January 2018

Interfacing The CFD Code MFiX With The PETSc Linear Solver Library To Achieve Reduced Computation Times

Lauren Clarke

INTERFACING THE CFD CODE MFIX WITH THE PETSC LINEAR SOLVER LIBRARY TO ACHIEVE REDUCED COMPUTATION TIMES

by

Lauren Elizabeth Clarke
Bachelor of Science, University of North Dakota, 2016

A Thesis

Submitted to the Graduate Faculty

of the

University of North Dakota

In partial fulfillment of the requirements

for the degree of

Master of Science

Grand Forks, North Dakota
May
2018

This thesis, submitted by Lauren Clarke in partial fulfillment of the requirements for the Degree of Master of Science from the University of North Dakota, has been read by the Faculty Advisory Committee under whom the work has been done and is hereby approved.

K. Gautham

Dr. Gautham Krishnamoorthy, Chairperson

Dr. Frank Bowman

Michael D. Mann

Dr. Michael Mann

This thesis is being submitted by the appointed advisory committee as having met all of the requirements of the School of Graduate Studies at the University of North Dakota and is hereby approved.

Dr. Grant McGimpsey
Dean of the School of Graduate Studies

May 1, 2018
Date

PERMISSION

Title:	Interfacing the CFD Code MFiX with the PETSc Linear Solver Library to Achieve Reduced Computation Times
Department:	Chemical Engineering
Degree:	Master of Science

In presenting this thesis in partial fulfillment of the requirements for a graduate degree from the University of North Dakota, I agree that the library of this University shall make it freely available for inspection. I further agree that permission for extensive copying for scholarly purposes may be granted by the professor who supervised my thesis work, or in his absence, by the Chairperson of the department or the dean of the School of Graduate Studies. It is understood that any copying or publication or other use of this thesis or part thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and the University of North Dakota in any scholarly use which may be made of any material in my thesis.

Lauren Elizabeth Clarke

May 2018

TABLE OF CONTENTS

CHAPTER

LIST OF FIGURES

Figure Page

LIST OF TABLES

Table Page

NOMENCLATURE

A	area of a control volume face, m^2
$\boldsymbol{A}$	matrix defining a linear system
a	coefficients containing flow properties from discretized equations
b	source term
$\boldsymbol{b}$	right-hand side vector of a linear system
C_p	pressure coefficient
C_{pg}	specific heat of the fluid-phase
C_{pm}	specific heat of the m^{th} solids phase
C, D, U, f	locations for TVD schemes
$\boldsymbol{D}$	diagonal matrix
$\mathcal{D}$	diffusion coefficient
dwf	downwind weighting factor for TVD schemes
$-\boldsymbol{E}$	lower triangular matrix
$-\boldsymbol{F}$	upper triangular matrix
F	interface transfer coefficient
f	fluid flow resistance due to porous media
g	acceleration due to gravity, m^2/s
H	total rate of enthalpy change

I	momentum transfer between two phases
$\mathcal{K}_m$	m^{th} Krylov subspace
$\boldsymbol{M}$	preconditioning matrix
$\boldsymbol{L}$	sparse lower triangular matrix for ILU
P	pressure
q	conductive heat flux
R	mass transfer of a chemical species due to reactions or other phenomena
$\mathcal{R}$	mass transfer between two phases
$\boldsymbol{R}$	residual matrix for ILU
$\boldsymbol{r}$	residual vector
t	time, s
t	temperature, K
U	velocity component, m/s
$\boldsymbol{U}$	sparse upper triangular matrix for ILU
u	x-velocity component, m/s
V	volume, m^3
x, y, z	coordinate directions
$\boldsymbol{x}$	solution vector of a linear system
X	species mass fraction

Greek symbols

ε	volume fraction
κ	condition number of a matrix
ρ	density, kg/m^3
τ	stress tensor
ω	relation factor for pressure correction equation
$\boldsymbol{\omega}$	relaxation parameter for the SOR preconditioner
γ	heat transfer coefficient
Φ	general representation of a variable being solved

Subscripts

c	close packed regions
e	east control volume face comparative to p
E	East control volume central point comparative to P
g	fluid-phase
i, j, k	vector direction components
$\boldsymbol{j}$	iteration number
m	solids phase
n	general phase number (fluid or solid) or species number
nb	neighbor control volume faces or central points
P	control volume central point at which a scalar variable is being solved

p	control volume face at which velocity component is being solved
$\boldsymbol{p}$	search direction vector
w	west control volume face comparative to p
W	west control volume central point comparative to P
∞	inlet air stream conditions
$'$	correction term for pressure or velocity
$*$	intermediate term for pressure or velocity or upwind biased estimate
$\sim$	normalized value

Superscripts

T	transpose of a matrix
-1	inverse of a matrix

ACKNOWLEDGEMENTS

I would first like to thank my academic advisor for the past six years, Dr. Gautham Krishnamoorthy. As a freshman in my undergraduate career at UND, he helped make my transition from high school to college less overwhelming. He also encouraged me to apply for the combined BS-MS degree program, which is the reason I pursued my master's degree and have completed this thesis! I am very thankful that I have been able to come to him for any questions I have had, whether it be for class, research, or other personal decisions. Without his guidance and support, completion of this research would not have been possible.

I must also thank my professors and committee members Dr. Frank Bowman and Dr. Michael Mann. They have both been very helpful in reviewing my thesis, setting up my defense date, and offering any advice.

I would not have made it to this point today without my fellow undergraduate and graduate students at UND, as well as the Chemical Engineering department faculty. I have learned a great deal from my peers and professors throughout all of the courses I have taken. Lastly, I would like to thank my parents for continually motivating and encouraging me throughout my education, as well as my brothers for their wonderful support. Without the love and help of my family, none of this would have been possible!

ABSTRACT

A computational bottleneck during the solution to multiphase formulations of the incompressible Navier-Stokes equations is often during the implicit solution of the pressure-correction equation that results from operator-splitting methods. Since density is a coefficient in the pressure-correction equation, large variations or discontinuities among the phase densities greatly increase the condition number of the pressure-correction matrix and impede the convergence of iterative methods employed in its solution. To alleviate this shortcoming, the open-source multiphase code MFiX is interfaced with the linear solver library PETSc. Through an appropriate mapping of matrix and vector data structures between the two software, the access to a suite of robust, scalable, solver options in PETSc is obtained.

Verification of the implementation of MFiX-PETSc is demonstrated through predictions that are identical to those obtained from MFiX's native solvers for a simple heat conduction case with a well-known solution. After verifying the framework, several cases were tested with MFiX-PETSc to analyze the performance of various solver and preconditioner combinations.

For a low Reynolds number, flow over a cylinder case, applying right-side Block Jacobi preconditioning to the BiCGSTAB iterative solver in MFiX-PETSc was 28-40% faster than MFiX's native solver at the finest mesh resolution. Similarly, the left-side Block Jacobi

preconditioner in MFiX-PETSc was 27–46% faster for the same fine meshing. Further assessments of these preconditioning options were then made for a fluidized bed problem involving different bed geometries, convergence tolerances, material densities, and inlet velocities.

For a three-dimensional geometry with uniform meshing, native MFiX was faster than MFiX-PETSc for each simulation. The difference in speed was minimized when a low density fluidization material (polypropylene) was used along with a higher order discretization scheme. With these settings, MFiX-PETSc was only 2-6% slower than native MFiX when right-side Block Jacobi preconditioning was employed. The fluidized bed was then represented by a two-dimensional geometry with fine meshing towards the center. When this bed was filled with glass beads, right-side Block Jacobi was 28% faster than MFiX's native solver, which was the largest speedup encountered throughout this 2D case.

CHAPTER 1

INTRODUCTION

1.1 Motivation

Although multiphase flows have been encountered in industry for decades, there still exists a lack of understanding in the effect of hydrodynamics for these flows. The recent advancements of CFD modeling have helped researchers better comprehend the relationship between hydrodynamics and other phenomena (such as reactions). However, to have a more significant impact in the design of multiphase flow technologies, the computational efficiency and scalability of CFD software must improve.

The inherently transient nature of most multiphase flows in conjunction with the large density variations among the phases, make them very difficult to simulate. For instance, in gas-solid contactors, the phase densities may vary by more than a factor of 1000. In the Two-Fluid Model (TFM) framework for simulating multiphase flows, the fluid and solids phases are treated as interpenetrating continua, for which all phases are represented by the Navier-Stokes equations. The coupling between these different phases is achieved through an appropriate modeling of the interaction and source terms in the respective phase equations [1].

Solution of the incompressible Navier-Stokes equations for the different phases is then undertaken using a semi-implicit method where a pressure-correction equation is formulated implicitly, requiring the solution of a linear system at each time step. The pressure-correction equation takes the form of a discrete Poisson equation with discontinuous coefficients [2]. This means that the matrix of the linear system representing the pressure equation should be symmetric and diagonally dominant. Although the operator is typically symmetric, the solution to this equation consumes the bulk of the computational time in multiphase simulations. This is because density is a coefficient in the pressure-correction equation and large variations or discontinuities among the phase densities greatly increase the condition number of the pressure-correction matrix and impede the convergence of iterative methods employed in its solution [3].

The computational bottleneck associated with the solution to the pressure-correction equation for the incompressible Navier-Stokes Equations has long been recognized. In single-phase fluid simulations, this bottleneck has been overcome by interfacing Computational Fluid Dynamic (CFD) codes with linear solver libraries in PETSc [4] and HYPRE [5] to achieve good scaling performance on a large number of cores [6]. The **P**ortable, **E**xtensible **T**oolkit for **S**cientific **C**omputation (PETSc) is a suite of data structures and routines which can be used in largely parallel environments to obtain solutions to systems modeled with partial differential equations. The PETSc [4] library is particularly interesting since it allows for the transparent use of various Krylov subspace solvers and preconditioner options in large-scale parallel environments without the need to write specialized code to access them. PETSc could have a promising application in solving the pressure-correction equation associated with multiphase flows to potentially reduce the computational cost associated with modeling these types of systems.

1.2 Objectives

This work was focused on achieving two mains objectives. The first objective was to build a robust, well-abstracted, interface to the PETSc linear solver library from the CFD code MFiX. **M**ultiphase **F**low with **I**nterphase e**X**change (MFiX) is an open-source software developed by the National Energy Technology Laboratory (NETL) to model fluid-solid flows. The MFiX-PETSc interface was tested by carrying out a simple heat conduction problem for which the results could be validated by comparing against an established analytical solution.

The second objective was to gain an understanding of the most effective solvers and preconditioners offered in PETSc for resolving the pressure-correction equation. The framework was tested on both single- and multi-phase transient problems. Pressure results were compared against published experimental data to analyze the accuracy achieved with these solver options. Furthermore, the computation time and average iterations were compared with native MFiX to gain insight into the speed and efficiency of each preconditioner/solver combination

The overall goal of these objectives was to create a framework between MFiX and PETSc that can be efficiently scaled to a parallel framework, where it will be most effective, with future work. Then this research aimed to create an understanding of best solver and preconditioning practices to resolve multiphase flows in order to guide this future work as well.

1.3 Thesis Outline

The rest of this thesis is structured as follows:

Chapter 2: Relevant background information for understanding the formation and solution of equations to model multiphase flows, including the conservation equations, equation discretization, solution procedure, linear algebra, and iterative methods.

Chapter 3: A brief explanation of building the software that was necessary in carrying out this study.

Chapter 4: Verification of the MFiX-PETSc interface with a simple, steady-state heat conduction problem.

Chapter 5: An investigation of pressure coefficient, CPU timing, and iteration results obtained with MFiX-PETSc for a problem characterized by single-phase flow over a cylinder.

Chapter 6: An investigation of pressure power spectra, CPU timing, and iteration results obtained with MFiX-PETSc for simulations of a 3D fluidized bed filled with either glass beads or polypropylene beads.

Chapter 7: An investigation of pressure power spectra, CPU timing, iteration, and time-step results obtained with MFiX-PETSc for simulations of a 2D fluidized bed filled with either glass beads or polypropylene beads.

Chapter 8: The overall conclusions of this thesis are discussed, as well as suggestions for future work.

Appendix: A list of the specific MFiX input files (mfix.dat) that were used to carry out the problems presented in Chapters 4 - 7.

CHAPTER 2

BACKGROUND

2.1 Multiphase Flows

A multiphase flow is defined as the simultaneous flow of materials with multiple phases or components. Flows with these properties have a wide application in industry, as they are found in slurries, cavitating flows, aerosols, debris flows, and fluidized beds, along with others. Due to this, nearly every process unit operation will have to handle multiphase flows, whether it's the flow of a slurry through piping or the gasification of coal particles in a reactor. Thus, being able to predict the fluid flow behavior of these multiphase processes is crucial to process efficiency and effectiveness [7].

One of the strategies used to model and predict these flows is a computational approach. **M**ultiphase **F**low with **I**nterface e**X**change (MFiX) is an open-source code developed with a purpose of computationally understanding the hydrodynamics, heat transfer, and chemical reactions of multiphase flows. MFiX currently offers Eulerian-Eulerian and Eulerian-Lagrangian approaches for solving fluid-solid flow problems. The Eulerian-Eulerian methodology, otherwise known as the Two-Fluid Model (TFM), describes both the fluid and solid phases as interpenetrating continua represented by the Navier-Stokes equations. Contrarily, the Eulerian-Lagrangian approach models only the fluid phase as a continuum while the position and trajectory of each solids particle is tracked [8].

Using the Lagrangian solids model does result in fewer and much simpler closures in comparison to the TFM which is why this model is considered to comprise of more certainty. However, considering that systems can contain millions or even billions of particles, it is easy to see how this type of approach can become computationally intensive, which is why it is generally limited to small-scale devices [8]. Due to this, the Two-Fluid Model is more commonly used for these types of applications, especially for the pilot- or industrial-scale systems.

2.2 Two-Fluid Model

The TFM represents both solids and fluids (gas or liquid) as interpenetrating continua with one fluid phase and one or more solids phases. Solids of one phase are assumed to move collectively, which is represented by the motion of a continuum. Particles with different sizes, densities, or compositions may be designated as a separate solids phase depending on the goals of the computational study. Figure 2-1 shows how a fluid-solids system can be represented as a two-phase or multiple-phase system using the MFiX-TFM [8].

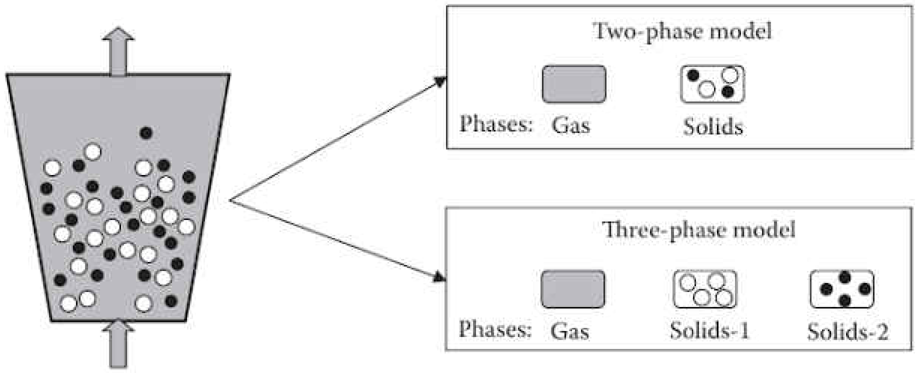


Figure 2-1. A multiphase flow system with two solid particle types can be represented as a two-phase system or a three-phase system using the MFiX-TFM [8].

Describing solids as a continuum avoids having to track the motion and collisions of each individual particle, which significantly reduces the computational cost. Consequently, this approach decreases the simulation resolution and constitutive equations must be included such as gas-solids drag to compensate. Listed below are the basic conservation equations implemented in the MFiX-TFM.

2.3 Partial Differential Equations

2.3.1 Conservation of Mass

The conservation of mass for the fluid and solids phases is represented respectively as follows [8]:

$$\frac{\partial}{\partial t}\varepsilon_g\rho_g + \frac{\partial}{\partial x_j}\left(\varepsilon_g\rho_g U_{gj}\right) = \sum_{n=1}^{N_g} R_{gn} \qquad (2.1)$$

$$\frac{\partial}{\partial t}\varepsilon_m\rho_m + \frac{\partial}{\partial x_j}\left(\varepsilon_m\rho_m U_{mj}\right) = \sum_{n=1}^{N_m} R_{mn} \qquad (2.2)$$

where ε_g is the fluid volume fraction, ε_m is the volume fraction of the m^{th} solids phase, ρ_g is the fluid-phase density, ρ_m is the density of the m^{th} solids phase, U_{gj} is the j^{th} velocity component of the fluid-phase, and U_{mj} is the j^{th} velocity component of the m^{th} solids phase. The right-hand term denotes interphase mass transfer due to chemical reactions or physical phenomena.

The fluid density (ρ_g) can be set to a constant value, representing an incompressible fluid, or can change according to the ideal gas law. The solids densities can also remain constant or vary as chemical reactions occur.

2.3.2 Conservation of Momentum

For fluid phases, the momentum balance is [8]:

$$\frac{\partial}{\partial t}\left(\varepsilon_g \rho_g U_{gi}\right) + \frac{\partial}{\partial x_j}\left(\varepsilon_g \rho_g U_{gj} U_{gi}\right) = \tag{2.3}$$

$$-\varepsilon_g \frac{\partial P_g}{\partial x_i} + \frac{\partial \tau_{gij}}{\partial x_j} \sum_{m=1}^{M} \left[\mathcal{R}_{mg} U_{mi} - \mathcal{R}_{gm} U_{gi} - I_{gmi}\right] + f_{gi} + \varepsilon_g \rho_g g_i$$

where P_g is the fluid-phase pressure, τ_{gij} is the stress tensor of the fluid-phase, $\mathcal{R}_{gm}$ is mass transfer from the fluid-phase to the m^{th} solids phase, $\mathcal{R}_{mg}$ is mass transfer from the m^{th} solids phase to the fluid-phase, I_{gmi} represents momentum transfer between the fluid and the m^{th} solids phase caused by interphase forces, f_{gi} is fluid flow resistance due to porous media, and g_i is acceleration due to gravity.

The momentum balance for the solids phase is represented similarly as [8]:

$$\frac{\partial}{\partial t}\left(\varepsilon_m \rho_m U_{mi}\right) + \frac{\partial}{\partial x_j}\left(\varepsilon_m \rho_m U_{mj} U_{mi}\right) = \tag{2.4}$$

$$-\varepsilon_m \frac{\partial P_g}{\partial x_i} - \varepsilon_m \frac{\partial P_c}{\partial x_i} + \frac{\partial \tau_{mij}}{\partial x_i} \sum_{m=1}^{M} \left[\mathcal{R}_{lm} U_{li} - \mathcal{R}_{ml} U_{mi} - I_{mli}\right] + \varepsilon_{gm} \rho_m g_i$$

where P_c is solids-phase pressure in close packed regions, τ_{mij} is the stress tensor of the m^{th} solids phase, $\mathcal{R}_{lm}$ is mass transfer from the l^{th} phase to the m^{th} phase, $\mathcal{R}_{ml}$ is mass transfer from the m^{th} phase to the l^{th} phase, and I_{mli} represents momentum transfer between the m^{th} phase and the l^{th}phase caused by interphase forces.

2.3.3 Conservation of Species Mass

Multiple chemical species can make up the fluid and solids phases. For the fluid-phase, the mass conservation of each species is represented as [8]:

$$\frac{\partial}{\partial t} \varepsilon_g \rho_g X_{gn} + \frac{\partial}{\partial x_j}\left(\varepsilon_g \rho_g U_{gj} X_{gn}\right) = \frac{\partial}{\partial x_j}\left(\mathcal{D}_{gn} \frac{\partial X_{gn}}{\partial x_j}\right) + R_{gn} \tag{2.5}$$

where X_{gn} is the mass fraction of the n^{th} chemical species in the fluid-phase, $\mathcal{D}_{gn}$ is diffusion coefficient of the n^{th} chemical species in the fluid-phase, and R_{gn} is the rate of formation or destruction of the n^{th} chemical species in the fluid-phase. Similarly, the mass conservation of each species in the solids phases is [8]:

$$\frac{\partial}{\partial t}\varepsilon_m\rho_m X_{mn} + \frac{\partial}{\partial x_j}\left(\varepsilon_m\rho_m U_{mj} X_{mn}\right) = R_{mn} \qquad (2.6)$$

where X_{mn} is the mass fraction of the n^{th} chemical species in the n^{th} solids phase and R_{mn} is the rate of formation or destruction of the n^{th} chemical species in the n^{th} solids phase.

2.3.4 Conservation of Energy

The energy conservation equations in MFiX are solved in terms of temperature. The conservation of energy for the fluid-phase is [8]:

$$\varepsilon_g\rho_g C_{pg}\left[\frac{\partial T_g}{\partial t} + U_{gj}\frac{\partial T_g}{x_j}\right] = -\frac{\partial q_{gj}}{\partial x_j} + \sum_{m=1}^{M}\gamma_{gm}\left(T_m - T_g\right) + \gamma_{R_g}\left(T_{R_g}^4 - T_g^4\right) - H_g \qquad (2.7)$$

where T_g is the temperature of the fluid-phase, C_{pg} is the specific heat of the fluid-phase, q_{gj} is the conductive heat flux in the fluid-phase, γ_{gm} is the heat transfer coefficient between the fluid and the m^{th} solids phase, T_m is the temperature of the m^{th} solids phase, γ_{R_g} is the radiative heat transfer coefficient for the fluid-phase, T_{R_g} is the background temperature of the fluid-phase in a radiation model, and H_g is the total rate of enthalpy change in the fluid-phase due to chemical reactions and phase changes. The solids-phase energy conservation equation is represented as [8]:

$$\varepsilon_m\rho_m C_{pm}\left[\frac{\partial T_m}{\partial t} + U_{mj}\frac{\partial T_m}{x_j}\right] = -\frac{\partial q_{mj}}{\partial x_j} - \gamma_{gm}\left(T_m - T_g\right) + \gamma_{R_m}\left(T_{R_m}^4 - T_m^4\right) - H_m \qquad (2.8)$$

for which C_{pm} is the specific heat of the m^{th} solids phase, q_{mj} is the conductive heat flux in the m^{th} solids phase, γ_{R_m} is the radiative heat transfer coefficient for the m^{th} solids phase,

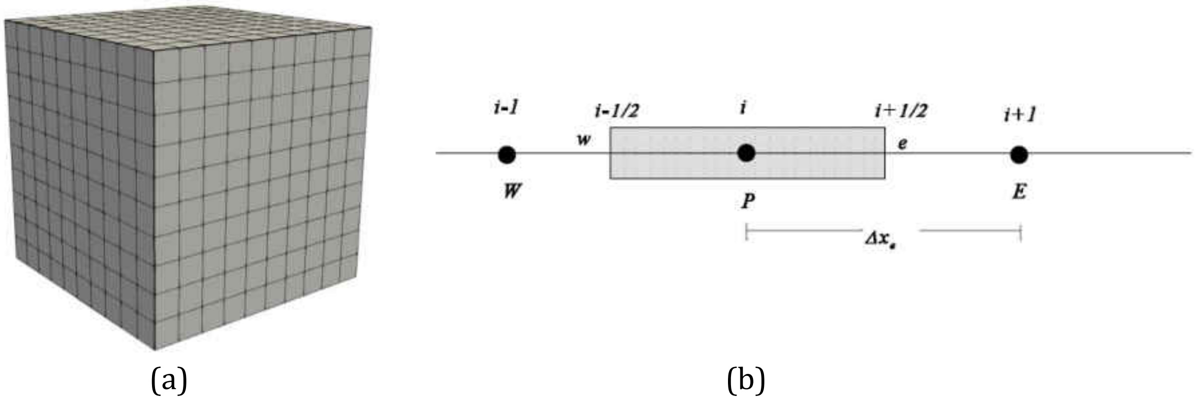
T_{R_m} is the background temperature of the m^{th} solids phase in a radiation model, and H_m is the total rate of enthalpy change in the m^{th} solids phase due to chemical reactions and phase changes.

2.3.5 Discretization of Convection-Diffusion Terms

The conservation equations include a combination of convection and diffusion terms of the form [2]:

$$\rho u \frac{\partial \phi}{\partial x} - \frac{\partial}{\partial x}\left(\Gamma \frac{\partial \phi}{\partial x}\right). \tag{2.9}$$

The way in which these terms are discretized can have a significant impact on the stability and accuracy of the numerical method. In CFD, a finite volume method is typically used to discretize these convection and diffusion terms. In this technique, the conservation laws are enforced within a small control volume. All of these small control volumes grouped together is defined as the computational mesh. Figure 2-2 (a) shows an example of a cube broken down into a computational mesh with 10x10x10 control volumes, while Figure 2-2 (b) defines a single control volume and its node locations in the x-direction.



(a) (b)

Figure 2-2. (a) A cube broken down into a 10x10x10 computational mesh, and (b) a single control volume defined by its node locations in the x-direction [2].

Integration of this convection-diffusion term over a control volume in the x-direction gives us [2]:

$$\int \left[\rho u \frac{\partial \phi}{\partial x} - \frac{\partial}{\partial x}\left(\Gamma \frac{\partial \phi}{\partial x}\right)\right] dV = \left[\rho u \phi_e - \left(\Gamma \frac{\partial \phi}{\partial x}\right)_e\right] A_e - \left[\rho u \phi_w - \left(\Gamma \frac{\partial \phi}{\partial x}\right)_w\right] A_w. \qquad (2.10)$$

The convection and diffusion terms are then accounted for in separate substeps. Solving for the diffusive fluxes at the control volume faces is straightforward, and can be calculated with a second-order accuracy as follows [2]:

$$\left(\Gamma \frac{\partial \phi}{\partial x}\right)_e = \Gamma_e \frac{\phi_E - \phi_P}{\delta x_e} + O(\delta x^2). \qquad (2.11)$$

Discretizing the convection term requires interpolating the face-centered velocity terms to their cell-centered values, and there are several methods to do so. The stability and accuracy of the numerical method can be strongly determined by the discretization strategy employed. This work incorporates one first-order scheme, and two higher-order schemes to discretize this convection term. In the first-order upwind (F.O.U.P.) scheme, face-centered velocity values are directly interpolated to their cell-centered values as [2]:

$$\phi_e = \begin{cases} \phi_P, & u \geq 0 \\ \phi_E, & u < 0 \end{cases} \qquad (2.12)$$

When flows are transient, multi-dimensional, or contain strong sources, first-order schemes may not provide enough accuracy. Higher-order discretization schemes for convection can help increase accuracy, but they can also create issues with overshoots and undershoots near discontinuities, known as oscillations. This can create problems with convergence and physically unrealistic intermediate solutions [2].

Total variation diminishing (TVD) schemes have been developed to resolve discontinuities without producing these oscillations. These techniques employ a limiter which bounds the value of ϕ. This limiter is defined using the notations for the node locations portrayed in Figure 2-3, which are based on the flow direction. The notation D represents downwind, U represents Upwind, C is the central point of the control volume, and f is the face of the control volume.

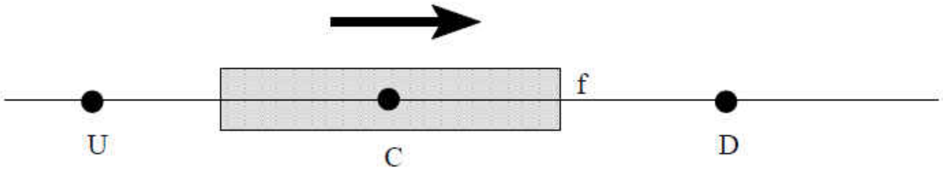


Figure 2-3. Notation used for node locations in TVD schemes, based on flow direction [2].

The limiter is expressed as a function of the normalized value of ϕ, which is defined as [2]:

$$\widetilde{\phi} = \frac{\phi - \phi_U}{\phi_D - \phi_U}. \qquad (2.13)$$

TVD schemes bound ϕ with this limiter when the variation in ϕ is monotonic, which occurs when $0 \leq \widetilde{\phi}_C \leq 1$. The overall goal is to calculate values at the control volume face (ϕ_f) based on the specific bounds that have been employed. There are four conditions which define how the limiter bounds ϕ_f, which are described by Syamlal [2].

A down-wind factor formulation for discretization, proposed by Leonard and Mokhtari [9], has been adopted into several existing codes due to its ability to retain the traditional septa-diagonal matrix structure in linear systems. This formulation applies the following steps [2]:

1. Calculate a high-order, multidimensional, upwind biased estimate of ϕ_f^*.
2. Calculate a preliminary downwind weighting factor (dwf^*):

$$dwf^* = \frac{\phi_f - \phi_C}{\phi_D - \phi_C} = \frac{\widetilde{\phi_f} - \widetilde{\phi_C}}{1 - \widetilde{\phi_C}}. \qquad (2.14)$$

3. Obtain dwf by limiting dwf^* to the monoatomic region.
4. Calculate the new estimate of ϕ_f as:

$$\phi_f = dwf\,\phi_D + (1 - dwf)\phi_C. \qquad (2.15)$$

The TVD schemes differ by how they calculate their downwind weighting factor in step 3. For all schemes, dwf is equal to 0 if $\widetilde{\phi_C}$ is less than 0 or greater than 1. Inside of these bounds however, if θ is a factor calculated as [2]:

$$\theta = \frac{\widetilde{\phi_C}}{1 - \widetilde{\phi_C}} \qquad (2.16)$$

then the downwind factor is equal to $\frac{1}{2}max[0, min(1,2\theta), min(2, \theta\,)]$ for the Superbee discretization scheme and $\widetilde{\phi_C}$ for the van Leer scheme.

2.3.6 MFiX Solution Procedure

MFiX uses a semi-implicit scheme, with automatic time-stepping to sequentially solve the discretized transport equations. The first step of the solution procedure involves discretizing the governing equations based on the schemes described in section 2.3.5. The finite volume method, which has been previously introduced, is applied with a staggered grid to discretize the governing equations. Using this approach, scalar values (i.e. fluid-pressure) are computed at the center of the control volume whereas velocity components are calculated along the faces of the control volume. Figure 2-4 shows how control volume centers and faces are defined for a two-dimensional grid in order to solve for scalar and vector values. This concept can be extended to a three-dimensional grid by envisioning that the center of a control volume coming out of the paper, adjacent to P, is labeled T for top and a control volume going into the paper is labeled B for bottom. Furthermore, the face between T and P is the top face t, and the face between B and P is the bottom face b.

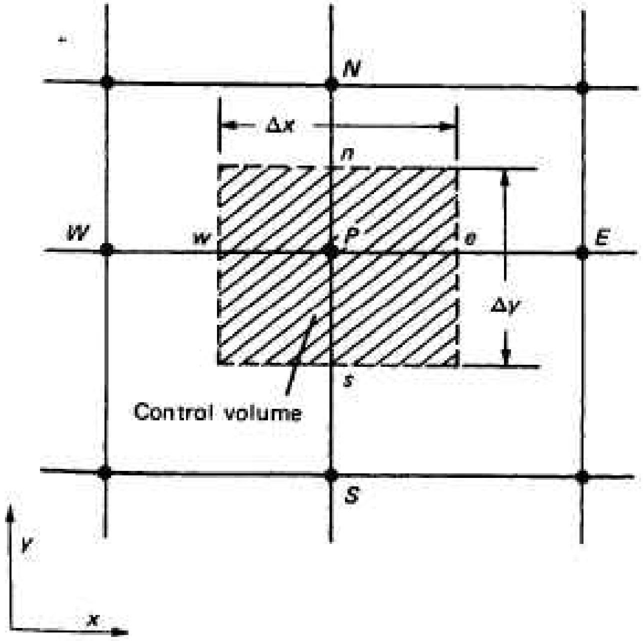


Figure 2-4. A two-dimensional representation of a control volume on a staggered grid which is used in the finite volume method for discretizing the transport equations [10].

Discretization of scalar transport equations for all phases can be represented as [2]:

$$(a_n)_P(\phi_n)_P = \sum_{nb}(a_n)_{nb}(\phi_n)_{nb} + b_n + \Delta V \sum_{l=0}^{M} F_{l,n}\,[(\phi_l)_P - (\phi_n)_P] \qquad (2.17)$$

for which coefficient a contains flow properties from the discretized equations, ϕ is a given scalar value such as temperature, b is a source term, and $F_{l,n}$ is the interface transfer coefficient between phases l and n. Subscript n represents the phase undergoing calculation, subscript P is the central point of the scalar quantity undergoing calculation, and subscript nb denotes its neighbor central points (E, W, N, S, T, and B).

Discretization of the momentum equations for the fluid and solids phases results in similar expressions. The discretized x-momentum equations for the gas and solids phases respectively are [2]:

$$\left(a_g\right)_e\left(u_g\right)_e = \textstyle\sum_{nb}\left(a_g\right)_{nb}\left(u_g\right)_{nb} + b_g \qquad (2.18)$$

$$-A_e\left(\varepsilon_g\right)_P\left[\left(P_g\right)_P - \left(P_g\right)_E\right] + \Delta V\sum_{l}^{M} F_{gl}\left[(u_l)_e - \left(u_g\right)_e\right]$$

$$(a_m)_e(u_m)_e = \textstyle\sum_{nb}(a_m)_{nb}(u_m)_{nb} + b_m - A_e(\varepsilon_m)_P\left[\left(P_g\right)_P - \left(P_g\right)_E\right] \qquad (2.19)$$

$$-A_e[(P_m)_P - (P_m)_E] + \Delta V\sum_{l}^{M} F_{lm}\left[(u_l)_e - (u_m)_e\right]$$

where a_g is a coefficient similar to Equation (2.17) that contains flow properties for the fluid-phase, a_m is a coefficient that contains flow properties for the m^{th} solids phase, u_g is the x-velocity of the fluid-phase, u_m ix the x-velocity of the m^{th} solids phase, b_g is the source term for the fluid-phase, and b_m is the source term for the m^{th} solids phase. Since velocity is a vector, subscript e describes face between control volumes P and E, subscript nb denotes neighbor faces. The y- and z-momentum equations are discretized in the same fashion.

After the transport equations are discretized, they are rearranged to yield a system of linear equations with large, sparse, septa-diagonal matrices that must be solved iteratively. When flow is incompressible, challenges arise in computing the fluid flow field. As shown in Equations (2.1) through (2.4), The velocity field can be computed from the momentum equations; however, the pressure field, which shows up in the momentum equation, cannot be solved directly from the continuity equation [11]. This results in a strong, implicit coupling between the pressure and velocity fields. In MFiX, the solids-phase pressure is resolved with a volume fraction correction equation and the fluid-phase pressure field is resolved with a fluid-pressure correction equation [8].

The SIMPLE algorithm is an operator-splitting numerical procedure that is widely employed in CFD to solve the discretized Navier-Stokes equations for incompressible systems. MFiX employs an extended version of the SIMPLE algorithm developed by Patankar [10] to account for multiphase systems. An outline of the steps followed during each time step is represented as follows [8]:

1. The time step starts. Physical properties and exchange coefficients are calculated.
2. Momentum equations are solved to obtain velocity fields using pressure and volume fractions from previous iteration.
3. Continuity equations for the solids phase are solved.
4. The gas-phase volume fraction is computed from the determined solids volume fraction.
5. The pressure of the solids phase is calculated using the solids volume fraction.
6. The face centered densities are computed.
7. The fluid-pressure correction equation is solved and the corrections are used to update the gas pressure and velocity fields.
8. Material densities and face-centered mass fluxes are resolved.
9. The scalar equations are solved (e.g. species mass, energy).

10. The normalized residuals are computed and used to assess convergence. If the convergence criterion is met, the time-step is advanced, otherwise another iteration is performed (back to step 1).

The number of times these ten steps are repeated represents the number of outer iterations per time-step.

2.3.7 Pressure Correction Equation

As described in section 2.3.6, the first step of each iteration involves solving the discretized momentum equations using the pressure field and volume fractions from the previous iteration. These intermediate values for pressure and velocity are represented by P* and u* respectively. The relationship between the intermediate value (ϕ*) and the actual value (ϕ) for these parameters is denoted as ϕ = ϕ* + ϕ', where ϕ' is the correction value.

Derivation of the fluid-pressure correction equation first requires replacing pressure (P) and velocity (u) terms in the discretized fluid-phase momentum equation with intermediate pressure (P*) and velocity (u*) terms. Then, the P* = P – P' and u* = u – u' expressions are substituted into this equation. The original discretized gas momentum equation, with actual pressure and velocity values, is subtracted to yield an expression only containing velocity (u') and pressure (P') correction terms. After several simplifications and rearrangements are made, the resultant fluid-pressure correction equation becomes [2]:

$$\left(a_g\right)_P\left(P_g'\right)_P = \sum_{nb}\left(a_g\right)_{nb}\left(P_g'\right)_{nb} + b_g. \tag{2.20}$$

The pressure correction terms calculated using Equation (2.20) are then used to calculate the actual gas-phase pressure [2]:

$$\left(P_g\right)_P = \left(P_g^*\right) + \omega_{P_g}\left(P_g'\right)_P \tag{2.21}$$

for which ω_{P_g} is a relation factor. This pressure correction is also used to update the gas-phase velocity [2]:

$$\left(u_g\right)_e = \left(u_g^*\right)_e + \left(d_g\right)_e \left[\left(P_g'\right)_P - \left(P_g'\right)_E\right] \quad (2.22)$$

where d_g is an interphase mass transfer factor.

2.4 Numerical Methods

2.4.1 Linear Algebra

Understanding linear algebra theory and operations is important when trying to learn and apply different numerical methods to solve the discretized transport equations in CFD. The objective of this section is to introduce the basic linear algebra concepts that will be used throughout section 2.4 to describe various iterative techniques.

Matrices and vectors form the foundation of linear algebra. A ***vector*** can be defined as a one-dimensional sequence of elements. An important operation with vectors is known as the ***dot product***, or the ***Euclidean inner product***. If $\boldsymbol{x}$ and $\boldsymbol{y}$ are vectors of the real coordinate space of n-dimensions ($\mathbb{R}^n$), then the dot product is commonly denoted as $\boldsymbol{x} \cdot \boldsymbol{y}$ or $(\boldsymbol{x}, \boldsymbol{y})$, and can be calculated as:

$$\boldsymbol{x} \cdot \boldsymbol{y} \text{ or } (\boldsymbol{x}, \boldsymbol{y}) = \sum_{i=1}^{n} x_i y_i = x_1 y_1 + x_2 y_2 + \cdots + x_n y_n. \quad (2.23)$$

By this definition, the dot product of a vector with itself can be expressed as:

$$\boldsymbol{x} \cdot \boldsymbol{x} = \sum_{i=1}^{n} x_i^2 = x_1^2 + x_2^2 + \cdots + x_n^2 \quad (2.24)$$

which is also equal to the length of the vector squared. Furthermore, the square root can be taken to obtain the ***vector length*** or ***Euclidean norm***:

$$\|\boldsymbol{x}\| = \sqrt{\boldsymbol{x} \cdot \boldsymbol{x}} = \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2}. \quad (2.25)$$

The Euclidean norm is also referred to as the ***$\boldsymbol{l^2}$-norm of a vector***.

A ***vector space*** contains a set V of vectors that can be added together or multiplied by a scalar. For all vectors $\boldsymbol{u}$**,** $\boldsymbol{v}$**,** and $\boldsymbol{w}$ in V, the vector addition operation must adhere to the following axioms:

1. Closure: $\boldsymbol{u}+\boldsymbol{v}$ also belongs to V
2. Communicative Law: $\boldsymbol{u}+\boldsymbol{v}=\boldsymbol{v}+\boldsymbol{u}$
3. Associative Law: $\boldsymbol{u}+(\boldsymbol{v}+\boldsymbol{w})=(\boldsymbol{u}+\boldsymbol{v})+\boldsymbol{w}$
4. Additive Identity: There is a zero vector $\mathbf{0}$ in V such that $\mathbf{0}+\boldsymbol{v}=\boldsymbol{v}$ and $\boldsymbol{v}+\mathbf{0}=\boldsymbol{v}$
5. Additive Inverses: For each vector $\boldsymbol{u}$ in V, there is an additive inverse denoted $-\boldsymbol{u}$ such that $\boldsymbol{u}+(-\boldsymbol{u})=\mathbf{0}$

Furthermore, multiplication of vectors with scalars c and d must satisfy these axioms:

6. Closure: $c\cdot\boldsymbol{v}$ also belongs to V
7. Distributive Law: $c\cdot(\boldsymbol{u}+\boldsymbol{v})=c\cdot\boldsymbol{u}+c\cdot\boldsymbol{v}$
8. Distributive Law: $(c+d)\cdot\boldsymbol{v}=c\cdot\boldsymbol{v}+d\cdot\boldsymbol{v}$
9. Associative Law: $c\cdot(d\cdot\boldsymbol{v})=(cd)\cdot\boldsymbol{v}$
10. Unitary Law: $1\cdot\boldsymbol{v}=\boldsymbol{v}$

A ***subset*** is a set of vectors from a vector space that do not need to follow any conditions. A ***subspace*** on the other hand is a set of vectors from a vector space that do need to adhere to certain conditions. A subspace is always a subset, but a subset is not necessarily a subspace. If W is a subset of V, then W is also a subspace of V if:

1. W is nonempty, meaning that at least the zero vector belongs to W.

2. If $\boldsymbol{u}$ and $\boldsymbol{v}$ are in W, then $\boldsymbol{u} + \boldsymbol{v}$ is also in W.
3. If $\boldsymbol{v}$ is a vector in W, and c is any real number, then $c \cdot \boldsymbol{v}$ is also in W.

Overall, a subspace can be thought of as a vector space contained within another vector space.

A set of vectors is said to be ***linearly independent*** when none of the vectors can be defined as a linear combination of the others. Given vector space V, a ***basis*** is a subset of V that contains a set of linearly independent vectors of V. A vector space can have various sets of basis vectors, but each must have the same number of elements which is known as the dimension of the vector space. In general, a vector space can be thought of as a linear combination of the basis vectors.

The set of all linear combinations of a vector set $\boldsymbol{v_1}, \boldsymbol{v_2}, \dots, \boldsymbol{v_n}$ is known as the ***span*** of the vectors, denoted $span\{\boldsymbol{v_1}, \boldsymbol{v_2}, \dots, \boldsymbol{v_n}\}$. For a subset S of vector space V, $span\{S\}$ is a subspace of V.

A ***matrix*** is defined as a rectangular array of elements arranged into rows and columns. Matrices can be denoted with a capital $\boldsymbol{A}$ and subscripts ij, where i represents the row number, and j represents the column number. If matrix $\boldsymbol{A}_{ij}$ is an $i \times j$ matrix, then:

$$\boldsymbol{A}_{ij} = \begin{pmatrix} a_{11} & \cdots & a_{1j} \\ \vdots & \ddots & \vdots \\ a_{i1} & \cdots & a_{ij} \end{pmatrix} \qquad (2.26)$$

where a is each element of the matrix. When the number of rows is equal to the number of columns, it is called a ***square matrix***, which can be denoted by $\boldsymbol{A}_{ii}$.

Matrices can be defined by their structure in several ways, which can be useful for computational purposes. The elements of a ***diagonal matrix*** are 0 when $i \neq j$. These matrices can be represented as $\boldsymbol{A} = diag(a_{11}, a_{22}, \dots, a_{nn})$. For an ***upper triangular matrix***,

$a_{ij} = 0$ below the matrix diagonal, or in other words when $i > j$ using the notation in Equation (2.26). In opposition, $a_{ij} = 0$ above the matrix diagonal $(i < j)$ for a ***lower triangular matrix***. A ***block diagonal matrix*** is a generalization for a diagonal matrix. It replaces each diagonal element with a smaller matrix, and is represented as $\boldsymbol{A} = diag(\boldsymbol{A}_{11}, \boldsymbol{A}_{22}, \dots, \boldsymbol{A}_{nn})$.

The Navier-Stokes equations are typically discretized in a way to form ***tridiagonal matrices*** (1-D problems), ***pentadiagonal matrices*** (2-D problems), and ***septadiagonal matrices*** (3-D problems). These types of matrices can be defined as:

- Tridiagonal Matrix (1-D problem)
  - $a_{ij} = 0$ for $j \neq i$ or $|j - i| > 1$
- Pentadiagonal Matrix (2-D problem with an $m \times n$ mesh)
  - $a_{ij} = 0$ for $j \neq i$ or $|j - i| \neq 1$ or $|j - i| \neq n$
- Septadiagonal Matrix (3-D problem with an $m \times n \times o$ mesh)
  - $a_{ij} = 0$ for $j \neq i$ or $|j - i| \neq 1$ or $|j - i| \neq n$ or $|j - i| \neq m \times n$

All three of these matrix types can be considered ***diagonally dominant*** matrices.

Matrix multiplication involves the dot product of rows of one matrix with columns of a second matrix. If $\boldsymbol{A}$ is a 2×3 matrix, and $\boldsymbol{B}$ is a 3×2 matrix, these two can be multiplied to yield a 2×2 product vector $\boldsymbol{C}$. The first step is to take the dot product between the first row of $\boldsymbol{A}$ and the first column of $\boldsymbol{B}$ and place it into the product matrix at $\boldsymbol{C}_{11}$. Furthermore, the dot product of the second row of $\boldsymbol{A}$ and the first column of $\boldsymbol{B}$ can be placed into $\boldsymbol{C}_{21}$, and so on. An example solution to matrix multiplication is shown in Figure 2-5.

$$\boldsymbol{B} = \begin{pmatrix} 1 & 2 \\ 1 & 4 \\ 5 & 2 \end{pmatrix}$$

$$\boldsymbol{A} = \begin{pmatrix} 1 & 0 & -4 \\ 3 & 4 & 0 \end{pmatrix} \qquad \boldsymbol{C} = \begin{pmatrix} -19 & -6 \\ 7 & 22 \end{pmatrix}$$

Figure 2-5: Multiplication of matrix $\boldsymbol{A}$ (2×3) with matrix $\boldsymbol{B}$ (3×2) to get matrix $\boldsymbol{C}$ (2×2).

In iterative methods, multiplication of a matrix by a vector is very common. This operation can be explained the same way as matrix-matrix multiplication, except one of the matrices has dimensions of $(n \times 1)$ or $(1 \times n)$, and the solution is a vector.

An important concept pertaining to matrices is an ***identity matrix*** $(\boldsymbol{I})$, where the diagonal entries are equal to 1 and all other entries are equal to 0. Furthermore, the ***inverse*** of a matrix $(\boldsymbol{A}^{-1})$ is a similar idea to finding the reciprocal of a number. Putting these two concepts together, the inverse of matrix $\boldsymbol{A}$ has to satisfy:

$$\boldsymbol{A} \times \boldsymbol{A}^{-1} = \boldsymbol{A}^{-1} \times \boldsymbol{A} = \boldsymbol{I}. \qquad (2.27)$$

A matrix is considered ***nonsingular*** if and only if it can admit an inverse.

The ***transpose*** of a matrix $\boldsymbol{A}_{ij}$ can be expressed by:

$$\boldsymbol{A}_{ij}^{T} = \boldsymbol{A}_{ji} \qquad (2.28)$$

where the rows of $\boldsymbol{A}_{ij}$ are the columns of $\boldsymbol{A}_{ij}^{T}$. A ***symmetric matrix*** is a special case where the transpose of a matrix is equivalent to the original matrix. When a linear system consists of a symmetric matrix, it can be solved using different iterative techniques than one with a non-symmetric matrix.

The norm of a matrix can be defined in many different ways, similar to a vector norm. If $\boldsymbol{A}$ is a matrix with M rows and N columns, then the ***Euclidean norm of the matrix*** (l^2-norm) is:

$$\|\boldsymbol{A}\|_2 = \sqrt{\sum_{i=1}^{M}\sum_{j=1}^{N}\left|a_{ij}\right|^2}. \tag{2.29}$$

2.4.2 Linear Systems

As previously described, discretization of the transport equations leads to a system of linear equations, $\boldsymbol{Ax} = \boldsymbol{b}$, defined as follows:

$$\begin{bmatrix} a_{11} & \cdots & a_{1i} \\ \vdots & \ddots & \vdots \\ a_{i1} & \cdots & a_{ii} \end{bmatrix}\begin{bmatrix} x_1 \\ \vdots \\ x_i \end{bmatrix} = \begin{bmatrix} b_1 \\ \vdots \\ b_i \end{bmatrix} \tag{2.30}$$

where $\boldsymbol{A}$ is an $n \times n$ square matrix of coefficients (a), $\mathbf{b}$ is the right-hand side vector containing source terms, and $\boldsymbol{x}$ is the unknown solution vector. In CFD, the number of columns and rows for the $\boldsymbol{A}$ matrix are both equal to the grid size (i.e. multiplication of the dimensions $I \times J \times K$).

For solving a linear system, the ***condition number*** can give insight as to how inaccurate the solution $(\boldsymbol{x})$ will be after it is approximated. A linear problem with a low condition number is said to be ***well-conditioned***, whereas one with a high condition number is ***ill-conditioned***. Linear systems with ill-conditioned matrices are more difficult to solve are prone to giving unreliable solutions. The condition number is relative to how the matrix norm is calculated. In general, the condition number (κ) of a linear system can be computed as:

$$\kappa = \|\boldsymbol{A}\|\|\boldsymbol{A}^{-1}\| \tag{2.31}$$

where $||\boldsymbol{A}||$ is any given method of calculating a matrix norm, such as the l^2-norm described with Equation (2.29).

2.4.3 Iterative Methods

In general, iterative methods start with an initial guess to the solution, $\boldsymbol{x}$, and repeat a certain algorithm to keep getting better and better solutions with a goal of minimizing the solution residual. The residual gives the error of the solution vector, and it is calculated at each iteration step, $\boldsymbol{j}$, as [13]:

$$\boldsymbol{r_j} = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x_j}. \qquad (2.32)$$

There are two main types of iterative methods commonly used to solve large linear systems: stationary iterative methods and Krylov subspace methods.

2.4.4 Stationary Iterative Methods

Stationary iterative methods are some of the oldest and simplest iteration techniques for indirectly solving linear systems. This class gets its name since the data in the equation to solve for the solution at each iteration remains fixed. The idea behind these methods is to split the $\boldsymbol{A}$ matrix into a sum of two matrices, $\boldsymbol{A} = M - N$, for which M must be easily invertible. By doing this, we can derive:

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b} \rightarrow (M - N)\boldsymbol{x} = \boldsymbol{b} \rightarrow M\boldsymbol{x} = N\boldsymbol{x} + \boldsymbol{b} \rightarrow \overbrace{\boldsymbol{x} = M^{-1}N\boldsymbol{x} + M^{-1}\boldsymbol{b}}. \qquad (2.33)$$

Using the final form of Equation (2.33), a stationary iteration takes the general form [13]:

$$\boldsymbol{x_{j+1}} = M^{-1}N\boldsymbol{x_j} + M^{-1}\boldsymbol{b} \qquad (2.34)$$

where the multiplication product of $M^{-1}N$ is known as the iteration matrix. The matrix $M^{-1}N$ and the vector $M^{-1}\boldsymbol{b}$ do not change as the iterations proceed. Overall, stationary iteration methods differ by how M and N are defined.

2.4.5 Krylov Subspace Methods

Krylov subspace methods are forms of nonstationary iterative methods where the data changes at each iteration step. They are considered some of the most important iterative methods for solving large linear systems. Given a matrix $\boldsymbol{A}$ and a vector $\boldsymbol{v}$, the mth Krylov subspace can be represented by [14]:

$$\mathcal{K}_m(\boldsymbol{A}, \boldsymbol{v}) = span\{\boldsymbol{v}, \boldsymbol{Av}, \boldsymbol{A}^2\boldsymbol{v}, \dots, \boldsymbol{A}^{m-1}\boldsymbol{v}\}. \qquad (2.35)$$

For each of these methods, an approximate solution is found within $\mathcal{K}_m(\boldsymbol{A}, \boldsymbol{v})$, and sometimes with another Krylov subspace $\mathcal{K}_m(\boldsymbol{A}^T, \boldsymbol{w})$. The $\boldsymbol{v}$ and $\boldsymbol{w}$ vectors are typically dependent on the initial residual vector, $\boldsymbol{r_0} = \boldsymbol{b} - \boldsymbol{Ax_0}$. In general, Krylov subspace iterations take the form [13]:

$$\boldsymbol{x_{j+1}} = \boldsymbol{x_j} + \boldsymbol{\alpha_j p_j} \qquad (2.36)$$

where $\boldsymbol{p_j}$ is known as the search direction vector and $\boldsymbol{\alpha_j}$ the step length.

Krylov subspace methods generally will follow one of two common procedures: The Arnoldi [15] iteration or the Lanczos biorthogonalization [16] iteration. Krylov subspace methods that are based on Lanczos Biorthorgonalization are significant due to their ability to solve linear systems with non-symmetric matrices, such as those produced in MFiX. The Biconjugate Gradient (BCG) algorithm is based on the method by Lanczos, and it solves both the linear system, $\boldsymbol{Ax} = \boldsymbol{b}$, and a dual linear system that includes the transpose, $\boldsymbol{A}^T\boldsymbol{x}^* = \boldsymbol{b}^*$. Each step of the BCG method requires a matrix-vector product with both matrix $\boldsymbol{A}$ and its transpose, $\boldsymbol{A}^T$. The Conjugate Gradient Squared (CGS) algorithm was created to avoid performing operations with the transpose at each step as well as to gain faster convergence using approximately the same computational cost as BCG. In the CGS method, the residual vector at step j is calculated as [13]:

$$\boldsymbol{r_j} = \phi_j^2(\boldsymbol{A})\boldsymbol{r_0} \qquad (2.37)$$

for which ϕ_j is a specific polynomial of degree j that satisfies $\phi_j(0) = 1$.

Although the CGS algorithm works well in many cases, the squaring of the polynomial, ϕ_j, can potentially lead to a large build-up of rounding errors, and possibly overflow. The Biconjugate Gradient Stabilized (BiCGSTAB) solver is a method that was established by van der Vorst [17] to prevent the error build-up and overflow phenomenon that can occur with CGS. The residual vector for the BiCGSTAB solver instead takes the form [13]:

$$\boldsymbol{r_j} = \psi_j(\boldsymbol{A})\phi_j(\boldsymbol{A})\boldsymbol{r_0} \qquad (2.38)$$

where ϕ_j is the same polynomial defined for the CGS method, and ψ_j is a different polynomial which is redefined every step to smooth convergence. Furthermore, the search direction vector is defined as:

$$\boldsymbol{p_j} = \psi_j(\boldsymbol{A})\pi_j(\boldsymbol{A})\boldsymbol{r_0} \qquad (2.39)$$

for which π_j is a different j-degree polynomial.

BiCGSTAB was used throughout this study due to its ability to solve matrices of a non-symmetric structure and alleviate the build-up of rounding errors, which can become problematic with multiphase flows. Overall, the BiCGSTAB algorithm adheres to the following framework [13]:

```
1.    Calculate $\boldsymbol{r_0} = \boldsymbol{b} - \boldsymbol{Ax_0}$, and arbitrarily choose $\boldsymbol{r_0^*}$ such that $\boldsymbol{r_0^*} \cdot \boldsymbol{r_0} \neq 0$
2.    Set $\boldsymbol{p_0} = \boldsymbol{r_0}$
3.    For $\boldsymbol{j}$ = 0, 1, … , until convergence, Do:
        $\alpha_{\boldsymbol{j}} = \frac{(\boldsymbol{r_j}, \boldsymbol{r_0^*})}{(\boldsymbol{Ap_j}, \boldsymbol{r_0^*})}$ (where $\alpha_{\boldsymbol{j}}$is a scalar)
        $\boldsymbol{s_j} = \boldsymbol{r}_j - \alpha_j \boldsymbol{Ap}_j$
        $\omega_{\boldsymbol{j}} = \frac{(\boldsymbol{As_j}, \boldsymbol{s_j})}{(\boldsymbol{As_j}, \boldsymbol{As_j})}$ (where $\omega_{\boldsymbol{j}}$ is a scalar)
        $\boldsymbol{x_{j+1}} = \boldsymbol{x_j} + \alpha_{\boldsymbol{j}} \boldsymbol{p_j} + \omega_{\boldsymbol{j}} \boldsymbol{s_j}$
        $\boldsymbol{r_{j+1}} = \boldsymbol{s_j} - \omega_j \boldsymbol{As_j}$
        $\beta_j = \frac{(\boldsymbol{r_{j+1}}, \boldsymbol{r_0^*})}{(\boldsymbol{r_j}, \boldsymbol{r_0^*})} \times \frac{\alpha_j}{\omega_j}$ (where $\beta_{\boldsymbol{j}}$ is a scalar)
        $\boldsymbol{p_{j+1}} = \boldsymbol{r_{j+1}} + \beta_j (\boldsymbol{p_j} - \omega_{\boldsymbol{j}} \boldsymbol{Ap_j})$
    End Loop
    Set solution $\boldsymbol{x} = \boldsymbol{x_{j+1}}$
```

For this algorithm, $\omega_{\boldsymbol{j}}$ is a stabilizing parameter defined to minimize the l^2-norm of the residual vector, $\boldsymbol{r_{j+1}}$.

2.4.6 Preconditioning

For many cases, stationary iterative methods have been replaced by Krylov subspace methods due to the sophistication of these techniques. However, these classical methods still find a role in numerical methods as preconditioners. Preconditioning is used to transform a linear system into one with the same solution, yet it becomes easier to solve using an iterative method. Preconditioners can do this by reducing the condition number of a given linear system. Overall, preconditioners can improve both the efficiency and the robustness of iterative numerical methods.

The first step of using preconditioning techniques is to identify a preconditioning matrix, $\boldsymbol{M}$. This matrix should be nonsingular, and it should be close to the original matrix,

$\boldsymbol{A}$, in some way. In addition, the preconditioner should be chosen to solve linear systems efficiently. After identifying the preconditioning matrix, there are three ways to apply this matrix to a linear system: from the left, from the right, and in a factored form. However, when a linear system is non-symmetric, the preconditioner should only be applied from the left or the right. Applying a preconditioner from the left to a linear system will yield [13]:

$$\boldsymbol{M}^{-1}\boldsymbol{A}\boldsymbol{x} = \boldsymbol{M}^{-1}\boldsymbol{b}, \tag{2.40}$$

And the Krylov subspace takes the form $\mathcal{K}_m(\boldsymbol{M}^{-1}\boldsymbol{A}, \boldsymbol{M}^{-1}\boldsymbol{b})$. Preconditioning can also be performed from the right [13]:

$$\boldsymbol{A}\boldsymbol{M}^{-1}\boldsymbol{u} = \boldsymbol{b}, \tag{2.41}$$

where $\boldsymbol{x} = \boldsymbol{M}^{-1}\boldsymbol{u}$, and the Krylov subspace takes the form $\mathcal{K}_m(\boldsymbol{A}\boldsymbol{M}^{-1}, \boldsymbol{b})$.

While it is true that left- and right-side preconditioners have similar asymptotic behavior, they can actually behave differently depending on the linear system. The termination criterion of Krylov subspace methods is generally related to the residual norm of the preconditioned system. When preconditioning is applied from the left, the preconditioned residual, defined as $\left\|\boldsymbol{M}^{-1}\boldsymbol{r}_{\boldsymbol{j}}\right\|$, can greatly differ from the true residual $\left\|\boldsymbol{r}_{\boldsymbol{j}}\right\|$ if the $\|\boldsymbol{M}^{-1}\|$ value is far from 1. Unfortunately, this can be a common problem when applying left preconditioning to large linear systems. On the other hand, right preconditioners use the unaltered, true residual with an insignificant increase in computational cost. Right preconditioners should not lead to large solution errors, unless the preconditioning matrix, $\boldsymbol{M}$, is extremely ill-conditioned [14] .

Four preconditioning methods were focused on throughout this thesis: line relaxation, diagonal scaling, successive over relaxation (SOR), and Block Jacobi. Line relaxation and diagonal scaling are the only two preconditioners available in MFiX. On the other hand, PETSc offers several preconditioning options, but only SOR and Block Jacobi were tested within this work.

Some preconditioners can be formulated by decomposing matrix $\boldsymbol{A}$ into $\boldsymbol{A} = \boldsymbol{D} - \boldsymbol{E} - \boldsymbol{F}$, where $\boldsymbol{D}$, $-\boldsymbol{E}$, and $-\boldsymbol{F}$ are the diagonal, lower triangular , and upper triangular matrices. When the preconditioner, $\boldsymbol{M}$, is equivalent to the matrix diagonal, $\boldsymbol{D}$, then this method is known as diagonal scaling (i.e. Jacobi). If instead $\boldsymbol{M} = \frac{1}{\omega}(\boldsymbol{D} - \omega\boldsymbol{E})$, for which ω is the relaxation parameter, then this is the SOR preconditioner [13].

Matrix $\boldsymbol{A}$ can also be decomposed into submatrices and subvectors as follows:

$$\boldsymbol{A} = \begin{pmatrix} \boldsymbol{A}_{11} & \cdots & \boldsymbol{A}_{1n} \\ \vdots & \ddots & \vdots \\ \boldsymbol{A}_{n1} & \cdots & \boldsymbol{A}_{nn} \end{pmatrix}, \quad \boldsymbol{x} = \begin{pmatrix} \xi_1 \\ \vdots \\ \xi_n \end{pmatrix}, \quad \boldsymbol{b} = \begin{pmatrix} \beta_1 \\ \vdots \\ \beta_n \end{pmatrix} \tag{2.42}$$

where the submatrices $\boldsymbol{A}_{ii}$ are consistent with the subvectors ξ_i and β_i. Similar to before, matrix $\boldsymbol{A}$ can be partitioned as $\boldsymbol{A} = \boldsymbol{D} - \boldsymbol{E} - \boldsymbol{F}$, where $\boldsymbol{D}$, $-\boldsymbol{E}$, and $-\boldsymbol{F}$ contain submatrices along the diagonal, submatrices in the lower triangular region, and submatrices in the upper triangular region. These block preconditioning methods typically use the submatrices along the matrix diagonal as the preconditioner for the linear system [13].

One standard approach is to formulate these submatrices (i.e. blocks) by breaking down the matrix and vectors by whole lines of the simulation mesh. For example, a 2D mesh can be partitioned by its columns or rows, which is known as line relaxation. Additionally, the submatrices can contain multiple consecutive columns or rows, or the submatrices and subvectors could overlap. Block Jacobi preconditioners are methods of block preconditioning geared towards parallel environments. Traditional Block Jacobi preconditioning employs a domain decomposition with no overlap. Figure 2-6 shows a 16x16 square matrix distributed across two processors, $\boldsymbol{A}_1^{(0)}$ and $\boldsymbol{A}_2^{(0)}$ [18]. In this example, each processor contains a local diagonal block, which is portrayed by the shaded regions.

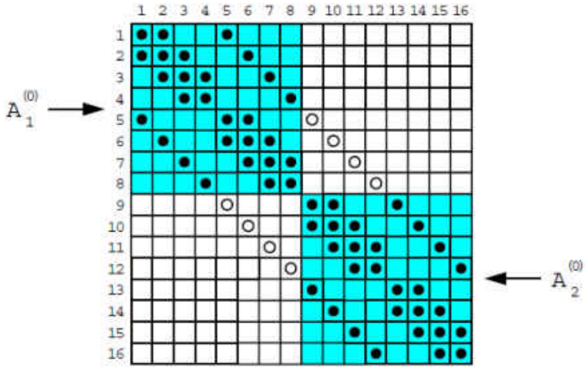


Figure 2-6. A 16x16 square matrix distributed across two processors, with each containing a shaded local diagonal block [18].

The Block Jacobi preconditioners in PETSc are obtained by applying incomplete LU factorizations with zero-fill in (ILU(0)) on each processor's local diagonal blocks. In general, the ILU factorization process formulates a sparse lower triangular matrix $\boldsymbol{L}$ and a sparse upper triangular matrix $\boldsymbol{U}$ which have the same nonzero structure as the lower and upper sections of $\boldsymbol{A}$. These matrices are computed so that $\boldsymbol{R} = \boldsymbol{LU} - \boldsymbol{A}$, the residual matrix, satisfies a certain constraint. For the ILU(0) method, this constraint is having zero entries in certain locations. This technique aims to define a preconditioner $\boldsymbol{M} = \boldsymbol{LU}$ such that the elements of the residual matrix are zero in the locations that the A matrix is non-zero [13].

2.4.7 Convergence

By default, both PETSc and MFiX test for convergence based upon the l_2-norm of the residual vector ($\boldsymbol{r_j}$). In PETSc, convergence is detected at iteration $\boldsymbol{j}$ if [12]:

$$\left\|\boldsymbol{r_j}\right\|_2 < max(rtol * \|\boldsymbol{b}\|_2, atol) \tag{2.43}$$

where $rtol$ is the decrease of the residual norm relative to the norm of the right hand side, $atol$ is the absolute size of the residual norm, and $dtol$ is the relative increase in the residual. The $rtol$, $atol$, and $dtol$ parameters can be set by the user.

2.5 Interfacing MFiX with PETSc

The PETSc suite of solvers consists of the following sub-components: Vectors, Matrices, Distributed Arrays, Preconditioners, Krylov Subspace Solvers, Non-Linear Solvers, Index Sets, and Time-steppers. PETSc allows for easy customization and extension to these components. Additionally, a comprehensive suite of data structures for parallel matrix and vector storage as well as unified interfaces to linear solvers and preconditioners are offered in PETSc for the purpose of achieving scalable, parallel computation. The matrices are computed and distributed among all processors involved in the simulation. PETSc enables us to work only with global indices even though, internally, local indices are used for accessing the distributed data structures. Moreover, PETSc provides various sparse matrix storage formats all of which have a uniform interface to the matrix operations [12].

With this work, the multiphase CFD code MFiX has been interfaced with the PETSc linear solver library to gain access to this suite of robust, scalable solver options available in PETSc. This framework allows the solver and preconditioner to be selected from a variety of options based upon the specific problem. The abstract solver interface for solving the generic linearized system, $\boldsymbol{Ax} = \boldsymbol{b}$, includes:

1. Problem Setup: Functionality for setting PETSc solver parameters such as solver tolerances, maximum number of iterations, and preconditioners as dictated by derived solver types.

2. Solver Setup: Solver object creation (allocation of $\boldsymbol{A}$, $\boldsymbol{x}$, and $\boldsymbol{b}$) initialization methods.
3. Communication Linear System: Handshake function for passing the linear system coefficients ($\boldsymbol{A}$) and right-hand-side values ($\boldsymbol{b}$) in the current native MFiX data-structure and subsequent conversion to the solver-specific types for PETSc.
4. Solve System: Using PETSc's native solver types, this function will compute the solution ($\boldsymbol{x}$) to the linear system.
5. Return/Copy Solution: Conversion of the solver type solution ($\boldsymbol{x}$) to the current, native MFiX type.
6. Cleanup: De-allocation and destruction of PETSc solver objects.

CHAPTER 3

BUILDING THE SOFTWARE

3.1 Native MFiX

MFiX version 2016.1 was used to conduct studies with native MFiX and to create the MFiX-PETSc interface. In the home “mfix-2016.1” directory, the “configure_mfix” script should be run to create a “Makefile". The following command was used throughout this study:

```
➢ ./configure_mfix FC=gfortran FCFLAGS=’-O3 –g –fbacktrace’
```

With this, the Fortran compiler is specified as gfortran by passing the FC argument to the “configure_mfix” script. Then, the script will test the compiler with certain flags specified with “FCFLAGS”. The “-g” flag is used to produce debugging information. The “-O3” flag refers to an optimization level, and “-fbacktrace’ provides a full backtrace when a runtime error is detected during debugging.

After creating the Makefile, the following command will build the MFiX executable:

```
➢ make
```

3.2 Native PETSc

The version of PETSc that was used throughout this work was 3.7.2. To configure PETSc without MPI, the following command can be used from the “petsc-3.7.2” directory:

- ./configure –with-mpi=0 –download-fblaslapack

Once PETSc is successfully configured, the code will walk you through the steps to build the program.

3.3 MFiX-PETSc Interface

The serial version of the MFiX-PETSc interface was successfully built and run on two different Linux operating systems. One of these was a Linux Ubuntu 17.10 system with a single-core. The second was a High Performance Computing (HPC) Linux cluster, that employed the Red Hat Enterprise Linux 7.3 distribution.

The first step of interfacing was to successfully configure and build PETSc. Then, MFiX was configured to create a Makefile. After these steps, three files from the MFiX code were altered to allow these two programs to talk with one another before building the interface executable.

3.3.1 leq_petsc.f

The ".f" files located in the MFiX "model" subdirectory contain modules, functions, and subroutines to carry out tasks such as placing the discretized equations into a linear system, calculating coefficients, applying solvers to linear systems, and many others. To create the interface, a new file had to be created in this subdirectory called "leq_petsc.f". This file contained a subroutine "LEQ_PETSC" which called out to PETSc in order to access its solvers and preconditioners to solve a given linear system.

First, this file reads in the coefficients for the matrix and right-hand side vector that have been created by MFiX for a given equation, such as the pressure-correction equation. New matrices and vectors are constructed using specific PETSc commands. Then, these coefficients are placed into the new matrix and vectors by employing a certain mapping function between the MFiX and PETSc matrix/vector indices.

After the linear system has been re-created, the solver is constructed and settings such as tolerance and preconditioner are specified. Once the solver is created, it is used to iteratively solve the linear system. Finally, the solution vector is passed back to MFiX to continue on with computation.

3.3.2 solve_lin_eq.f

The "solve_lin_eq.f" file redirects each equation to the respective solver. A command was added into this file to pass the following elements to the "LEQ_PETSC" subroutine:

- Name of the variable to be solved for (such as fluid pressure)
- Number denoted in MFiX for the variable (fluid pressure = 1)
- Current solution vector, $\boldsymbol{x}$
- Coefficient matrix, $\boldsymbol{A}$
- Right-hand side vector, $\boldsymbol{b}$

3.3.3 Makefile

When MFiX is built, object files denoted with ".o" and dependencies denoted with ".d" files are generated from the ".f" files. A few lines of code were added into the "Makefile" so that these files would be generated for "leq_petsc.f" as well.

Paths to the PETSc "include" subdirectories also had to be added to the Makefile. Then, the following environment variables were added to the Makefile as well:

- "PETSC_DIR" – points to the petsc-3.7.2 directory

- “PETSC_ARCH” – points to the subdirectory “arch-linux2-c-debug” which contains a specific set of libraries depending on the compiler and machine type
- “PETSCLIB_DIR” – points to the “lib” subdirectory which contains variables, rules, and more

CHAPTER 4

3D, STEADY-STATE HEAT CONDUCTION

4.1 Problem Overview

The first case carried out using MFiX-PETSc was heat conduction throughout a three-dimensional cube at steady-state. Temperature boundary conditions were set at the top and bottom walls, while the other four walls were adiabatic. This problem was chosen as the initial case since the solution is well-known, therefore it was used to test the interface framework. Theoretically, at steady-state conditions, there should be a linear temperature profile between the top and bottom walls. Figure 4-1 shows the dimensions and temperature boundary conditions used throughout Case 1.

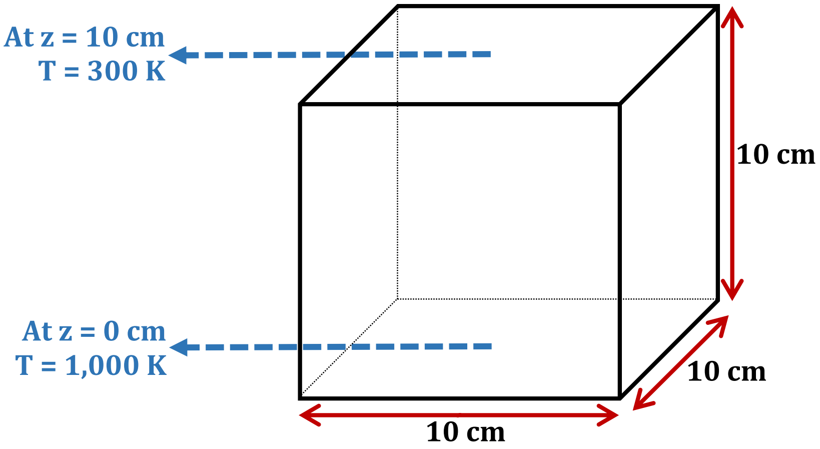


Figure 4-1: Geometry dimensions and boundary conditions used to carry out the 3D, steady-state heat conduction case.

Simulating this case only required solving the energy conservation equation. The medium inside of the geometry was chosen as air, since the material should not affect the temperature profile at steady-state. For each simulation, the viscosity, density, and molecular weight of air were 10^{-2} g/(cm·s), 2.5 g/cm^3, and 29 g/mol respectively. Additionally, the gas conductivity was 1.0 cal/(s·cm·K) and the heat capacity was 1.0 cal/(g·s·K). The mfix.dat input file for this case can be found in the Appendix.

For both native MFiX and the MFiX-PETSc Interface, the BiCGSTAB solver was used to solve the energy equation with an outer iteration tolerance of 10^{-4}. In native MFiX, line relaxation preconditioning was applied to BiCGSTAB with a solver (inner iteration) tolerance of 10^{-4}. The maximum number of inner iterations for solving the energy equation in MFiX is set to 15 by default. This maximum value was increased to 10,000, which allowed the solution to converge due to the residual norm rather than by reaching a maximum iteration value. Using MFiX-PETSc, SOR and Block Jacobi preconditioning were applied from both the left- and right-side. Solver tolerances of both 10^{-4} and 10^{-7} were tested with the interface, while the maximum number of solver iterations was also set to 10,000.

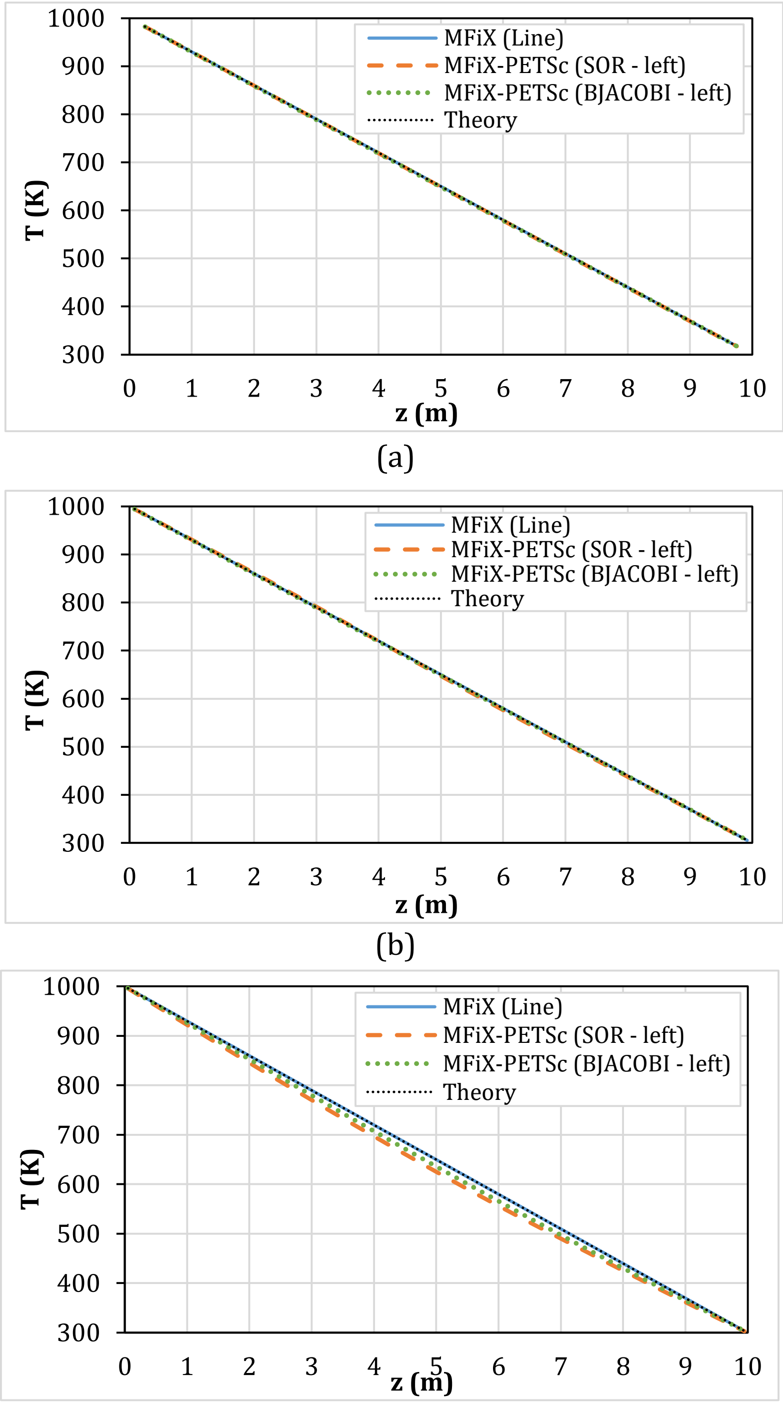
The meshes used varied from 8,000 elements (20x20x20) to 1,000,000 elements (100x100x100). Table 4-1 summarizes all of the simulations tested throughout Case 1. The main goal of this exercise was to verify that the MFiX-PETSc interface was working correctly by comparing the solutions obtained with MFiX-PETSc against native MFiX and theoretical results. An additional objective was to make an initial assessment of the preconditioners and solver settings offered in PETSc. To do this, CPU times and solution accuracy were compared between the different preconditioners.

Table 4-1. Summary of grid dimensions, meshing, tolerances, solvers, and preconditioners employed in Case 1.

Case 1: 3D, Steady-State Heat Conduction					
Case	**Dimensions**	**Solver**	**Tolerance**	**Preconditioner**	**Mesh**
1.1	10x10x10 cm^3	BCGS	Outer: 10^{-4} Solver: 10^{-4}	MFiX: Line	1. 20x20x20 2. 40x40x40 3. 60x60x60 4. 80x80x80 5. 100x100x100
1.2				MFiX-PETSc: SOR (left)	1. 20x20x20 2. 40x40x40 3. 60x60x60 4. 80x80x80 5. 100x100x100
1.3				MFiX-PETSc: Block Jacobi (left)	1. 20x20x20 2. 40x40x40 3. 60x60x60 4. 80x80x80 5. 100x100x100
1.4			Outer: 10^{-4} Solver: 10^{-7}	MFiX-PETSc: SOR (left)	1. 20x20x20 2. 40x40x40 3. 60x60x60 4. 80x80x80 5. 100x100x100
1.5				MFiX-PETSc: Block Jacobi (left)	1. 20x20x20 2. 40x40x40 3. 60x60x60 4. 80x80x80 5. 100x100x100
1.6			Outer: 10^{-4} Solver: 10^{-4}	MFiX-PETSc: SOR (right)	1. 100x100x100
1.7				MFiX-PETSc: Block Jacobi (right)	1. 100x100x100

4.2 Results

For each simulation, the fluid temperature was recorded in the z-direction (from 0 to 10 cm) when x = 2.25 cm and y = 5 cm. The temperature profiles were then compared against theoretical values. The plots in Figure 4-2 show these temperature distributions for a solver tolerance of 10^{-4} (Cases 1.1-1.3) for the following meshes: (a) 20x20x20, (b) 60x60x60, and (c) 100x100x100. At this intermediate solver tolerance (10^{-4}), it’s evident that the left-side preconditioning options in PETSc became less accurate as the problem size was increased.



(a)

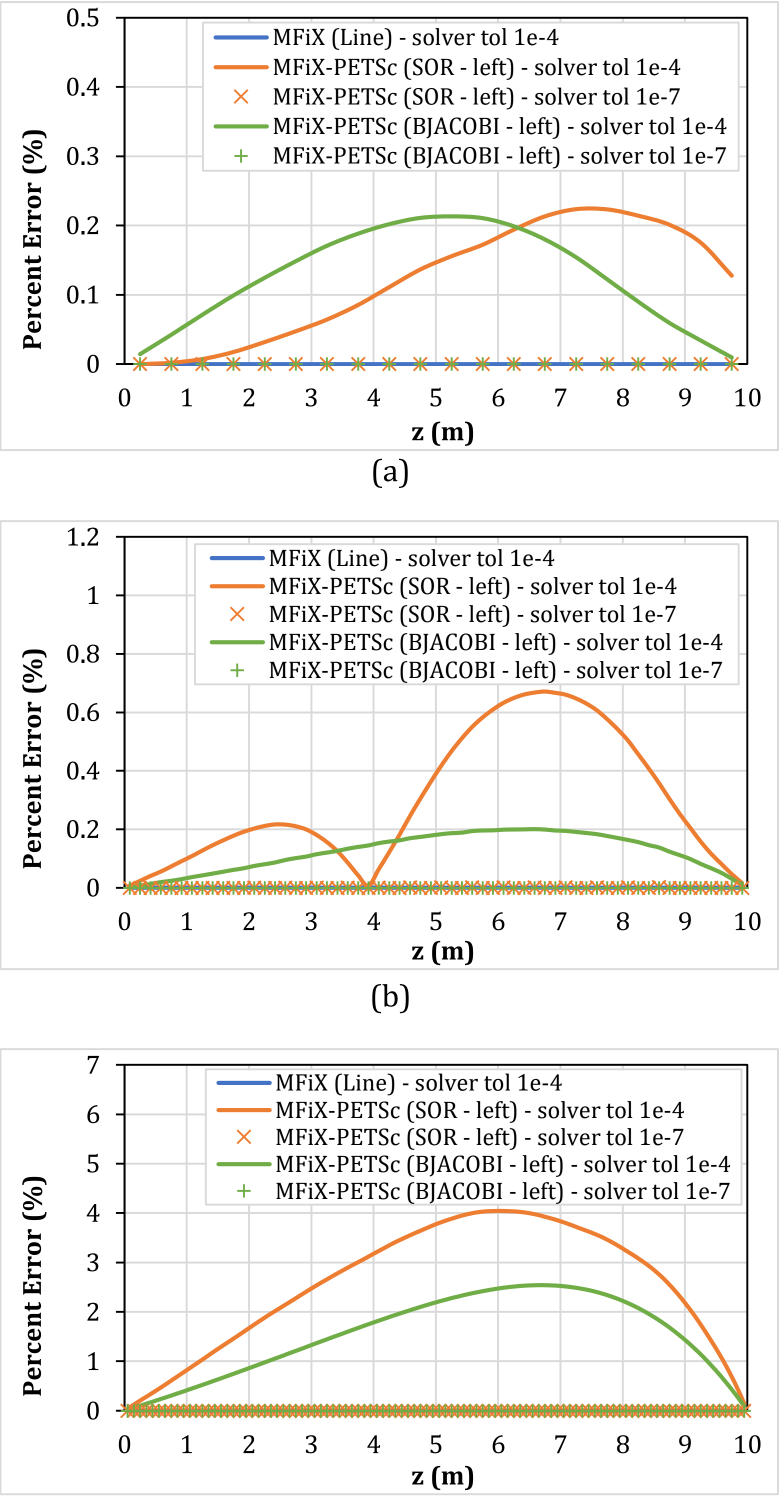
(b)

(c)

Figure 4-2. Temperature distributions in the z-direction when x = 2.25 cm and y = 5 cm for Cases 1.1-1.3 and mesh sizes of (a) 20x20x20, (b) 60x60x60, and (c) 100x100x100.

Based upon these results, the solver tolerance for MFiX-PETSc was refined to (10^{-7}) in Cases 1.4 and 1.5. Then, the temperature distributions obtained using an intermediate solver tolerance (10^{-4}) and a refined solver tolerance (10^{-7}) were compared by calculating percent errors from the theoretical solution. Figures 4-3 (a, b, c) compare these temperature percent errors for mesh sizes of (a) 20x20x20, (b) 60x60x60, and (c) 100x100x100. Similar to the results shown throughout Figure 4-2, temperature results obtained with MFiX-PETSc's preconditioners using an intermediate solver tolerance (10^{-4}) resulted in more error when compared to native MFiX at the same tolerance. This error generally increased at points further away from the boundary conditions (employed at z = 0 and 10). As the solver tolerance of MFiX-PETSc was refined to 10^{-7}, the error became minimal for both Block Jacobi and SOR preconditioners. Therefore, implementation of the MFiX-PETSc code has been verified by obtaining identical results to native MFiX and well-known theory.

Each simulation was repeated ten times to collect timing data. Figure 4-4 compares CPU time as a function of problem size (number of unknowns) for Cases 1.1-1.5. Overall, native MFiX with line relaxation preconditioning was significantly faster than any of the preconditioners tested in MFiX-PETSc for this steady-state problem. Additionally, refining the solver tolerance within the interface resulted in slower simulations, as expected. The interface was at best 168% slower than MFiX's native solver to obtain the same accuracy at a fine mesh resolution (100x100x100).



(a)

(b)

(c)

Figure 4-3. Percent errors for temperature in the z-direction when x = 2.25 cm and y = 5 cm for Cases 1.1-1.5 and mesh sizes of (a) 20x20x20, (b) 60x60x60, and (c) 100x100x100.

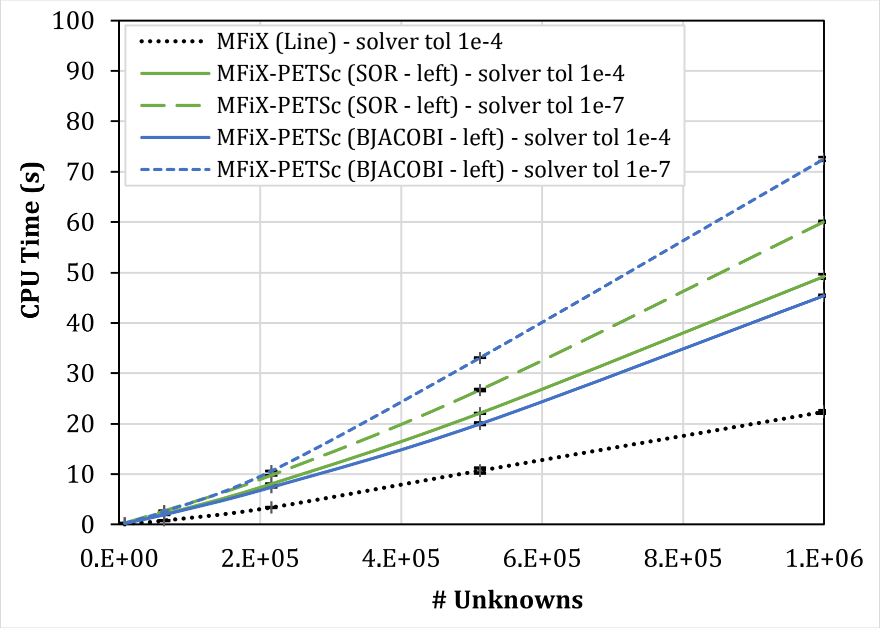


Figure 4-4. Comparison of CPU time as a function of problem size for Cases 1.1-1.5, with standard deviation error bars.

It is known that the application of a preconditioner from the left and the right can behave differently depending on the system being solved. For left preconditioning, the calculated residual can greatly differ from the true residual since it is computed based on the preconditioned system. This issue can escalate as the problem size increases. As demonstrated with this problem, the SOR and Block Jacobi preconditioners applied from the left-side became less accurate as the problem size was increased from 8,000 to 1,000,000 unknowns. Theoretically, if this loss in accuracy is related to how the residual is being calculated, applying these preconditioners from the right-side should alleviate this problem to achieve more accurate solutions. Cases 1.6 and 1.7 explore right-side SOR and right-side Block Jacobi preconditioning respectively, with an intermediate solver tolerance (10^{-4}) and a fine mesh (100x100x100). Figure 4-5 compares the temperature percent errors obtained using right-side preconditioning versus left-side preconditioning in MFiX-PETSc. The temperature profiles achieved with right-side preconditioning did not follow expectations. Even more error was introduced into this system when Block Jacobi, and especially SOR, were applied from the right-hand side. Moreover, the CPU times required to solve the system were similar for left and right preconditioning, as demonstrated in Figure 4-6.

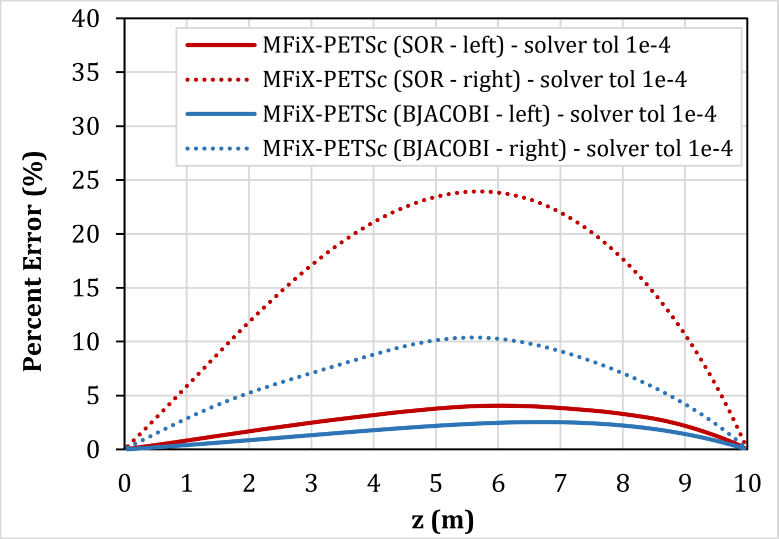


Figure 4-5. Comparison of temperature percent errors obtained for Case 1 using left-side (Cases 1.2 and 1.3) and right-side (Cases 1.6 and 1.7) preconditioning in MFiX-PETSc for an intermediate solver tolerance (10^{-4}) and a fine mesh (100x100x100).

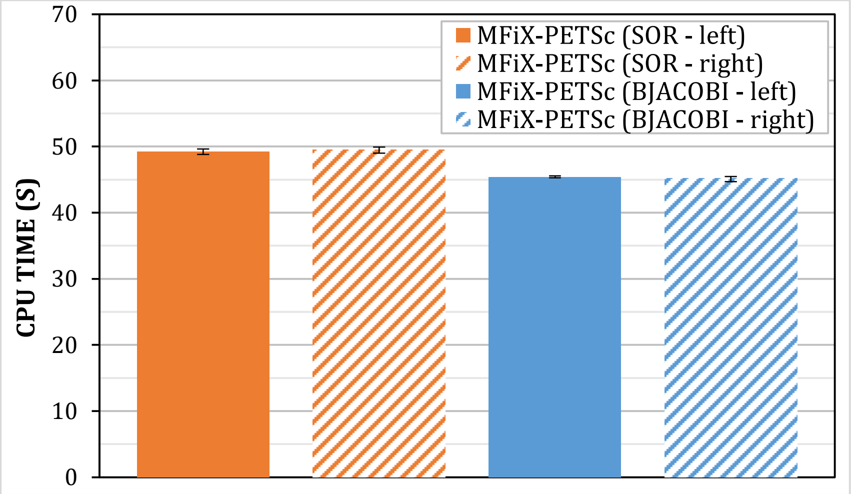


Figure 4-6. Comparison of the CPU time required to solve Case 1 using left-side (Cases 1.2 and 1.3) and right-side (Cases 1.6 and 1.7) preconditioning in MFiX-PETSc for an intermediate solver tolerance (10^{-4}) and a fine mesh (100x100x100)

CHAPTER 5

SINGLE-PHASE FLOW OVER A CYLINDER

5.1 Problem Overview

The second case investigated the isothermal, unsteady flow of air with a low Reynolds number past a cylindrical boundary on a two-dimensional domain. This is a well-studied problem dating back to the 1950's both experimentally and numerically due to its relevance in engineering applications. Obtaining an accurate numerical solution for this case can be challenging since the location of flow separation from the cylinder surface is only influenced by the flow regime and/or the upstream conditions [19].

This problem was simulated with three different meshes: a coarse mesh (120x80), an intermediate mesh (240x160), and a fine mesh (480x320). This was done to analyze the effect of meshing on the performance MFiX-PETSc. Figure 5-1 displays the coarse mesh that was used throughout Case 2 along with the cylinder and flow-field dimensions used for all levels of meshing. An inlet of ambient air was placed along the left side of the geometry, where x is equal to 0. The velocity of the inlet was set to 3 cm/s resulting in a Reynold's number (Re) of 200. The initial condition of the inlet air was set to 0 Pa, representing the gauge pressure. Additionally, the viscosity and density of air were assumed to remain constant at 1.8 x 10^{-5} Pa·s and 1.2 kg/m^3 respectively.

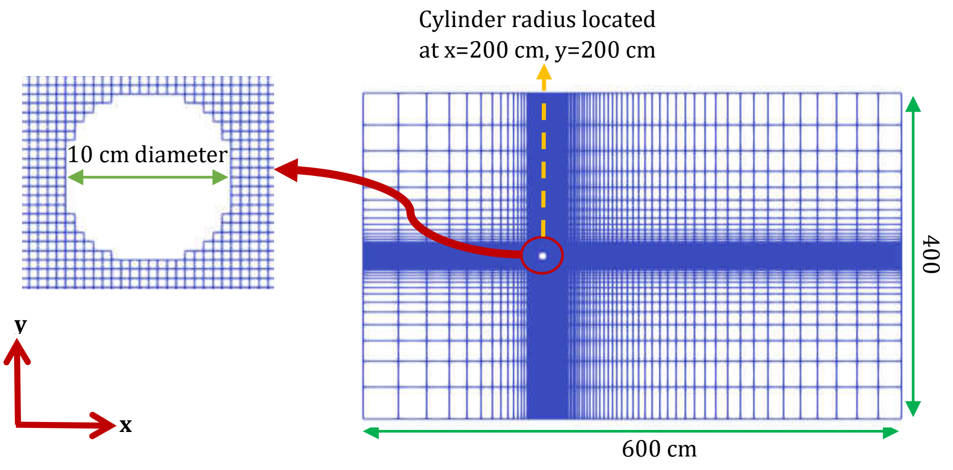


Figure 5-1. The coarse mesh (120x80) and dimensions used to simulate Case 2.

Simulating this case required solving the momentum conservation equation in both the x- and y- directions as well as the fluid pressure-correction equation. First-Order Upwind (F.O.U.P.), Superbee, and van Leer discretization schemes were applied to these equations to monitor their effect on pressure solutions and CPU time.

With native MFiX, line relaxation and diagonal scaling preconditioners were applied to BiCGSTAB to solve the pressure-correction equation. In the MFiX-PETSc solver, Block Jacobi and SOR preconditioning were applied on both sides (left and right) individually. The solver tolerance for all equations was set to 10^{-4}, and the maximum number of iterations to solve the pressure equation was 10,000 for both native MFiX and MFiX-PETSc to ensure that the linear solver for the pressure-correction equation did reach the specified tolerance every time. Table 5-1 summarizes the simulations run in Case 2 along with their settings. These simulations were each run for 300 seconds of simulation time with a constant time step of 0.25 seconds. The mfix.dat input file for this case can be found in the Appendix.

Table 5-1. Summary of grid dimensions, meshing, time steps, tolerances, solvers, discretization schemes, and preconditioners (P.C.) employed in Case 2.

Case 2: 2D Flow Over a Cylinder						
Case	**Dimensions + Mesh**	**Time Step**	**Tolerance**	**Solver**	**Scheme**	**P.C.**
2.1	6x4 m^2 120x80	DT: 0.25	Outer: 10^{-6} Solver: 10^{-4}	BCGS	1. F.O.U.P. 2. Superbee 3. van Leer	1. MFiX Line 2. MFiX Diag 3. Interface SOR (left) 4. Interface SOR (right) 5. Interface BJACOBI (left) 6. Interface BJACOBI (right)
2.2	6x4 m^2 240x160	DT: 0.25	Outer: 10^{-6} Solver: 10^{-4}	BCGS	1. F.O.U.P 2. Superbee 3. van Leer	1. MFiX Line 2. MFiX Diag 3. Interface SOR (left) 4. Interface SOR (right) 5. Interface BJACOBI (left) 6. Interface BJACOBI (right)
2.3	6x4 m^2 480x320	DT: 0.25	Outer: 10^{-6} Solver: 10^{-4}	BCGS	1. F.O.U.P. 2. Superbee 3. van Leer	1. MFiX Line 2. MFiX Diag 3. Interface SOR (left) 4. Interface SOR (right) 5. Interface BJACOBI (left) 6. Interface BJACOBI (right)

5.2 Results

Pressure data was collected along the surface of the cylinder by representing surface points as angles, which is illustrated in Figure 5-2. The pressure data was recorded at 11 surface points and then transformed into the Pressure Coefficient (C_p) as follows [20]:

$$C_p = \frac{P - P_\infty}{\frac{1}{2}\rho_\infty {U_\infty}^2} \tag{5.1}$$

where P is the time-averaged surface pressure, P_∞ is the constant pressure of the inlet air stream, ρ_∞ is the constant density of the inlet air stream, and U_∞ is the constant velocity of the inlet air stream. The surface pressure measurements were averaged after steady-state fluctuations were observed.

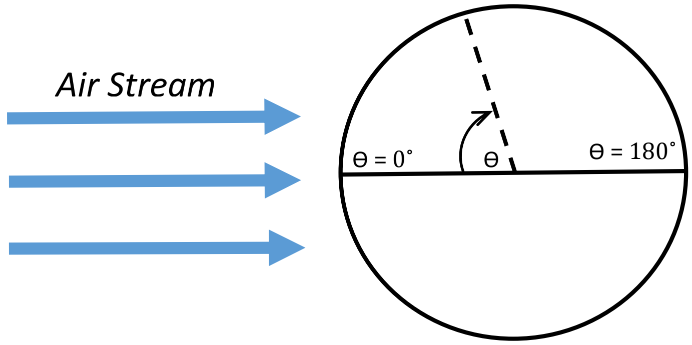
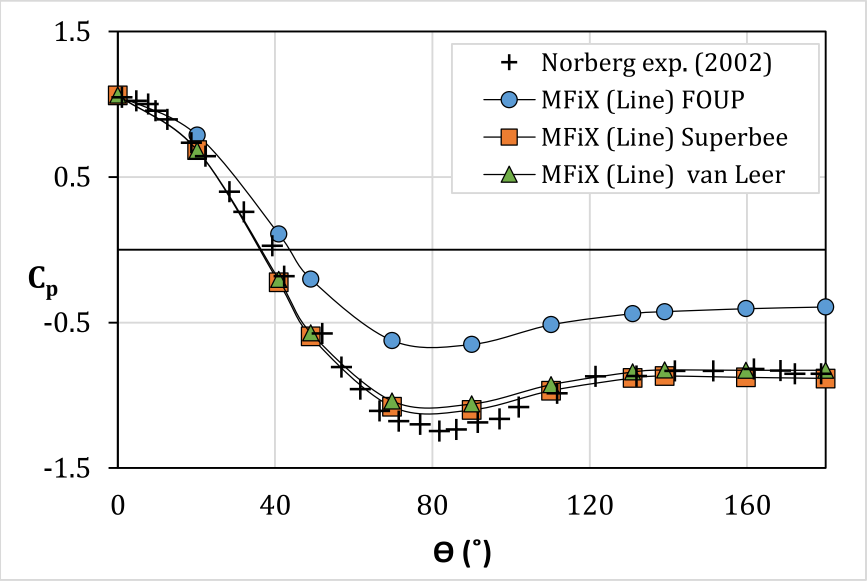
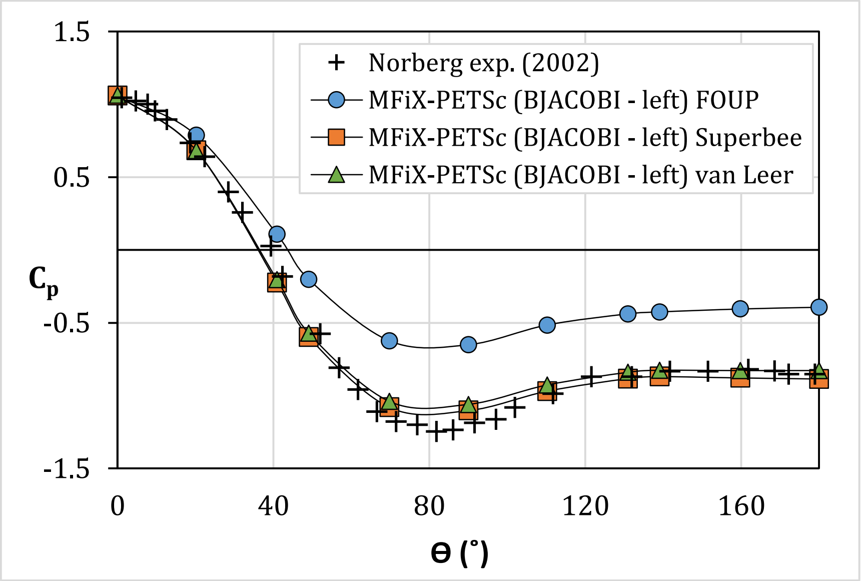


Figure 5-2. Surface points along the cylinder represented as angles.

Time-averaged pressure coefficients along the cylinder surface obtained using native MFiX and MFiX-PETSc were compared against experimental measurements from Norberg [20]. Figure 5-3 shows these pressure coefficient results achieved with (a) MFiX's line relaxation preconditioner and (b) MFiX-PETSc's left-side Block Jacobi preconditioner for a coarse mesh (Case 2.1). Similarly, Figures 5-4 (a, b) compare these two preconditioning methods for an intermediate mesh (Case 2.2). Results from the other MFiX-PETSc preconditioning options mentioned in Table 5-1 were exactly identical to those from the Block Jacobi (left) preconditioner and are not shown for brevity. First off, it is evident that the results obtained with MFiX-PETSc's left-side Block Jacobi preconditioner are identical to the results from MFiX's line relaxation preconditioner at the coarse and intermediate meshing levels. Therefore, with coarse and intermediate meshing, good agreement was demonstrated between native MFiX and MFiX-PETsc. Figures 5-3 and 5-4 also show how the higher-order discretization schemes (van Leer, Superbee) result in more accurate pressure predictions compared to the first-order upwind (F.O.U.P.) scheme with its results improving at higher mesh resolutions. A visual representation of these predictions is shown in Figure 5-5 where pressure and velocity contours of the different discretization and preconditioning options are compared for the coarse mesh simulations. Again, F.O.U.P. results in contours that are diffuse while the higher order schemes are able to better represent the fine vortical structures behind the cylinder.



(a)



(b)

Figure 5-3. Comparison of time-averaged pressure coefficients obtained with (a) MFiX's line relaxation and (b) MFiX-PETSc's left-side Block Jacobi preconditioners against experimental measurements from Norberg [20] using a coarse mesh (Case 2.1).

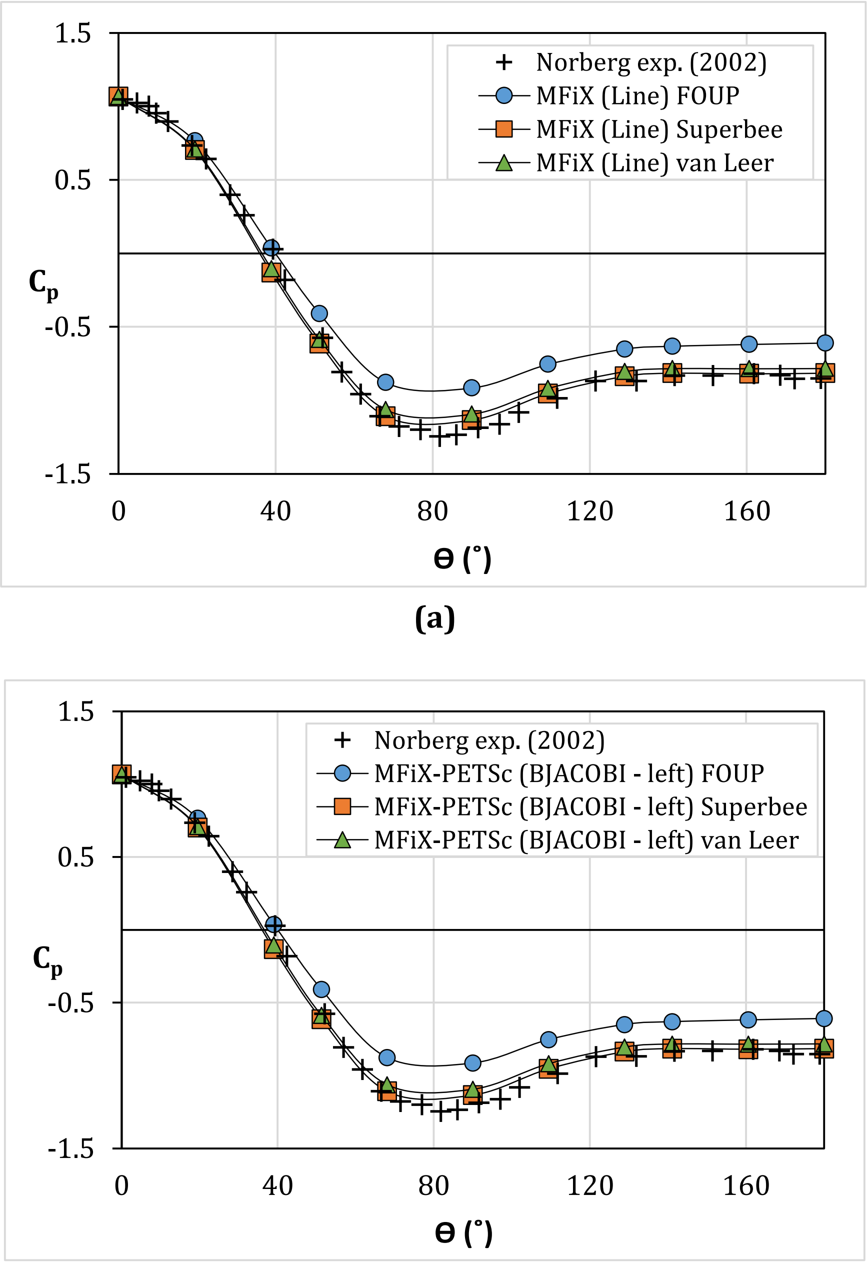
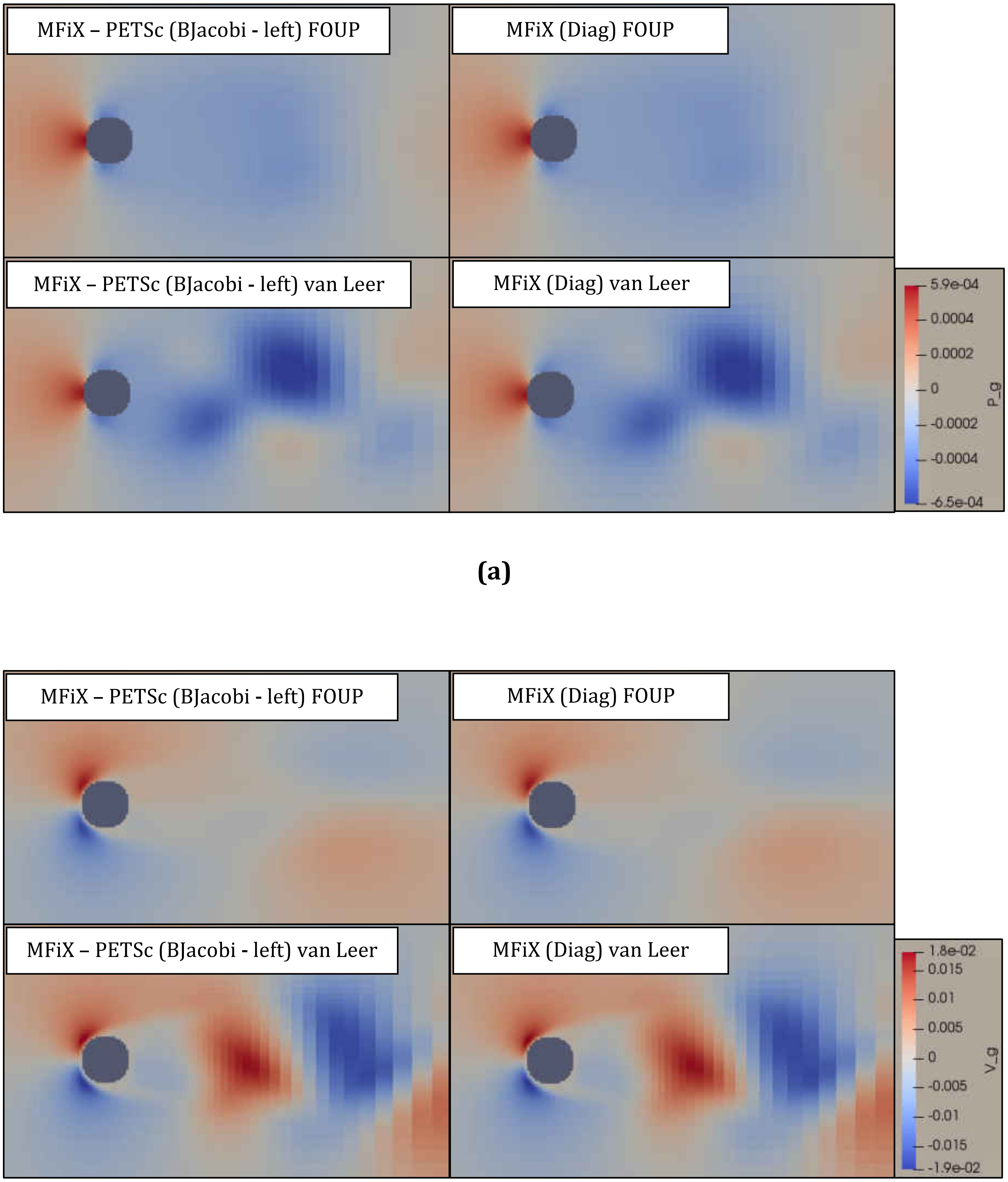


Figure 5-4. Comparison of time-averaged pressure coefficients obtained with (a) MFiX's line relaxation and (b) MFiX-PETSc's left-side Block Jacobi preconditioners against experimental measurements from Norberg [20] using an intermediate mesh (Case 2.2).



(a)

(b)

Figure 5-5. Comparison of the (a) pressure and (b) y-velocity contours between F.O.U.P. and van Leer discretization schemes at 100 seconds using left-side Block Jacobi preconditioning in MFiX-PETSc and diagonal scaling preconditioning in the native MFiX solver.

As the meshing was refined even more in Case 2.3, the preconditioning techniques had an effect on the pressure coefficient solutions. Figures 5-6, 5-7, and 5-8 show the time-averaged pressure coefficients as a function of the surface angle for F.O.U.P., Superbee, and van Leer discretization schemes employed throughout Case 2.3 respectively. For all types of discretization at this fine meshing level, both right- and left-side Block Jacobi, as well as left-side SOR agreed well with the experimental results. In opposition, both of the preconditioners offered in MFiX and right-side SOR from MFiX-PETSc were not able to achieve these high levels of accuracy as the mesh was refined.

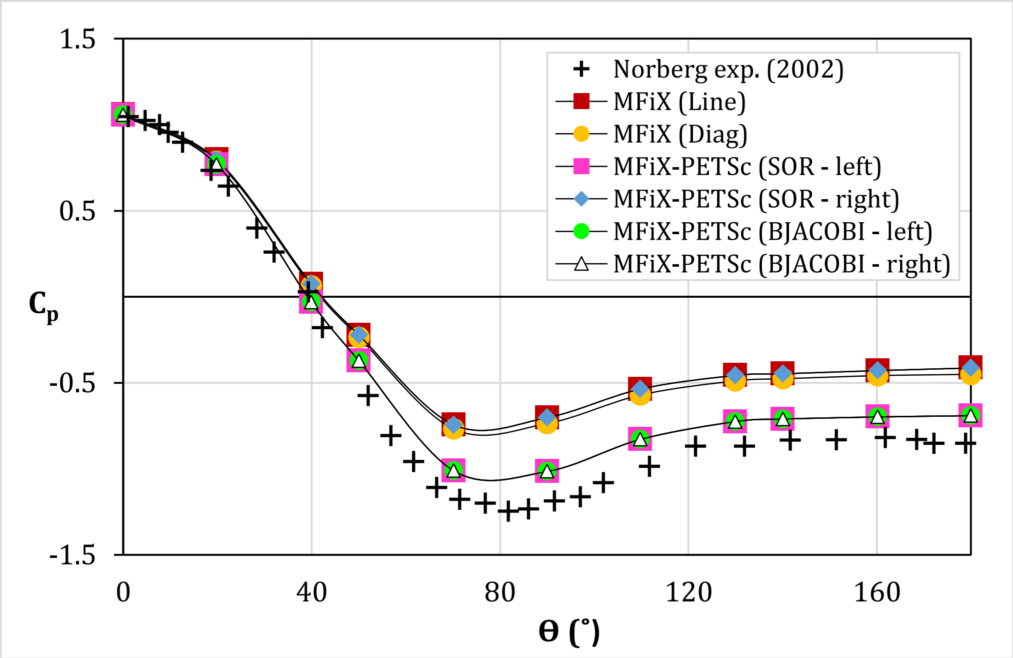


Figure 5-6. Comparison of time-averaged pressure coefficients obtained in Case 2.3 against previous experimental measurements from Norberg [20] for F.O.U.P discretization.

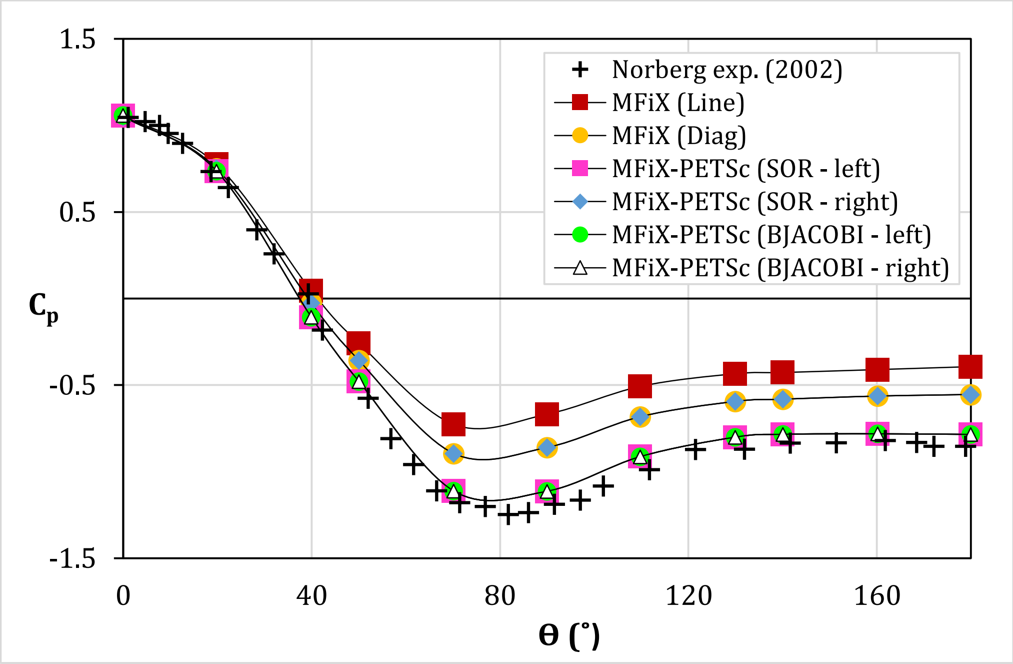


Figure 5-7. Comparison of time-averaged pressure coefficients obtained in Case 2.3 against previous experimental measurements from Norberg [20] for Superbee discretization.

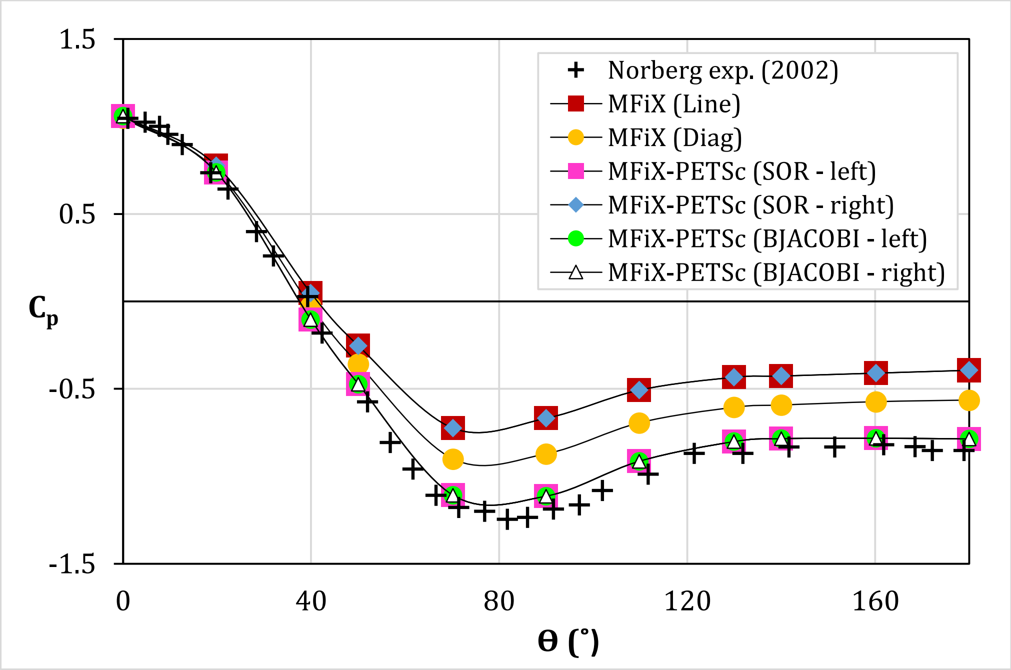


Figure 5-8. Comparison of time-averaged pressure coefficients obtained in Case 2.3 against previous experimental measurements from Norberg [20] for van Leer discretization.

Back in Case 1, the heat conduction problem, different behavior was observed when Block Jacobi and SOR were applied from the left-side versus the right-side. For that specific problem, preconditioning from the left was significantly more accurate. For this case, that trend remained when SOR was applied from the right-side; however, left-side and right-side Block Jacobi both yielded accurate pressure solutions. These findings support the idea presented in section 2.5.2 that applying preconditioning from the left or the right can exhibit extremely different performance depending on the linear system.

For the fine mesh used in Case 2.3, applying MFiX's preconditioners with a constant time step caused convergence issues, with the exception of line relaxation combined with Superbee discretization. When the diagonal scaling preconditioner was used, the results immediately diverged due to extremely high velocity values near the inlet. The constant time step setting was relaxed with diagonal scaling to allow these cases to reach 300 seconds of simulation time. As these simulations approached steady pressure fluctuations, the time step approached 0.9 seconds, which was larger compared to the constant 0.25 second time step used in the other cases. Applying line relaxation with the F.O.U.P. and van Leer discretization schemes diverged after 296 and 235 seconds of simulation time respectively. Pressure and timing results were still analyzed for these simulations up until the divergence errors.

Figures 5-9, 5-10, and 5-11 compare the CPU time as a function of problem size for F.O.U.P., Superbee, and van Leer discretization respectively. The line relaxation preconditioning option which is the default in native MFiX was significantly slower than the other preconditioning options explored in this study. While switching over to MFiX's diagonal scaling preconditioner did speed up the time to solution, the right- and left-side Block Jacobi (BJACOBI) preconditioning options (to the BiCGSTAB solver) in MFiX-PETSc emerged as the faster solver-preconditioner combination for this test problem. Altogether, right-side Block Jacobi was 40%, 30% and 28% faster than native MFiX's diagonal scaling for the F.O.U.P., Superbee, and van Leer discretization schemes respectively. Moreover, left-side Block Jacobi was 46%, 27%, and 31% faster than MFiX's diagonal scaling for F.O.U.P., Superbee, and van Leer discretization schemes respectively.

Figure 5-12 compares the average number of iterations (with standard deviations) to solve the pressure-correction equation in Cases (a) 2.1, (b) 2.2, and (c) 2.3. The Block Jacobi preconditioner in MFiX-PETSc required a lower number of solver iterations compared to the other preconditioners across all mesh sizes and discretization schemes. Consequently, these also translate to the trends in CPU time observed in Figures 5-9 – 5-11. Overall, Block Jacobi preconditioning from both the left and right was faster, more accurate, and exhibited better convergence properties than MFiX's preconditioning options for 2D flow past a cylinder.

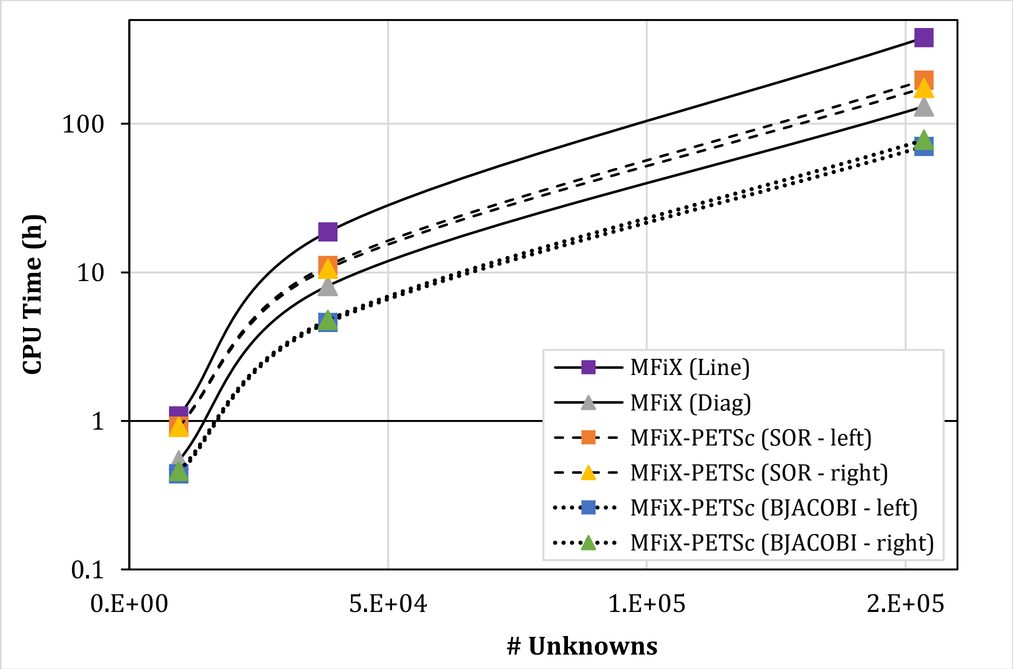


Figure 5-9. CPU time as a function of problem size for Case 2 using the F.O.U.P. discretization scheme.

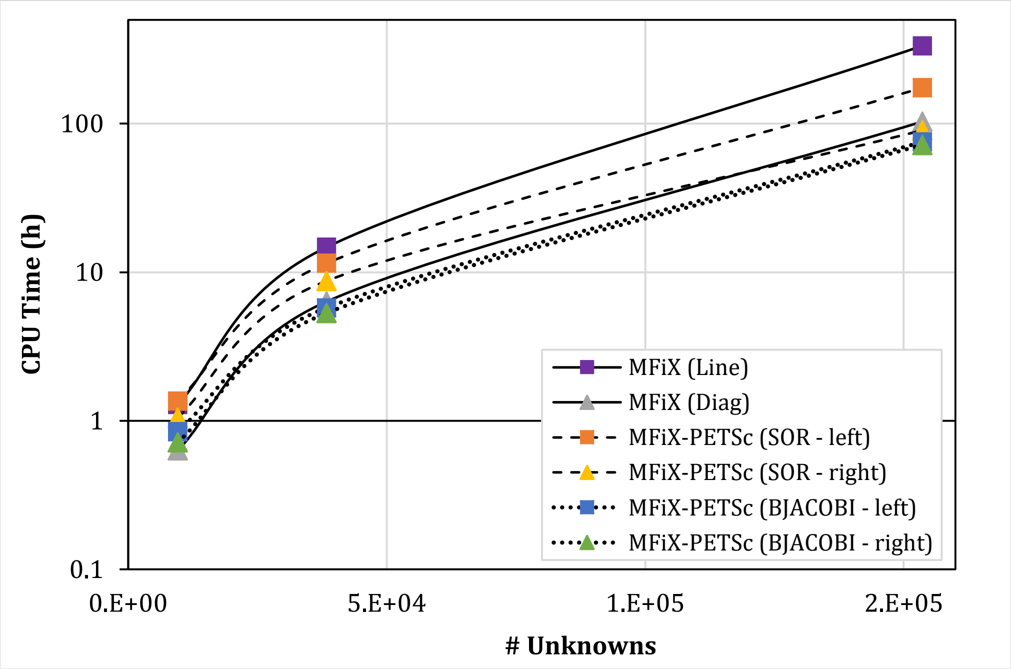


Figure 5-10. CPU time as a function of problem size for Case 2 using the Superbee discretization scheme.

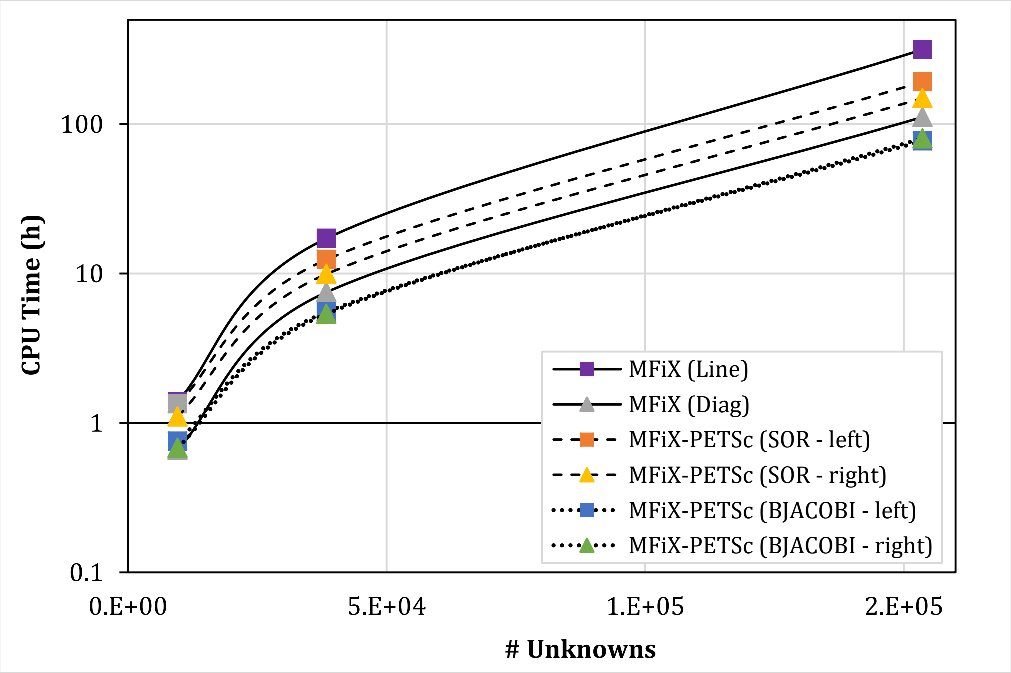
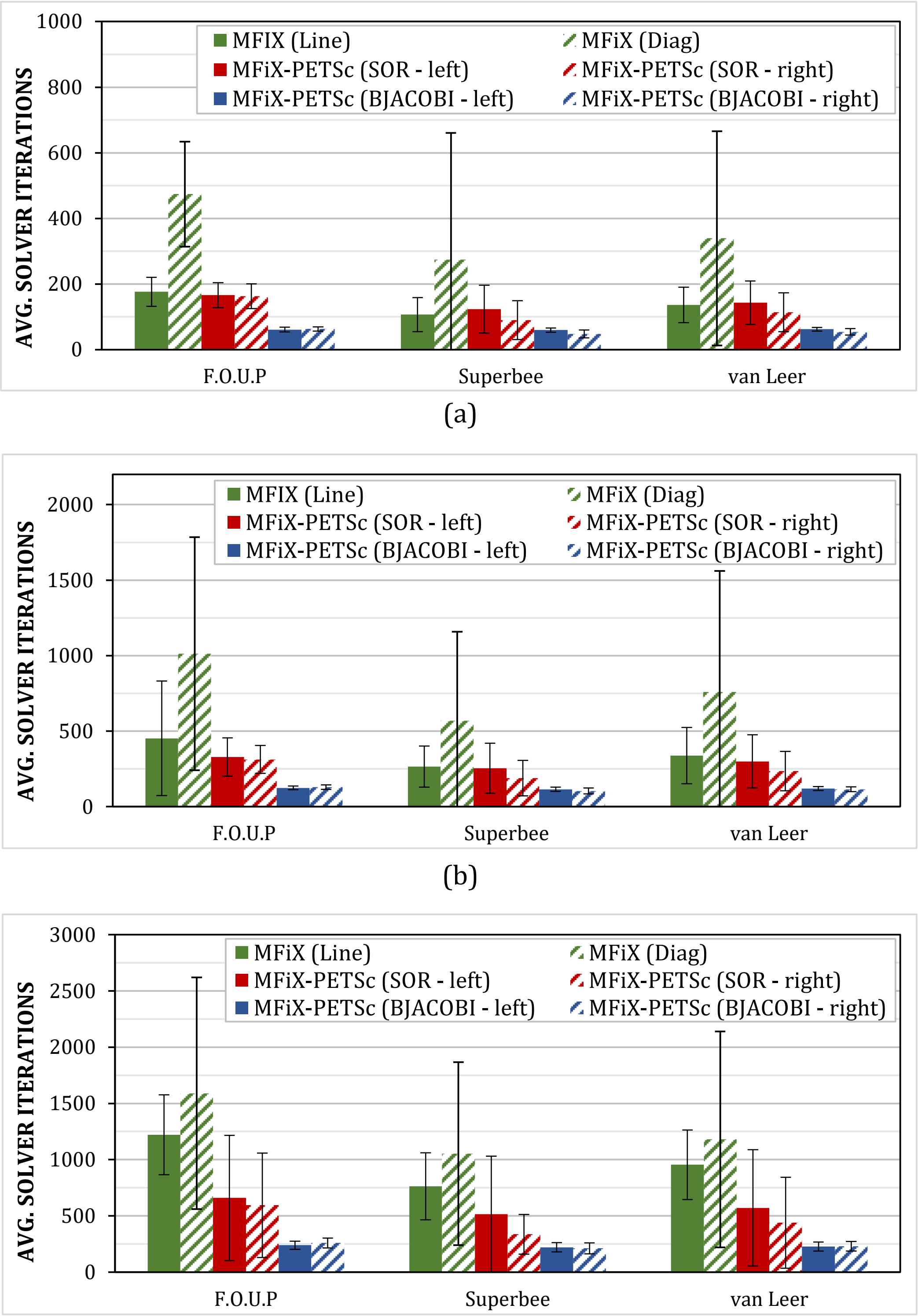


Figure 5-11. CPU time as a function of problem size for Case 2 using the van Leer discretization scheme.



(a)

(b)

(c)

Figure 5-12. Average pressure solver iterations required throughout Cases (a) 2.1, (b) 2.2, and (c) 2.3 (with standard deviations).

CHAPTER 6

3D Fluidized Bed

6.1 Problem Overview

For the third and fourth cases presented in Chapter 6, MFiX-PETSc was used to solve the pressure-correction equation for a three-dimensional, rectangular fluidized bed operated with a central jet. The geometry of the bed and simulation settings were based on a previous study performed by Utikar and Ranade [21]. Figure 6-1 shows the dimensions of the bed and central jet used to carry out this study. The 20x100x2 cm^3 bed geometry was broken down into a 40x250x10 mesh to carry out simulations.

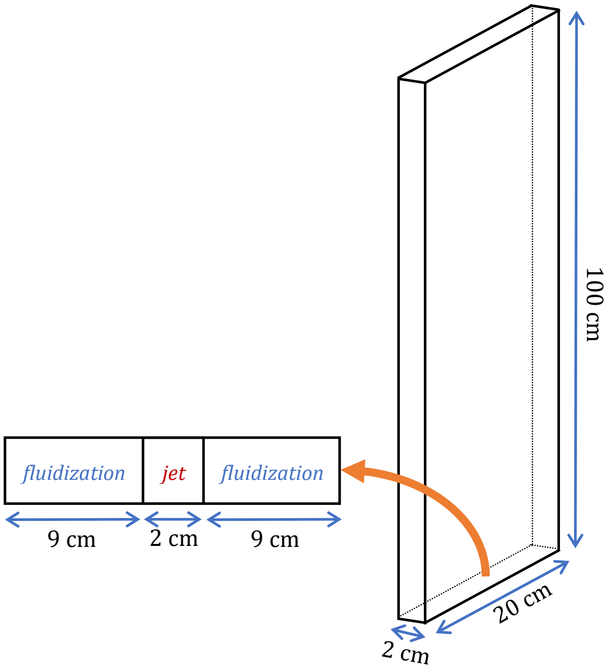


Figure 6-1. Dimensions and central jet location of the 3D rectangular fluidized bed.

The bed was filled with spherical beads at a height to diameter ratio (H/D) of 1. Beads of two different materials, glass (Case 3) and polypropylene (Case 4), were tested separately. Additionally, two different air inlets were used: air through the central jet at velocities of 5 and 20 m/s and air at the minimum fluidization velocity from the bottom of the bed. The air was assumed to have constant density of 1.2 kg/m^3 and viscosity of 1.8x10^{-5} Pa·s.

6.2 Results

6.2.1 Glass Particles

In Case 3, the fluidized bed was filled with glass particles at a H/D of 1. Table 6-1 contains the material properties used to represent the glass particles.

Table 6-1. List of material properties used for glass particles throughout Case 3.

Glass Particle Properties	
Density (kg/m^3)	2545
Coefficient of Restitution	0.9
Angle of Internal Friction	30°
Diameter (μm)	425
Minimum Fluidization Velocity (m/s)	0.30

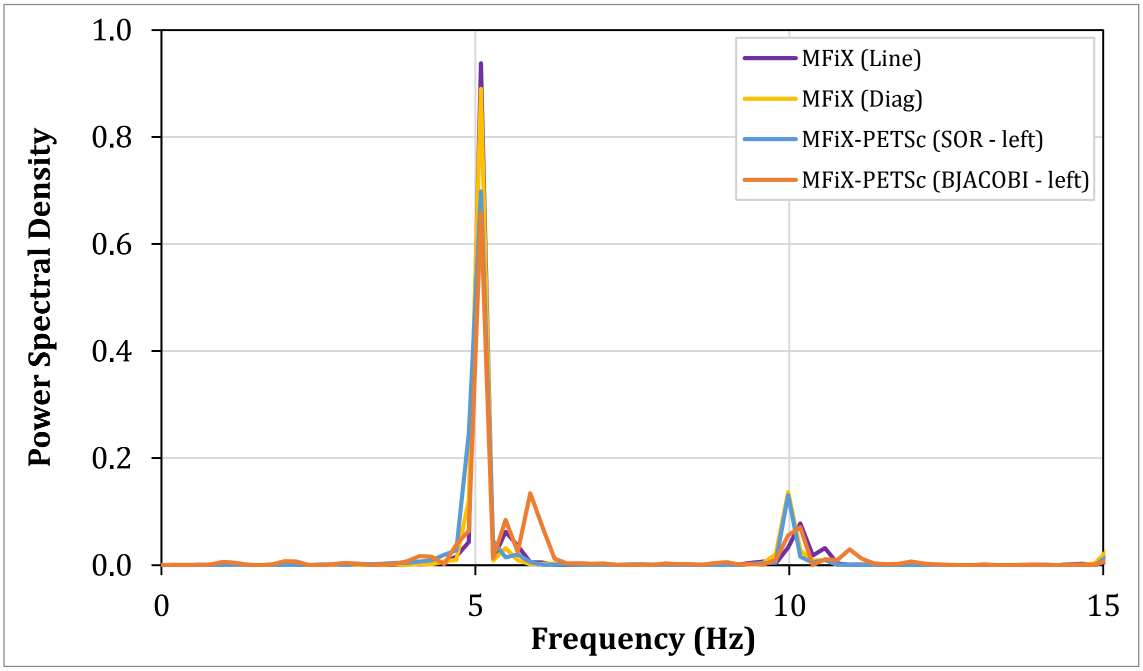
Similar to Case 2, the line relaxation and diagonal scaling preconditioners in native MFiX, and the Block Jacobi and SOR preconditioners in MFiX-PETSc were applied to the BiCGSTAB solver in Case 3. Block Jacobi and SOR were applied from both the left- and right-hand side in MFiX-PETSc. Throughout this case, three different tolerance combinations were tested: an outer tolerance of 10^{-1} with a solver tolerance of 10^{-1}, an outer tolerance of 10^{-1} with a solver tolerance of 10^{-3}, and an outer tolerance of 10^{-3} with a solver tolerance of 10^{-3}. Additionally, both the F.O.U.P. and van Leer discretization schemes were used. Table 6-2 summarizes the various settings applied to different simulations for Case 3. The mfix.dat input file for this case can be found in the Appendix.

Table 6-2. Summary of grid dimensions, meshing, time steps, tolerances, inlet velocities, solvers, discretization schemes, and preconditioners (P.C.) employed in Case 3.

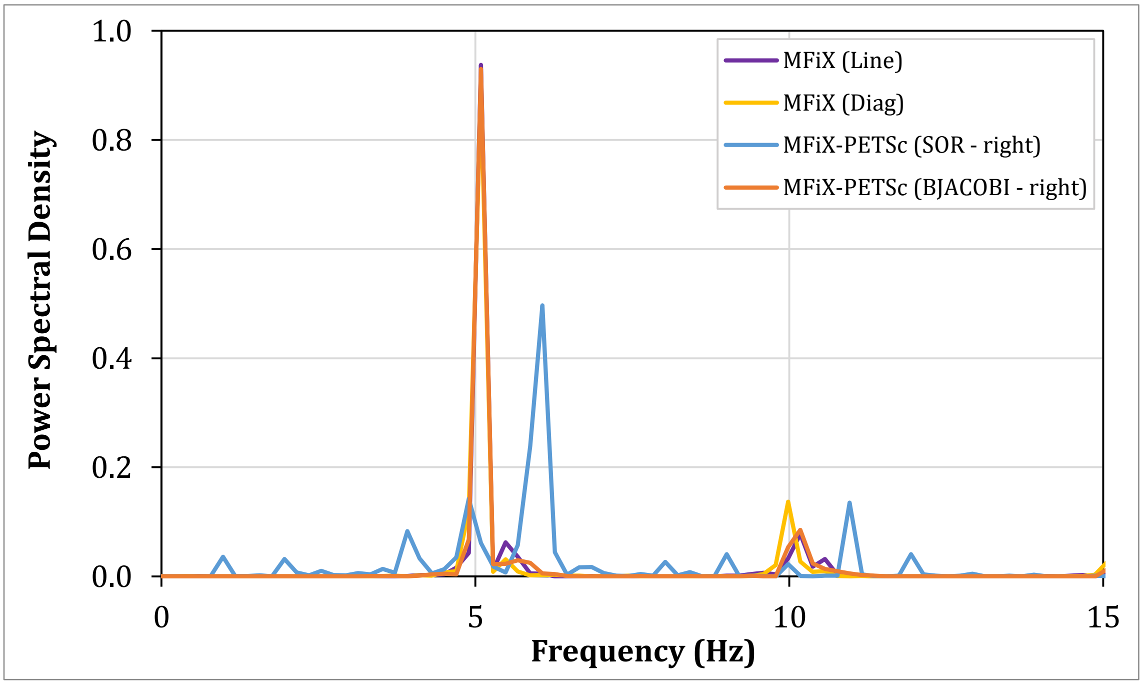
Case 3: 3D Fluidized Bed with Glass Particles							
Case	**Dimensions + Mesh**	**Time Step**	**Tolerance**	**U_{in}**	**Solver**	**Scheme**	**P.C.**
3.1	20x100x2 cm^3 40x250x10	Max: 10^{-1} Min: 10^{-6}	Outer: 10^{-1} Solver: 10^{-1}	5 m/s	BCGS	F.O.U.P.	1. MFiX Line 2. MFiX Diag 3. PETSc SOR (left) 4. PETSc SOR (right) 5. PETSc BJACOBI (left) 6. PETSc BJACOBI (right)
3.2	20x100x2 cm^3 40x250x10	Max: 10^{-1} Min: 10^{-6}	Outer: 10^{-1} Solver: 10^{-3}	5 m/s	BCGS	F.O.U.P.	1. MFiX Line 2. MFiX Diag 3. PETSc SOR (left) 4. PETSc SOR (right) 5. PETSc BJACOBI (left) 6. PETSc BJACOBI (right)
3.3	20x100x2 cm^3 40x250x10	Max: 10^{-1} Min: 10^{-6}	Outer: 10^{-3} Solver: 10^{-3}	5 m/s	BCGS	F.O.U.P.	1. MFiX Line 2. MFiX Diag 3. PETSc SOR (left) 4. PETSc SOR (right) 5. PETSc BJACOBI (left) 6. PETSc BJACOBI (right)
3.4	20x100x2 cm^3 40x250x10	Max: 10^{-1} Min: 10^{-6}	Outer: 10^{-1} Solver: 10^{-1}	5 m/s	BCGS	van Leer	1. MFiX Line 2. MFiX Diag 3. PETSc SOR (left) 4. PETSc SOR (right) 5. PETSc BJACOBI (left) 6. PETSc BJACOBI (right)
3.5	20x100x2 cm^3 40x250x10	Max: 10^{-1} Min: 10^{-6}	Outer: 10^{-1} Solver: 10^{-3}	5 m/s	BCGS	van Leer	1. MFiX Line 2. MFiX Diag 3. PETSc SOR (left) 4. PETSc SOR (right) 5. PETSc BJACOBI (left) 6. PETSc BJACOBI (right)
3.6	20x100x2 cm^3 40x250x10	Max: 10^{-1} Min: 10^{-6}	Outer: 10^{-3} Solver: 10^{-3}	5 m/s	BCGS	van Leer	1. MFiX Line 2. MFiX Diag 3. PETSc SOR (left) 4. PETSc SOR (right) 5. PETSc BJACOBI (left) 6. PETSc BJACOBI (right)
3.7	20x100x2 cm^3 40x250x10	Max: 10^{-1} Min: 10^{-6}	Outer: 10^{-1} Solver: 10^{-3}	20 m/s	BCGS	van Leer	1. MFiX Diag 2. PETSc BJACOBI (left) 3. PETSc BJACOBI (right)

Pressure fluctuations were measured at a bed height of 15 mm above the central jet. The pressure data was then normalized after 2 seconds of simulation time by using the standard deviation of the fluctuations. Estimates of the power spectral density (PSD) of the normalized pressure fluctuations were made using the Fast Fourier Transform (FFT) method. The power spectra predicted by MFiX-PETSc were compared against native MFiX for each case. Figures 6-2 (a, b), 6-3 (a, b), and 6-4 (a, b) show the PSD plots for Cases 3.1, 3.2, and 3.3 respectively. All three cases employed a 5 m/s central jet and F.O.U.P. discretization.

When both the outer and solver tolerance were 10^{-1} (Case 3.1), all preconditioners besides right-side SOR followed similar trends and predicted a dominant frequency of 5.1 Hz. As the solver tolerance was decreased to 10^{-3} in Case 3.2, two main peaks started to appear at approximately 5.1 and 5.9 Hz. Additionally, results obtained with all preconditioning methods agreed well. Finally, Case 3.3 refined the outer tolerance to match the solver tolerance at 10^{-3}. The power spectra still exhibited two peaks; however, the PSD trends captured using these different preconditioners were not as similar compared to Case 3.2, which employed a higher outer tolerance. This observation can be explained by looking at the pressure fluctuations. The pressure data captured between 2- 10 seconds for each of the preconditioners in Case 3.3 is shown in Figure 6-5. These images indicate that the bed was undergoing a transition between fluidization regimes. Therefore, simulations should have been run longer than 10 seconds for a lower outer tolerance level, as was used in Case 3.3. Overall, power spectra predicted by MFiX-PETSc agreed fairly well with native MFiX considering the complexity of the case compared with the heat conduction and 2D flow problems.

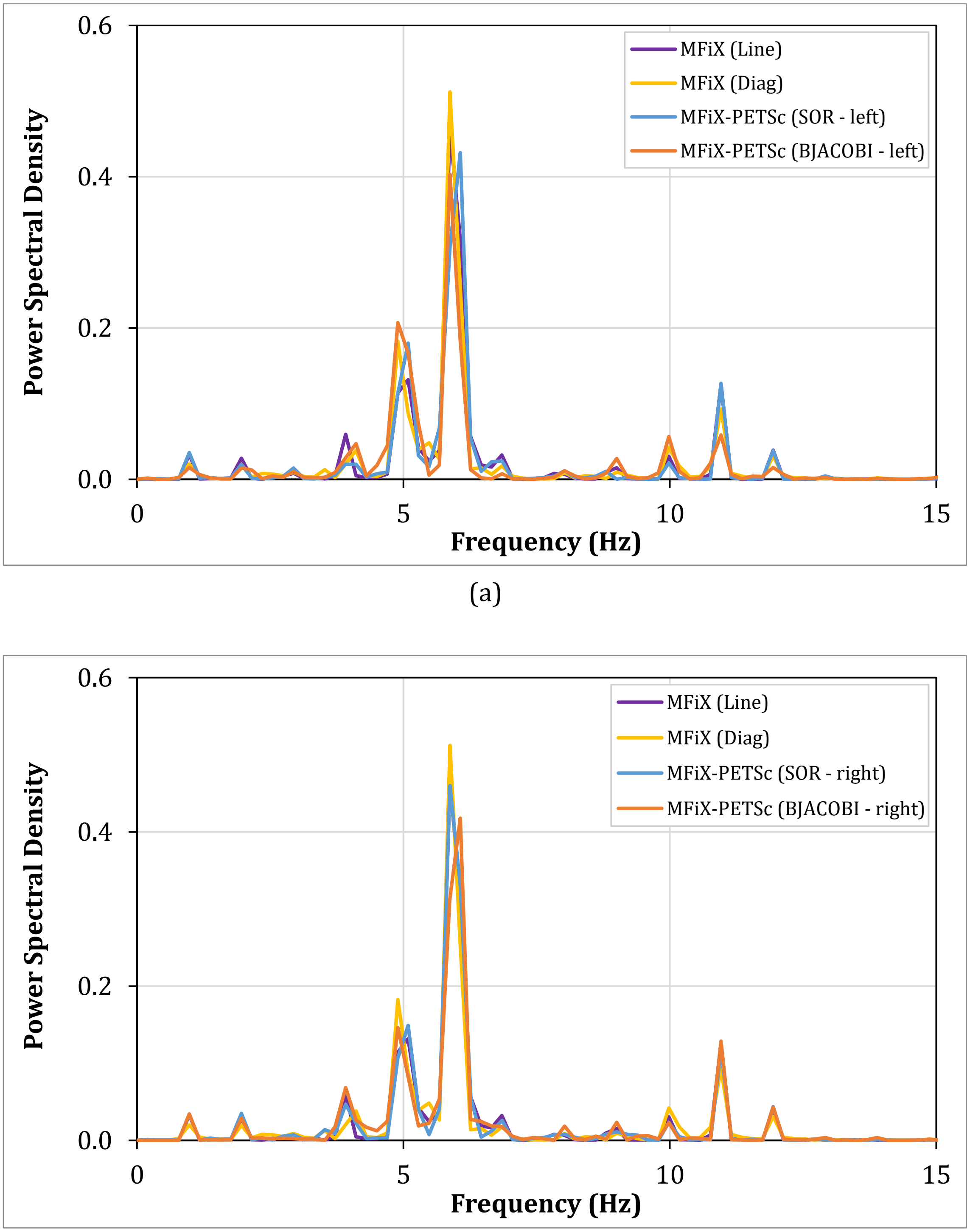


(a)



(b)

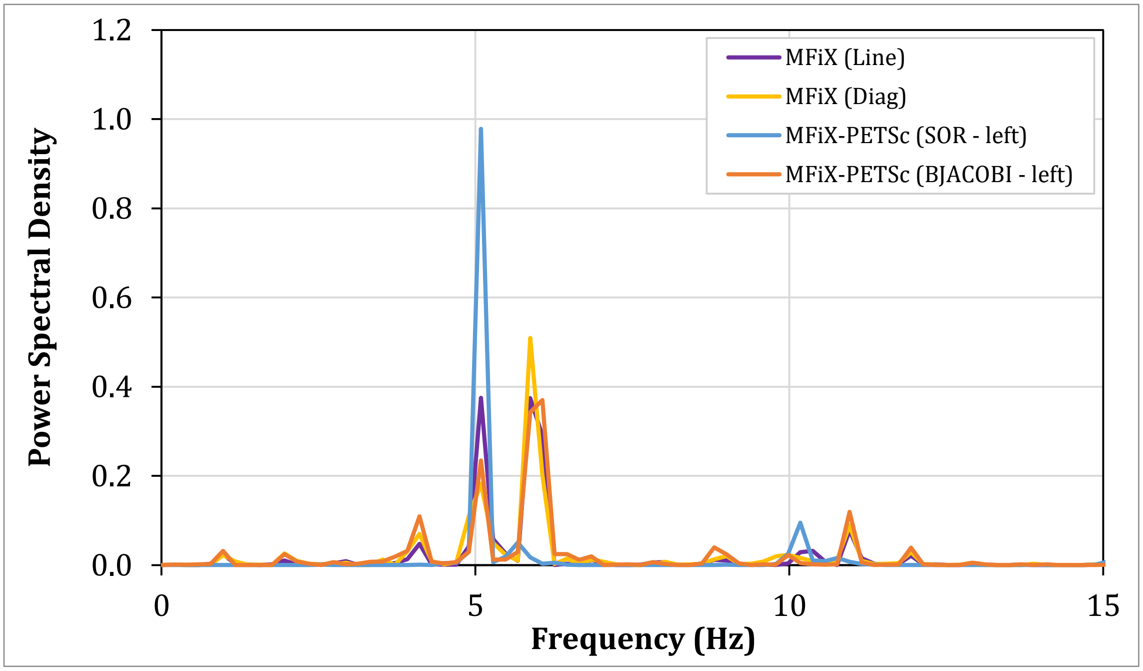
Figure 6-2. Comparison of power spectra obtained using (a) left and (b) right preconditioning in MFiX-PETSc against native MFiX for an outer tolerance of 10^{-1}, a solver tolerance of 10^{-1}, and F.O.U.P. discretization (Case 3.1).



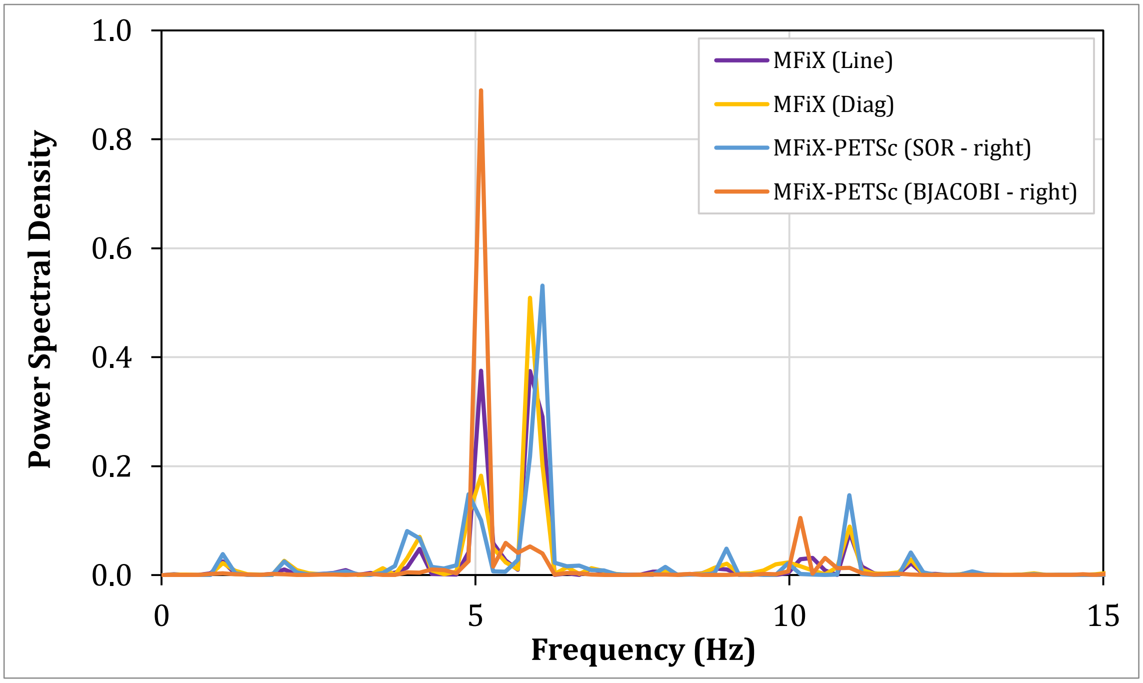
(a)

(b)

Figure 6-3. Comparison of power spectra obtained using (a) left and (b) right preconditioning in MFiX-PETSc against native MFiX for an outer tolerance of 10^{-1}, a solver tolerance of 10^{-3}, and F.O.U.P. discretization (Case 3.2).



(a)



(b)

Figure 6-4. Comparison of power spectra obtained using (a) left and (b) right preconditioning in MFiX-PETSc against native MFiX for an outer tolerance of 10^{-3}, a solver tolerance of 10^{-3}, and F.O.U.P. discretization (Case 3.3).

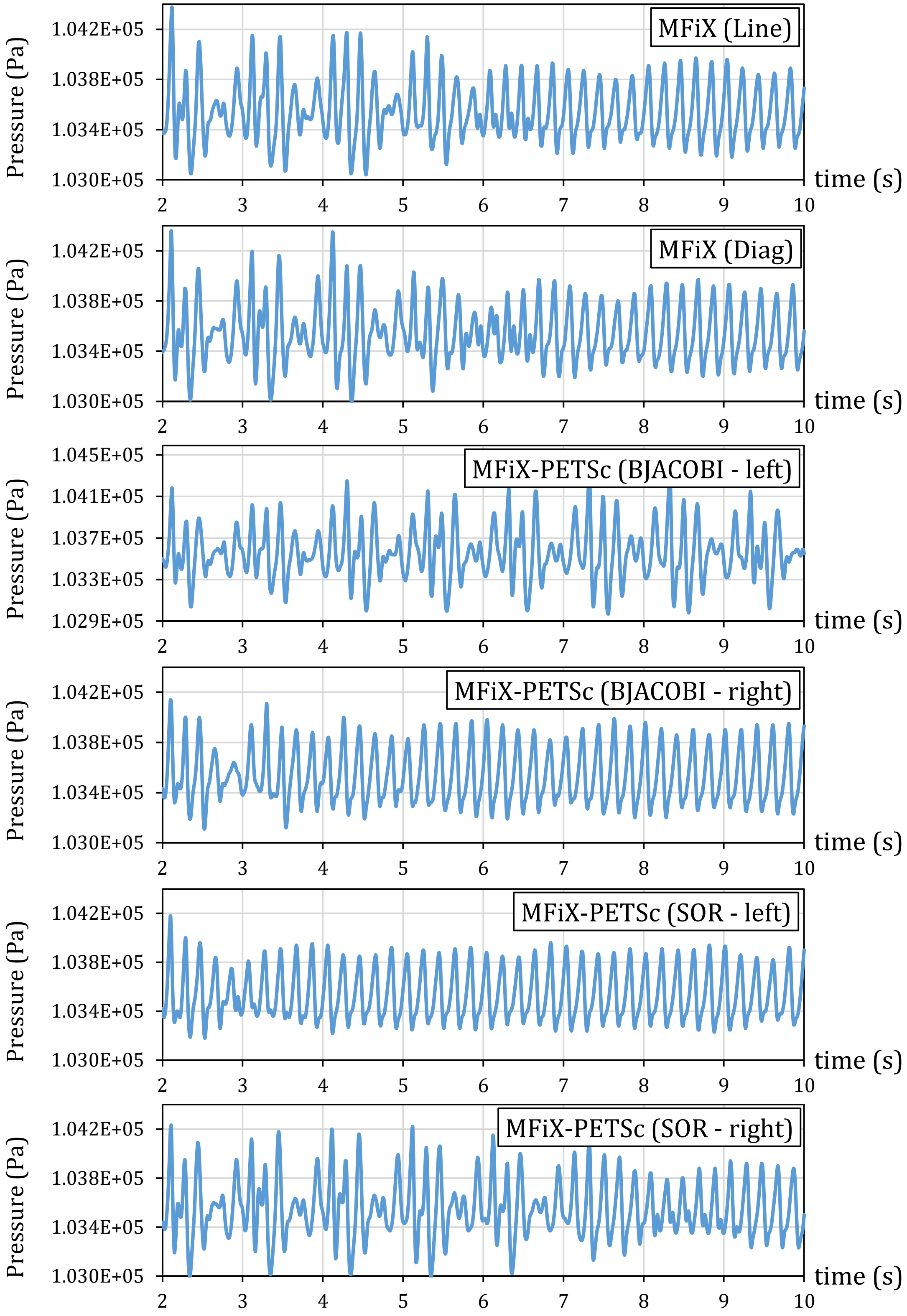
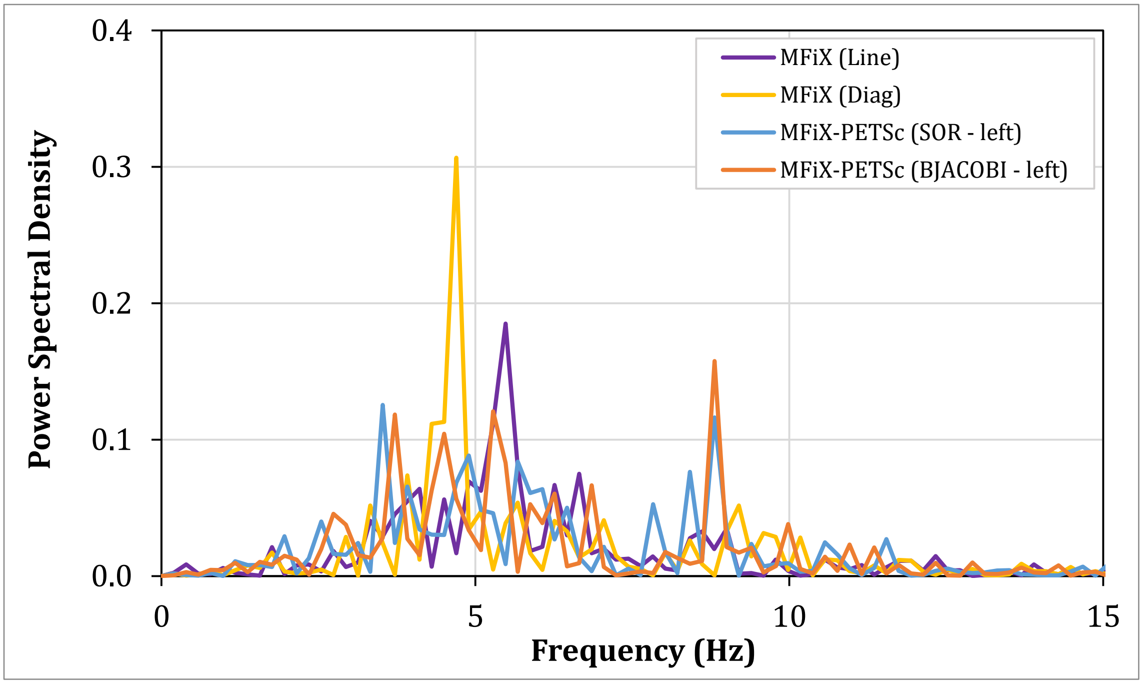


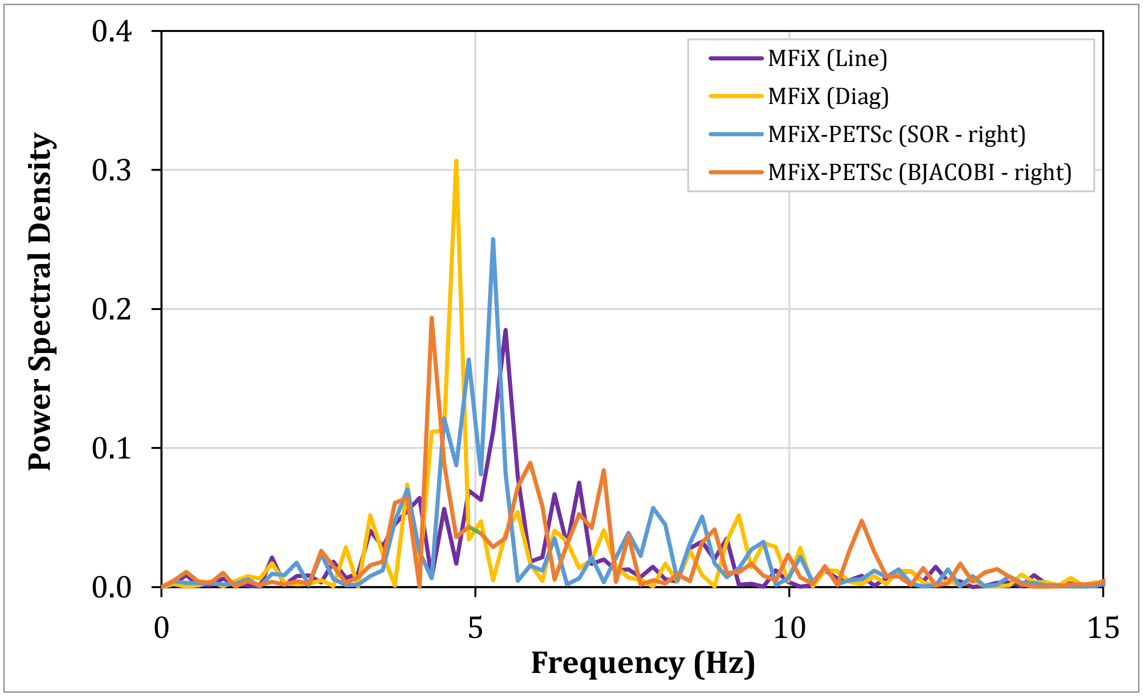
Figure 6-5. Comparison of pressure fluctuations for different preconditioners employed in Case 3.3 which indicates a fluidization regime transition occurred.

Figures 6-6 (a, b), 6-7 (a, b), and 6-8 (a, b) show the PSD plots for Cases 3.4, 3.5, and 3.6 respectively. These simulations employed a 5 m/s central jet and van Leer discretization. Looking at the power spectra of these cases, it is evident that the PSD trends are less smooth than those observed for F.O.U.P. discretization. For Case 3.4, which uses an outer and solver tolerance of 10^{-1}, the dominant frequency predicted using left-side Block Jacobi was 8.8 Hz. This value is significantly higher than the peak frequencies achieved with the other preconditioning techniques, which ranged from 3.5 – 5.5 Hz. In Case 3.5, when the solver tolerance was reduced to 10^{-3}, the dominant frequency predicted with left-side Block Jacobi became more reasonable. Overall, peak frequencies of these simulations in Case 3.5 ranged from 4.3 - 6.1 Hz. Lastly, the outer tolerance was decreased to 10^{-3} in Case 3.6. The trends of the power spectra were similar to Cases 3.4 and 3.5 in that there was a lot of noise in the data and there was a range of dominant frequencies, which was 3.7 – 6.1 Hz for this specific case.

To gain insight into why the PSD plots were less smooth, Figures 6-9 shows an example comparison of the pressure fluctuations captured using F.O.U.P. against van Leer discretization using the same tolerance levels and preconditioning method. While a pattern develops after about 6 seconds with F.O.U.P. discretization, there is no evident pattern in the fluctuations when van Leer is used. This indicates that simulations might have needed to be run longer than 10 seconds for a pattern to be observed. For example, pressure signals were recorded for 200 seconds in the Utikar and Ranade [21] experiments. However, since these cases were run in a serial fashion, the computation time needed to run simulations for more than 10 seconds would have been too large for the purposes of this study.

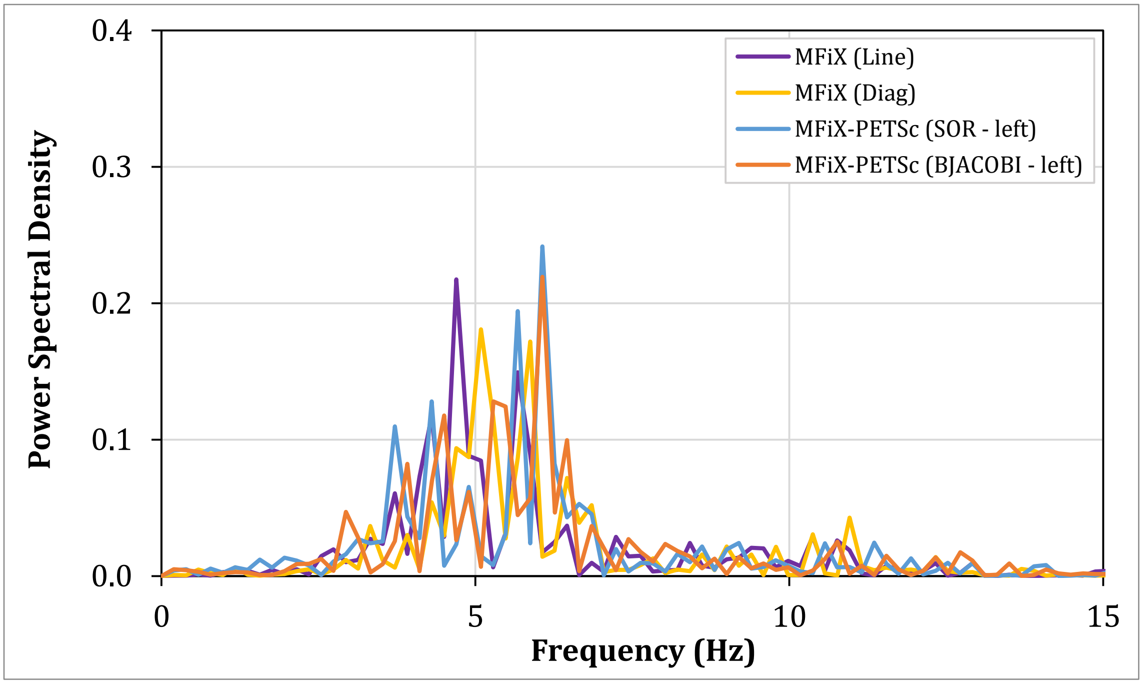


(a)

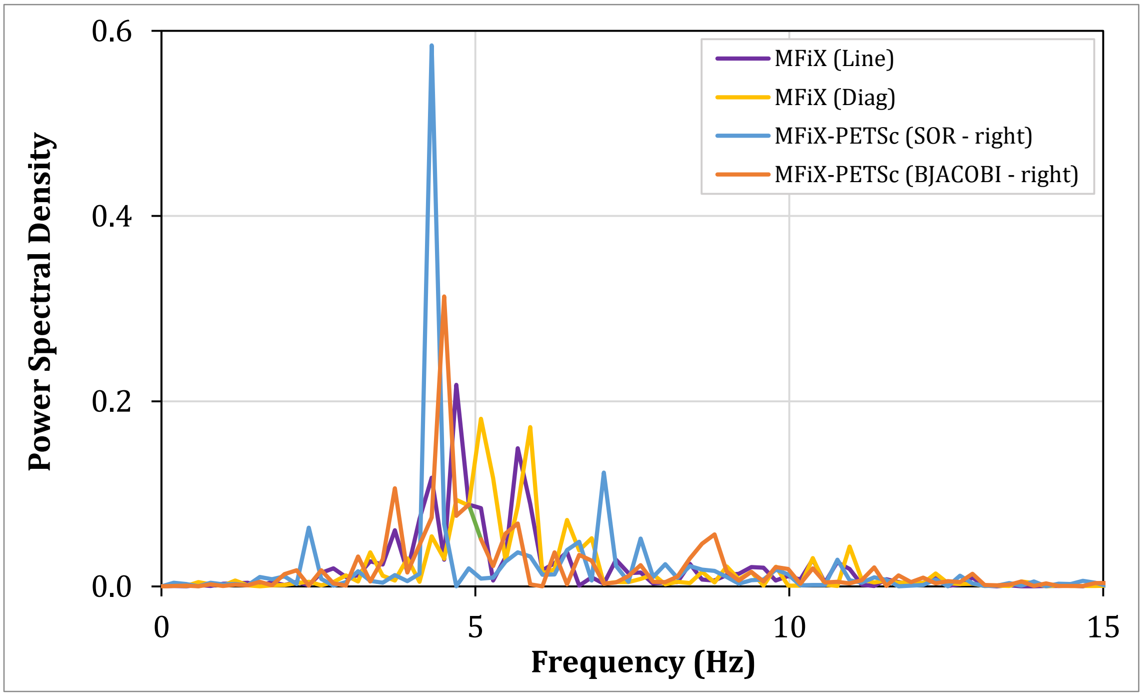


(b)

Figure 6-6. Comparison of power spectra obtained using (a) left and (b) right preconditioning in MFiX-PETSc against native MFiX for an outer tolerance of 10^{-1}, a solver tolerance of 10^{-1}, and van Leer discretization (Case 3.4).

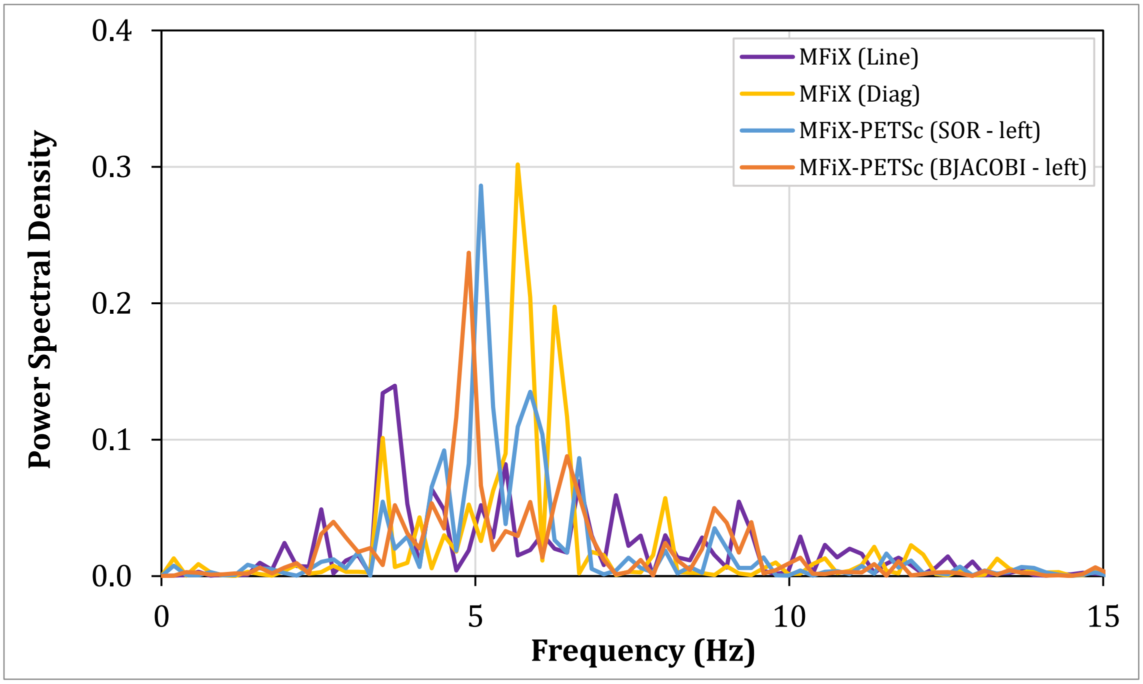


(a)

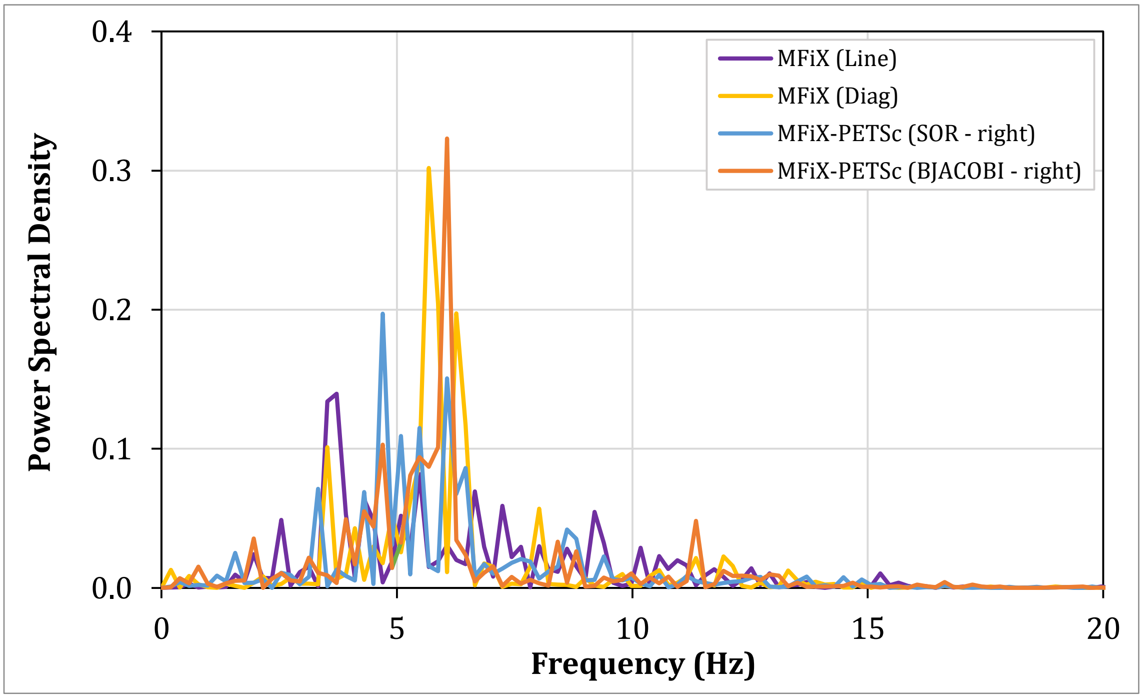


(b)

Figure 6-7. Comparison of power spectra obtained using (a) left and (b) right preconditioning in MFiX-PETSc against native MFiX for an outer tolerance of 10^{-1}, a solver tolerance of 10^{-3}, and van Leer discretization (Case 3.5).



(a)



(b)

Figure 6-8. Comparison of power spectra obtained using (a) left and (b) right preconditioning in MFiX-PETSc against native MFiX for an outer tolerance of 10^{-3}, a solver tolerance of 10^{-3}, and van Leer discretization (Case 3.6).

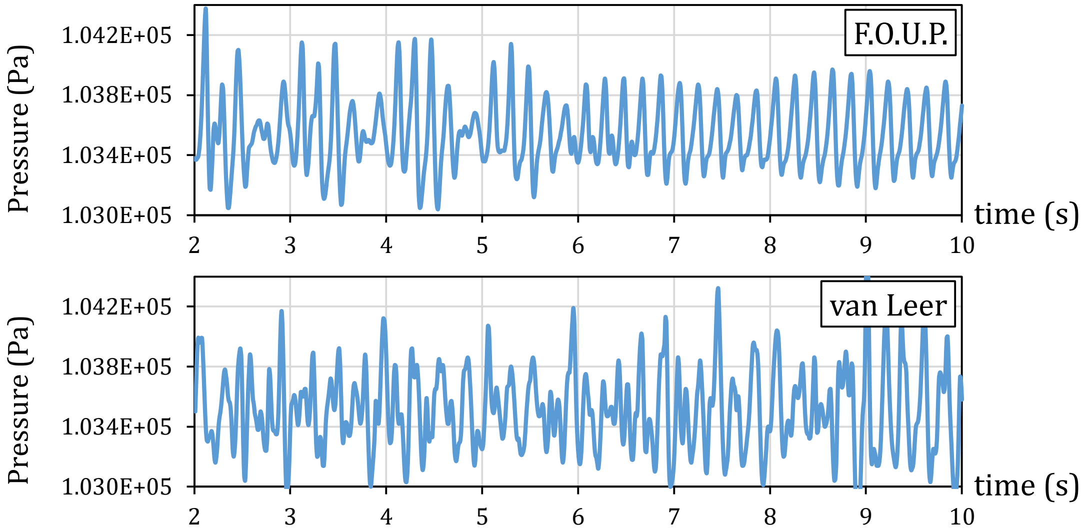
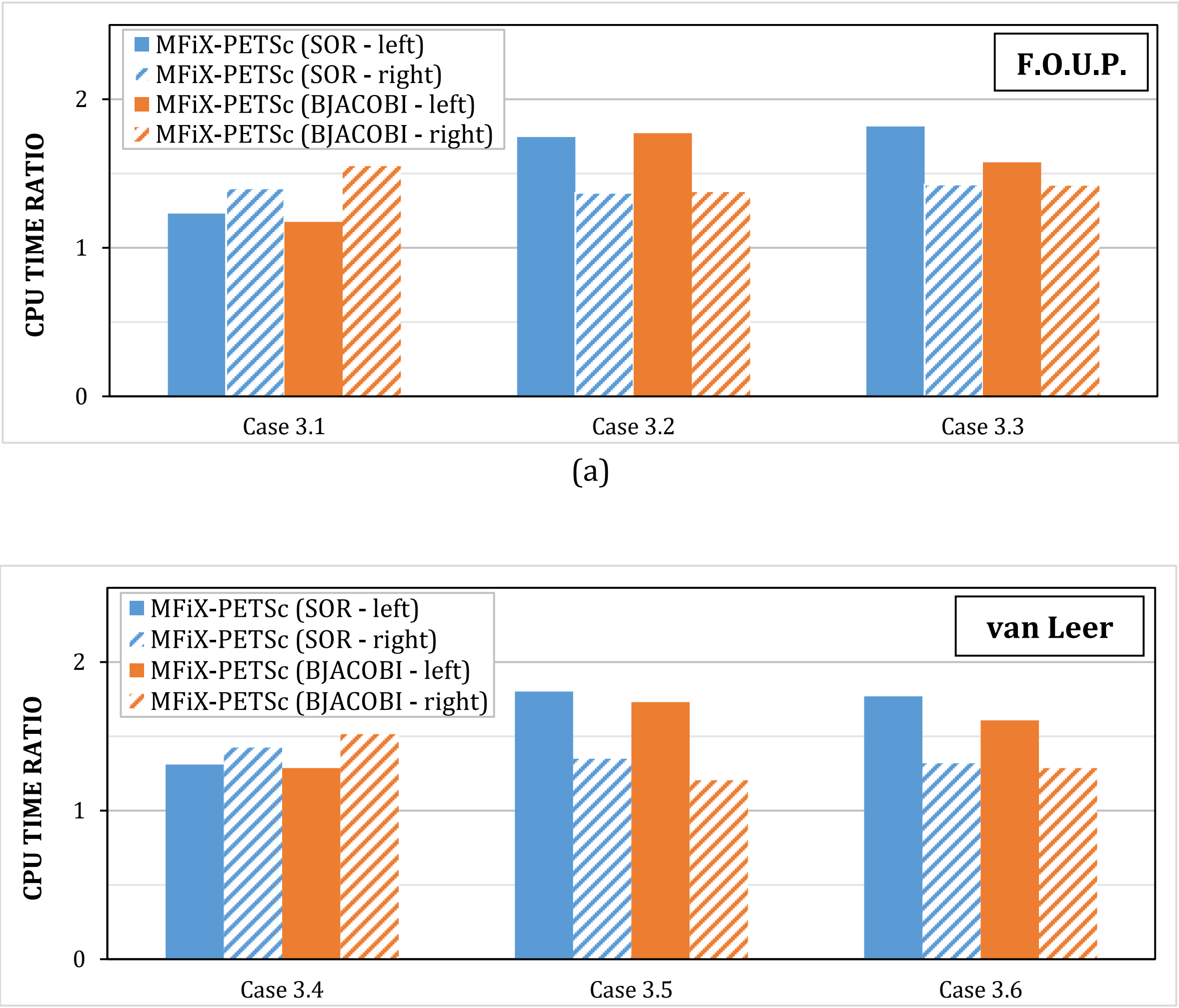


Figure 6-9. An example of the pressure fluctuations obtained with F.O.U.P. versus van Leer discretization schemes using the same tolerance levels and preconditioning method.

Figures 6-10 (a, b) compare the CPU time ratios (MFiX-PETSc CPU time / MFiX CPU time) for the F.O.U.P. and van Leer discretization cases respectively. In computing these ratios, the CPU time utilizing MFiX's default line preconditioning method was used since it is the default method. The CPU times achieved with the diagonal scaling and line relaxation preconditioners were similar; therefore, the choice did not significantly affect the time ratios. For all simulations employing a central jet of 5 m/s (Cases 3.1 – 3.6), native MFiX was faster than MFiX-PETSc, resulting in a CPU time ratio greater than one. However, this case still provided insight into the performance of MFiX-PETSc's preconditioners.

For a solver tolerance of 10^{-1}, which was applied in Case 3.1 (F.O.U.P.) and Case 3.4 (van Leer), preconditioning from the left was faster for both SOR and Block Jacobi. As the solver tolerance was tightened to 10^{-3} in Cases 3.2 and 3.3 (F.O.U.P.) and Cases 3.5 and 3.6 (van Leer), right preconditioning became significantly more efficient. With F.O.U.P. discretization and a lower solver tolerance (10^{-3}), right-side SOR preconditioning was 37-

43% slower than MFiX's line relaxation preconditioner. Similarly, right-side Block Jacobi preconditioning was 38-42% slower. With a higher-order discretization scheme (van Leer) at this low tolerance level, right-side SOR was 32-35% slower than MFiX's line relaxation, while right-side Block Jacobi was 21-29% slower. Therefore, at a low solver tolerance level, the timing differences between MFiX-PETSc and native MFiX were minimized by employing right-side Block Jacobi with a higher-order discretization scheme.

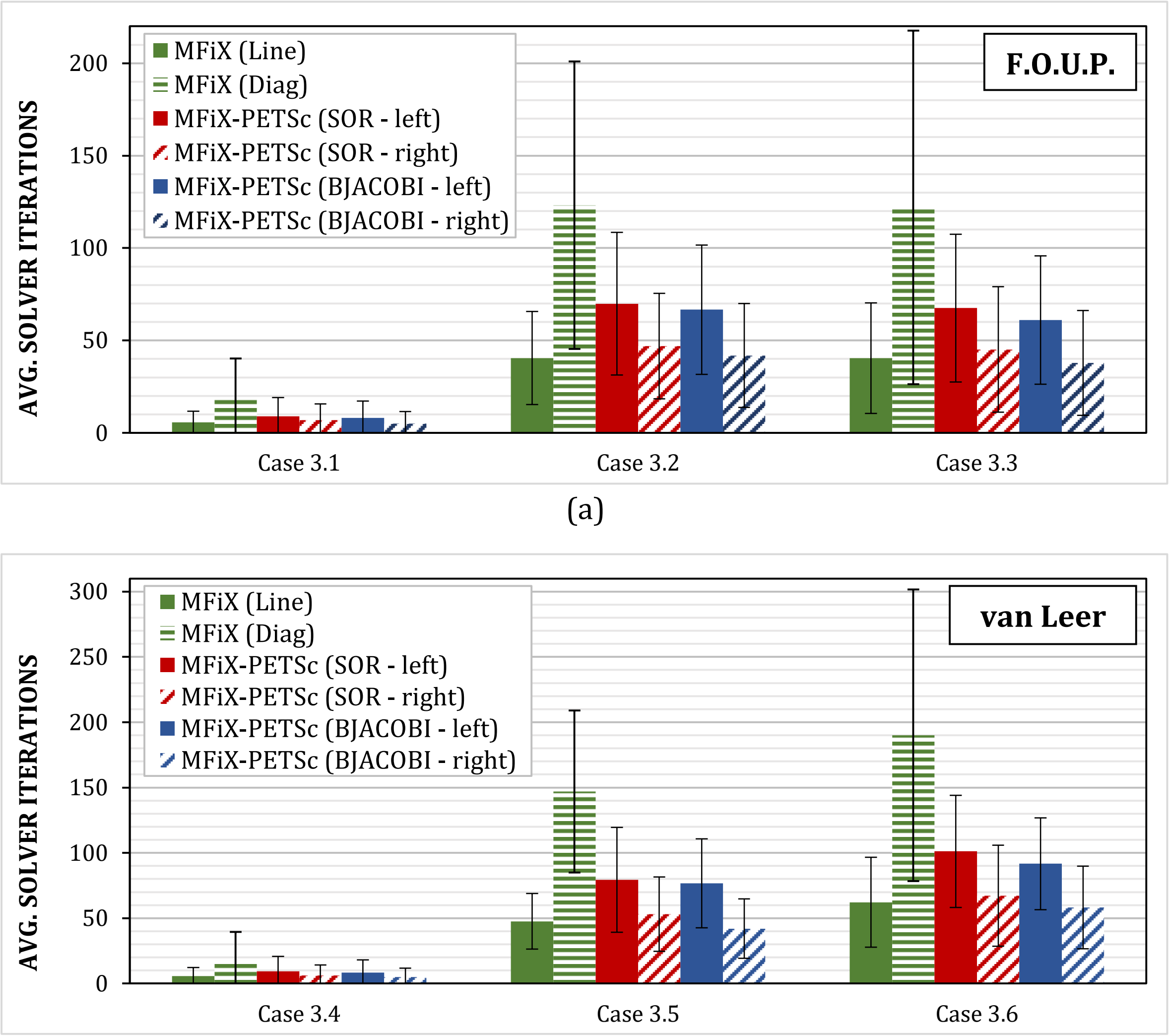


(a)

(b)

Figure 6-10. CPU time ratios (MFiX-PETSc CPU time / native MFiX CPU time) for (a) F.O.U.P. and (b) van Leer discretization schemes employed throughout Cases 3.1 - 3.6**.**

Figures 6-11 (a, b) show the average solver iterations (with standard deviations) required to solve the pressure-correction equation for the F.O.U.P. and van Leer cases. An important trend shown in these figures is that for the cases with a lower solver tolerance (Cases 3.2, 3.3, 3.4, and 3.5), right-side preconditioning results in fewer average iterations, which correlates with the timing results portrayed throughout Figure 6-10.



(a)

(b)

Figure 6-11. Comparison of the average iterations (with standard deviations) required by each preconditioner-solver combination to solve the pressure-correction equation when (a) F.O.U.P. and (b) van Leer discretization schemes were employed throughout Cases 3.1 – 3.6.

Since Block Jacobi was shown to be a more efficient preconditioning technique compared to SOR, specifically for a higher-order discretization scheme, it was further tested with an inlet velocity of 20 m/s in Case 3.7. For these simulations, van Leer was used as the discretization scheme, the outer tolerance was 10^{-1}, and the solver tolerance was 10^{-3}. Then, the PSD, timing, and iteration results achieved with right- and left-side Block Jacobi were compared against native MFiX. At this increased velocity, line relaxation did not converge well using these settings; therefore, diagonal scaling was used as the preconditioner in MFiX for Case 3.7. Figure 6-12 shows the power spectra for Case 3.7 obtained with both MFiX and MFiX-PETSc. At this higher inlet velocity, the PSD results using van Leer discretization were more smooth compared to a 5 m/s inlet velocity. It can clearly be seen that the power spectra trends captured by using both left-side and right-side Block Jacobi preconditioning are in agreement with MFiX's diagonal scaling.

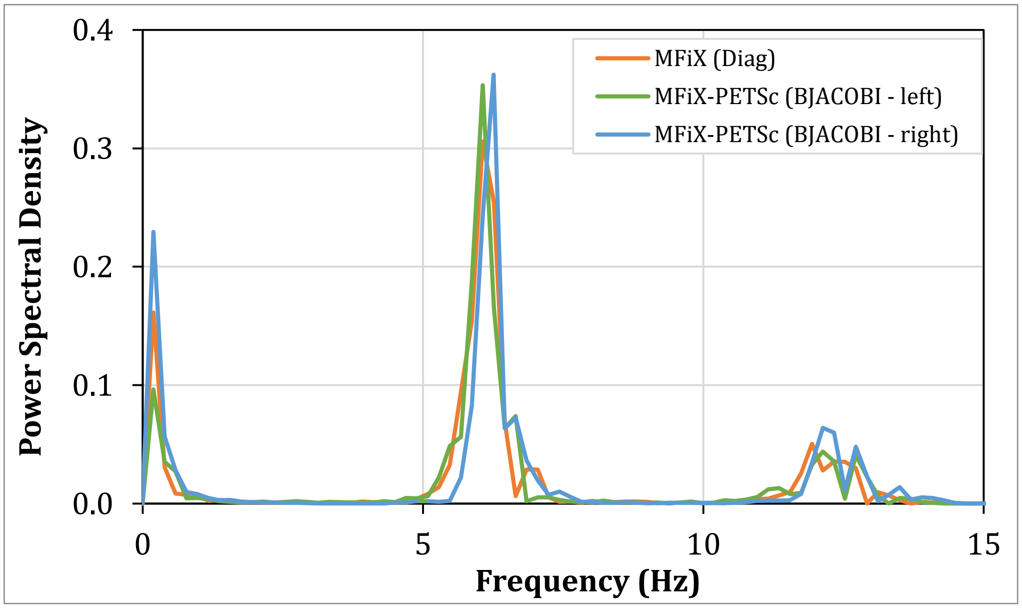
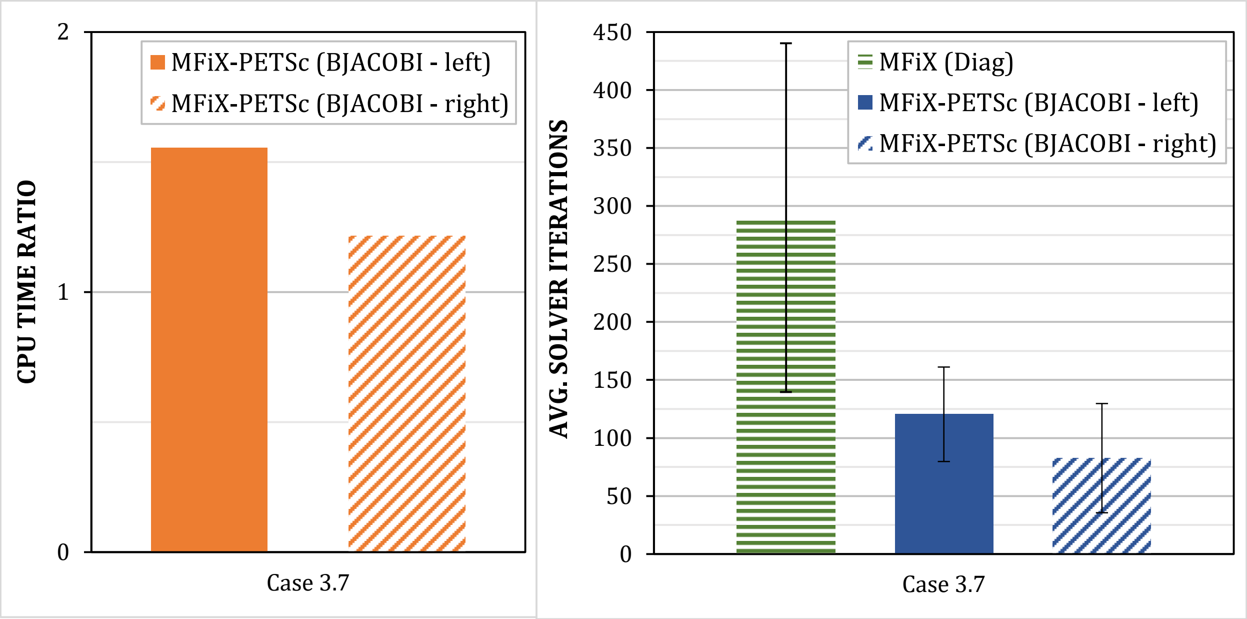


Figure 6-12. Comparison of power spectra obtained for Case 3.7, which used a 20 m/s jet.

The CPU time ratios (MFiX-PETSc CPU time / MFiX CPU time) are shown in Figure 6-13 (a) for Case 3.7. With a higher inlet velocity of 20 m/s, the CPU time ratios were greater than one showing that native MFiX was still faster than the MFiX-PETSc solver. Applying Block Jacobi from the right-side was significantly faster than from the left, which supports previous results found with an inlet velocity of 5 m/s. Overall, using right-side Block Jacobi resulted in a simulation that was 22% slower than using MFiX's diagonal scaling preconditioner. Therefore, inlet velocity did not significantly affect the timing difference between MFiX and MFiX-PETSc. Figure 6-13 (b) shows the average iterations (with standard deviations) required to solve the pressure-correction equation for each preconditioner tested in Case 3.7. Right-side Block Jacobi preconditioning resulted in a lower number of iterations required, which correlates with the CPU times shown in Figure 6-13 (a).



(a) (b)

Figure 6-13. The (a) CPU time ratios and (b) average solver iterations with standard deviations for Case 3.7, which used a 20 m/s jet.

6.2.2 Polypropylene Particles

Case 4 used polypropylene beads in the 3D fluidized bed with a H/D of 1. Table 6-3 lists the material properties that were used for polypropylene.

Table 6-3. List of material properties used for polypropylene particles throughout Case 4.

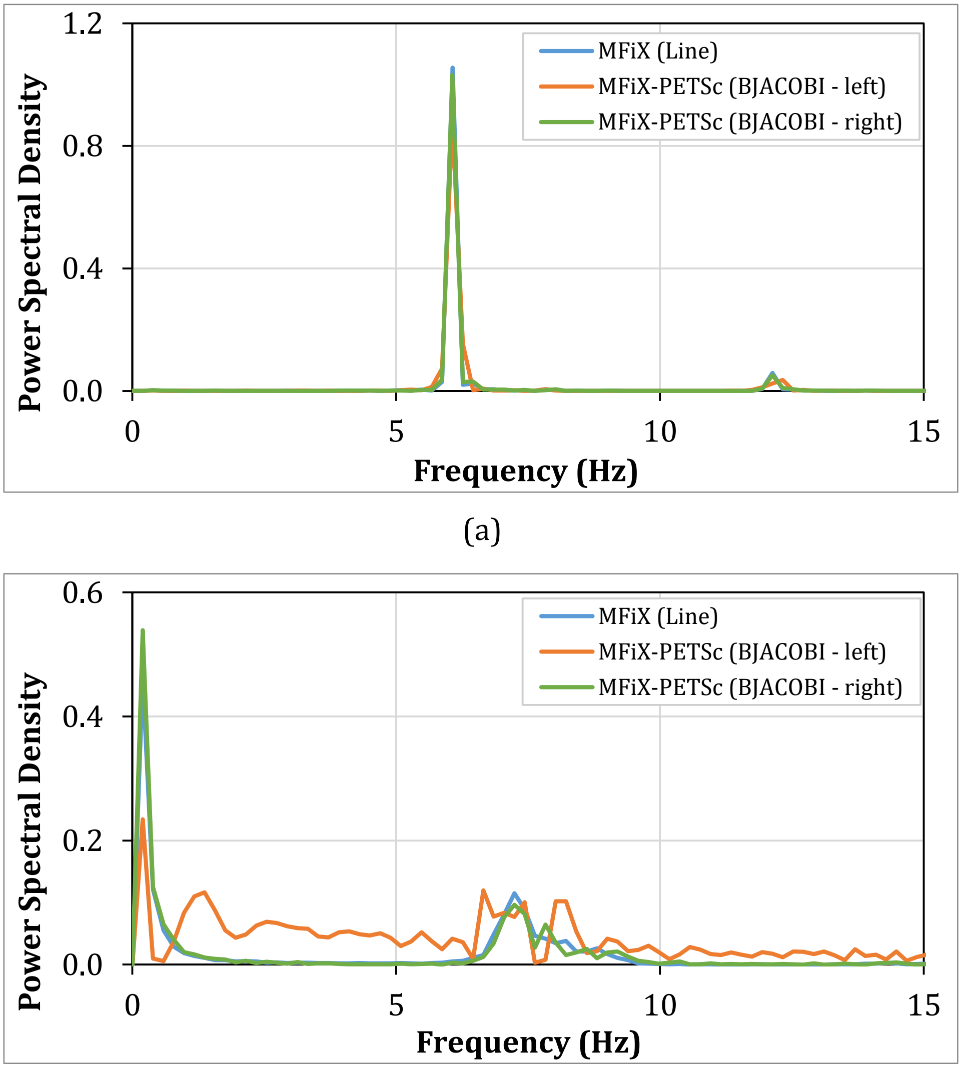
Polypropylene Particle Properties	
Density (kg/m^3)	900
Coefficient of Restitution	0.6
Angle of Internal Friction	30°
Diameter (μm)	425
Minimum Fluidization Velocity (m/s)	0.11

In Case 3, it was shown that Block Jacobi was faster than SOR for a higher-order discretization scheme. Furthermore, line relaxation and diagonal scaling in MFiX were similar in terms of efficiency. Due to these findings, the number of simulations carried out in Case 4 was reduced by only looking at line relaxation in MFiX and Block Jacobi (left- and right-side) in MFiX-PETSc. These preconditioners were applied with van Leer discretization, an outer tolerance of 10^{-1}, and a solver tolerance of 10^{-3}. Additionally, inlet velocities of 5 and 20 m/s were both tested. Table 6-4 outlines all of the simulations that were run in Case 4, along with their settings. The mfix.dat input file for this case can be found in the Appendix.

Table 6-4. Summary of inlet velocities, meshing, time steps, tolerances, solvers, discretization schemes, and preconditioners (P.C.) employed in Case 4.

Case 4: 3D Fluidized Bed with Polypropylene Particles							
Case	**U$_{in}$**	**Dimensions + Mesh**	**Time Step**	**Tolerance**	**Solver**	**Scheme**	**P.C.**
4.1	5 m/s	20x100x2 cm^3 40x250x10	Max: 10^{-1} Min: 10^{-6}	Outer: 10^{-1} Solver: 10^{-3}	BCGS	van Leer	1. MFiX Line 2. PETSc BJACOBI (left) 3. PETSc BJACOBI (right)
4.2	20 m/s						

Figures 6-14 (a, b) show the power spectra for Cases 4.1 and 4.2 respectively. With a lower inlet velocity (Case 4.1), the PSD plots from both the left- and right-side Block Jacobi preconditioner agreed well with the native MFiX power spectrum using line relaxation preconditioning. As the inlet velocity was increased to 20 m/s in Case 4.2, left-side Block Jacobi preconditioning yielded a power spectrum much different from MFiX while Block Jacobi from the right maintained agreement.



(a)

(b)

Figure 6-14. Comparison of power spectra for Cases (a) 4.1 and (b) 4.2.

Figure 6-15 shows the CPU time ratios for Cases 4.1 and 4.2. Left-side Block Jacobi preconditioning was 31% slower than native MFiX for a 5 m/s inlet velocity and 37% slower for a 20 m/s inlet velocity. As Block Jacobi preconditioning was switched from left- to right-side, the CPU times obtained with MFiX-PETSc approached those of MFiX. Right-side Block Jacobi was 6% slower than MFiX with an inlet velocity of 5 m/s and only 2% slower for a 20 m/s inlet. When comparing these results to Case 3, it appears that the timing differences between MFiX-PETSc and native MFiX were minimized when a lower density fluidization material (polypropylene) was used.

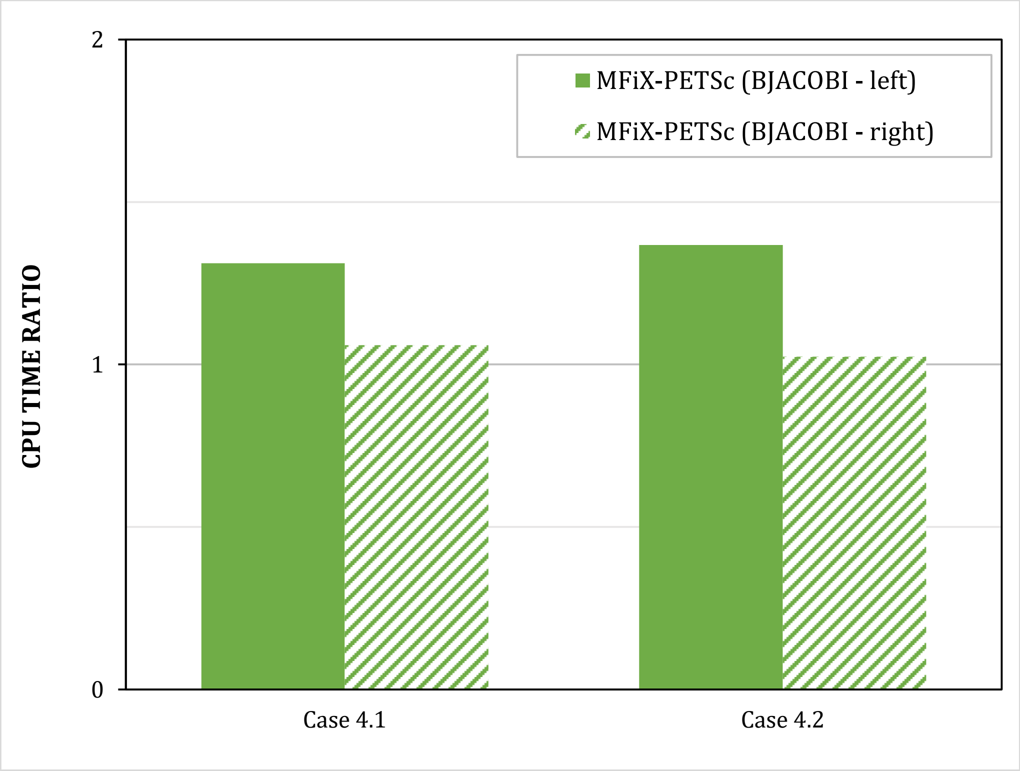


Figure 6-15. CPU time ratios (MFiX-PETSc CPU time / native MFiX CPU time) for Cases 4.1 and 4.2.

Figure 6-16 shows the average iterations (with standard deviations) required to solve the pressure-correction equation for Cases 4.1 and 4.2. As was found previously, right-side Block Jacobi results in fewer solver iterations when compared to left-side Block Jacobi. This also correlates with the timing results presented in Figure 6-15.

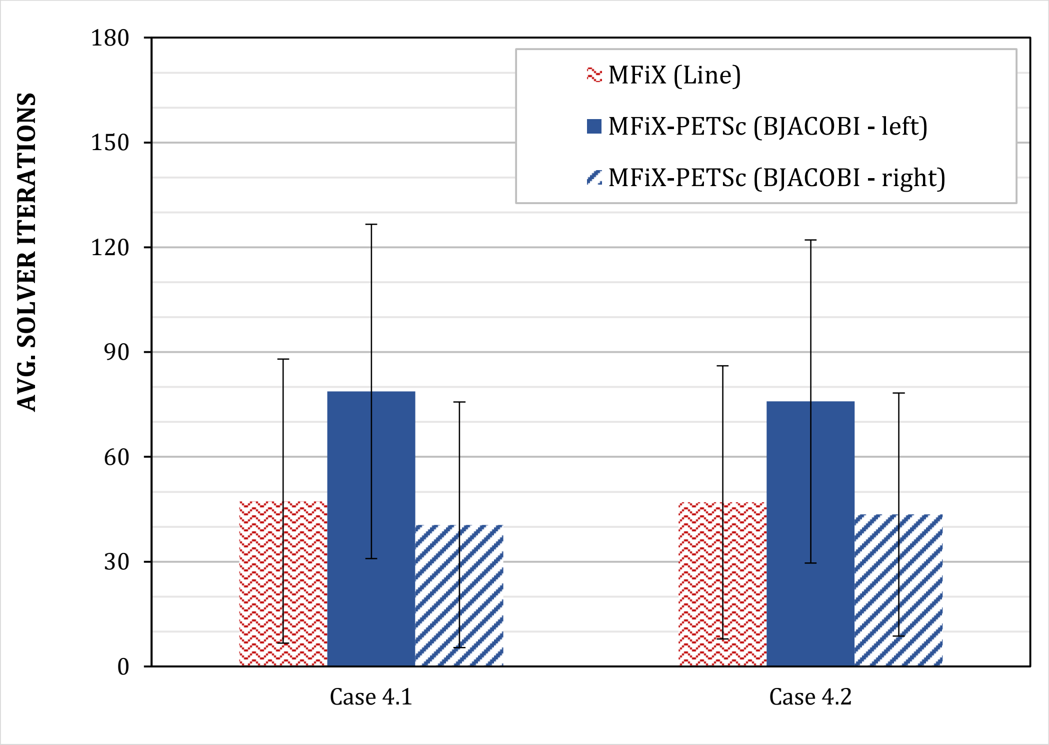


Figure 6-16. Average solver iterations, with standard deviations, for Cases 4.1 and 4.2.

CHAPTER 7

2D FLUIDIZED BED WITH FINE CENTRAL MESHING

7.1 Problem Overview

The same fluidized bed problem as presented in Cases 3 and 4 was represented by a two-dimensional geometry with fine meshing towards the center. This fine central meshing was used since the jet is in the center and thus this is where the major pressure fluctuations occurred. Figure 7-1 shows the 2D bed geometry (20x100 cm^2) and the computational mesh (56x250) that was used throughout this study.

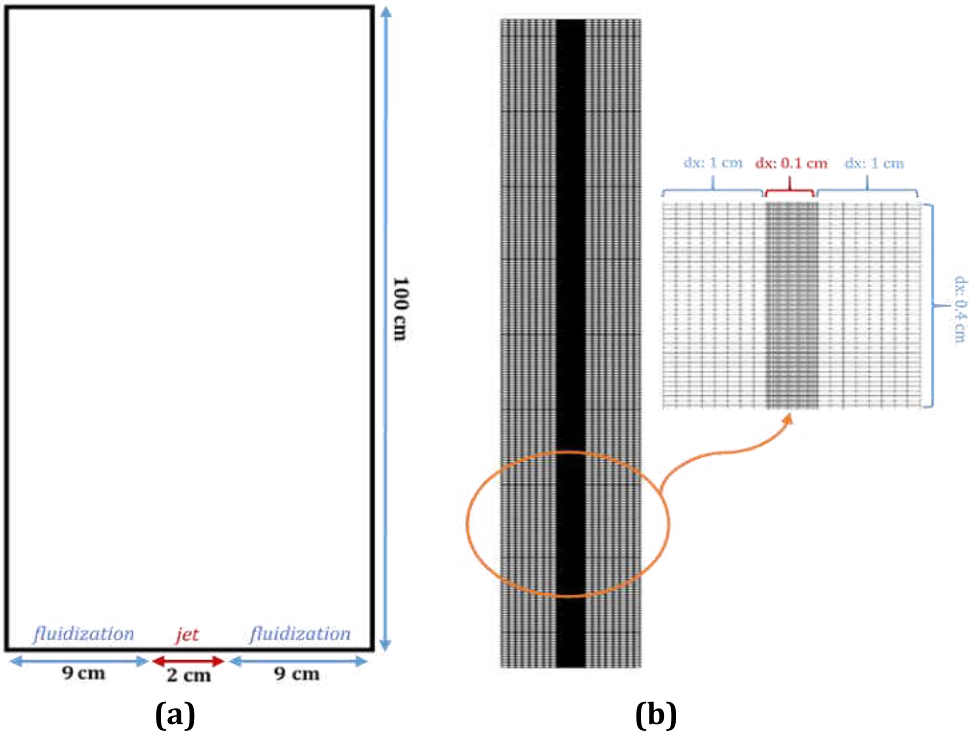


Figure 7-1. Case 5: (a) Dimensions of the rectangular fluidized bed (2D); (b) Mesh size

For this study the domain was filled with glass and polypropylene particles separately at a H/D of 1, and the same properties presented in Tables 6-1 and 6-3 were used to represent these materials. Again, two different air inlets were used: air through the central jet at a velocity of 5 m/s and air at the minimum fluidization velocity (0.11 or 0.3 m/s) from the bottom of the bed. Air was assumed to have constant density of 1.2 kg/m^3 and viscosity of 1.8x10^{-5} Pa·s. The numerical simulation parameters specified in this investigation are summarized in Table 3. The mfix.dat file for Case 5 can be found in the Appendix.

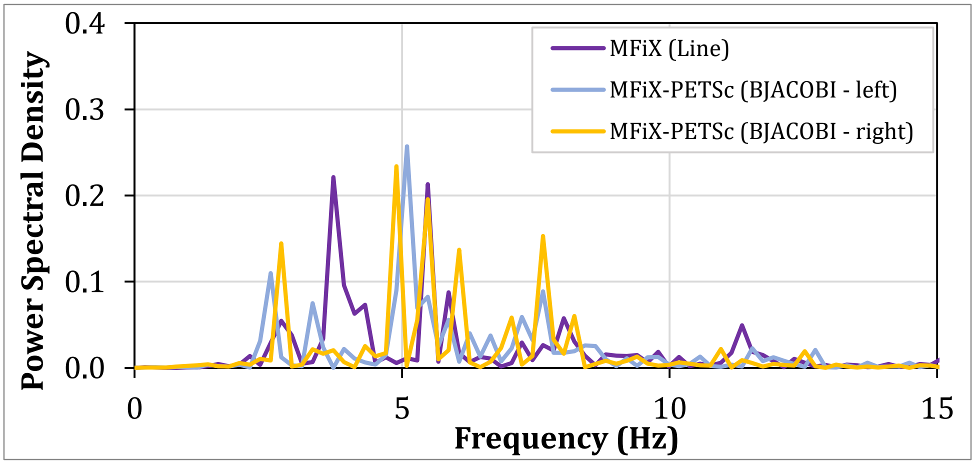
Table 7-1. Summary of fluidization materials, meshing, time steps, tolerances, solvers, discretization schemes, and preconditioners employed in Case 5.

Case 5: 2D Fluidized Bed with Fine Central Meshing							
Case	**Particle**	**Dimensions + Mesh**	**Time Step**	**Tolerance**	**Solver**	**Scheme**	**P.C.**
5.1	Glass	20x100 cm^2 56x250	Max: 10^{-3} Min: 10^{-6}	Outer: 10^{-3} Solver: 10^{-3}	BCGS	van Leer	1. MFiX: Line 2. MFiX-PETSc: BJACOBI (left) 3. MFiX-PETSc: BJACOBI (right)
5.2	P.P.						

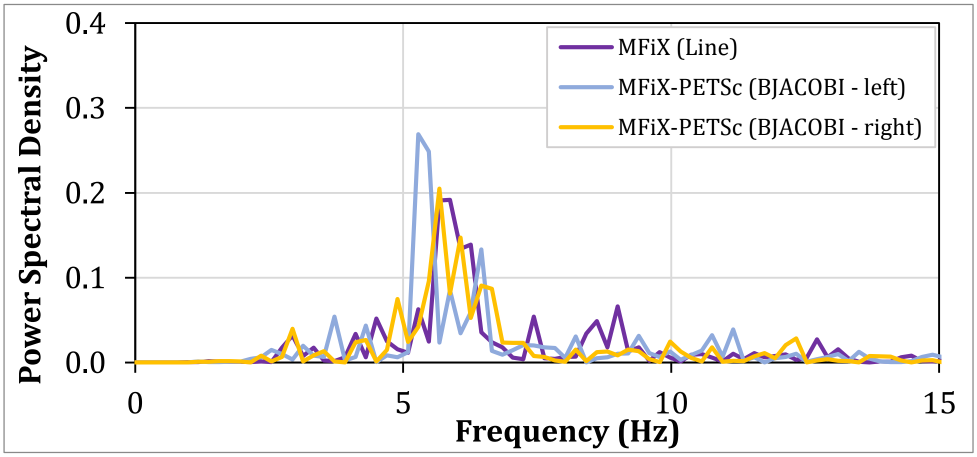
Similar to Cases 3 and 4, pressure was monitored 15 mm above the central jet. The normalized pressure fluctuations were transformed into Power Spectral Density (PSD) plots via the Fast Fourier Transform (FFT) method. Additionally, CPU times and average solver iterations (with standard deviations) were compared between the preconditioners. Case 5 also monitored the total number of time steps and total number of outer solver iterations required to complete 10 seconds of simulation time. Looking at these two properties helped indicate which preconditioning method resulted in the largest time step, which generally leads to lower CPU times.

7.2 Results

Figure 7-2 (a) shows the PSD trends predicted in Case 5.1 for glass particles, while Figure 7-2 (b) shows these trends predicted in Case 5.2 for polypropylene particles. As was seen in the 3D fluidized bed cases, the power spectra predicted with van Leer are not very smooth for 10 seconds of simulation. The results obtained by the MFiX-PETSc solver for polypropylene particles are fairly similar to native MFiX. For glass particles however, no conclusions can clearly be drawn regarding power spectra similarities between the different preconditioners.



(a)



(b)

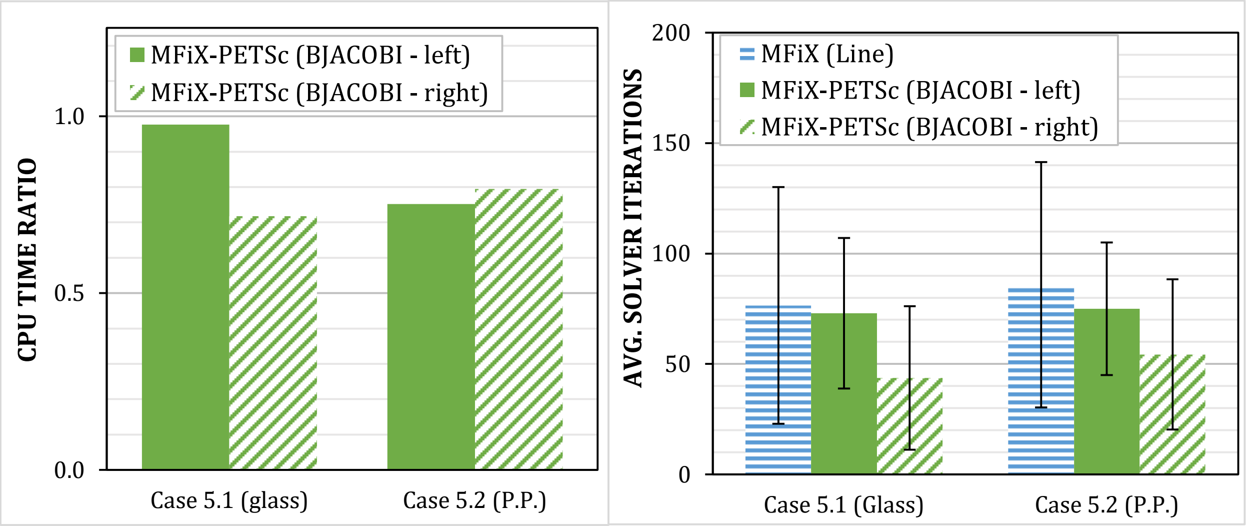
Figure 7-2. Comparison of power spectra obtained for Cases (a) 5.1 and (b) 5.2.

Figure 7-3 (a) shows the CPU time ratios relative to that of the native MFiX solver (MFiX-PETSc CPU time / native MFiX CPU time) to simulate 10 seconds of fluidization in Case 5. For a high density fluidization material (glass), MFiX-PETSc was 2% faster than native MFiX when left-side Block Jacobi preconditioning was employed, and 28% faster with right-side Block Jacobi preconditioning. When a low density fluidization material was used (polypropylene), MFiX-PETSc was 25% and 20% faster than native MFiX with left-side Block Jacobi and right-side Block Jacobi respectively.

Figure 7-3 (b) shows the average iterations required to solve the pressure-correction equation, with standard deviations, for each preconditioning method throughout Case 5. When compared to MFiX's native solver, Block Jacobi (both left and right) takes less iterations to converge to the specified tolerance and thereby causes the CPU times to be lower than those of native MFiX. Additionally, for both Cases 5.1 and 5.2, right-side Block Jacobi resulted in fewer solver iterations compared to left-side Block Jacobi. This trend correlates with the CPU timing difference between left and right preconditioning that was observed for glass particles. However, it was surprising for Case 5.2 (P.P.) that while left-side Block Jacobi is faster overall, the average number of solver iterations is higher than that of right-side Block Jacobi preconditioning. This may be attributed to the number of times the solution to the linear pressure-correction system is invoked over the 10 seconds of simulation time, which is determined by the number of "outer iterations".

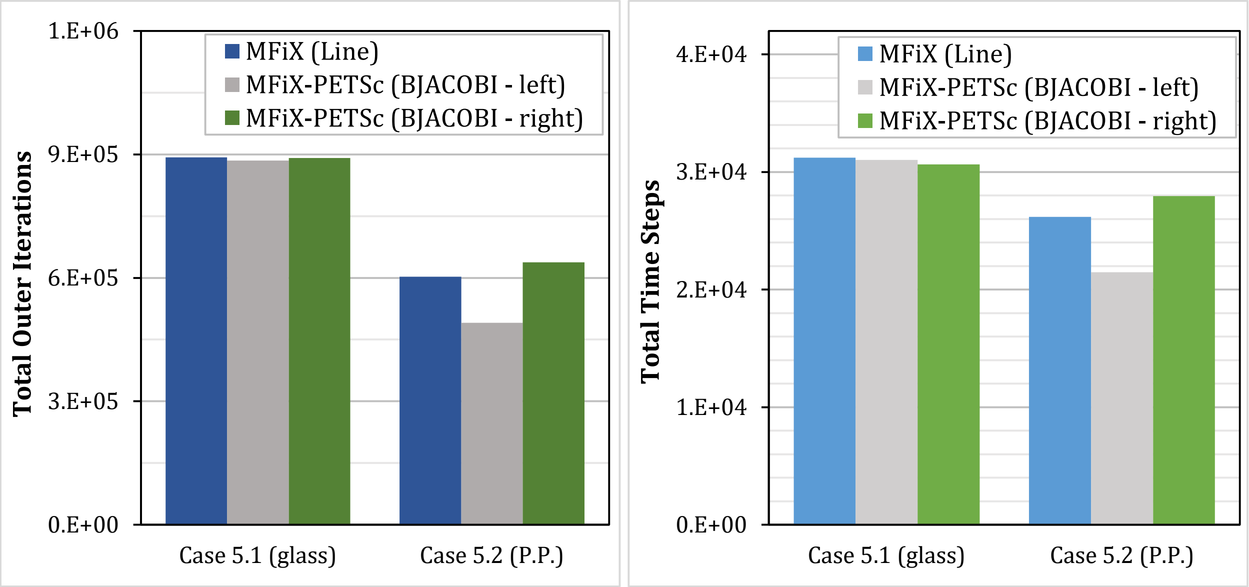
Figure 7-4 (a) shows the total number of outer iterations when employing different preconditioner options for both Cases 5.1 (glass) and 5.2 (P.P.). The CPU time is in general proportional to the product of: number of outer iterations and the number of solver iterations for the pressure correction equation. For glass beads, the difference in total outer iterations between preconditioning methods is insignificant. Therefore, right-side Block Jacobi emerged as the best preconditioner option solely due to its reduced solver iterations, as shown in Figure 7-3 (b). With polypropylene particles, Block Jacobi (left-side) resulted in significantly fewer outer iterations when compared to native MFiX and right-side Block Jacobi. Overall, this reduction in outer iterations proved to be the factor making left-side Block Jacobi the fastest preconditioning option throughout Case 5.2 (P.P.).

The number of time-steps to achieve 10 seconds of simulation time are shown in Figure 7-4 (b). For Case 5, we see that there is a good correlation between the total number of outer iterations and the number of time steps.



(a) (b)

Figure 7-3. The (a) CPU time ratios and (b) average number of solver iterations with standard deviations over 10 seconds of fluidized bed simulations (Case 5).



(a)

(b)

Figure 7-4. The (a) number of time steps and (b) number of outer iterations over 10 seconds of fluidized bed simulations (Case 5).

CHAPTER 8

SUMMARY

8.1 Conclusions

With this work, we were able to interface the MFiX multiphase flow software with the PETSc linear solver library for serial execution. This interface was successfully implemented and verified by comparing its predictions against those obtained from MFiX's solver options for a simple heat conduction problem, as well as a class of single-phase and multiphase flow problems. Different preconditioning methods were applied to the BiCGSTAB solver to carry out this study.

For a steady-state heat conduction case, MFiX-PETSc was at best 168% slower than MFiX's native solver to obtain the same accuracy with a fine mesh resolution. Furthermore, preconditioning the conduction case from the right-side resulted in significantly more error compared to the left-side.

The second case was the single-phase flow of air past a cylinder, and it was found that applying Block Jacobi from both the left- and right-side was more accurate and faster than MFiX's preconditioning options as the meshing was refined. For the finest mesh, left-side Block Jacobi was 27-46% faster than the diagonal scaling preconditioner in MFiX, while right-side Block Jacobi was 28-40% faster. It was found that Block Jacobi preconditioning (both left and right) required fewer solver iterations compared to other methods, which correlates with the lower CPU times.

As the problem changed to a multiphase system, significant differences in the speeds of the preconditioning options offered in MFiX-PETSc were observed as the simulation settings were altered. First, a fluidized bed with a central jet was represented using a 3D geometry with a uniform mesh in Cases 3 and 4. Although native MFiX was faster than MFiX-PETSc for every simulation, this case gave us insight into the performance of MFiX-PETSc's preconditioners based on settings such as solver tolerance, fluidization material, and discretization schemes. First off, with a high solver tolerance (10^{-1}), left-side preconditioning was significantly faster than right-side preconditioning for both SOR and Block Jacobi. As the solver tolerance was reduced (10^{-3}), right-side preconditioning resulted in lower CPU times compared to the left-side. This can be attributed to a fewer number of solver iterations when right-side preconditioning is employed at this low tolerance level. Furthermore, the timing differences between MFiX and MFiX-PETSc were reduced when a higher order discretization scheme was used (van Leer). With a low solver tolerance and van Leer discretization, right-side Block Jacobi was the fastest preconditioner in MFiX-PETSc. This preconditioning method was 21-25% slower than MFiX for a high density material (glass) and only 2-6% slower for a low density material (polypropylene).

The fluidized bed was then represented by a 2D geometry with fine central meshing in the fifth case. Unlike the 3D case, preconditioning options in MFiX-PETSc proved to be faster than MFiX's native line relaxation preconditioner. With glass as the fluidization material, right-side Block Jacobi preconditioning was 28% faster than MFiX's default solver, whereas left-side Block Jacobi was only 2% faster. When the material was changed to polypropylene, Block Jacobi was 25% faster than native MFiX when applied from the left-side, and 20% faster from the right. This result was surprising since right-side Block Jacobi required fewer iterations to solve the pressure-correction equation. However, it was found that for this specific case, left-side Block Jacobi resulted in a lower number of outer iterations, which also impacts the CPU time.

By considering all of these cases, preconditioning with MFiX-PETSc's Block Jacobi was shown to be the best option with an application to the pressure-correction equation. Choosing between left- and right-side Block Jacobi was proven to be case dependent. Therefore, these results demonstrate the importance of having multiple solvers and preconditioners available, as their performance and accuracy strongly depend on the linear system at hand.

8.2 Future Work

The main objective for future studies will be creating a parallelized version of the MFiX-PETSc interface. The PETSc linear solver library is best known for its scalable computation and thus, the interface software will be most beneficial in a parallel environment. With a new parallel framework, various timing studies can be performed with these same cases. Additionally, the interface can be tested on combustion or other reactions for which the solution to the fluid-phase temperature equation is a computational bottleneck.

Overall, PETSc contains 39 Krylov subspace methods and 14 preconditioners. Testing different combinations of solvers and preconditions for different types of systems would be valuable for any future work with the MFiX-PETSc interface. The end goal for this research would be a successful implementation of a parallelized MFiX-PETSc interface with a written manual describing solver and preconditioning options that work most effectively based upon properties of the system.

REFERENCES

1. M. Syamlal, W. Rogers, and T.J. O'Brien, "MFIX documentation: Theory guide," U.S. Dept. of Energy, Office of Fossil Energy, Technical Note No. DOE/METC-94/1004 (1993).
2. M. Syamlal, "MFIX documentation: Numerical technique," National Energy Technology Laboratory, U.S. Dept. of Energy, Technical Note No. DOE/MC31346-5824 (1998).
3. H. Jinsong and J. Lou. "Numerical simulation of bubble rising in viscous liquid," Journal of Computational Physics 222, no. 2 (2007): 769-795.
4. S. Balay, W.D. Gropp, L.C. McInnes, and B.F. Smith, "The Portable Extensible Toolkit for Scientific Computing (PETSc)," version 28, (2000).
5. R.D. Falgout, J.E. Jones, and U.M. Yang. "The design and implementation of hypre, a library of parallel high performance preconditioners", In: *Numerical Solution of Partial Differential Equations on Parallel Computers*, pages 267–294. Springer-Verlag, (2006).
6. J. Schmidt, J. Thornock, J. Sutherland, and M. Berzins. "Large scale parallel solution of incompressible flow problems using uintah and hypre," Technical Report UUSCI-2012-002, Scientific Computing and Imaging Institute, (2012).
7. C. E. Brennan, "Fundamentals of multiphase flow," Cambridge Univ. Press, (2005).
8. M. Syamlal, J. Musser, J. F. Dietiker, "The two-fluid model in MFIX," in: *Multiphase Flow Handbook,* 2nd ed., Boca Raton, FL: Taylor & Francis Group, LLC, (2017), pp. 242-275

9. B. P. Leonard, S. Mohktari, “Beyond first-order upwinding: the ultra-sharp alternative for non-oscillatory steady-state simulation of convection,” Int. J. Numer. Methods Eng., 30, 729 (1990).

10. S. V. Patankar, “Numerical heat transfer and fluid flow,” New York: Hemisphere Publishing Corporation, (1980).

11. F. Moukalled, L. Mangani, and M. Darwish, “Fluid flow computation: Incompressible flows,” In: *The Finite Volume Method in Computational Fluid Dynamics*, Springer International Publishing Switzerland, (2016), pp. 561-654.

12. S. Balay, S. Abhyankar, M. Adams, J. Brown, P. Brune, et al., “PETSc users manual,” Technical Report, ANL-95/11, Rev. 3.8, Argonne National Laboratory, Lemont, IL, (2017).

13. Y. Saad, *Iterative Methods for Sparse Linear Systems,* 2nd ed., Society for Industrial and Applied Mathematics, (2013).

14. A. Ghai, C. Lu, and X. Jiao, “A comparison of preconditioned krylov subspace methods for large-scale nonsymmetric linear systems,” (2017).

15. W. E. Arnoldi, “The principle of minimized iteration in the solution of the matrix eigenvalue problem,” Quart. Appl. Math., 9 (1951), pp. 17-29.

16. C. Lanczos, “An iteration method for the solution of the eigenvalue problem of linear differential and integral operators,” Journal of Research of the National Bureau of Standards, 45 (1950), pp. 255-282.

17. H. A. van der Vorst, “BiCGSTAB: a fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems,” SIAM J. Sci. Stat. Comp., 13(2): pp. 631-644, (1992).

18. “Sparse linear algebra,” In: *NAG Parallel Library Manual*.

19. M. Rahman, M. Karim, A. Alim, “Numerical investigation of unsteady flow past a circular cylinder using 2-d finite volume method,” Journal of Naval Architecture and Marine Engineering, 4(2007). pp. 27-42.

20. C. Norberg, “Pressure distributions around a circular cylinder in cross-flow,” In: Hourigan, K., Leweke, T., Thompson, M.C., Williamson, C.H.K. (Eds.), Symposium on Bluff Body Wakes and Vortex-Induced Vibrations (BBVIV3), 17–20 Dec. 2002, Port Arthur, Queensland, Australia, 2002. Monash University, Melbourne, Australia, pp. 1–4.

21. R. P. Utikar, V. V. Ranade, “Single jet fluidized beds: experiments and CFD simulations with glass and polypropylene particles,” Chemical Engineering Science 62 (2007), pp. 167-183.

APPENDIX

MFIX.DAT FILES

A.1 CASE 1

```
#
#  Conduction in a 3D cube
#

#
# Run-control section
#
  RUN_NAME = 'COND01'
  DESCRIPTION = 'Steady conduction'
  RUN_TYPE = 'new'
  UNITS = 'cgs'

  LEQ_METHOD(6) = 1          ! Comment out if using native MFiX
  LEQ_IT(6) = 10000          ! Comment out if using MFiX-PETSc
  LEQ_TOL(6) = 1.0d-4        ! comment out if using MFiX-PETSc
  MAX_NIT = 500

  ENERGY_EQ = .TRUE.
  SPECIES_EQ = .FALSE.    .FALSE.
  MOMENTUM_X_EQ = .FALSE.    .FALSE.
  MOMENTUM_Y_EQ = .FALSE.    .FALSE.
  MOMENTUM_Z_EQ = .FALSE.    .FALSE.

#
# Geometry Section
#
  COORDINATES = 'Cartesian'

  XLENGTH  = 10.0    IMAX = 100
  YLENGTH  = 10.0    JMAX = 100
  ZLENGTH  = 10.0    KMAX = 100

#
# Gas-phase Section
#
  RO_g0 = 2.5
  MU_g0 = 0.01
  MW_avg = 29.
  GRAVITY = 0.0
  K_g0 = 1.0
  C_pg0 = 1.

#
# Solids-phase Section
#
  MMAX        = 0

#
# Initial Conditions Section
#
  IC_X_w          =  0.0
  IC_X_e         = 10.0
  IC_Y_s         =  0.0
  IC_Y_n        = 10.0
  IC_Z_b        = 0.0
  IC_Z_t        = 10.0

  IC_EP_g         =  1.0
  IC_U_g         =  0.0
  IC_V_g         =  0.0
  IC_W_g          =  0.0
  IC_T_g         = 300.

#
#  Boundary Conditions Section
#
     !              bottom      Top
  BC_X_w         =  0.0         0.0
  BC_X_e         =  10.0    10.0
  BC_Y_s         =  0.0         0.0
  BC_Y_n         =  10.0    10.0
  BC_Z_b         =  0.0       10.0
  BC_Z_t         =  0.0       10.0

  BC_TYPE       =  'NSW'   'NSW'

  BC_EP_g        =  1.0     1.0

  BC_U_g         =  2*0.0
  BC_V_g         =  2*0.0
  BC_W_g         =  2*0.0
  BC_Tw_g        = 1000.0   300.

#
#  Output Control
#
  RES_DT = 0.01
  OUT_DT = 10.

! Interval at which .SPX files are written
  SPX_DT(1) = 0.10   ! EP_g
  SPX_DT(2) = 0.10   ! P_g, P_star
  SPX_DT(3) = 0.10   ! U_g, V_g, W_g
  SPX_DT(4) = 0.10   ! U_s, V_s, W_s
  SPX_DT(5) = 100.   ! ROP_s
  SPX_DT(6) = 100.   ! T_g, T_s
  SPX_DT(7) = 100.   ! X_g, X_s
  SPX_DT(8) = 100.   ! theta
  SPX_DT(9) = 100.   ! Scalar

  NLOG   = 25
  FULL_LOG = .TRUE.
  RESID_STRING = 'T0'


# DMP control
  NODESI = 1   NODESJ = 1   NODESK = 1
```

A.2 CASE 2

```
#
#  Single-phase flow past a cylinder
#

#
# Run-control section
#
   RUN_NAME            =  'CYL'

   DESCRIPTION= 'Flow over a Cylinder , Re = 200'
   RUN_TYPE      = 'new'
   UNITS            = 'SI'
   TIME              = 0.0
```

```
  TSTOP           = 300.0
  DT              = 0.25
  ENERGY_EQ           = .FALSE.
  SPECIES_EQ          = .FALSE.    .FALSE.

  DT_FAC = 1.0
  DETECT_STALL = .FALSE.

  GRAVITY = 0.0

  LEQ_METHOD(1) = 1          ! Comment out if using native MFiX
  LEQ_IT(1) = 10000          ! Comment out if using MFiX-PETSc
  DISCRETIZE(1) = 2
  DISCRETIZE(3) = 2
  DISCRETIZE(4) = 2
  DEF_COR         = .TRUE.
  FPFOI           = .FALSE.

  TOL_RESID    = 1.0E-6

  NORM_g = 0.0

  MOMENTUM_X_EQ(1) = .FALSE.
  MOMENTUM_Y_EQ(1) = .FALSE.

!======================================================================
==========
! Cartesian Grid - Quadric definition:
! Quadric surface Normal form :
! f(x,y,z) = lambda_x * x^2 + lambda_y * y^2 + lambda_z * z^2 + d = 0
! Regions where f(x,y,z) < 0 are part of the computational domain.
! Regions where f(x,y,z) > 0 are excluded from the computational domain.
!
! Predefined quadrics: set QUADRIC_FORM to one of the following:
! Plane:                   'PLANE'
! Cylinder (internal flow): 'X_CYL_INT' or 'Y_CYL_INT' or 'Z_CYL_INT'
! Cylinder (external flow): 'X_CYL_EXT' or 'Y_CYL_EXT' or 'Z_CYL_EXT'
! Cone     (internal flow): 'X_CONE'    or 'Y_CONE'    or 'Z_CONE'
!======================================================================
==========

 CARTESIAN_GRID = .TRUE.

 N_QUADRIC = 1
```

```
  QUADRIC_FORM(1) = 'Z_CYL_EXT'
  RADIUS(1) = 0.05

  t_x(1)     =  2.0
  t_y(1)     =  2.0

  BC_ID_Q(1) = 12

  PRINT_WARNINGS = .TRUE.

  PRINT_PROGRESS_BAR = .TRUE.
  WRITE_DASHBOARD = .TRUE.

!======================================================================
==========
! VTK file options
!======================================================================
==========
  WRITE_VTK_FILES  = .TRUE.
  TIME_DEPENDENT_FILENAME = .TRUE.
  VTK_DT = 0.5

! Available flags for VTK_VAR are :
!  1 : Void fraction (EP_g)
!  2 : Gas pressure, solids pressure (P_g, P_star)
!  3 : Gas velocity (U_g, V_g, W_g)
!  4 : Solids velocity (U_s, V_s, W_s)
!  5 : Solids density (ROP_s)
!  6 : Gas and solids temperature (T_g, T_s1, T_s2)
!  7 : Gas and solids mass fractions (X_g, X_s)
!  8 : Granular temperature (G)
! 11 : Turbulence quantities (k and ε)
! 12 : Gas Vorticity magnitude and Lambda_2 (VORTICITY, LAMBDA_2)
!100 : Processor assigned to scalar cell (Partition)
!101 : Boundary condition flag for scalar cell (BC_ID)


  VTK_VAR = 2 3 12

! Geometry Section

  COORDINATES         = 'cartesian'

  XLENGTH            =  6.0          ! length
  YLENGTH            =  4.0          ! height
  IMAX              =  120          ! cells in i direction
  JMAX              =  80          ! cells in j direction

  NO_K = .TRUE.

! Using Control points to define grid spacing

  CPX        = 1.90    2.10   6.00
  NCX        = 20      40     60

  LAST_DX(1)  = -1.0  ! Match Last  DX from segment 1 with First DX of segment 2
  FIRST_DX(3) = -1.0  ! Match First DX from segment 3 with Last  DX of segment 2

  CPY        = 1.90    2.10   4.00
  NCY        = 20      40     20

  LAST_DY(1)  = -1.0  ! Match Last  DY from segment 1 with First DY of segment 2
  FIRST_DY(3) = -1.0  ! Match First DY from segment 3 with Last  DY of segment 2


! Gas-phase Section

 ! Gas-phase Section
   MU_g0             = 1.8E-5          !constant gas viscosity
   RO_g0             = 1.2             !constant gas density

! Solids-phase Section

  MMAX = 0

! Initial Conditions Section

IC_X_w(1)         =    0.0
IC_X_e(1)         =    6.0
IC_Y_s(1)         =    0.0
IC_Y_n(1)         =    4.0

IC_EP_g(1)        =    1.0
IC_U_g(1)         =    0.0
IC_V_g(1)         =    0.0


! Boundary Conditions Section
! Inlet
```

```
BC_X_w(  1)     =   0.0
BC_X_e(  1)    =   0.0
BC_Y_s(  1)    =   0.0
BC_Y_n(  1)    =   4.0
BC_TYPE(  1)    =   'MI'
BC_Ep_g(  1)    =   1.0
BC_U_g(  1)    =   0.03
BC_V_g(  1)    =   0.0
BC_P_g(  1)    =   0.0

! Outlet

BC_X_w(2)          =    6.0
BC_X_e(2)          =    6.0
BC_Y_s(2)          =    0.0
BC_Y_n(2)          =    4.0
BC_TYPE(2)         =    'PO'
BC_P_g(2)          =    0.0

! Top wall
BC_X_w(3)          =    0.0
BC_X_e(3)          =    6.0
BC_Y_s(3)          =    4.0
BC_Y_n(3)          =    4.0

BC_TYPE(3)         =    'FSW'


! Bottom wall
BC_X_w(4)          =    0.0
BC_X_e(4)          =    6.0
BC_Y_s(4)          =    0.0
BC_Y_n(4)          =    0.0

BC_TYPE(4)         =    'FSW'


! cut-cell boundary condition
  BC_TYPE(12)      = 'CG_NSW'


!
!  Output Control
!
  OUT_DT            = 10.                     !write text file CYL.OUT  every 10 s
  RES_DT          = 100.0                  !write binary restart file CYl.RES every 100.0 s
  NLOG            = 25                      !write logfile CYL.LOG ever 25 time steps
  FULL_LOG        = .TRUE.                  !display residuals on screen

  Resid_string    = "P0", "U0", "V0"

SPX_DT = 100. 0.1      0.1  0.1  100.     100. 100.    100.   100.

!  The decomposition in I, J, and K directions for a Distributed Memory Parallel machine

  NODESI = 1   NODESJ = 1  NODESK = 1

!  Sweep Direction

 LEQ_SWEEP(1) = 'ISIS'
 LEQ_SWEEP(2) = 'ISIS'
 LEQ_SWEEP(3) = 'ISIS'
 LEQ_SWEEP(4) = 'ISIS'
 LEQ_SWEEP(5) = 'ISIS'
 LEQ_SWEEP(6) = 'ISIS'
 LEQ_SWEEP(7) = 'ISIS'
 LEQ_SWEEP(8) = 'ISIS'
 LEQ_SWEEP(9) = 'ISIS'
```

A.3 CASE 3

```
#
#  3D Fluidized Bed
#  Glass Particles
#

#
# Run-control section
#
  RUN_NAME  = 'BUB02'
  DESCRIPTION = 'Bubbling Fluidized Bed Simulation'
  RUN_TYPE    = 'new'
  UNITS         = 'SI'
  TIME           = 0.0
  TSTOP         = 10.0
  DT_MAX       = 1.0E-1
  DT               = 1.0E-3
  DT_MIN        = 1.0E-6
  ENERGY_EQ = .FALSE.
  SPECIES_EQ = .FALSE.    .FALSE.

#
# Numerical Section
#
  MAX_NIT      = 100
  TOL_RESID   = 1.0d-3
  LEQ_IT(1)     = 10000          ! Comment out if using MFiX-PETSc
  LEQ_TOL(1)  = 1.0d-3           ! Comment out if using MFiX-PETSc
  LEQ_METHOD(1) = 1              ! Comment out if using native MFiX
  DISCRETIZE(1) = 7
  DISCRETIZE(3) = 7
  DISCRETIZE(4) = 7

#
# Geometry Section
#
  COORDINATES = 'cartesian'

  XLENGTH  =  0.2    IMAX = 40
  YLENGTH  =  1.0    JMAX = 250
  ZLENGTH  =  0.02   KMAX = 10

#
# Gas-phase Section
#
  MU_g0 = 1.8E-5
  RO_g0 = 1.2

#
# Solids-phase Section
#
  RO_s0   = 2545.0
  D_p0    = 4.25E-4

  C_E     = 0.9
  Phi     = 30.0
  EP_star = 0.45

#
# Initial Conditions Section
#
     !            Bed       Freeboard
  IC_X_w         =  0.0          0.0
  IC_X_e         =  0.2          0.2
  IC_Y_s         =  0.0          0.2
  IC_Y_n         =  0.2          1.0
  IC_Z_b         =  0.0          0.0
  IC_Z_t         =  0.02         0.02

  IC_EP_g        =  0.45         1.0

  IC_U_g         =  0.0          0.0
  IC_V_g         =@(0.3/0.45)    0.3
  IC_W_g         =  0.0          0.0

  IC_U_s(1,1)    =  0.0          0.0
  IC_V_s(1,1)    =  0.0          0.0
  IC_W_s(1,1)    =  0.0          0.0

  IC_P_star      =  0.0          0.0
  IC_T_g         = 300.0         300.0

#
#  Boundary Conditions Section
#
    !           Fluidization     Jet      Exit
  BC_X_w         =  0.0    0.11    0.09    0.0
  BC_X_e         =  0.09   0.2     0.11    0.2
  BC_Y_s         =  0.0    0.0     0.0     1.0
  BC_Y_n         =  0.0    0.0     0.0     1.0
  BC_Z_b         =  0.0    0.0     0.0     0.0
  BC_Z_t         =  0.02   0.02    0.02    0.02

  BC_TYPE        = 'MI'    'MI'    'MI'    'PO'

  BC_EP_g        =  1.0    1.0     1.0

  BC_U_g         =  0.0    0.0     0.0
  BC_V_g         =  0.3    0.3     5.0
  BC_W_g         =  0.0    0.0     0.0

  BC_P_g         = 1.013d+5  1.013d+5  1.013d+5  1.013d+5
  BC_T_g         = 300.0    300.0    300.0

#
#  Output Control
#
```

```
  RES_DT = 0.01
  SPX_DT = 0.01 0.01      0.1  0.1  100.      100. 100.   100.0  100.0

  NLOG   = 100
  full_log = .true.
```

A.4 CASE 4

```
#
#  3D Fluidized Bed
#  Polypropylene Particles
#

#
# Run-control section
#
  RUN_NAME  = 'BUB02'
  DESCRIPTION = 'Bubbling Fluidized Bed Simulation'
  RUN_TYPE    = 'new'
  UNITS          = 'SI'
  TIME            = 0.0
  TSTOP          = 10.0
  DT_MAX       = 1.0E-1
  DT                = 1.0E-3
  DT_MIN        = 1.0E-6
  ENERGY_EQ = .FALSE.
  SPECIES_EQ = .FALSE.    .FALSE.

#
# Numerical Section
#
  MAX_NIT       = 100
  TOL_RESID   = 1.0d-3
  LEQ_IT(1)     = 10000          ! Comment out if using MFiX-PETSc
  LEQ_TOL(1)  = 1.0d-3           ! Comment out if using MFiX-PETSc
  LEQ_METHOD(1) = 1              ! Comment out if using native MFiX
  DISCRETIZE(1) = 7
  DISCRETIZE(3) = 7
  DISCRETIZE(4) = 7

#
# Geometry Section
```

```
#
  COORDINATES = 'cartesian'

  XLENGTH  =  0.2    IMAX = 40
  YLENGTH  =  1.0    JMAX = 250
  ZLENGTH  =  0.02   KMAX = 10

#
# Gas-phase Section
#
  MU_g0 = 1.8E-5
  RO_g0 = 1.2

#
# Solids-phase Section
#
  RO_s0   = 900.0
  D_p0    = 4.25E-4

  C_E     = 0.6
  Phi     = 30.0
  EP_star = 0.45

#
# Initial Conditions Section
#
     !            Bed       Freeboard
  IC_X_w         =  0.0          0.0
  IC_X_e         =  0.2          0.2
  IC_Y_s         =  0.0          0.2
  IC_Y_n         =  0.2          1.0
  IC_Z_b         =  0.0          0.0
  IC_Z_t         =  0.02         0.02

  IC_EP_g        =  0.45          1.0

  IC_U_g         =  0.0          0.0
  IC_V_g         =@(0.11/0.45)     0.11
  IC_W_g         =  0.0          0.0

  IC_U_s(1,1)     =  0.0          0.0
  IC_V_s(1,1)     =  0.0          0.0
  IC_W_s(1,1)     =  0.0          0.0

  IC_P_star       =  0.0          0.0
```

```
 IC_T_g         = 300.0        300.0

#
#  Boundary Conditions Section
#
     !            Fluidization     Jet      Exit
 BC_X_w         =  0.0     0.11     0.09     0.0
 BC_X_e         =  0.09    0.2      0.11     0.2
 BC_Y_s         =  0.0     0.0      0.0      1.0
 BC_Y_n         =  0.0     0.0      0.0      1.0
 BC_Z_b         =  0.0     0.0      0.0      0.0
 BC_Z_t         =  0.02    0.02     0.02     0.02

 BC_TYPE        = 'MI'     'MI'     'MI'     'PO'

 BC_EP_g        =  1.0     1.0      1.0

 BC_U_g         =  0.0     0.0      0.0
 BC_V_g         =  0.11    0.11     5.0
 BC_W_g         =  0.0     0.0      0.0

 BC_P_g         = 1.013d+5  1.013d+5  1.013d+5  1.013d+5
 BC_T_g         = 300.0    300.0    300.0

#
#  Output Control
#
 RES_DT = 0.01
 SPX_DT = 0.01 0.01     0.1  0.1  100.     100. 100.   100.0  100.0

 NLOG   = 100
 full_log = .true.
```

A.5 CASE 5

```
#
#  2D Fluidized Bed
#  Glass or Polypropylene Particles
#

#
# Run-control section
```

```
#
  RUN_NAME = 'BUB02'
  DESCRIPTION = 'Bubbling Fluidized Bed Simulation'
  RUN_TYPE = 'new'
  UNITS = 'SI'
  TIME  = 0.0
  TSTOP = 10.0
  DT_MAX = 1.0E-3
  DT = 1.0E-3
  DT_MIN = 1.0E-6
  ENERGY_EQ = .FALSE.
  SPECIES_EQ = .FALSE.    .FALSE.

#
# Numerical Section
#
  MAX_NIT   = 1000
  TOL_RESID = 1.0d-3
  LEQ_METHOD(1) = 1          ! Comment out if using native MFiX
  LEQ_IT(1) = 10000          ! Comment out if using MFiX-PETSc
  LEQ_TOL(1) = 1.0d-3        ! Comment out if using MFiX-PETSc
  DISCRETIZE(:) = 7
#
# Geometry Section
#
  COORDINATES = 'cartesian'

  XLENGTH  =  0.2    IMAX = 56
  YLENGTH  =  1.0    JMAX = 250
  NO_K     =  .TRUE.

  DX(0)    =  0.01
  DX(1)    =  0.01
  DX(2)    =  0.01
  DX(3)    =  0.01
  DX(4)    =  0.01
  DX(5)    =  0.01
  DX(6)    =  0.01
  DX(7)    =  0.01
  DX(8)    =  0.001
  DX(9)    =  0.001
  DX(10)    =  0.001
  DX(11)    =  0.001
  DX(12)    =  0.001
  DX(13)    =  0.001
```

```
 DX(14)   =  0.001
 DX(15)   =  0.001
 DX(16)   =  0.001
 DX(17)   =  0.001
 DX(18)   =  0.001
 DX(19)   =  0.001
 DX(20)   =  0.001
 DX(21)   =  0.001
 DX(22)   =  0.001
 DX(23)   =  0.001
 DX(24)   =  0.001
 DX(25)   =  0.001
 DX(26)   =  0.001
 DX(27)   =  0.001
 DX(28)   =  0.001
 DX(29)   =  0.001
 DX(30)   =  0.001
 DX(31)   =  0.001
 DX(32)   =  0.001
 DX(33)   =  0.001
 DX(34)   =  0.001
 DX(35)   =  0.001
 DX(36)   =  0.001
 DX(37)   =  0.001
 DX(38)   =  0.001
 DX(39)   =  0.001
 DX(40)   =  0.001
 DX(41)   =  0.001
 DX(42)   =  0.001
 DX(43)   =  0.001
 DX(44)   =  0.001
 DX(45)   =  0.001
 DX(46)   =  0.001
 DX(47)   =  0.001
 DX(48)  =  0.01
 DX(49)  =  0.01
 DX(50)  =  0.01
 DX(51)  =  0.01
 DX(52)  =  0.01
 DX(53)  =  0.01
 DX(54)  =  0.01
 DX(55)  =  0.01

#
# Gas-phase Section
```

```
#
  MU_g0 = 1.8E-5
  RO_g0 = 1.2

#
# Solids-phase Section
#
  RO_s0   = 2545.0      ! Comment out if using polypropylene particles
!  RO_s0   = 900.0      ! Comment out if using glass particles
  D_p0    = 4.25E-4

  C_E     = 0.9         ! Comment out if using polypropylene particles
!  C_E     = 0.6        ! Comment out if using glass particles
  Phi     = 30.0
  EP_star = 0.45

#
# Constants to ensure correct drag correlation
#
  drag_c1 = 0.771
  drag_d1 = 2.8781

#
# Initial Conditions Section
#
     !          Bed      Freeboard
  IC_X_w        =  0.0          0.0
  IC_X_e        =  0.2          0.2
  IC_Y_s        =  0.0          0.2
  IC_Y_n        =  0.2          1.0

  IC_EP_g       =  0.45         1.0

  IC_U_g        =  0.0          0.0
  IC_V_g        =@(0.3/0.45)      0.3            ! Comment out if using polypropylene particles
!  IC_V_g        =@(0.11/0.45)      0.11         ! Comment out if using glass particles
!  IC_W_g        =  0.0          0.0

  IC_U_s(1,1)     =  0.0          0.0
  IC_V_s(1,1)     =  0.0          0.0
!  IC_W_s(1,1)     =  0.0          0.0

  IC_P_star      =  0.0          0.0
  IC_T_g        = 300.0        300.0
```

```
#
#  Boundary Conditions Section
#
    !           Fluidization    Jet     Exit
  BC_X_w        =  0.0    0.11    0.09    0.0
  BC_X_e        =  0.09   0.2     0.11    0.2
  BC_Y_s        =  0.0    0.0     0.0     1.0
  BC_Y_n        =  0.0    0.0     0.0     1.0
!  BC_Z_b       =  0.0    0.0     0.0     0.0
!  BC_Z_t       =  0.02   0.02    0.02    0.02

  BC_TYPE       = 'MI'    'MI'    'MI'    'PO'

  BC_EP_g       =  1.0    1.0     1.0

  BC_U_g        =  0.0    0.0     0.0
  BC_V_g        =  0.3    0.3     5.0              ! Comment out if using polypropylene particles
!  BC_V_g       = 0.11  0.11    5.0               ! Comment out if using glass particles

  BC_P_g        = 1.013d+5  1.013d+5  1.013d+5  1.013d+5
  BC_T_g        = 300.0    300.0    300.0


#
#  Output Control
#
  RES_DT = 0.01
  SPX_DT = 0.01 0.01     0.1  0.1  100.     100. 100.   100.0  100.0

  NLOG   = 100
  full_log = .true.
```