymbol{b}$ in the Krylov subspace $K_i(A,\boldsymbol{v})$ which is

$$K_i(A,\boldsymbol{v}) = span\ \{\boldsymbol{v}, A\boldsymbol{v}, A^2\boldsymbol{v}, \ldots, A^{(i-1)}\boldsymbol{v}\}\ ,$$

where $\boldsymbol{v}\in\mathbb{R}^n$ is a suitably chosen vector. A vector $\boldsymbol{v}$ is said to be of grade d with respect to A if d is the smallest integer for which the set $\{\boldsymbol{v}, A\boldsymbol{v}, A^2\boldsymbol{v}, \ldots, A^d\boldsymbol{v}\}$ is linearly dependent. Since $d\leq n$ the Krylov subspace methods converge in at-most n iterations [6], but practically these methods converge much before n iterations. Further we discuss some of the solvers which are either used in the available code, or can be used to increase its performance.

4.1. RESTARTED GMRES

GMRES [6] is a Krylov subspace method which is widely used to iteratively solve $A\boldsymbol{x}=\boldsymbol{b}$ where A is a non hermitian matrix. In this section we discuss the GMRES method in detail.

In the GMRES method, like in all the Krylov subspace methods, the solution $\boldsymbol{x}$ is approximated by $\boldsymbol{x}_i$ which lives in the Krylov subspace $K_i(A,\boldsymbol{r}_0)$ where $\boldsymbol{r}_0$ is the residue of initial guess ($\boldsymbol{r}_i = \boldsymbol{b} - A\boldsymbol{x}_i$). The GMRES method reduces to finding $\boldsymbol{x}_i$ in each iteration such that it minimizes the residue $\boldsymbol{r}_i$. This gives

$$\boldsymbol{x}_i=\boldsymbol{x}_0+\boldsymbol{y},\ \ \boldsymbol{y}\in K_i(A,\boldsymbol{r}_0),\ \ where\ K_i(A,\boldsymbol{r}_0)=span\{\boldsymbol{r}_0, A\boldsymbol{r}_0, \ldots .\ A^{(i-1)}\boldsymbol{r}_0\}\ \ .$$

Choosing the Krylov subspace defined as above does not generally give a stable algorithm, therefore we form Gram-Schmidt type vectors as the orthogonal basis of $K_i(A,\boldsymbol{r}_0)$. We form the orthogonal basis vectors using the Arnoldi decomposition:

$$\hat{\boldsymbol{v}}_{i+1}=A\boldsymbol{v}_i-\sum_{j=1}^{i}(A\boldsymbol{v}_i,\boldsymbol{v}_j)\boldsymbol{v}_j,\quad \boldsymbol{v}_{i+1}=\hat{\boldsymbol{v}}_{i+1}/\|\hat{\boldsymbol{v}}_{i+1}\|,\quad h_{ji}=(A\boldsymbol{v}_i,\boldsymbol{v}_j),\quad \boldsymbol{v}_1=\boldsymbol{r}_0\ .$$

This gives,

$$A[\boldsymbol{v}_1\ \boldsymbol{v}_2\ \ldots\ \boldsymbol{v}_i]=[\boldsymbol{v}_1\ \boldsymbol{v}_2\ \ldots\ \boldsymbol{v}_i\ \boldsymbol{v}_{i+1}]\begin{bmatrix} h_{11} & h_{12} & \cdots & \\ \|\hat{\boldsymbol{v}}_2\| & h_{22} & \cdots & \\ 0 & \|\hat{\boldsymbol{v}}_3\| & \cdots & \\ 0 & 0 & \ddots & \\ 0 & 0 & 0 & \|\hat{\boldsymbol{v}}_{i+1}\| \end{bmatrix}\ .$$

In matrix form we get $A V_i = V_{i+1} H_{i+1,i}$, where

$$H_{i+1,i} = \begin{bmatrix} h_{11} & h_{12} & \cdots & \\ \|\hat{\boldsymbol{v}}_2\| & h_{22} & \cdots & \\ 0 & \|\hat{\boldsymbol{v}}_3\| & \cdots & \\ 0 & 0 & \ddots & \\ 0 & 0 & 0 & \|\hat{\boldsymbol{v}}_{i+1}\| \end{bmatrix}$$

$H_{i+1,i} \in \mathbb{R}^{(i+1)\times i}$ is the unreduced upper Hessenberg matrix. If $\boldsymbol{r}_0$ is of grade d with respect to A we get $A V_d = V_d H_{dd}$, where $H_{dd} \in \mathbb{R}^{d\ x\ d}$ is given by

$$H_{d,d} = \begin{bmatrix} h_{11} & h_{12} & \cdots & & \\ \|\hat{\boldsymbol{v}}_2\| & h_{22} & \cdots & & \\ 0 & \|\hat{\boldsymbol{v}}_3\| & \ddots & & \\ 0 & 0 & \ddots & \ddots & \\ 0 & 0 & 0 & \|\hat{\boldsymbol{v}}_d\| & h_d \end{bmatrix}$$

GMRES finds a $\boldsymbol{z} \in \boldsymbol{x}_0 + K_i(A, \boldsymbol{r}_0)$ in each iteration i such that the residual at that iteration is minimized, i.e., find $\boldsymbol{z}$ such that $\|\boldsymbol{r}_i\|_2 = \min_{\boldsymbol{z} \in \boldsymbol{x}_0 + K_i(A, \boldsymbol{r}_0)} \|\boldsymbol{b} - A\boldsymbol{z}\|_2$.

The above equation leads to finding a vector $\boldsymbol{t}_i \in \mathbb{R}^i$ such that it minimizes $\|V_{i+1}(\|r_0\|_2 \boldsymbol{e}_1 - H_{i+1,i}\boldsymbol{t})\|_2$. If we form the QR decomposition of the Hessenberg matrix as $H_{k+1,k} = QR$, where $Q \in \mathbb{R}^{(i+1)\times(i+1)}$ is an orthogonal matrix and $R \in \mathbb{R}^{(i+1)\times i}$ is an upper diagonal matrix, we get the GMRES method as

$$\min_{\boldsymbol{t}_i \in \mathbb{R}^i} \left\| Q^H \boldsymbol{e}_1 \|\boldsymbol{r}_0\|_2 - \begin{bmatrix} R_i \\ 0 \end{bmatrix} \boldsymbol{t}_i \right\|_2 .$$

Choosing $t_i = \|r_0\|_2 R_i^{-1} \begin{bmatrix} q_1 \\ q_2 \\ \vdots \\ q_i \end{bmatrix}$ gives us the solution to the above problem, with the minimum residual equal to $|q_{i+1}|\ \|\boldsymbol{r}_0\|_2$ for each iteration.

Now that we have discussed the GMRES method, it shall be prudent to discuss its convergence properties. By analyzing the GMRES method as a polynomial problem one can derive a bound on the residue at each iteration:

$$\frac{\|\boldsymbol{r}_i\|_2}{\|\boldsymbol{b}\|_2} \leq \inf_{p_i \in P_i} \kappa(X) \max_{\lambda \in \sigma(A)} \|p_i(\lambda)\|_2$$

where P_i are the polynomials of degree i having $P_i(0) = 1$ and $\kappa(X)$ is the conditioning number of X where X is obtained from diagonalization of A . The above bound shows that the reduction in residual per iteration is large if

1. Conditioning number $\kappa(X)$ is small, i.e., A is nearly normal.
2. Eigenvalues of A are clustered far away from the origin.

One drawback of GMRES is that all the Arnoldi vectors need to be stored for computation of the solution vector. If the dimension of the system is large, this results in an excessive computational and storage overhead. To avoid this, the Restarted GMRES method can be employed in which, after some predefined iterations, all the Arnoldi vectors are deleted and the iterations are restarted with the available solution as the initial guess. In the present code the GMRES method is used to solve the implicit predictor step, and the iterations are restarted after every 50 internal iterations.

4.2. CONJUGATE GRADIENT (CG)

The Conjugate Gradient (CG) method [6] is another class of Krylov subspace methods, and is the most preferred solver for the systems having Symmetric Positive Definite (SPD) matrices. In the CG method, as in the GMRES method, we express the approximate solution as a member of the Krylov subspace and form the Krylov subspace by spanning the space generated by the orthogonal Gram-Schmidt style vectors. But because of SPDness of the matrix, Gram-Schmidt style vectors reduces to Lanczos vectors and we get a SPD Hessenberg matrix giving a short recursion formula, hence we need not store all the Lanczos vectors. A similar analysis as GMRES can be easily performed for the CG method and an elegant algorithm can be easily derived, for brevity the (serial) CG algorithm is not presented in this report.

We further discuss the convergence properties of the CG method. Useful bounds on the error for the CG method can be derived by approximating it as a polynomial problem. We get

$$\frac{\|\boldsymbol{x}_i-\boldsymbol{x}\|_A^2}{\|\boldsymbol{x}\|_A^2}\leq 2\left(\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right)^i ,$$

where $\kappa=\kappa(A)$ is the conditioning number of matrix A, and $\boldsymbol{x}$ is the actual solution. This convergence bound tells us that the CG method will converge much faster if the matrix is well conditioned. More details about CG can be found in [6].

4.3. IDR(s)

IDR(s) is another method to solve $A\boldsymbol{x}=\boldsymbol{b}$, where A is a general matrix, which has certain advantages over GMRES. As discussed above, the GMRES method has an ever increasing depth of recursion with the number of iterations, i.e., the amount of computational work increase as the number of iterations increase, also the memory requirements scale with the number of iterations. Though restarting of GMRES helps to partially alleviate these two problems it also slows down the convergence of the GMRES method, thus increasing the total computational work.

The IDR(s) method due to Sonneveld, P. & van Gijzen, M.B. [8] has the same number of matrix vector products per iteration as GMRES but has a fixed depth of recursion (s) which is smaller than GMRES, hence the overhead of IDR(s) is lower [8]. Also since we do not delete all the preconstructed Arnoldi vectors, the convergence property of IDR(s) may be better than that of the restarted GMRES method (depending on the restarting frequency of GMRES). Below we discuss the IDR(s) algorithm.

IDR(s) Theorem (4.1): Consider any matrix $A\in\mathbb{R}^{N\times N}$ and non zero vector $\boldsymbol{v}_0\in\mathbb{R}^N$, and let $G_0\in K^N(A,\boldsymbol{v}_0)$. Let S be a proper subspace of $\mathbb{R}^N$, such that S and G_0 does not share a nontrivial invariant subspace of A , and for nonzero $\omega_j's$ define sequence $G_j=(I-\omega_j A)(G_{j-1}\cap S)$, $j=1,2,3,4....$. Then the following holds,

1. $G_j\subset G_{j-1}\ \forall j>0.$
2. $G_j=\{0\}\ \textit{for some}\ j\leq N.$

The proof of this theorem can be found in [8]. The above theorem can be applied by generating residuals $\boldsymbol{r}_k$ which are forced to live in the subspace G_j , here j is non-decreasing with k . This implies that the system will be solved in at-most N iterations. The residual $\boldsymbol{r}_k$ belongs to G_{j+1} if

$$\boldsymbol{r}_{k+1}=(I-w_{j+1}A)\boldsymbol{v}_k\ \ \textit{where}\ \ \boldsymbol{v}_k\in G_j\cap S\ .$$

Now if we choose,

$$\boldsymbol{v}_k=\boldsymbol{r}_k-\sum_{i=1}^{\hat{l}}\gamma_i\Delta\boldsymbol{r}_{k-i}\ \text{we get}$$

$$\boldsymbol{r}_{k+1}=\boldsymbol{r}_k-\omega_{j+1}A\boldsymbol{v}_k-\sum_{i=1}^{\hat{l}}\gamma_i\Delta\boldsymbol{r}_{k-i}=\boldsymbol{v}_k-\omega_{j+1}A\boldsymbol{v}_k$$

This is similar to the residual in the general Krylov subspace methods [8]. Let us assume S to be in the left nullspace of some $N\times s$ matrix T . Now, since $\boldsymbol{v}_k \in G_j \cap S \in S$ we have $T^H v_k = 0$. It follows that we get a linear system of size $s\times\hat{l}$ for $\hat{l}$ coefficients γ_i , and the system is uniquely solvable if $\hat{l}=s$. Hence, the first vector in G_j requires $s+1$ vectors in G_{j-1} and $\boldsymbol{r}_k$ lies in G_j only if $k\geq j(s+1)$. Defining

$$\Delta R_k = (\Delta \boldsymbol{r}_{k-1}, \Delta \boldsymbol{r}_{k-2}, \ldots\ldots, \Delta \boldsymbol{r}_{k-s}) \text{ and}$$

$$\Delta X_k = (\Delta \boldsymbol{x}_{k-1}, \Delta \boldsymbol{x}_{k-2}, \ldots\ldots, \Delta \boldsymbol{x}_{k-s}) \text{ then}$$

$\boldsymbol{r}_{k+1} \in G_{j+1}$ can be computed by, calculate $\boldsymbol{\gamma} \in \mathbb{R}^s$ from

$$(T^H \Delta R_k)\boldsymbol{\gamma} = T^H \boldsymbol{r}_k \text{ then compute}$$

$$\boldsymbol{v} = \boldsymbol{r}_k - \Delta R_k \boldsymbol{\gamma} \text{ giving}$$

$$\boldsymbol{r}_{k+1} = \boldsymbol{v} - \omega_{j+1} A \boldsymbol{v} \ .$$

Since $G_{j+1} \subset G_j$, the new residuals $\boldsymbol{r}_{k+2}, \boldsymbol{r}_{k+3}, \ldots., \boldsymbol{r}_{k+s+1} \subset G_{j+1}$ can be produced by performing the above calculations repeatedly. The next residual, however, will belong to G_{j+2} . Also to be noted is the fact that for calculation of the first residual in G_j we can use any value for ω_j but this value must be same for calculation of the remaining residuals in G_j . Further the algorithm for IDR(s) is presented.

Algorithm 4.1 : IDR(s) Solver (to solve $A\boldsymbol{x}=\boldsymbol{b}$)

```
Required: $A$, $\boldsymbol{x}_{0,}$ $\boldsymbol{b}$, $T$
Initialize $\boldsymbol{r}_0 = \boldsymbol{b} - A\boldsymbol{x}_0$ , calculate first $s$ residuals by
 $\boldsymbol{v} = A\boldsymbol{r}_k;\ \omega = \frac{\boldsymbol{v}^H \boldsymbol{r}_k}{\boldsymbol{v}^H \boldsymbol{v}}$ and $\Delta \boldsymbol{x}_k = \omega \boldsymbol{r}_k;\ \Delta \boldsymbol{r}_k = -\omega \boldsymbol{v}$
 $\boldsymbol{r}_{k+1} = \boldsymbol{r}_k + \Delta \boldsymbol{r}_k;\ \boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \Delta \boldsymbol{x}_k$ where $k \in [0, s-1]$ and form
 $\Delta R_{k+1} = (\Delta \boldsymbol{r}_k, \Delta \boldsymbol{r}_{k-1}, \ldots\ldots, \Delta \boldsymbol{r}_0)$ and $\Delta X_{k+1} = (\Delta \boldsymbol{x}_k, \Delta \boldsymbol{x}_k, \ldots\ldots, \Delta \boldsymbol{x}_0)$

//Building $G_j$ spaces for $j=1,2,3,\ldots$
 $n=s$
//loop over $G_j$ spaces
 while $\|r_k\| > Tol \wedge k < Max\ iter$ do
    //loop inside $G_j$ space-time
    for $k = 0, s$ do
        Solve for $\boldsymbol{\gamma}$ by $T^H \Delta R_k \boldsymbol{\gamma} = T^H \boldsymbol{r}_k$
        $\boldsymbol{v} = \boldsymbol{r}_k - \Delta R_k \boldsymbol{\gamma}$
        $if\ k = 0\ then$
            //First vector in $G_{j+1}$
            $\boldsymbol{t} = A\boldsymbol{v}$
            $\omega = (\boldsymbol{t}^H \boldsymbol{v})/(\boldsymbol{t}^H \boldsymbol{t})$
            $\Delta \boldsymbol{r}_k = -\Delta R_k \boldsymbol{\gamma} - \omega \boldsymbol{t}$
            $\Delta \boldsymbol{x}_k = -\Delta X_k \boldsymbol{\gamma} - \omega \boldsymbol{v}$
        $else$
            //Subsequent vectors in $G_{j+1}$
            $\Delta \boldsymbol{x}_k = -\Delta X_k \boldsymbol{\gamma} - \omega \boldsymbol{v}$
            $\Delta \boldsymbol{r}_k = -A \Delta \boldsymbol{x}_k$
        $end\ if$
        $\boldsymbol{r}_{k+1} = \boldsymbol{r}_k + \Delta \boldsymbol{r}_k$
        $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \Delta \boldsymbol{x}_k$
        $k = k+1$
        $\Delta R_k = (\Delta \boldsymbol{r}_{k-1}, \Delta \boldsymbol{r}_{k-2}, \ldots.., \Delta \boldsymbol{r}_{k-s})$
        $\Delta X_k = (\Delta \boldsymbol{x}_{k-1}, \Delta \boldsymbol{x}_{k-2}, \ldots.., \Delta \boldsymbol{x}_{k-s})$
    $end\ for$
$end\ while$
```

As proven by Theorem 4.1, the IDR(s) method converges in at-most N outer steps, with each outer step having $s+1$ inner steps. Hence we have at-most $N \times (s+1)$ matrix vector multiplications. It can be proven however, that the rate at which the dimension of G_j reduces is “almost always” equal to s [8], hence the total number of outer iterations reduces to $\frac{N}{s}$.

4.4. GCR METHOD

The GCR method is yet another method to solve $A\boldsymbol{x} = \boldsymbol{b}$ where A is a general matrix. For a constant preconditioner, the GCR method and the GMRES method are mathematically the same. The GCR method, although may not be the fastest algorithm, is very simple to implement, minimizes the residual norm and has a very important property that it does not require a constant preconditioner. The usefulness of the last property will be shown in the later sections, here we just discuss the basic formulation.

Let $\{\boldsymbol{v}_1\ \boldsymbol{v}_2 \ldots\ \boldsymbol{v}_k\}$ be the orthonormal basis of $K_k(A, \boldsymbol{r}_0)$, we construct $\boldsymbol{r}_k$ orthonormal to $K_k(A, \boldsymbol{r}_0)$. Then we have

$$\boldsymbol{r}_k = \boldsymbol{r}_0 - \sum_1^m \alpha_j \boldsymbol{v}_j \quad \textit{where} \quad \alpha_j = (\boldsymbol{r}_0, \boldsymbol{v}_j) = (\boldsymbol{r}_0 - \sum_{m=1}^{j-1} (\boldsymbol{r}_0, \boldsymbol{v}_m)\boldsymbol{v}_m \, , \, \boldsymbol{v}_j) = (\boldsymbol{r}_{j-1}, \boldsymbol{v}_j)$$

implying

$$\boldsymbol{r}_k = \boldsymbol{r}_{k-1} - (\boldsymbol{r}_{k-1}, \boldsymbol{v}_k)\boldsymbol{v}_k \text{ . Which gives}$$

$$\boldsymbol{x}_k = \boldsymbol{x}_{k-1} + (\boldsymbol{r}_{k-1}, \boldsymbol{v}_k)\boldsymbol{s}_k, \qquad \boldsymbol{v}_k = A\boldsymbol{s}_k \ .$$

Now all that remains is to find $\{\boldsymbol{v}_1, \boldsymbol{v}_2 \ldots \boldsymbol{v}_k\}$ and $\{\boldsymbol{s}_1, \boldsymbol{s}_2 \ldots \boldsymbol{s}_k\}$. This can be done by using the Gram-Schmidt type orthogonalization processes. The GCR method with preconditioning is presented later.

Evidently the computational overhead of the GCR method is more than that of the GMRES method (nearly twice), since both $\{\boldsymbol{v}_1, \boldsymbol{v}_2 \ldots \boldsymbol{v}_k\}$ and $\{\boldsymbol{s}_1, \boldsymbol{s}_2 \ldots \boldsymbol{s}_k\}$ must be stored and computed. But an advantage of GCR is that we can truncate it easily, unlike the GMRES method, because of which the GCR method may converge faster than GMRES.

5. PRECONDITIONERS

This section is divided into two parts, the first section discusses the general preconditioners used to improve the convergence of Krylov subspace methods. These preconditioners, though effective for most type of matrices, are inadequate for saddle point problems because of the presence of a zero diagonal block. Thus special kind of preconditioners are required for saddle type problems which are the topic of discussion of section 5.2.

5.1. GENERAL PRECONDITIONERS

The above discussion of convergence for the GMRES and CG methods motivates the use of preconditioning, i.e., improve the spectral properties of A to solve $A\boldsymbol{x}=\boldsymbol{b}$ efficiently. The main idea is to premultiply matrix A with another easily invertible matrix P^{-1} , which is close to A , such that the iterative solver for the system having $P^{-1}A$ as the system matrix converges faster than the iterative solver for the system having system matrix A . That is instead of solving $A\boldsymbol{x}=\boldsymbol{b}$ we solve the system

$$P^{-1}A\boldsymbol{x}=P^{-1}\boldsymbol{b} \ .$$

Another way of forming a preconditioned system is by right preconditioning. That is, we solve $AP^{-1}P\boldsymbol{x}=\boldsymbol{b}$. In this system we first solve for $\boldsymbol{y}$ in $AP^{-1}\boldsymbol{y}=\boldsymbol{b}$ and then extract solution by solving $P\boldsymbol{x}=\boldsymbol{y}$. Which way to model a preconditioned system is a topic open for debate with no conclusive answers yet. However, for this study both left and right preconditioning is considered.

As described above, in choosing a preconditioner we are faced with two requirements [7]

1. The preconditioner must be easily invertible, i.e., $Px=y$ is easily solvable.
2. It must improve the convergence of the iterative method.

If we look at the first condition the Identity matrix is a perfect choice, but it does not help us at all with the second requirement. While, if we look at the second condition A^{-1} is the perfect choice, but it does not help us at all with the first condition. Therefore, choosing a preconditioner is an optimization problem between the above two requirements. Further, we discuss some of the preconditioners used in this code.

5.1.1. JACOBI PRECONDITIONER

Jacobi is an easy to implement preconditioner, which and can improve the convergence in the cases having jumps in the diffusion coefficient [7] (which is also the case in this endeavor due to the presence of the interface). Jacobi preconditioning is also easy to parallelize. It is basically scaling the equation with the diagonal of system matrix, presenting us with very little extra calculations. Here,

$$A=\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & \cdots & & a_{2n} \\ \vdots & & & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \quad \& \quad P=\begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & 0 & 0 \\ \vdots & & \ddots & \vdots \\ 0 & \cdots & & a_{nn} \end{bmatrix}$$

with P^{-1} simply

$$P^{-1}=\begin{bmatrix} 1/a_{11} & 0 & \cdots & 0 \\ 0 & 1/a_{22} & 0 & 0 \\ \vdots & & \ddots & \vdots \\ 0 & \cdots & & 1/a_{nn} \end{bmatrix}.$$

5.1.2. INCOMPLETE CHOLESKY

For methods like CG, which are applicable for a Symmetric Positive Definite (SPD), we would like to have a preconditioner such that the resulting system is also SPD. Let LL^T be the Cholesky decomposition of system matrix A , then $P=LL^T$ preserves the the SPD properties of the system if used as a dual-sided preconditioner. The problem with this preconditioner is that it is not sparse, hence the resulting system may also not be sparse requiring more memory space and incurring more construction cost. To preserve the sparsity we can use an Incomplete Cholesky factorization [9] which is in some sense close to the Cholesky factorization. The Incomplete Cholesky factorization is constructed by setting the non-zero elements of L which are zero in A to zero.

Below, we describe the algorithm of the factorization used in the present code. For a 2-d case, we know that we get a five diagonal matrix from the Poisson equation (for which CG method is used) which requires storage of only 3 diagonal (due to symmetry). We form the preconditioner as $P = L D^{-1} L^T$ where, lower triangular matrix L and diagonal matrix D satisfies

1. $L_{ij} = 0 \ if \ A_{ij} = 0, i > j$
2. $L_{ii} = D_{ii}$
3. $(LD^{-1}L^T)_{ij} = a_{ij} \ for \ (i, j)$ where $A_{ij} \neq 0 \ \ i \geq j$

Therefore, if A and L are

$$A = \begin{bmatrix} a_1 & b_1 & 0 & & c_1 & & & & & \\ b_1 & a_2 & b_2 & \ddots & & c_2 & & & & 0 \\ \vdots & \ddots & \ddots & \ddots & & & \ddots & & & \\ c_1 & & b_m & a_{m+1} & b_{m+1} & 0 & & c_{m+1} & & \\ 0 & \ddots & 0 & \ddots & \ddots & \ddots & 0 & & \ddots & \end{bmatrix}$$

$$L = \begin{bmatrix} d_1^1 & & & & \\ b_1^1 & d_2^1 & & & \\ \vdots & \ddots & \ddots & & 0 \\ c_1^1 & & b_m^1 & d_{m+1}^1 & \\ 0 & \ddots & 0 & \ddots & \ddots \end{bmatrix}$$, then

$$\left.\begin{aligned} d_i^1 &= a_i - \frac{b_{i-1}^2}{d_{i-1}} - \frac{c_{i-m}^2}{d_{i-m}} \\ b_i^1 &= b_i \\ c_i^1 &= c_i \end{aligned}\right\} i=1,\ldots,n \ .$$

Very easily this method can be extended for a 3d case in which case we will get a seven diagonal matrix from the Poisson equation. More details about this method can be found in [9].

5.2. SADDLE POINT PRECONDITIONERS

For the saddle point problems broadly two types of preconditioners are available; the block preconditioners and the ILU type preconditioners. The ILU type preconditioners are better for finite element solvers where both matrix builder and solver must be adapted, since the splitting of velocity and pressure unknowns is required [10]. In this study we intend to use the block type preconditioners, hence the ILU type is not discussed any further.

Block preconditioners are based on the block LDU decomposition of the coefficient matrix A in equation (2.3). We write

$$A = LDU = \begin{bmatrix} F & B^T \\ B & 0 \end{bmatrix} = \begin{bmatrix} I & 0 \\ BF^{-1} & I \end{bmatrix} \begin{bmatrix} F & 0 \\ 0 & S \end{bmatrix} \begin{bmatrix} I & F^{-1}B^T \\ 0 & I \end{bmatrix} ,$$

where $S = -BF^{-1}B^T$ is the Schur complement matrix. Almost all the preconditioners are some form of combination of these blocks with an appropriate Schur complement matrix approximation.

If we choose a preconditioner (P) based on the product of only the diagonal matrix D and the upper diagonal matrix U, i.e.,

$$P = \begin{bmatrix} F & B^T \\ 0 & S \end{bmatrix} ,$$

then it is easy to show that the eigenvalues of the preconditioned system are all 1, hence GMRES converges in 2 iterations in exact arithmetic [11]. Computing the inverse of S and F is naturally not practical, hence a cheap approximation of S is used and the system $F\boldsymbol{u}=\boldsymbol{f}$ is solved approximately using iterative methods. Application of such a preconditioner requires solving $P\boldsymbol{z}=\boldsymbol{r}$ where $\boldsymbol{z}=[\boldsymbol{z}_1;\boldsymbol{z}_2]$ and $\boldsymbol{r}=[\boldsymbol{r}_1;\boldsymbol{r}_2]$. Now all that remains is to form an approximation of the Schur complement. As it turns out, there is a plethora of ways we can approximate the Schur complement. The way we do it leads to various block preconditioners. In the following subsections we discuss a few of these preconditioners.

5.2.1. PRESSURE CONVECTION-DIFFUSION PRECONDITIONER

Based on [12], let the Convection Diffusion (C-D) operator L be defined on the velocity space. Also let $\boldsymbol{w}_h$ be the approximate discrete velocity computed in the most recent Piccard iteration. Then L is given by

$$L = -\nabla.(\nu\nabla)+\boldsymbol{w}_h.\nabla \ .$$

Let the commutator of L be $\epsilon = L\nabla - \nabla L_p$, where L_p is analogous to L but does not carry any physical meaning. If $\boldsymbol{w}_h$ is constant, the commutator is zero in the interior of domain and is small for smooth $\boldsymbol{w}$, hence the discrete commutator (in terms of matrices) defined as,

$$\epsilon_h = (Q_v^{-1}F)(Q_v^{-1}B^T) - (Q_v^{-1}B^T)(Q_p^{-1}F_p)$$

will also be small. Here, Q_v is the velocity mass matrix, Q_p is the pressure mass matrix and F_p is discrete C-D operator on the pressure space. Assuming the commutator is indeed small then left multiplication of the discrete commutator by $BF^{-1}Q_v$ and right multiplication by $F_p^{-1}Q_p$ gives an approximation to the Schur complement

$$-BF^{-1}B^T \approx -BQ_v^{-1}B^{-1}F_p^{-1}Q_p \ . \qquad (5.1)$$

Since $BQ_v^{-1}B^T$ is expensive to compute, we replace it with its spectral equivalent matrix A_p known as pressure Laplacian matrix, thus giving

$$S = -BF^{-1}B^T = A_pF_p^{-1}Q_p \ .$$

This preconditioner gives a very good convergence if used with the Krylov subspace methods for enclosed flows (if the convective terms are linearized by the Piccard linearization). But for other problems this preconditioner may not be ideal as A_p is defined only for the enclosed flow problems [10].

5.2.2. LEAST SQUARES COMMUTATOR PRECONDITIONER

The Least Square Commutator (LSC) preconditioner was given by Elman et al. [13], and is based on the same principle as the PCD preconditioner discussed above. We approximate F_p in a way which gives us a small discrete commutator. Therefore we solve the following least squares problem

$$min\|[Q_v^{-1} F Q_v^{-1} B^T]_j - Q_v^{-1} B^T Q_p^{-1} [F_p]_j\|_{Q_v} ,$$

where the j-th column of matrix F_p is represented by $[F_p]_j$, the j-th column of matrix $Q_v^{-1} F Q_v^{-1}$ is $[Q_v^{-1} F Q_v^{-1}]_j$, and $\|.\|_{Q_v}$ is the energy norm with respect to Q_v . The associated normal equations are

$$Q_p^{-1} B Q_v^{-1} B^T Q_p^{-1} [F_p]_j = [Q_p^{-1} B Q_v^{-1} F Q_v^{-1} B^T]_j ,$$

which gives the following equation for F_p :

$$F_p = Q_p (B Q_v^{-1} B^T)^{-1} (B Q_v^{-1} F Q_v^{-1} B^T) . \tag{5.2}$$

Equation 5.1 and 5.2 gives the Schur complement approximation

$$B F^{-1} B^T \approx (B Q_v^{-1} B^T)(B Q_v^{-1} F Q_v^{-1} B^T)^{-1} (B Q_v^{-1} B^T) , \tag{5.3}$$

where Q_V is approximated by its diagonal elements to reduce the complexity of inverting it. This gives rise to the following algorithm:

Algorithm 5.1 : LSC preconditioner

```
First compute $\boldsymbol{r}_u = \boldsymbol{f}$ and $r_p = -BK^{-1}\boldsymbol{f} + \boldsymbol{g}$ then,
1. Solve $S_p z_p = r_p$, where $S_p = B D_v^{-1} B^T, D_v = diag(Q_V)$
2. Update $r_p = B D_V^{-1} F D_v^{-1} B^T z_p$
3. Solve $S_p z_p = -r_p$
4. Update $r_u = r_u - B^T z_p$
5. Solve $F z_u = r_u$
```

5.2.3. SIMPLE PRECONDITIONER

SIMPLE method was first introduced by Patankar & Spalding as a method to solve the coupled system iteratively. The following steps outline the method proposed by Patankar and Spalding

- Initialize pressure and velocity with the pressure velocity from previous step.
- Then we solve for velocity using the momentum equation, and pressure from the poisson equation obtained while imposing the solenoidicity of

velocity.

- We continue this procedure until desired convergence is reached.

The SIMPLE method, though very easy to implement, shows poor convergence properties for most of the problems. Although if used as a preconditioner [10], the spectral properties of Krylov subspace methods is much improved. It can be proven that some of the eigenvalues are clustered near 1, while the others are dependent on the approximation of the Schur complement. Further, we discuss the formulation of such a preconditioner.

Let the system we have to solve be $A\boldsymbol{x}=\boldsymbol{b}$ where,

$$A=\begin{bmatrix} F_r & 0 & 0 & G_r \\ 0 & F_\theta & 0 & G_\theta \\ 0 & 0 & F_z & G_z \\ G_r^T & G_\theta^T & G_z^T & 0 \end{bmatrix}=\begin{bmatrix} F & B^T \\ B & 0 \end{bmatrix},\quad \boldsymbol{x}=\begin{bmatrix} \boldsymbol{u}_r \\ \boldsymbol{u}_\theta \\ \boldsymbol{u}_z \\ \boldsymbol{p} \end{bmatrix} \text{ and } b=\begin{bmatrix} b_r \\ b_\theta \\ b_z \\ b_p \end{bmatrix}=\begin{bmatrix} \boldsymbol{f} \\ \boldsymbol{g} \end{bmatrix}.$$

Then we base the SIMPLE preconditioner on the approximation of LU where L & U are lower and upper diagonal of matrix A . Thus

$$\begin{bmatrix} F & B^T \\ B & 0 \end{bmatrix}\approx\begin{bmatrix} F & 0 \\ B & S_a \end{bmatrix}\begin{bmatrix} I & D^{-1}B^T \\ 0 & I \end{bmatrix},$$

where S_a is an approximate Schur complement constructed by approximating F by its diagonals in the definition of S , i.e., $S=-BF^{-1}B^T\approx-BD^{-1}B^T=S_a$. Thus one iteration of SIMPLE is to solve

$$\begin{bmatrix} F & 0 \\ B & S_a \end{bmatrix}\begin{bmatrix} I & D^{-1}B^T \\ 0 & I \end{bmatrix}\begin{bmatrix} \delta\boldsymbol{u} \\ \delta\boldsymbol{p} \end{bmatrix}=\begin{bmatrix} \boldsymbol{f} \\ \boldsymbol{g} \end{bmatrix}-\begin{bmatrix} F & B^T \\ B & 0 \end{bmatrix}\begin{bmatrix} \boldsymbol{u}^{(k)} \\ \boldsymbol{p}^{(k)} \end{bmatrix}=\begin{bmatrix} \boldsymbol{r}_u \\ \boldsymbol{r}_p \end{bmatrix},$$

where $\delta\phi=\phi^{k+1}-\phi^k$. The above equation can be solved in two steps:

$$\begin{bmatrix} F & 0 \\ B & S_a \end{bmatrix}\begin{bmatrix} \delta\hat{\boldsymbol{u}} \\ \delta\hat{\boldsymbol{p}} \end{bmatrix}=\begin{bmatrix} \boldsymbol{r}_u \\ \boldsymbol{r}_p \end{bmatrix}$$

and

$$\begin{bmatrix} I & D^{-1}B^T \\ 0 & I \end{bmatrix}\begin{bmatrix} \delta\boldsymbol{u} \\ \delta\boldsymbol{p} \end{bmatrix}=\begin{bmatrix} \delta\hat{\boldsymbol{u}} \\ \delta\hat{\boldsymbol{p}} \end{bmatrix}.$$

Then we update the velocity and pressure based on

$$\begin{bmatrix} \boldsymbol{u}^{(k+1)} \\ \boldsymbol{p}^{(k+1)} \end{bmatrix}=\begin{bmatrix} \boldsymbol{u}^{(k)} \\ \boldsymbol{p}^{(k)} \end{bmatrix}+\begin{bmatrix} \delta\boldsymbol{u} \\ \delta\boldsymbol{p} \end{bmatrix}.$$

Here, one of the above iterations is used as the preconditioner. The SIMPLE algorithm has some problems; firstly, the SIMPLE preconditioner may return poor results for convection dominated flows due to the approximation of F by its diagonal in construction of the Schur complement, which does not capture the convection operator accurately. Moreover, the method suffers a lot if the Reynolds number increases or the mesh size decreases.

Another point to note is that many of the Krylov subspace methods, for instance CG and GMRES, require a constant preconditioner, or more specifically, a constant inverse of the preconditioner. If a preconditioner is changing due to the requirement of the problem to be solved, or if we invert the preconditioner iteratively (i.e., we solve $P\boldsymbol{x}=\boldsymbol{y}$ iteratively where P is the preconditioner) this requirement of the Krylov subspace methods can not be fulfilled. The SIMPLE preconditioner is one of the above iterations, which itself is solved iteratively, hence the preconditioner changes in each iteration [10]. Therefore we use the GCR solver which, though a bit expensive, can handle variable preconditioners.

Algorithm 5.2 : SIMPLE preconditioner

```
1. $\boldsymbol{u}^{(k)}$ and $\boldsymbol{p}^{(k)}$ are known from previous iterations.
2. Set $\boldsymbol{r}_u=\boldsymbol{f}-F\boldsymbol{u}^{(k)}-B^T\boldsymbol{p}^{(k)}, \boldsymbol{r}_p=\boldsymbol{g}-B\boldsymbol{u}^{(k)}$
3. Solve $F\delta\hat{\boldsymbol{u}}=\boldsymbol{r}_u$
4. Solve $S_a\delta\hat{\boldsymbol{p}}=\boldsymbol{r}_p-B\delta\hat{\boldsymbol{u}}$
5. Update $\delta\boldsymbol{p}=\delta\hat{\boldsymbol{p}}$
6. Update $\delta\boldsymbol{u}=\delta\hat{\boldsymbol{u}}-D^{-1}B^T\delta\boldsymbol{p}$
7. Update $\boldsymbol{u}^{(k+1)}=\boldsymbol{u}^{(k)}+\delta\boldsymbol{u}, \quad \boldsymbol{p}^{(k+1)}=\boldsymbol{p}^{(k)}+\delta\boldsymbol{p}$
```

5.2.4. SIMPLER PRECONDITIONER

There are many variants of the SIMPLE preconditioner available, one such preconditioner is SIMPLER [10]. In SIMPLER we first solve for the pressure $\hat{\boldsymbol{p}}$ instead of assuming it to be the same as the previous iteration's pressure $\boldsymbol{p}^{(k)}$, and then we apply the SIMPLE algorithm with $\hat{\boldsymbol{p}}$ instead of $\boldsymbol{p}^{(k)}$. The algorithm is given below

$$\textbf{\textit{Solve}}\, S_a \hat{\boldsymbol{p}} = \boldsymbol{g} - B\boldsymbol{u}^{(k)} - BD^{-1}(\boldsymbol{f} - F\boldsymbol{u}^{(k)})$$, then

$$\begin{bmatrix} F & 0 \\ B & S_a \end{bmatrix}\begin{bmatrix} I & D^{-1}B^T \\ 0 & I \end{bmatrix}\begin{bmatrix} \boldsymbol{u} \\ \delta\boldsymbol{p} \end{bmatrix} = \begin{bmatrix} \boldsymbol{f} \\ \boldsymbol{g} \end{bmatrix} - \begin{bmatrix} F & B^T \\ B & 0 \end{bmatrix}\begin{bmatrix} 0 \\ \boldsymbol{p}^{(k)} \end{bmatrix} = \begin{bmatrix} \boldsymbol{r}_u \\ \boldsymbol{r}_p \end{bmatrix}$$

which can be solved in two steps

$$\begin{bmatrix} F & 0 \\ B & S_a \end{bmatrix}\begin{bmatrix} \hat{\boldsymbol{u}} \\ \delta\hat{\boldsymbol{p}} \end{bmatrix} = \begin{bmatrix} \boldsymbol{r}_u \\ \boldsymbol{r}_p \end{bmatrix}$$

and

$$\begin{bmatrix} I & D^{-1}B^T \\ 0 & I \end{bmatrix}\begin{bmatrix} \boldsymbol{u} \\ \delta\boldsymbol{p} \end{bmatrix} = \begin{bmatrix} \hat{\boldsymbol{u}} \\ \delta\hat{\boldsymbol{p}} \end{bmatrix}.$$

Then we update the velocity and pressure.

Algorithm 5.3 : SIMPLER preconditioner without pressure update damping

```
1. Solve $S_a \hat{\boldsymbol{p}} = \boldsymbol{g} - B\boldsymbol{u}^{(k)} - BD^{-1}(f - F\boldsymbol{u}^{(k)})$
2. Set $\boldsymbol{r}_u = \boldsymbol{f} - B^T\hat{\boldsymbol{p}}, \boldsymbol{r}_p = \boldsymbol{g}$
3. Solve $F\hat{\boldsymbol{u}} = \boldsymbol{r}_u$
4. Solve $S_a \delta\hat{\boldsymbol{p}} = \boldsymbol{r}_p - B\hat{\boldsymbol{u}}$
5. Update $\delta\boldsymbol{p} = \delta\hat{\boldsymbol{p}}$
6. Update $\boldsymbol{u} = \hat{\boldsymbol{r}} - D^{-1}B^T\delta\boldsymbol{p}$
7. Update $\boldsymbol{p} = \hat{\boldsymbol{p}} + \delta\boldsymbol{p}$
```

The SIMPLER preconditioner is supposed to have convergence independent of the Reynolds number. Unfortunately, in practice, not much improvement is seen when graduating from SIMPLE to SIMPLER preconditioners, for many test cases SIMPLER performs even worse than SIMPLE [10]. Hence, this preconditioner is not further discussed, however we discuss an another variant of SIMPLER, the so called MSIMPLER which is supposed to give better results than SIMPLER.

5.2.5. MSIMPLER PRECONDITIONER

The MSIMPLER preconditioner is an improved SIMPLER precondtioner presented by Segal et al. in [10]. It is inspired by the similarities between SIMPLE and commutator preconditioners presented by Elman et al. [13]. As presented in equations (5.2) and (5.3) for the commutator preconditioners, the more general form of Schur decomposition is given by

$$BF^{-1}B^T \approx (BM_1^{-1}B^T)F_p^{-1} \text{ with,}$$

$$F_p=(BM_2^{-1}B^T)^{-1}(BM_2^{-1}F\,M_1^{-1}B^T) \ .$$

In the equations (5.2) and (5.3) we took $M_1=M_2=diag(Q_v)$. Now, if we were to create a new block factorization preconditioner in which Schur complement is built on SIMPLE's approximate block factorization while being based on a commutator approximation, we get

$$P=LU\begin{bmatrix} I & 0 \\ 0 & F_p^{-1} \end{bmatrix} . \tag{5.4}$$

If $S=-BF^{-1}B^T\approx-BD^{-1}B^T=S_a$, $M_1=D$ and F_p is identity then equation (5.4) corresponds to the SIMPLE preconditioner. Similarly, if the pressure update step (step 4) in the SIMPLE algorithm is solved with $S_a=-(BM_1^{-1}B^T)F_p^{-1}$, then equation (5.4) is equivalent to the SIMPLE preconditioner.

In our case we deal with the time dependent multiphase flows. Hence, while discretizing the time dependent Navier-Stokes equations, we get a $1/timestep$ term on the diagonal of matrix F (which is obtained from the implicit time discretization of the momentum equation). The presence of this $1/timestep$ term on the diagonal makes the diagonal entries larger than the off diagonal entries, hence FD^{-1} is “close to” identity. This also implies that F_p is also close to identity. For the time dependent problems, D is also close to the diagonal of the velocity mass matrix. Hence, if we choose again $M_1=M_2=diag(Q_v)=Q_{vd}$ we get

$$F_p=(BQ_{vd}^{-1}B^T)^{-1}(BQ_{vd}^{-1}F\,Q_{vd}^{-1}B^T) \ .$$

From the above definition it is easy to see that if FQ_{vd}^{-1} is close to unity then F_p is also “close to” identity, hence the Schur complement becomes

$$BF^{-1}B^T\approx BQ_{vd}^{-1}B^T \ .$$

Therefore, if we replace D with the diagonal of the mass matrix of velocity Q_{vd} in the SIMPLE/ SIMPLER method we get the MSIMPLER method. The algorithm is presented later.

For time dependent Navier-Stokes problems, the MSIMPLER preconditioner is better than SIMPLER because in SIMPLER we must form the Schur complement after every iteration (as it is based on diagonal elements of F which needs to be updated after each iteration), while the MSIMPLER preconditioner has the velocity mass matrix in its definition of the Schur complement which does not change per iteration, hence we save time in building the Schur complement.

Algorithm 5.4 : MSIMPLER preconditioner without pressure update damping

```
1. Solve $S_a \hat{\boldsymbol{p}} = \boldsymbol{g} - B\boldsymbol{u}^{(k)} - BQ_{vd}^{-1}(\boldsymbol{f} - F\boldsymbol{u}^{(k)})$
2. Set $\boldsymbol{r}_u = \boldsymbol{f} - B^T\hat{\boldsymbol{p}}, \boldsymbol{r}_p = \boldsymbol{g}$
3. Solve $F\hat{\boldsymbol{u}} = \boldsymbol{r}_u$
4. Solve $S_a \delta \boldsymbol{p} = \boldsymbol{r}_p - B\hat{\boldsymbol{u}}$
5. Update $\delta \boldsymbol{p} = \delta \hat{\boldsymbol{p}}$
6. Update $\boldsymbol{u} = \hat{\boldsymbol{u}} - Q_{vd}^{-1} B^T \delta \boldsymbol{p}$
7. Update $\boldsymbol{p} = \hat{\boldsymbol{p}} + \delta \boldsymbol{p}$
```

Below we present the GCR algorithm with SIMPLE/SIMPLER type preconditioners, for more details about the algorithm one is referred to [14]. We choose GCR because it is stable, minimizes the residual norm, and allows for variable preconditioning which is typical of SIMPLE type precondtioners as discussed above.

Algorithm 5.5 : GCR - MSIMPLER preconditioner to solve $A\boldsymbol{x} = \boldsymbol{b}$

```
$\boldsymbol{r}_0 = \boldsymbol{b} - A\boldsymbol{x}_0$
for $k = 0, 1 \ldots, ngcr$
$\boldsymbol{s}_{k+1} = P^{-1}\boldsymbol{r}_k$
$\boldsymbol{v}_{k+1} = A\boldsymbol{s}_{k+1}$
for $i = 0, 1, \ldots, k$
    $\boldsymbol{v}_{k+1} = \boldsymbol{v}_{k+1} - (\boldsymbol{v}_{k+1}, \boldsymbol{v}_i)\boldsymbol{v}_i$
    $\boldsymbol{s}_{k+1} = \boldsymbol{s}_{k+1} - (\boldsymbol{v}_{k+1}, \boldsymbol{v}_i)\boldsymbol{s}_i$
    end for
    $\boldsymbol{v}_{k+1} = \boldsymbol{v}_{k+1} / \|\boldsymbol{v}_{k+1}\|_2$
    $\boldsymbol{s}_{k+1} = \boldsymbol{s}_{k+1} / \|\boldsymbol{s}_{k+1}\|_2$
    $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + (\boldsymbol{r}_k, \boldsymbol{v}_{k+1}) \boldsymbol{s}_{k+1}$
    $\boldsymbol{r}_{k+1} = \boldsymbol{r}_k - (\boldsymbol{r}_k, \boldsymbol{v}_{k+1}) \boldsymbol{v}_{k+1}$
end for
```

Where matrix P depends on the type of SIMPLE/SIMPLER preconditioner used, for MSIMPLER it is

$$P = H_r M_r^{-1} - H_r M_r^{-1} A M_l^{-1} H_l + M_l^{-1} H_l \text{ where}$$

$$H_r = \begin{bmatrix} I & 0 & 0 & -Q_{vr\,d}^{-1} G_r \\ 0 & I & 0 & -Q_{v\theta\,d}^{-1} G_\theta \\ 0 & 0 & I & -Q_{vz\,d}^{-1} G_z \\ 0 & 0 & 0 & I \end{bmatrix}, \quad M_r = \begin{bmatrix} F_r & 0 & 0 & 0 \\ 0 & F_\theta & 0 & 0 \\ 0 & 0 & F_z & 0 \\ G_r^T & G_\theta^T & G_z^T & S_a \end{bmatrix}$$

$$H_l = \begin{bmatrix} I & 0 & 0 & 0 \\ 0 & I & 0 & 0 \\ 0 & 0 & I & 0 \\ -G_r Q_{vr\,d}^{-1} & -G_\theta Q_{v\theta\,d}^{-1} & -G_z Q_{vz\,d}^{-1} & I \end{bmatrix}, \quad M_r = \begin{bmatrix} F_r & 0 & 0 & G_r \\ 0 & F_\theta & 0 & G_\theta \\ 0 & 0 & F_z & G_z \\ 0 & 0 & 0 & S_a \end{bmatrix}$$

and W is the block diagonal of $M_l + M_r - A$.

5.3. DEFLATION PRECONDITONER

As discussed above the aim of preconditioning is to improve the spectral properties of matrix A so that the Krylov subspace methods solve $A\boldsymbol{x} = \boldsymbol{b}$ efficiently. Presence of the small eigenvalues is one such spectral property which deteriorates the performance of the solver considerably. Therefore, it makes sense to have a preconditioner which deflates the small eigenvalues of A. This is the main motivation behind the deflation preconditioners [15, 16, 17, 18].

As will be discussed below, the deflation preconditioner requires construction of the deflation matrix which points to the eigenvalues that need to be deflated. Naturally, the construction of an *efficient* deflation matrix is quite important as bad choice of deflation matrix may deflate *good* eigenvalues rather than *bad* ones. One obvious way to construct the deflation matrix is to run the Arnoldi iterations to find the smallest eigenvalues, this however is a very expensive task and would definitely mitigate the speed-up achieved by deflation to a great extent, if not completely.

Another way is to find the eigenvalues based on the physics of the problem, for example, if an interface is present in the domain, the interface location points to the small eigenvalues and this information could be used to construct the deflation matrix. The third way, which pertains more to this research, is to construct the deflation matrix using the algebraic deflation vectors, for example in the case of domain decomposition. We shall discuss this more in detail later, for now it suffices to say that since in this research deflation preconditioners are only used to improve the performance of domain decomposition, from now on we will not consider the first two methods of constructing deflation matrix. Further, we discuss the deflation algorithm.

Suppose we have to solve $A\boldsymbol{x}=\boldsymbol{b}$ where $A\in\mathbb{R}^{n\times n}$. We consider a matrix $Z\in\mathbb{R}^{n\times m}$, where $m<n$ and rank of Z is m , i.e., columns of Z are all linearly independent. The columns of Z are called the deflation vectors and Z is called the <u>deflation matrix</u>. The columns of Z are spanned by the deflation subspace in which the *bad* (small) eigenvalues of A reside (which are to be projected out of the residual). For this we define two projectors

$$\Pi=I-AZE^{-1}Z^T \text{ and } Q=I-ZE^{-1}Z^T A \text{ , with}$$

$$\Pi^2=\Pi \text{ , } Q^2=Q \text{ and } E=Z^T AZ \text{ ,}$$

where $E\in\mathbb{R}^{m\times m}$ and I is identity matrix of appropriate size. To solve for $\boldsymbol{x}$ we write

$$\boldsymbol{x}=(I-Q)\boldsymbol{x}+Q\boldsymbol{x}=ZE^{-1}Z^T\boldsymbol{b}+Q\boldsymbol{x} \text{ .} \tag{5.5}$$

Here $ZE^{-1}Z^T\boldsymbol{b}$ is easily computed, hence we turn our attention towards the remaining expression $Q\boldsymbol{x}$. Since $\Pi A=AQ$ we compute $Q\boldsymbol{x}=\boldsymbol{x}_a$ by

$$\Pi A\boldsymbol{x}_a=\Pi AQ\boldsymbol{x}=\Pi^2 A\boldsymbol{x}=\Pi A\boldsymbol{x}=\Pi\boldsymbol{b} \text{ , i.e., we solve}$$

$$\Pi A\boldsymbol{x}_a=\Pi\boldsymbol{b} \text{ .}$$

Then if we multiply $\boldsymbol{x}_a$ by Q we get $Q\boldsymbol{x}_a=Q^2\boldsymbol{x}=Q\boldsymbol{x}$ hence we replace $Q\boldsymbol{x}=Q\boldsymbol{x}_a$ in equation (5.5) to solve for $\boldsymbol{x}$. For further discussion on deflation [15, 16, 17, 18] are recommended. In the above method, E is presumed to be easily invertible. It can be shown that the deflation method works correctly if E^{-1} is computed with high accuracy. Also, we do not explicitly compute E^{-1}

rather solve the system $E\boldsymbol{q}=\boldsymbol{t}$ with high accuracy using some direct method.

Now, we discuss the way to formulate the deflation matrix for the domain decomposition technique. The need for deflation for domain decomposition is discussed later (in the section where we discuss domain decomposition), here we only present how to formulate the deflation vectors. We base our deflation vectors on the decomposition of domain Ω , i.e., if we decompose the domain into d non-overlapping sub-domains we define the deflation vector $\boldsymbol{z}_i$ as

$$(\boldsymbol{z}_i)_j = \left\{ \begin{matrix} 0 & if\ x_i \notin \Omega_j \\ 1 & if\ x_i \in \Omega_j \end{matrix} \right\} \quad 1 \le i \le d \quad .$$

That is, the total number of deflation vectors is the number of non-overlapping sub-domains, and each vector has a non homogeneous entry only when that index belongs to the domain to which the corresponding deflation vector belongs. The exact deflation vector in our case will be presented in the domain decomposition sub-section. Further, we present the algorithms of a few Krylov subspace methods with deflation technique.

Algorithm 5.6: Deflated Preconditioned CG for solving $A\boldsymbol{x}=\boldsymbol{b}$ with preconditioner P and deflation matrix Z .

```
$Select\ \boldsymbol{x_0}.\ \ Compute\ r_0=(\boldsymbol{b}-A\boldsymbol{x_0}),\ \ set\ \boldsymbol{r_0^1}=\Pi\boldsymbol{r_0}$
$Solve\ P\boldsymbol{x_0}=\boldsymbol{r_0^1}$ and $set\ \boldsymbol{p_0}=\boldsymbol{y_0}$
for $j=0,1,\ldots until\ convergence$ do
  $\boldsymbol{w_j^1} = \Pi A\boldsymbol{p_j}$
  $\alpha = \frac{(\boldsymbol{r_j^1},\ \boldsymbol{y_j})}{(\boldsymbol{w_j^{1,}}\ \boldsymbol{p_j})}$
  $\boldsymbol{x_{j+1}^1} = \boldsymbol{x_j^1}+\alpha_j\boldsymbol{p_j}$
  $\boldsymbol{r_{j+1}^1} = \boldsymbol{r_j^1}-\alpha_j\boldsymbol{w_j^1}$
  $Solve\ P\boldsymbol{y_{j+1}} = \boldsymbol{r_{j+1}^1}$
  $\beta = \frac{(\boldsymbol{r_{j+1}^1},\boldsymbol{y_{j+1}})}{(\boldsymbol{r_j^1},\boldsymbol{y_j})}$
  $\boldsymbol{p_{j+1}} = \boldsymbol{y_{j+1}}+\beta_j\boldsymbol{p_j}$
end for
$\boldsymbol{x} = ZE^{-1}Z^T\boldsymbol{b} + Q\boldsymbol{x_{j+1}^1}$
```

Similarly the deflation technique can be used for non-symmetric systems as well. In this case instead of solving $A\boldsymbol{x}=\boldsymbol{b}$ (A non-symmetric), we use the above described method and solve $\Pi A\boldsymbol{x}_a=\Pi b$ using Krylov subspace methods like GMRES/IDR(s). Then we multiply $\boldsymbol{x}_a$ by Q and put it in the equation (5.5) to solve for $\boldsymbol{x}$. The first part ($ZE^{-1}Z^T\boldsymbol{b}$) is computed like before.

If the non-symmetric system is left preconditioned with preconditioner P then we instead solve $P^{-1}\Pi A\boldsymbol{x}_a=P^{-1}\Pi\boldsymbol{b}$ and form $Q\boldsymbol{x}_a$ to solve for $\boldsymbol{x}$. While if the system is right preconditioned we solve $\Pi AP^{-1}\boldsymbol{x}_a=\Pi\boldsymbol{b}$ and form $QP^{-1}\boldsymbol{x}_a$ to solve for $\boldsymbol{x}$.

6. PARALLELIZATION

The first step of parallelization involves making a choice regarding the architecture for which the parallelization of the code will be done. One has to choose between the shared memory systems and the distributed memory systems. We discuss both the systems in brief below.

6.1. SHARED MEMORY SYSTEM

In the shared memory systems, all the processors are directly connected to all the available memories via an Interconnect. Illustration 5 shows a conceptual design of such a machine. Shared memory systems can be further sub-categorized between the Uniform Memory Architecture (UMA) and Non-Uniform Memory Architecture (NUMA) architectures. The UMA, systems have all the processors at equal distance from all the memories. The more practical architecture is NUMA, where different processors have different distances from the different memories.

Parallelization for a shared memory systems has the following attributes,

- Memory access time for all the processors is same (for UMA systems only).
- Every processor accesses the same address space.
- Communication between the processors is done through shared variables.
- Every processor maintains a local view of the memory and this view must be updated if some updated variable is to be used by the processor, which gives rise to synchronization problems.

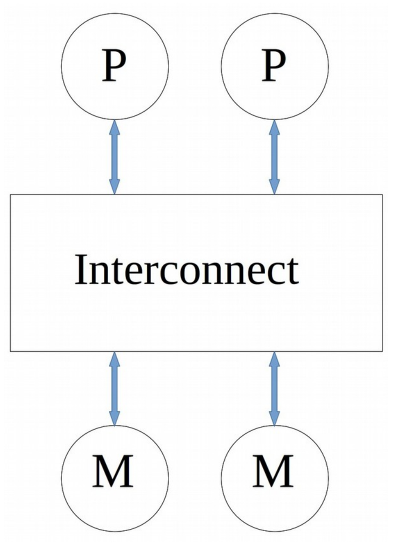

Illustration 5: Conceptual design of shared memory system

For shared memory parallelization one can employ OpenMP (for CPU's) and OpenCL/CUDA (for GPU's). Parallelization for GPU's is a relatively new area with huge potential due to massive numbers of processors/workers available in a GPU processor.

The disadvantage of parallelizing for a shared memory architecture is that such systems have scalability limit on both the memory and the processors. Furthermore, programming for GPU's is a highly involved and complex task for systems like the one we have in hand. Additionally, it is heavily plagued by synchronization problems. Further, we discuss the distributed system.

6.2. DISTRIBUTED SYSTEMS

The distributed systems are those in which all the processors have their own memories and do not have a direct access to each others memory. A distributed system has all the nodes/processors connected via an interconnect and theoretically has no requirement of close proximity of all the participating nodes. In practice many of the clusters and supercomputers have participating nodes which are several hundred miles away. Illustration 6 shows a conceptual design of such a system.

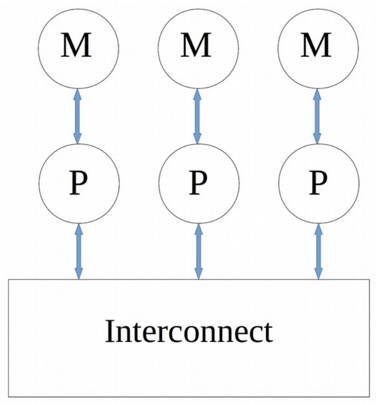

Illustration 6: Conceptual design of distributed systems

Parallelization for the distributed memory systems has the following attributes,

- Access time depends on distance to data, i.e., distance between processors.
- Every processor has its own local address space.
- Non local data can be accessed through a request to the owning processor.
- Communication through message passing.
- Data distribution is important and expensive.
- OpenMPI can be used to parallelize a software on distributed memory architectures.

In the distributed memory systems synchronization problems are much less of an issue since the communication is through explicit messages and no processor has direct access to other processor's memory. Another very important advantage of distributed systems is that they are theoretically infinitely scalable, hence we can run our parallel application over a much large numbers of processors as compared to the shared memory systems. Although the parallelization using OpenMPI is harder than Openmp, due to high scalability of distributed architectures we shall use OpenMPI to parallelize the available code.

6.3. DOMAIN DECOMPOSITION

After we have decided to choose distributed architectures, we must then focus our attention towards the parallelization methodology. Since fine grain parallelization is not very suitable for the distributed architectures we choose domain decomposition as a valid and efficient parallelization technique.

Now all that remains is to choose what type of domain decomposition we shall choose, i.e., shall we do column/row wise domain decomposition or 2d/3d block domain decomposition. The number of elements to be communicated is minimum in the case of 3d block domain decomposition, leading to substantial savings on the communication time, hence we choose a 3d block domain decomposition for the present study.

Computing the solution in parallel on a decomposed domain reduces the computational time per iteration of a Krylov iterative solver. On the other hand however, it increases the total number of iterations to be performed [15, 16, 17, 18], thus reducing the total efficiency of the parallelized system. As reasoned in [18] though it is simple to implement a non-overlapping preconditioner in parallel, the convergence behavior of the preconditioner deteriorates considerably when the domain is split into a high number of sub-domains. The loss of convergence is attributed to the small eigenvalues arising from the domain decomposition. This motivates the use of deflation for improving the convergence of the incomplete Cholesky preconditioned CG method (used to solve the Poisson equation). Deflation technique can also be used as a preconditioner for the coarse grid correction [15], for this reason we shall use deflation in combination with the diagonal scaling as a preconditioner for the predictor equation.

As previously discussed, for applying the deflation technique we must construct the deflation matrix whose construction is also discussed above. Here we present the deflation matrix for our case. Let us subdivide our domain into d sub-domains and let us have total n number of unknowns in the total domain and $m_{ri}, m_{\theta}i, m_{zi}, m_{pi}$ be the number of velocity and pressure unknowns in the i^{th} sub-domain $1 \le i \le d$. Further let $\{ri, \theta i, zi, pi\}$ be the index set (of velocity and pressure unknowns in the i^{th} sub-domain) giving the location of corresponding unknowns in $n \times 1$ vector, then deflation vectors $\mathbf{z}_i$ can be given by two ways;

$$z_{(r/\theta/z/p)i}(k) = \begin{cases} 1 & if\ k \in \{ri/\theta i/zi/pi\} \\ 0 & otherwise \end{cases} \text{ or } z_i(k) = \begin{cases} 1 & if\ k \in \{ri, \theta i, zi, pi\} \\ 0 & otherwise \end{cases}$$

Example: the deflation vectors for a 2-d square, having a square domain divided into 4 sub-domains (as shown in Illustration 7), can be constructed in two ways as indicated below:

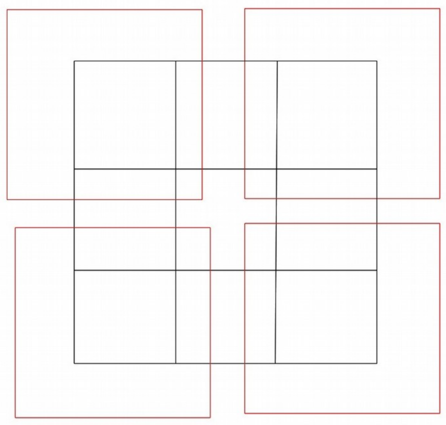

Illustration 7: Description of 2-d domain for forming deflation vectors

1st way

$$\mathbf{z}_{x1}=\begin{bmatrix}1\\0\\0\\0\\0\\0\\0\\0\\0\\0\\0\\0\end{bmatrix},\ \mathbf{z}_{y1}=\begin{bmatrix}0\\0\\0\\0\\1\\0\\0\\0\\0\\0\\0\\0\end{bmatrix},\ \mathbf{z}_{p1}=\begin{bmatrix}0\\0\\0\\0\\0\\0\\0\\0\\1\\0\\0\\0\end{bmatrix},\ Z_{x2}=\begin{bmatrix}0\\1\\0\\0\\0\\0\\0\\0\\0\\0\\0\\0\end{bmatrix},\ \mathbf{z}_{y2}=\begin{bmatrix}0\\0\\0\\0\\0\\1\\0\\0\\0\\0\\0\\0\end{bmatrix},\ \mathbf{z}_{p2}=\begin{bmatrix}0\\0\\0\\0\\0\\0\\0\\0\\0\\1\\0\\0\end{bmatrix},\ \mathbf{z}_{x3}=\begin{bmatrix}0\\0\\1\\0\\0\\0\\0\\0\\0\\0\\0\\0\end{bmatrix},\ \mathbf{z}_{y3}=\begin{bmatrix}0\\0\\0\\0\\0\\0\\1\\0\\0\\0\\0\\0\end{bmatrix},\ \mathbf{z}_{p3}=\begin{bmatrix}0\\0\\0\\0\\0\\0\\0\\0\\0\\0\\1\\0\end{bmatrix}$$

$$,\ \mathbf{z}_{x4}=\begin{bmatrix}0\\0\\0\\1\\0\\0\\0\\0\\0\\0\\0\\0\end{bmatrix},\ \mathbf{z}_{y4}=\begin{bmatrix}0\\0\\0\\0\\0\\0\\0\\1\\0\\0\\0\\0\end{bmatrix},\ \mathbf{z}_{p4}=\begin{bmatrix}0\\0\\0\\0\\0\\0\\0\\0\\0\\0\\0\\1\end{bmatrix}\quad \mathbf{z}_{i}=\begin{bmatrix}\mathbf{u}_1\\\mathbf{u}_2\\\mathbf{u}_3\\\mathbf{u}_4\\\mathbf{v}_1\\\mathbf{v}_2\\\mathbf{v}_3\\\mathbf{v}_4\\\mathbf{p}_1\\\mathbf{p}_2\\\mathbf{p}_3\\\mathbf{p}_4\end{bmatrix}\ \text{and}$$

$$Z=[\mathbf{z}_{x1}\quad \mathbf{z}_{y1}\quad \mathbf{z}_{p1}\quad \mathbf{z}_{x2}\quad \mathbf{z}_{y2}\quad \mathbf{z}_{p2}\quad \mathbf{z}_{x3}\quad \mathbf{z}_{y3}\quad \mathbf{z}_{p3}\quad \mathbf{z}_{x4}\quad \mathbf{z}_{y4}\quad \mathbf{z}_{p4}]$$

2nd Way

$$\boldsymbol{z}_1=\begin{bmatrix}\mathbf{1}\\\mathbf{0}\\\mathbf{0}\\\mathbf{0}\\ \\\mathbf{1}\\\mathbf{0}\\\mathbf{0}\\\mathbf{0}\\ \\\mathbf{1}\\\mathbf{0}\\\mathbf{0}\\\mathbf{0}\end{bmatrix},\ \boldsymbol{z}_2=\begin{bmatrix}\mathbf{0}\\\mathbf{1}\\\mathbf{0}\\\mathbf{0}\\ \\\mathbf{0}\\\mathbf{1}\\\mathbf{0}\\\mathbf{0}\\ \\\mathbf{0}\\\mathbf{1}\\\mathbf{0}\\\mathbf{0}\end{bmatrix},\ \boldsymbol{z}_3=\begin{bmatrix}\mathbf{0}\\\mathbf{0}\\\mathbf{1}\\\mathbf{0}\\ \\\mathbf{0}\\\mathbf{0}\\\mathbf{1}\\\mathbf{0}\\ \\\mathbf{0}\\\mathbf{0}\\\mathbf{1}\\\mathbf{0}\end{bmatrix},\ \boldsymbol{z}_4=\begin{bmatrix}\mathbf{0}\\\mathbf{0}\\\mathbf{0}\\\mathbf{1}\\ \\\mathbf{0}\\\mathbf{0}\\\mathbf{0}\\\mathbf{1}\\ \\\mathbf{0}\\\mathbf{0}\\\mathbf{0}\\\mathbf{1}\end{bmatrix}\quad \boldsymbol{z}_i=\begin{bmatrix}\boldsymbol{u}_1\\\boldsymbol{u}_2\\\boldsymbol{u}_3\\\boldsymbol{u}_4\\\boldsymbol{v}_1\\\boldsymbol{v}_2\\\boldsymbol{v}_3\\\boldsymbol{v}_4\\\boldsymbol{p}_1\\\boldsymbol{p}_2\\\boldsymbol{p}_3\\\boldsymbol{p}_4\end{bmatrix}\ \text{and}\ Z=[\boldsymbol{z}_1\quad \boldsymbol{z}_2\quad \boldsymbol{z}_3\quad \boldsymbol{z}_4]\ .$$

Where bold numerals denote the vectors of the size m_{xi}, m_{yi}, m_{pi} in each sub-domain i . It is not clear which of the above two ways of constructing the deflation vectors will give a better performance. In this research however we plan to use the 1st way as it *seems* to be more suitable for mechanical problems having discontinuous coefficients [19].

6.4. COMMUNICATION OVERLAPPING CG

Since we aim to parallelize the present code in this endeavor, perhaps it is also prudent to analyze the computational bottlenecks of the parallel CG method. CG requires three global dot products, which corresponds to high communication cost on a parallel architecture if number of processors is fairly large. To suppress the global communication cost we might think of overlapping the communication with computation [20]. If we write preconditioner as $P = LL^T$, then Algorithm 6.1 gives a communication overlapping CG (CoCG).

Algorithm 6.1: Communication overlapping CG (CoCG)

```
$\boldsymbol{x}_{-1} = \boldsymbol{x}_0 = intial\ guess;\quad \boldsymbol{r}_0 = b - A\boldsymbol{x}_0$

$\boldsymbol{p}_{-1}=0;\quad \alpha_{-1}=0$

$\boldsymbol{s} = L^{-1}\boldsymbol{r}_0$

$\rho_{-1} = 1$

$for\ \ i = 0, 1, 2, ...\ \ do$

(1)    $\rho_i=(\boldsymbol{s},\boldsymbol{s})$
```

```
        $\boldsymbol{w}_i = L^{-T}\boldsymbol{s}$
        $\beta_{i-1} = \rho_i/\rho_{i-1}$
        $\boldsymbol{p}_i = \boldsymbol{w}_i + \beta_{i-1}\boldsymbol{p}_{i-1}$
        $\boldsymbol{q}_i = A\,\boldsymbol{p}_i$
(2)     $\gamma = (\boldsymbol{p}_i, \boldsymbol{q}_i)$
        $\boldsymbol{x}_i = \boldsymbol{x}_{i-1} + \alpha_{i-1}\boldsymbol{p}_{i-1}$
        $\alpha_i = \rho_i/\gamma$
        $\boldsymbol{r}_{i+1} = \boldsymbol{r}_i - \alpha_i\boldsymbol{q}_i$
(3)     $compute\|\boldsymbol{r}\|$
        $\boldsymbol{s} = L^{-1}\boldsymbol{r}_{i+1}$
        If accurate enough then
            $\boldsymbol{x}_{i+1} = \boldsymbol{x}_i + \alpha_i\boldsymbol{p}_i$
            quit
end
```

Algorithm 6.1 only reschedules some operations and is numerically just as stable as the normal CG. In the above algorithm the communication of the inner products (1), (2), (3) can be overlapped by the computation which follows in the subsequent lines. The total run-time of parallel CG is $c+a$ where a is the total computational time of one iteration and c is the communication time. If for $0\leq\beta\leq 1$, βa give the overlapping computation time, then the run-time of CoCG is $(1-\beta)a + max(\beta a, c)$. As number of processors increase a decreases while c increases, hence if the value of β is high we can hide higher communication time behind the computational time. Although for a rather high number of processors it would not be possible to completely overlap communication and computation, but we will still save on some communication time. For more details one can refer to [20].

6.5. COMMUNICATION REDUCING IDR(s)

The IDR(s) method also involves global communication which, like in CG, will affect the parallelization efficiency. An algorithm can be found in [21] which reduces the number of global communications per iteration of IDR(s). This algorithm is different from the Algorithm (4.1) presented earlier, in the sense that this algorithm exploits the bi-orthogonality property of the residual and change in the residual at each step [21]. We present the algorithm below. For a more rigorous derivation, the interested readers can refer to [21].

Algorithm 6.2: Minsync IDR(s) to solve $A\boldsymbol{x}=b$

```
Required: $A \in \mathbb{R}^{N\times N}$, $\boldsymbol{x}_{0,}$ $\boldsymbol{b} \in \mathbb{R}^{N}$, $T \in \mathbb{R}^{N\times s}$
Initialize $\boldsymbol{r}_0 = \boldsymbol{b} - A\boldsymbol{x}_0$ ,
 $\Delta R = \Delta X = 0 \ \mathbb{R}^{N\times s}$ ; $M_l = I \in \mathbb{R}^{s\times s}$ ; $M_t = M_c = 0$ ; $\omega = 1$
 $\boldsymbol{\phi} = P^H \boldsymbol{r}$ , $\boldsymbol{\phi} = (\phi_{1,}\dots,\phi_s)^T$
//Loop over nested $G_j$ spaces, $j = 0,1,\dots$
 $while\ \|r\| > Tol$ and $k < Max\ iter\ do$
    //Compute s linearly independent vectors $\Delta\boldsymbol{r}_k$ in $G_j$
    $for\ k = 1, s\ do$
        $Solve\ for\ \boldsymbol{\gamma}\ by\ M_l\boldsymbol{\gamma}_{(k:s)} = \boldsymbol{\phi}_{(k:s)}$
        $\boldsymbol{v} = \boldsymbol{r} - \sum_{i=k}^{s} \Delta\boldsymbol{r}_i\gamma_i$
        $\Delta\hat{\boldsymbol{x}}_k = \sum_{i=k}^{s} \gamma_i\Delta\boldsymbol{x}_i + \omega\boldsymbol{v}$ //Intermediate vector $\Delta\hat{\boldsymbol{x}}_k$ .
        $\Delta\hat{\boldsymbol{r}}_k = A\delta\hat{\boldsymbol{x}}_k$ //Intermediate vector $\Delta\hat{\boldsymbol{r}}_k$ .
        $\boldsymbol{\psi} = P^H\Delta\hat{\boldsymbol{r}}_k$ **// s inner products (combined)**
        $Solve\ M_t\boldsymbol{\alpha}_{(1:k-1)} = \boldsymbol{\psi}_{(1:k-1)}$
      //Make $\Delta\hat{\boldsymbol{r}}_k$ orthogonal to $\boldsymbol{p}_{1,}\dots,\boldsymbol{p}_k$ $update\ \Delta\hat{\boldsymbol{x}}_k\ accordingly$
        $\Delta\boldsymbol{r}_k = \Delta\hat{\boldsymbol{r}}_k - \sum_{i=1}^{k-1}\alpha_i\Delta\boldsymbol{r}_i$ and $\Delta\boldsymbol{x}_k = \Delta\hat{\boldsymbol{x}}_k - \sum_{i=1}^{k-1}\alpha_i\Delta\boldsymbol{x}_i$
      //Update column k of $M_l$
        $\mu^l_{i,k} = \psi_i - \sum_{j=i}^{k-1}\alpha_j\mu^c_{i,j}\ \ for\, i = k,\dots,s$
      //Make $\boldsymbol{r}$ orthogonal to $\boldsymbol{p}_{1,}\dots,\boldsymbol{p}_k$ $update\ \boldsymbol{x}\ accordingly$
        $\beta = \phi_k/\mu^l_{k,k}$
        $\boldsymbol{r} = \boldsymbol{r} - \beta\Delta\boldsymbol{r}_k$
      //Update $\boldsymbol{\phi} = P^H\boldsymbol{r}$
        $if\ \ k+1 \le s\ \ then$
            $\phi_i = 0\ \ for\ i=1,\dots,k$
            $\phi_i = \phi_i - \beta\mu^l_{i,k}\ \ for\ i = k+1,\dots,s$
        $endif$
    $end\ for$
    //Entering $G_{j+1}$.  $Note: \boldsymbol{r} \perp P$
     $\boldsymbol{v} = \boldsymbol{r}$
     $\boldsymbol{t} = A\boldsymbol{v}$
     $\omega = (\boldsymbol{t}^H \boldsymbol{v})/(\boldsymbol{t}^H \boldsymbol{t})$ ;  $\boldsymbol{\phi} = -P^H \boldsymbol{t}$  **//s+2 inner products (combined)**
     $\boldsymbol{r} = \boldsymbol{r} - \omega \boldsymbol{t}$
     $\boldsymbol{x} = \boldsymbol{x} + \omega \boldsymbol{v}$
     $\boldsymbol{\phi} = \omega \boldsymbol{\phi}$
  end while
```

From the above algorithm it is clear that each iteration requires synchronizations only once [21]. This reduces the global communication time since the communication overhead (latency), which is incurred anytime communication happens, is reduced. Moreover, this formulation is more stable than the one presented in Algorithm 4.1 which is also an added benefit.

7. PROFILING & IMPROVING THE PRESENT CODE

To increase the speed of a code it is important to identify the parts which take the longest time per iteration. It is not very wise to improve the efficiency of those parts of the program which take little time in comparison to others, because it does not help in improving the overall speed by much. Hence, the profiling of the available code was done to identify the components which take the maximum time. The profiling results of the original code for various density to viscosity ratios (which correspond to Reynolds number) are presented in Table 1, with percentage denoting the percentage of total time taken by a particular module.

Table 1: Profiling results of original code for 5 time steps on a 14x45x45 grid.

Density to viscosity ratio (Fluid 1 & Fluid 2)	Flow Time(s) - percentage			Interface Time(s) - percentage		
	Predictor Time(s) / percentage	Poisson Time(s) / percentage	Corrector Time(s) / percentage	Advect VOF & LS Time(s) / percentage	LS_reinit Time(s) / percentage	Coupling Time(s) / percentage
1e3 & 1e5	14.36 (s) – 89.7%			1.36 (s) – 8%		
	14.46 (s) 84.5%	1.004 (s) 5.8%	0.05 (s) 0.1%	0.05 (s) 0.1%	0.93 (s) 5.4%	0.35 (s) 2.2%

1e2 & 1e4	54.87 (s) – 96.3%			1.53 (s) – 2.7%		
	54.0 (s) 94.8%	0.97 (s) 1.9%	0.05 (s) 0.1%	0.05 (s) 0.1%	1.09 (s) 1.9%	0.35 (s) 0.6%
1e2 & 1e3	57.13 (s) – 96.4%			1.52 (s) – 2.6%		
	56.25 (s) 94.9%	1.0 (s) 1.7%	0.04 (s) 0.1%	0.05 (s) 0.1%	1.09 (s) 1.8%	0.34 (s) 0.6%

As evident from the above profiling results, the predictor part takes the longest time followed by the interface calculation part. The Poisson solver, contrary to our expectations, took much less time.

In the current code, restarted GMRES is used to solve the predictor part. To speed it up, various modifications are performed on the original code. The first improvement is motivated by the knowledge that Krylov subspace solvers converge faster with preconditioning. Hence, the Jacobi preconditioner was used to speed up the predictor module. This preconditioner was chosen because it is easy to implement, delivers considerable speedups for our case [7], and also lends itself well to parallel programming which is a highly desirable property for the present endeavor. Table 2 shows the results obtained by performing the Jacobi scaling. As is evident from the results, the Jacobi preconditioned GMRES performs better than the unconditioned GMRES, and for low Reynolds number a speedup of 2 is obtained.

Table 2: Comparison of Jacobi preconditioned GMRES (used to solve the predictor step) with non preconditioned GMRES (50 time steps on a 14x45x45 grid).

Case	Density to viscosity ratio	GMRES			
		Not Preconditioned		Preconditioned	
		# Iter.	Time	# Iter.	Time
1	1e4 & 1e6	30	24 (s)	22	19 (s)
2	1e3 & 1e5	100	87 (s)	60	52 (s)
3	1e2 & 1e4	387	320 (s)	182	157.5 (s)
4	1e2 & 1e3	390	337.5 (s)	191	162.5 (s)

The second major improvement was motivated by the fact that the calculations of matrix entries are performed in each iteration of GMRES unnecessarily. Hence, if we could save the matrix in some *diagonal* form it would decrease the computational time. This modification, though quite complex and involved to perform, was rather fruitful, as it gave an overall speedup of more than factor 2 irrespective of the Reynolds number. Table 3 presents the results obtained after the 2nd improvement.

Table 3: Comparison of GMRES with storing the matrix and without storing the matrix (50 time steps on a 15x45x45 grid).

Case	Density to viscosity ratio	GMRES			
		Without matrix storing		With matrix storing	
		# Iter.	Time	# Iter.	Time
1	1e4 & 1e6	22	19 (s)	22	7.32 (s)
2	1e3 & 1e5	60	52 (s)	60	20.7 (s)
3	1e2 & 1e4	182	157.5 (s)	182	61.05 (s)
4	1e2 & 1e3	191	162.5 (s)	191	64.02 (s)

The above modifications improve the performance of the serial version substantially. For the low Reynolds number cases, which are typical of the target applications, a speed up of approximately 5 is obtained, for example, case 4 initially took 337.5 (s) but after the two improvements it took only 64.02 (s). Similarly, for the high Reynolds number cases a speed up of approximately 3-4 can be expected as a result of the above improvements.

8. TEST CASE

To validate the accuracy and speedup given by the modified code (parallelized version with improved solvers and preconditioners) we plan to apply it to two test cases. The first test case is an academic problem used to validate the modified code, while the second test case is more practical, and is typical of the problems for which the code is developed.

8.1. DAM BREAK / BENJAMIN BUBBLE PROBLEM

In the dam break problem we consider two fluids with different densities in a closed pipe, initially at rest. The fluids are separated by a vertical dam as shown in Illustration 8. At t=0 the dam breaks, and the fluids start to move so as to attain an equilibrium where the heavier fluid is below the lighter one.

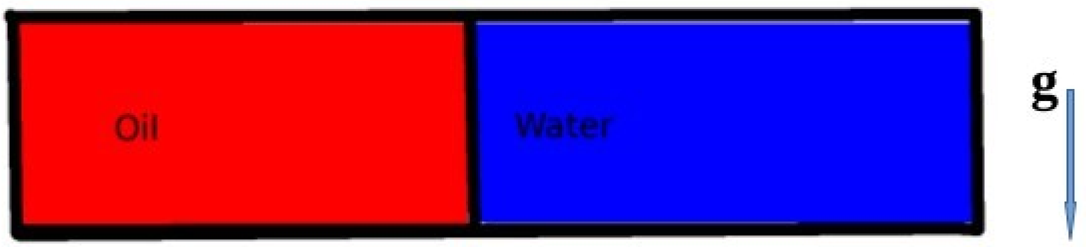

Illustration 8: Initial condition for test case 1.

We impose a slip boundary condition at the walls, with the two fluids initially at rest. The properties of the oil and water are indicated in Table 4. To check the accuracy, we shall compare the interface velocities obtained by the modified and serial codes. There also exist analytical solutions to such problems which can be used to check the accuracy gain given by the coupled solver. Finally, we shall present the speedup obtained and scaling behavior of the parallel solver for a medium sized grid.

8.2. PIPE FLOW

As mentioned in the introduction, TNO-Netherlands, Shell and Deltares are interested in multiphase flow happening in pipes. In such cases, pipes are initially filled with oil, and water is pumped to flush the oil out. Many experiments to visualize the shape and propagation velocity of the liquid-liquid interface during the flushing process have been performed in the past. We shall simulate one such test case provided by TNO to check the speed-up given by the parallelism, faster solvers, and better preconditioners. We also aim to check the accuracy improvement given by the coupled solver. Below, we provide details of the simulated test case.

We simulate the flow field inside a pipe, configured as indicated in Illustration 9. Initially the pipe is assumed to be filled with stagnant oil, and at t=0, water is provided at the inlet to flush the oil out. After some time the water reaches the bend, and instead of rising further, it creeps horizontally, displacing the oil. The resulting interface is tricky to capture numerically. It is seen in previous simulations done at TNO, that a no-slip boundary condition gives an unphysical interface shape [22] (where an oil film is formed between water and the wall), while a slip boundary condition over-predicts the speed of the interface. Hence a Navier-slip boundary condition is implemented in the present code, in which the shear stress near the contact point of the interface and wall is reduced, i.e., we have a degree of freedom with which we can specify the slip near the interface-wall contact point. For more details one could refer to [22]. Table 4 gives the relative properties of oil and water used in the test.

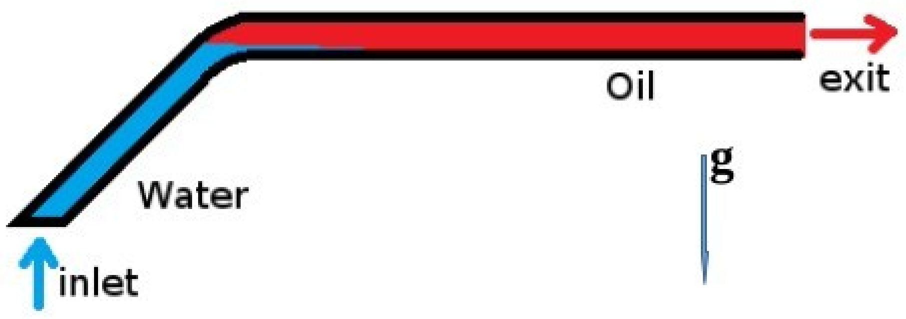

Illustration 9: Configuration for test case 2.

Table 4: Properties of the oil and water used in the test case.

Fluid	Relative density (to water) @15^0 C	Kinematic viscosity @40^0 C
Water	1	0.658 mm^2/s
Nexbase 3080	0.84	48 mm^2/s

The volumetric water flow at the inlet is 0.0003023 m^3/s, and its superficial velocity is 0.1069 m/s. Due to the complexity of the analysis we cannot, at this stage, provide the suitable boundary conditions and length of the pipe that will be simulated. They will be experimentally derived at a later stage.

To check the accuracy of the modified code we plan to compare the velocity and pressure profiles obtained by the serial and parallel codes, and contrast it against the experimentally available results. We shall then present the speed-up and scalability obtained by the parallelism.

9. RESEARCH QUESTIONS & METHODOLOGY

Based on the current study we are posed with several research questions:

- By how much can we speed-up the code using better solvers and parallelism?
- How much is the accuracy improved by solving the coupled system?
- Which is the most efficient preconditioner to solve the coupled system?
- By how much does the deflation preconditioner (for coarse grid correction) improve the convergence of the non-symmetric convection-diffusion equation?
- As discussed above, for our problem we can choose the deflation vector in 2 different ways, which of these two ways gives a better performance?

- How does the IDR(s) method perform in comparison to the GMRES method to solve the non-symmetric convection-diffusion equation?

To address of these questions the following track can be taken:

- Implement parallelization in the code.
- Improve the performance of the resulting system by adding deflation to it.
- Extend the system to a coupled system.
- Implement the IDR(s) solver and compare the performance with already available GMRES.
- Run the test case to access the speed up given by the parallelization, IDR(s) and deflation. Further we check the scaling behavior of the parallelization. The accuracy improvement by solving the coupled system is also accessed.

10. REFERENCES

1. S. van der Pijl. *“Computation of bubbly flows with a Mass-Conserving Level-Set Method.”* PhD Thesis, TU-Deflt (2005) .

2. Y. Morinishi, O.V. Vasilyev, Takeshi Ogi. *“Fully Conservative finite difference scheme in cylindrical coordinates for incompressible flow simulations”.* Journal of Computational Physics 197, (2004) pp. 686-710.

3. A. Arakawa, V.R. Lamb. *"Computational design of the basic dynamical processes of the UCLA general circulation model".* Methods of Computational Physics 17, (1977) pp. 173–265.

4. T.W.H Sheu, R.K. Lin. *“Newton linearization of the incompressible Navier–Stokes equations”.* Int. J. Numer. Meth. Fluids 44, (2004) pp. 297-312.

5. J. van Kan. *“A Second-Order Accurate Pressure-Correction Scheme For Viscous Incompressible Flow.”* SIAM J. Sci. Stat. Comput. 7(3), (1986) pp. 870-891.

6. L.N. Trefethen, D. Bau.*“Numerical Linear Algebra.”* Philadelphia, SIAM (1997).

7. A. J. Wathen. *“Preconditioning”.* Acta Numerica, 24, (2015) pp. 329-376.

8. P. Sonneveld, M.B. van Gijzen. *“IDR(s): A Family of Simple and Fast Algorithms for Solving Large Nonsymmetric Stsrems of Linear Equations.”* SIAM J. Sci. Comput., 31 (2), (2008) pp. 1035-1062.

9. J.A. Meijerink, H.A. van der Vorst. *“An iterative solution method for linear systems of which the coefficient matrix is a symmetric M-matrix.”* Math. Comp., 31, (1977) pp. 148–162.

10. A. Segal, M. ur Rehman, C. Vuik. *“Preconditioners for Incompressible Navier-*

Stokes Solvers”. Numer. Math. Theor. Meth. Appl. 3(3), (2010) pp 245-275.

11. M. F. Murphy, G. H. Golub, A. J. Wathen. *“A Note on Preconditioning for Indefinite Linear Systems”*. SIAM J. Sci. Comput., 21(6), (2000) pp. 1969–1972.

12. D. Kay, D. Loghin, A. J. Wathen. *“A preconditioner for the steady-state Navier–Stokes equations.”* SIAM J. Sci. Comput., 24, (2002) pp. 237–256.

13. H.C. Elman, V.E. Howle, J. Shadid, R. Shutterworth, R. Tumirano. *“Block Preconditioner Based on Approximate Commutators.”* SIAM J. Sci. Comput., 27 (5) , (2006) pp. 1651-1668.

14. C. Vuik, A. Saghir, G.P. Boerstoel. *“The Krylov Accelerated SIMPLE(R) Method for Flow Problems in Industrial Furnaces”*. Int. J. Numer. Meth. Fluids 33, (2000) pp. 1027-1040.

15. R. Nabben, C. Vuik. *“Domain Decomposition Methods and Deflated Krylov Subspace Iterations.”* ECCOMAS CFD (2006).

16. J. Verkaik, C. Vuik, B.D. Paarhuis, A. Twerda. *“The Deflation Accelerated Schwarz Method for CFD.”* ICCS, (2005) pp 868-875.

17. J. Frank, C. Vuik, A. Segal. *“On The Construction of Deflation-Based Preconditioners.”* SIAM J. Sci. Comput., 23 (2), (2001) pp. 442-462.

18. C. Vuik, J. Frank. *“Coarse Grid Acceleration of a Parallel Block Preconditioner.”* Future Generation Computer Systems. 17 (2001), pp. 933-940.

19. T.B. Jonsthovel, M.B. van Gijzen, C. Vuik, A. Scarpas. *“On The Use Of Rigid Body Modes In The Deflated Preconditioned Conjugate Gradient Method.”* SIAM J. Sci. Comput., 35 (1), (2012) pp. B207-B225.

20. E. de Sturler, H.A. van der Vorst. *“Reducing the effect of the global communication in GMRES(m) and CG on the parallel distributed memory computers.”* Appl. Numerical Mathematics 18, (1995) pp. 441-459.

21. T. P. Collignon, M.B van Gijzen. *“Minimizing Synchronization in IDR(s).”* Numer. Liner Algebra Appl. 18, (2011)pp. 805-825.

22. B. de Jong. *“Contact Line Dynamics in Oil Water Simulations.”* Internship Report, TNO (2015).