



Michigan Technological University
Create the Future Digital Commons @ Michigan Tech

Dissertations, Master's Theses and Master's
Reports - Open

Dissertations, Master's Theses and Master's
Reports

2013

SIMULATIONS OF NEWTONIAN AND NON-NEWTONIAN FLOWS IN DEFORMABLE TUBES

Abdallah A. Al-Habahbeh
Michigan Technological University

Follow this and additional works at: <https://digitalcommons.mtu.edu/etds>


 Part of the [Applied Mathematics Commons](#)

Copyright 2013 Abdallah A. Al-Habahbeh

Recommended Citation

Al-Habahbeh, Abdallah A., "SIMULATIONS OF NEWTONIAN AND NON-NEWTONIAN FLOWS IN DEFORMABLE TUBES", Dissertation, Michigan Technological University, 2013.
<https://doi.org/10.37099/mtu.dc.etds/593>

Follow this and additional works at: <https://digitalcommons.mtu.edu/etds>

 Part of the [Applied Mathematics Commons](#)

SIMULATIONS OF NEWTONIAN AND NON-NEWTONIAN FLOWS IN
DEFORMABLE TUBES

By

Abdallah A. Al-Habahbeh

A DISSERTATION

Submitted in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

In Mathematical Sciences

MICHIGAN TECHNOLOGICAL UNIVERSITY

2013

© 2013 Abdallah A. Al-Habahbeh

This dissertation has been approved in partial fulfillment of the requirements for the
Degree of DOCTOR OF PHILOSOPHY in Mathematical Sciences.

Department of Mathematical Sciences

Dissertation Co-Advisor Prof. Tanner, Franz X

Dissertation Co-Advisor Prof. Feigl, Kathleen A

Committee Member Prof. Yang, Song L

Committee Member Prof. Labovsky, Alexander E

Department Chair/School Dean Prof. Gockenbach, Mark S

Contents

List of Figures	ix
List of Tables	xix
Preface	xxi
Acknowledgments	xxiii
ABBREVIATIONS	xxiv
Abstract	xxvii
1 Introduction	1
1.1 Background	1
1.2 Previous and Related Studies	2
1.3 Contributions of this thesis	5
2 Fluid Dynamics Background and Numerical Methods	9
2.1 Conservation Principles	10
2.2 Rheological Model	13

2.3	Relevant Dimensionless Numbers	21
2.4	Numerical Methods	23
2.4.1	Finite Volume Method	23
2.4.2	Pressure-velocity coupling	42
2.4.3	Linear Solvers	48
2.5	Mesh Motion	59
3	Simulation of Flow in a Collapsed Elastic Tube	63
3.1	Experimental Details	64
3.1.1	Material Properties	66
3.2	Simulation Details	67
3.2.1	Mesh Dependence and Convergence	69
3.3	Results and Discussion	72
3.3.1	Velocity Comparison	72
3.3.2	Shear Rate Comparison	78
3.3.3	Pressure Drop Comparison	82
3.4	Summary and Conclusions	85
4	Simulations of Peristaltic Motion	87
4.1	Moving Frame Simulations	88
4.1.1	Initial and Boundary Conditions	89
4.1.2	Computational Details	90
4.1.3	Mesh Dependence Study	92

4.1.4	Parameter Study	97
4.2	Fixed Frame Simulations	115
4.2.1	Initial and Boundary Conditions	116
4.2.2	Computational Details	118
4.2.3	Mesh Dependence Study	126
4.2.4	Parameter Study	130
4.2.5	Comparison between the Moving and Fixed Simulations	144
4.2.6	Comparison with Experiment	146
4.2.7	Three Dimensional Channel	148
4.2.8	Other Types of Moving Waves	153
4.3	Conclusions	157
5	Summary and Future Work	159
	References	163
A	Case Files: Elastic Tube Simulations	169
B	Case Files: Peristaltic Motion Simulations	189
B.1	Moving Frame	189
B.2	Fixed Frame	198
C	OpenFOAM Codes	207
C.1	movingWallNormalVel boundary condition	207

C.2	dynPerCircle boundary condition	214
C.3	Parabolic wave	225
C.4	Sinusoidal wave	237
C.5	transientSimpleDyMFoam	248
C.6	shearRate	257

List of Figures

2.1	Viscosity versus shear rate curves for typical time-independent fluids. . . .	16
2.2	Stress versus shear rate curves for typical time-independent fluids.	16
2.3	Log-log plot of viscosity versus shear rate for shear-thinning fluid.	17
2.4	Arbitrary polyhedral control volume. Source: Jasak and Tukovic [1]. . . .	25
2.5	Vectors \mathbf{S} and \mathbf{d}_f on a non-orthogonal mesh.	31
2.6	Non-orthogonality treatment in the over-relaxed approach.	32
3.1	Schematic representation of the Starling Resistor for fluid flow behavior study through a collapsible elastic tube. The schematic of collapsed elastic tube and position of ultrasound transducer for velocity profile measurement is also inserted.	65
3.2	Viscosity curve for the non-Newtonian fluid.	67
3.3	Schematic illustration of the flow through the collapsed tube in the $y=15$ mm plane. (The tube is collapsed in the y -direction.)	68
3.4	Three-dimensional view of the computational mesh.	69
3.5	Mesh dependence study for the non-Newtonian velocity profile at $x=70$ mm.	71

3.6	Convergence study for the non-Newtonian velocity profile at x=70 mm.	71
3.7	Velocity profiles at x=70 mm.	73
3.8	Cross-section of the non-Newtonian velocity magnitude at x=70 mm. The cross-sectional coordinates are in mm and the velocity is in m/s.	73
3.9	Velocity profiles at x=90 mm.	74
3.10	Cross-section of the non-Newtonian velocity magnitude at x=90 mm. The cross-sectional coordinates are in mm and the velocity is in m/s.	74
3.11	Velocity profiles at x=150 mm.	75
3.12	Cross-section of the non-Newtonian velocity magnitude at x=150 mm. The cross-sectional coordinates are in mm and the velocity is in m/s.	76
3.13	Three-dimensional view of the non-Newtonian shear rates on the periphery of the collapsed tube.	79
3.14	Cross-section of the non-Newtonian shear rates at x=50 mm.	79
3.15	Shear rate profile for the non-Newtonian and Newtonian simulations at x=50 mm.	82
3.16	Pressure curves for the non-Newtonian and Newtonian simulations. (The flow is from right to left.)	84
4.1	Computation domain for moving frame simulations.	88

4.2	Moving frame mesh dependence study for the x-component of the velocity of the Newtonian fluid along the centerline at $y=0.05$ mm. (The left and right vertical dashed lines represent the ends of the roller, the middle vertical dashed line is the center of the roller.)	94
4.3	Moving frame mesh dependence study for the kinematic pressure of the Newtonian fluid along the centerline at $y=0.05$ mm. (The left and right vertical dashed lines represent the ends of the roller, the middle vertical dashed line is the center of the roller.)	95
4.4	Moving frame mesh dependence study for the shear rate of the Newtonian fluid along the centerline at $y=0.05$ mm. (The left and right vertical dashed lines represent the ends of the roller, the middle vertical dashed line is the center of the roller.)	96
4.5	Shear rate dependent viscosity curves.	98
4.6	Moving frame x-component of the velocity [m/s] of the Newtonian fluid. (The gap half-width is 4 mm, and the roller speed is 5 mm/s.)	101
4.7	Moving frame kinematic pressure [m^2/s^2] of the Newtonian fluid. (The gap half-width is 4 mm, and the roller speed is 5 mm/s.)	101
4.8	Moving frame shear rate [s^{-1}] of the Newtonian fluid. (The gap half-width is 4 mm, and the roller speed is 5 mm/s.)	101
4.9	Moving frame velocity vectors [m/s] under the roller of the Newtonian fluid. (The gap half-width is 4 mm, the roller speed is 5 mm/s.)	102

4.10 Moving frame x-component of the velocity for the Newtonian and non-Newtonian fluids along the centerline at $y=0.05$ mm. (The left and right vertical dashed lines represent the ends of the roller, the middle vertical dashed line is the center of the roller.)	102
4.11 Moving frame x-component velocity profile for the Newtonian and non-Newtonian fluids near the right boundary at $x=89$ mm.	103
4.12 Moving frame kinematic pressure for the Newtonian and non-Newtonian simulations along the centerline at $y=0.05$ mm. (The left and right vertical dashed lines represent the ends of the roller, the middle vertical dashed line is the center of the roller.)	104
4.13 Moving frame shear rate for the Newtonian and non-Newtonian simulations along the centerline at $y=0.05$ mm. (The left and right vertical dashed lines represent the ends of the roller, the middle vertical dashed line is the center of the roller.)	106
4.14 Moving frame shear rate profile for the Newtonian and non-Newtonian fluids near the right boundary at $x=89$ mm.	107
4.15 Moving frame x-component velocity profile for the Newtonian fluid near the right boundary at $x=89$ mm for different gap half-widths. (The roller speed is 10 mm/s.)	110

4.16 Moving frame x-component of the velocity of the Newtonian fluid along the centerline at $y=0.05$ mm for different gap half-widths. (The roller speed is 10 mm/s.)	111
4.17 Moving frame velocity vectors [m/s] under the roller. (The gap half-width is 8 mm, the roller speed is 10 mm/s.)	111
4.18 Moving frame x-component of the velocity of the Newtonian fluid along the centerline at $y = 0.05$ mm for different roller speeds. (The gap half-width is 8 mm.)	113
4.19 Moving frame x-component velocity profile of the Newtonian fluid near the right boundary at $x=89$ mm for different roller speeds. (The gap half-width is 8 mm.)	114
4.20 Computation domain for fixed frame simulations.	116
4.21 The circular wave.	117
4.22 Residual of the discrete x-momentum equation at the beginning of each pressure-velocity iteration in the moving frame simulations (using simpleFoam).	122
4.23 Residual of the discrete pressure equation at the beginning of each pressure-velocity iteration in the moving frame simulations (using simpleFoam).	123

4.24	Residual of the discrete x-momentum equation at the beginning of the last (i.e. 20th) PISO iteration in each time step in the fixed frame simulations (using transientSimpleDyMFoam).	124
4.25	Residual of the discrete pressure equation at the beginning of the last (i.e. 20th) PISO iteration in each time step in the fixed frame simulations (using transientSimpleDyMFoam).	125
4.26	Fixed frame mesh dependence study for the x-component of the velocity of the Newtonian fluid along the centerline at $y=0.04$ mm and time $t=14$ s. (The left and right vertical dashed lines represent the ends of the roller, the middle vertical dashed line is the center of the roller.)	128
4.27	Fixed frame mesh dependence study for the kinematic pressure of the Newtonian fluid along the centerline at $y=0.04$ mm and time $t=14$ s. (The left and right vertical dashed lines represent the ends of the roller, the middle vertical dashed line is the center of the roller.)	129
4.28	Fixed frame x-component of the velocity [m/s] of the Newtonian fluid at time $t=28$ s. (The gap half-width is 4 mm, the roller speed is 5 mm/s.) . . .	131
4.29	Fixed frame kinematic pressure [m^2/s^2] of the Newtonian fluid at time $t=28$ s. (The gap half-width is 4 mm, the roller speed is 5 mm/s.)	131
4.30	Fixed frame shear rate [s^{-1}] of the Newtonian fluid at time $t=28$ s. (The gap half-width is 4 mm, and the roller speed is 5 mm/s.)	132

4.31	Fixed frame velocity vectors [m/s] under the roller of the Newtonian fluid at $t=28$ s. (The gap half-width is 4 mm, and the roller speed is 5 mm/s.) . .	132
4.32	Fixed frame x-component of the velocity for the Newtonian and non-Newtonian fluids along the centerline at $y=0.04$ mm and time $t=28$ s. (The left and right vertical dashed lines represent the ends of the roller, the middle vertical dashed line is the center of the roller.)	134
4.33	Fixed frame x-component velocity profile for the Newtonian and non-Newtonian fluids near the right boundary at $x=179$ mm and time $t=28$ s.	135
4.34	Fixed frame kinematic pressure for the Newtonian and non-Newtonian fluids along the centerline at $y=0.04$ mm and time $t=28$ s. (The left and right vertical dashed lines represent the ends of the roller, the middle vertical dashed line is the center of the roller.)	136
4.35	Fixed frame shear rate for the Newtonian and non-Newtonian fluids along the centerline at $y=0.04$ mm and time $t=28$ s. (The left and right vertical dashed lines represent the ends of the roller, the middle vertical dashed line is the center of the roller.)	137
4.36	Fixed frame x-component velocity profile of the Newtonian fluid near the right boundary at $x=179$ mm and time $t=14$ s for different gap half-widths. (The roller speed is 10 mm/s.)	140

4.37	Fixed frame x-component of the velocity of the Newtonian fluid along the centerline at $y=0.04$ mm and time $t=14$ s for different gap half-widths. (The roller speed is 10 mm/s.)	141
4.38	Fixed frame x-component velocity profile for the Newtonian fluid near the right boundary at $x=179$ mm and time $t=14$ s for different roller speed. (The gap half-width is 8 mm.)	142
4.39	Fixed frame x-component of the velocity of the Newtonian fluid along the centerline at $y=0.04$ mm and time $t=14$ s for different roller speeds. (The gap half-width is 8 mm.)	143
4.40	Experimental setup of the peristaltic motion.	147
4.41	Fixed frame x-component of the velocity of the non-Newtonian fluid along the centerline at $y=0.04$ mm. (The gap half-width is 4 mm.)	147
4.42	Three-dimensional view of the deformed computational mesh.	148
4.43	Fixed frame x-component of the velocity [m/s] in the three dimensional channel simulation at $t = 28$ s. (The gap half-width is 4 mm, the roller speed is 5 mm/s.)	150
4.44	Fixed frame velocity vectors [m/s] under the roller in the three dimensional channel simulation at $t = 28$ s. (The gap half-width is 4 mm, the roller speed is 5 mm/s.)	151

4.45	Fixed frame kinematic pressure [m^2/s^2] in the three dimensional channel simulation at $t = 28$ s. (The gap half-width is 4 mm, the roller speed is 5 mm/s.)	151
4.46	Fixed frame shear rate [s^{-1}] in the three dimensional channel simulation at $t = 28$ s. (The gap half-width is 4 mm/s, the roller speed is 5 mm/s.)	152
4.47	The parabolic wave.	154
4.48	The sinusoidal wave.	155
4.49	View of the computational mesh using parabolic wave, with $A = 41.6667$ mm, $x_0 = 0$ mm, $c = 5$ mm/s, and $y_0 = 4$ mm.	155
4.50	View of the computational mesh using sinusoidal wave, with $A = 6$ mm, $\lambda = 60$ mm, and $c = 8$ mm/s.	156

List of Tables

3.1	Comparison between the Newtonian and non-Newtonian simulations. . . .	77
4.1	Number of cells for the different meshes. (Moving frame simulations.) . .	92
4.2	Non-Newtonian fluid parameters	98
4.3	Moving frame transport efficiency for different fluids	108
4.4	Moving frame transport efficiency for different normalized gap half-widths .	108
4.5	Moving frame transport efficiency for different roller speeds	112
4.6	Parameters for the circular deformation. (Fixed frame simulations.) . . .	118
4.7	Absolute Tolerances	125
4.8	Number of cells for the different meshes.(Fixed frame simulations) . . .	126
4.9	Fixed frame transport efficiency for different fluids	135
4.10	Fixed frame transport efficiencies	138
4.11	Transport efficiency for different fluids	144
4.12	Transport efficiency for different normalized gap half-widths.	145
4.13	Parameters for the parabolic deformation.	153
4.14	Parameters for the parabolic deformation.	154

Preface

This dissertation is submitted for the degree of Doctor of Philosophy at Michigan Technological University. The research described herein was conducted under the supervision of Prof. F. Tanner and Prof. K. Feigl in the Department of Mathematical Sciences, Michigan Technological University, between Jan. 2009 and Apr. 2013.

This work is to the best of my knowledge original, except where references are made to previous work. Neither this, nor any substantially similar dissertation has been or is being submitted for any other degree, diploma or other qualification at any other university.

Chapter three of this work has been presented in the following publication:

F. Tanner, A. Al-Hababbeh, K. Feigl, S. Nahar, S. Jeelani, W. Case, and E. Windhab, Numerical and experimental investigation of a non-Newtonian flow in a collapsed elastic tube, *Appl. Rheol.*, vol. 22, 2012.

The co-Author: Abdallah Al-Hababbeh set up the model, performed the numerical simulations and analyzed the results under the supervision of Prof. Franz Tanner and Prof. Kathleen Feigl.

All experimental work presented in the paper was performed by Samsun Nahar, under the supervision of Dr. Shaik Jeelani and Prof. Erich Windhab.

The contributions from William Case for providing advice on model setup and simulation techniques were also noteworthy.

Acknowledgments

Foremost, I would like to express my sincere gratitude to my advisors Prof. Dr. Franz Tanner and Prof. Dr. Kathleen Feigl for the continuous support of my Ph.D study and research, for their patience, motivation, and enormous knowledge. Their guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better advisors for my Ph.D study.

Besides my advisors, I would like to thank the rest of my thesis committee: Prof. Dr. Song Yang, and Prof. Dr. Alexander Labovsky, for their encouragement, and insightful comments.

Special thanks to my friends and colleagues in the CFD group: Ahmad Baniabedalruhman, William Case, Chao Liang, Samer Alokaily, and Olabanji Shonibare for sharing the literature and invaluable assistance.

I would like to acknowledge the generous financial support of the Tafil Technical University and the Department of Mathematical Sciences at Michigan Technological University.

I would also like to thank my father, two elder brothers, three elder sisters, my mother and sister in law, and my uncle Hashim. They were always supporting me and encouraging me with their best wishes.

Last, but not least, I would like to thank my wife, Hebah for her understanding, love and personal support.

ABBREVIATIONS

Acronyms

BiCG	Bi-conjugate Gradient Method
CD	Central Differencing
CG	Conjugate Gradient Method
CT	Computer Tomography
CV	Control Volume
DIC	Simplified Diagonal-based Incomplete Cholesky Method
FVM	Finite Volume Method
PISO	Pressure-Implicit with Splitting of Operators
SCL	Space Conservation Law

SIMPLE Semi-Implicit Method for Pressure-Linked Equations

UD Upwind Differencing

UVP Ultrasound Velocity Profiling

Non-dimensional parameters

Co Courant number

De Deborah number

Re Reynolds number

Greek symbols

δ unit tensor

σ Cauchy stress tensor

τ extra stress tensor

$\dot{\gamma}$ shear rate

η_0 viscosity at zero shear rate

η_∞ viscosity at infinity shear rate

μ	Dynamic viscosity, molecular viscosity
ρ	Density
τ_0	yield stress
π	Production term in the continuous phase of the ϕ equation
ϕ	Source term in the continuous phase of the ϕ equation
ς	Supply term in the continuous phase of the ϕ equation

Roman symbols

\boldsymbol{g}	Body force per unit mass
p	Pressure
\boldsymbol{u}	Fluid velocity

Abstract

Computational models for the investigation of flows in deformable tubes are developed and implemented in the open source computing environment OpenFOAM. Various simulations for Newtonian and non-Newtonian fluids under various flow conditions are carried out and analyzed. First, simulations are performed to investigate the flow of a shear-thinning, non-Newtonian fluid in a collapsed elastic tube and comparisons are made with experimental data. The fluid is modeled by means of the Bird-Carreau viscosity law. The computational domain of the deformed tube is constructed from data obtained via computer tomography imaging. Comparison of the computed velocity fields with the ultrasound Doppler velocity profile measurements show good agreement, as does the adjusted pressure drop along the tube's axis. Analysis of the shear rates show that the shear-thinning effect of the fluid becomes relevant in the cross-sections with the biggest deformation.

The peristaltic motion is simulated by means of upper and lower rollers squeezing the fluid along a tube. Two frames of reference are considered. In the moving frame the computational domain is fixed and the coordinate system is moving with the roller speed, and in the fixed frame the roller is represented by a deforming mesh. Several two-dimensional simulations are carried out for Newtonian and non-Newtonian fluids. The effect of the shear-thinning behavior of the fluid on the transport efficiency is examined. In addition, the influence of the roller speed and the gap width between the rollers on the

transport efficiency is discussed. Comparison with experimental data is also presented and different types of moving waves are implemented.

Chapter 1

Introduction

1.1 Background

Many biofluid mechanical processes encountered in the human body can be modeled as the flow of a non-Newtonian fluid in a deformable or collapsible elastic tube which is subjected to external and/or internal forces. Examples include the transport of food or digestive liquids in the esophagus, stomach and intestines, and the flow of blood through veins, capillaries and arteries. This is an inherently multi-physics problem which can be modeled by means of a fluid in a deforming tube. Specifically, as the fluid flows through the tube, it produces surface forces on the internal tube wall, which may cause the tube to deform. This change in the tube's shape causes a change in the flow field and hence the

surface forces on the tube, which again affects the tube's deformation. Moreover, there may be large additional external forces applied to the outer wall of the tube which further cause the tube to deform, thus contributing to the complex interplay between the flow field and the tube deformation. Under many relevant conditions, the tube experiences very large deformations and may approach complete collapse.

There are two main approaches to describe such phenomena. The first approach is to solve the coupled fluid-structure problem, that is, both, the flow field and the tube deformation are simultaneously solved as a coupled system. The second approach, called peristalsis, describes a fluid system whose flow is driven by the deformation of the boundaries due to outside forces. In a biofluid mechanical system, the peristaltic motion is the transport of fluids in a tube by means of muscular contraction and expansion such as in the esophagus, stomach, intestines, blood vessels etc. A non-biological application of peristalsis is the peristaltic pump which is used to move clean/sterile or aggressive fluids through a tube without cross contamination between the exposed pump components and the fluid. As discussed in Jaffrin and Shapiro [2], the presence of viscous forces can produce effective pumping.

1.2 Previous and Related Studies

Studies involving the coupled fluid-structure approach with varying degrees of sophistication have been conducted by different authors including Heil [3, 4], Hazel and

Heil [5], and Grotberg and Jensen [6]. Heil [3, 4] presented a three-dimensional solution of flow in a coupled system by using the nonlinear Kirchhoff-Love shell model to describe the tube wall deformation and a Stokes flow approximation for the fluid. The fluid traction has been approximated by lubrication theory. Hazel and Heil [5] used the finite element method to simulate three dimensional flows in a non-axisymmetric buckled tube at finite Reynolds number. Grotberg and Jensen [6] discussed some of physiological applications of collapsible tube flows.

To understand the peristaltic motion in different situations, several theoretical and experimental attempts have been made by the pioneering work of Latham [7] which investigated the mechanism of peristalsis in relation to mechanical pumping. Specifically, a theoretical and experimental analysis of a simplified model of peristaltic pumping have been presented. Shapiro et al. [8] studied a Newtonian fluid with a periodic train of sinusoidal peristaltic waves. They found that for a given amplitude ratio (the ratio of the wave amplitude to the channel width or tube radius), the theoretical pressure rise per wavelength decreases linearly with an increase of the flow rate. Also, reflux in peristaltic flow has been discussed. Using the work of Shapiro et al. [8], Kleinstreuer [9] presented an analytical solution of a peristaltic problem by using a reduced form of the Navier-Stokes equations with a sinusoidal displacement wave, as a boundary condition, traveling on the channel wall with constant speed; he computed the fluid's axial pressure gradient resulting from the pressure difference at both channel ends, the volumetric flow rate and the conditions of reflux.

Analytic investigations of a Newtonian fluid flow induced by the peristaltic motion of a flexible tube have been made by Barton and Raynor [10]; they computed the instantaneous flow rate for a wide range of tube geometries. Jaffrin and Shapiro [2] presented a review of much of the early literature on peristaltic flows.

The first attempt to understand the peristaltic motion of non-Newtonian fluids has been made by Raju and Devanathan [11]; they considered the steady motion of a non-Newtonian fluid in a rigid tube with a sinusoidal deformation at the boundary, and they discussed the influence of the applied pressure gradient along with non-Newtonian parameters on the streamlines and velocity profiles. In 1985, Srivastava and Srivastava [12] showed that the magnitude of pressure rise, under a zero Reynolds number and long wavelength approximation, is smaller in the case of a shear-thinning non-Newtonian fluid.

The peristaltic flow of a Jeffrey six-constant viscoelastic fluid in a uniform inclined tube has been investigated by Nadeem et al. [13], while the peristaltic flow of a Herschel-Bulkley yield-value fluid in a nonuniform inclined tube was studied by Nadeem and Akbar [14]. A theoretical analysis for the axisymmetric peristaltic motion of an incompressible Johnson-Segalman viscoelastic fluid through a circular deformable tube has been carried out by Hayat and Ali [15].

Merrill [16] mentioned that the temperature has no significant effect on the rheological properties of normal blood in the non-Newtonian regime. Specifically, he found that the yield stress is independent of temperature at least in the range of 15-37 C. Also, blood flow is laminar except in severely stenosed arteries. This laminar behavior is achieved with the

controlled pumping action of the heart and the viscoelasticity of the blood vessels reacting locally to variations in blood pressure. For more details, refer to Kleinstreuer [9].

1.3 Contributions of this thesis

This dissertation has investigated several issues in Computational Fluid Dynamics applied to flow in deformable tubes and channels, and the major contributions are:

- **For Collapsed Elastic Tube:**
 - Simulations have been performed to investigate the flow of a shear-thinning, non-Newtonian fluid in a collapsible elastic tube that reflect an experimental setup. The computational domain has been constructed from data obtained via Computer Tomography (CT) imaging.
 - The numerical simulations have been compared with the experimental data, and this comparison study shows a generally good agreement. Results show that crude info about tube geometry is sufficient to get accurate simulations, without need to solve the more complex coupled fluid-structure problem.
 - Plenty of insight into the flow and the material properties has been offered. Some of these properties cannot be easily obtained by means of measurement.
 - The effect of a shear-thinning non-Newtonian viscosity on the flow behavior in

the collapsed tube has been determined.

- **For Peristaltic Motion:**

- Two-dimensional computer models have been developed to simulate peristaltic-motion-driven flow based on experiments where the peristaltic motion is induced by means of rollers which deform the tube. Two different frames of references are considered to describe the peristaltic-motion-driven flow. In the moving frame, the computational domain is fixed and the coordinate system is moving with the roller speed, and in the fixed frame the roller motion is represented by a deforming mesh.
- A moving-mesh boundary condition has been modified to account for the boundary motion of interest in the fixed frame of reference. This boundary condition moves the points of the mesh at a boundary according to a prescribed mathematical formula.
- To examine the flexibility of our moving-mesh boundary condition, several shapes of traveling waves such as parabolic and sinusoidal waves have been tested.
- A boundary condition for velocity has been implemented to ensure zero flux on the moving boundary during the deformation.
- Good agreement has been obtained between the simulation and experimental results.

- The effect of several parameters on the transport efficiency and other flow behavior has been investigated.
- The extension to a three-dimensional channel and comparison of the results to those of the two-dimensional channel have been made.

The computations performed in the framework of this thesis have been performed with the open-source software OpenFOAM [17]. OpenFOAM is an object-oriented C++ continuum mechanics software library. OpenFOAM currently uses a second-order Finite Volume Method on structured and unstructured meshes. It is written in operator form and has a class hierarchy designed to be shared between various discretization practices. Lower level objects, including mesh, matrix, field, boundary conditions, linear solvers etc. are re-used without change.

To study peristaltic motion by means of rollers moving axially along an elastic tube, different codes of OpenFOAM had to be modified. The code changes and additions are documented in the appendix.

Chapter 2

Fluid Dynamics Background and Numerical Methods

Fluid dynamics is the science which describes the motion of fluids and their interactions with solid bodies. The term fluid is used to describe a substance that flows continuously under an applied stress. In most cases of interest, a fluid can be regarded as a continuum, i.e., a continuous substance. Every point in space has finite values for physical properties such as velocity, stress, temperature, etc. These properties may change their values from a point to the next one or there could be a surface where some properties jump discontinuously. However, the continuum assumption does not allow properties to become infinite or to jump discontinuously at a single isolated point.

Applying external forces causes the fluid to flow. These driving forces can be classified as

surface forces (e.g. pressure, viscous forces in a moving fluid, etc.) and body forces (e.g. gravity, electromagnetic forces, etc.).

2.1 Conservation Principles

The fluid motion is described by means of the equations of conservation for mass, momentum, and energy. The general form of the local balance equations for a physical variable γ in the Eulerian frame of reference takes the form (see Hutter and Jöhnk [18])

$$\frac{\partial \gamma}{\partial t} + \nabla \cdot (\gamma \mathbf{u}) = -\nabla \cdot \phi + \pi + \zeta, \quad (2.1)$$

in which \mathbf{u} is the velocity of the fluid, ϕ is the flux of γ from the outside into the body through its bounding surface, π is the production term, and ζ denotes the supply term.

Taking $\gamma = \rho$, the mass density (mass per unit volume) in Eq. (2.1), and assuming that mass is neither produced nor supplied and it does not flow through a surface, we make the following substitutions $\pi = \zeta = \nabla \cdot \phi = 0$. This leads to the mass conservation equation (continuity equation)

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{u}) = 0, \quad (2.2)$$

where the first term on the left hand side of the equation represents the local rate of change of ρ and the second term represents the convective flow of mass out of the fluid particle across its boundaries.

If ρ is constant, the continuity equation Eq. (2.2) becomes

$$\nabla \cdot \mathbf{u} = 0, \quad (2.3)$$

which is called incompressibility constraint. A fluid which satisfies the incompressibility constraint is called incompressible.

The momentum equation can be obtained by taking $\gamma = \rho \mathbf{u}$ in Eq. (2.1) with $\pi = 0$ (assuming that momentum is conserved). Moreover, the supply of momentum is governed by the external volume forces or by densities of the volume forces (e.g. the gravitational force or weight), i.e., $\zeta = \rho \mathbf{g}$, where \mathbf{g} is the gravitational acceleration vector. The flux of momentum is the result of the surface-force densities; these are represented by the stress tensor, i.e., $\phi = -\boldsymbol{\sigma}$, where $\boldsymbol{\sigma}$ is the Cauchy stress tensor whose component σ_{ij} refers to the force per unit area acting in the j direction on a surface (plane) which is perpendicular to the i direction. These substitutions lead to the momentum conservation equation

$$\frac{\partial(\rho \mathbf{u})}{\partial t} + \nabla \cdot (\rho \mathbf{u} \mathbf{u}) = \nabla \cdot \boldsymbol{\sigma} + \rho \mathbf{g}. \quad (2.4)$$

The Cauchy stress tensor is determined by the type of material and can be written as

$$\boldsymbol{\sigma} = -p\boldsymbol{\delta} + \boldsymbol{\tau}, \quad (2.5)$$

where p represents the pressure, $\boldsymbol{\delta}$ is the unit tensor, and $\boldsymbol{\tau}$ is the viscous stress (or extra-stress) tensor. Substituting Eq. (2.5) into the momentum conservation equation Eq. (2.4) yields

$$\frac{\partial(\rho\mathbf{u})}{\partial t} + \nabla \cdot (\rho\mathbf{u}\mathbf{u}) = -\nabla p + \nabla \cdot \boldsymbol{\tau} + \rho\mathbf{g}. \quad (2.6)$$

For an incompressible fluid, the momentum conservation equation becomes

$$\rho\left(\frac{\partial\mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla\mathbf{u}\right) = -\nabla p + \nabla \cdot \boldsymbol{\tau} + \rho\mathbf{g}. \quad (2.7)$$

In this thesis, the flows considered are isothermal so that the energy conservation equation will not be discussed. In fact, for the isothermal flows, the mass and momentum conservation equations suffice. In addition, the constitutive equations, which relate the stress tensor to the fluid deformation, will be discussed in the next section.

2.2 Rheological Model

The system of the conservation equations has more unknowns than equations, and so to close the system we need additional equations called constitutive equations. For the viscous or extra-stress tensor $\boldsymbol{\tau}$, a constitutive equation is a rheological equation of state which describes the stress in the fluid as a function of the rate-of-strain or strain that the fluid experiences.

Fluids can be classified as Newtonian and non-Newtonian. A Newtonian fluid is one in which viscous stress is linear in the rate-of-strain. More specifically, for a general isotropic (no directional preference) isothermal Newtonian fluid, the constitutive equation is given by

$$\boldsymbol{\tau} = \mu \dot{\boldsymbol{\gamma}} - \frac{2}{3} \mu (\nabla \cdot \mathbf{u}) \boldsymbol{\delta}, \quad (2.8)$$

where the rate-of-strain (or rate-of-deformation) tensor $\dot{\boldsymbol{\gamma}}$, is defined by $\dot{\boldsymbol{\gamma}} = \nabla \mathbf{u} + (\nabla \mathbf{u})^T$ and μ is the constant dynamic viscosity.

For an incompressible fluid under isothermal conditions the constitutive equation becomes

$$\boldsymbol{\tau} = \mu \dot{\boldsymbol{\gamma}}. \quad (2.9)$$

A non-Newtonian fluid is one whose stress cannot be described by Eq. (2.8).

There are many ways for a fluid to be non-Newtonian. The following are some of these ways:

- A fluid whose stress is a nonlinear function of strain or rate-of-strain.
- A fluid that exhibits nonzero normal stress differences in shear flows.
- A fluid with a memory effect due to micro-structure.

Two main classifications of many non-Newtonian fluids are (time-independent) inelastic fluids and (time-dependent) viscoelastic fluids.

In either case, an important rheological characteristic of a non-Newtonian fluid is a shear-rate-dependent viscosity, $\eta(\dot{\gamma})$, where the shear rate $\dot{\gamma}$ is given by

$$\dot{\gamma} = \sqrt{\frac{1}{2}(\dot{\boldsymbol{\gamma}} : \dot{\boldsymbol{\gamma}})} = \sqrt{\frac{1}{2} \sum_{i,j} \dot{\gamma}_{ij} \dot{\gamma}_{ji}}. \quad (2.10)$$

If the predominant rheological characteristic of a fluid is a time-independent shear-rate-dependent viscosity, then the fluid may be modeled with a generalized Newtonian constitutive equation

$$\boldsymbol{\tau} = \eta(\dot{\gamma}) \dot{\boldsymbol{\gamma}}, \quad (2.11)$$

where a viscosity model must be specified for $\eta(\dot{\gamma})$. These models are empirical in nature.

Fluids with a shear-rate-dependent viscosity $\eta(\dot{\gamma})$ can be classified into:

- **Shear-thinning** or pseudoplastic fluids when $\eta(\dot{\gamma})$ decreases as the shear rate increases.
- **Shear-thickening** or dilatant fluids when $\eta(\dot{\gamma})$ is an increasing function of the shear rate.
- **Yield-value** or viscoplastic fluids when the fluid can sustain an applied (nonzero) stress without flowing. The stress below which the fluid does not flow is called a yield-value or yield-stress.

Figure 2.1 illustrates these shear-rate-dependent viscosity curves, while Fig. 2.2 shows the relationship between the shear stress and the shear rate for different types of fluid.

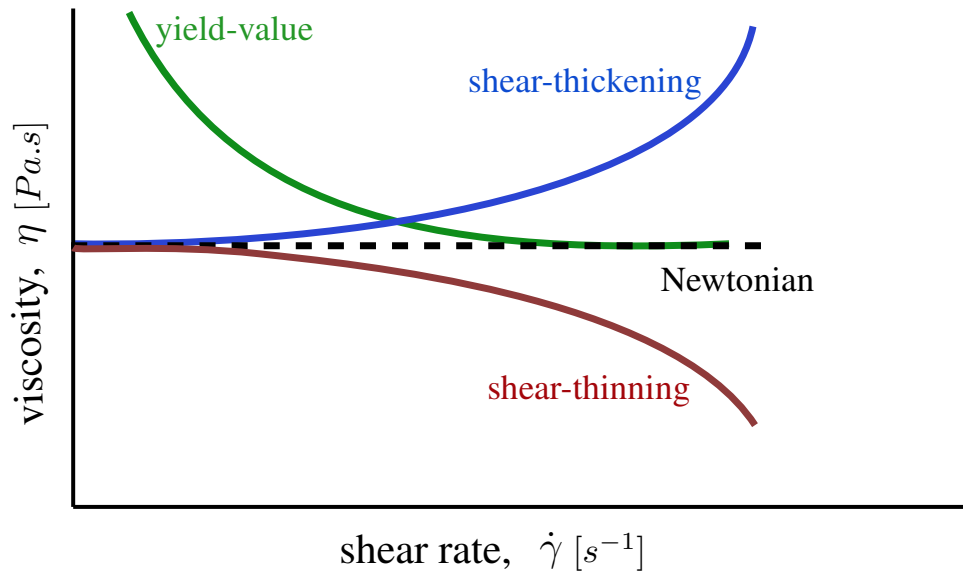


Figure 2.1: Viscosity versus shear rate curves for typical time-independent fluids.

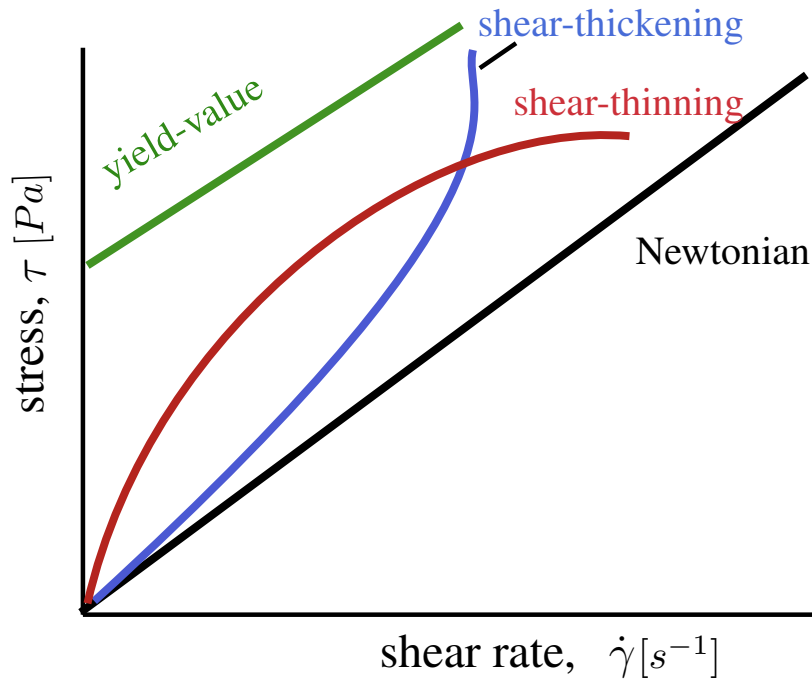


Figure 2.2: Stress versus shear rate curves for typical time-independent fluids.

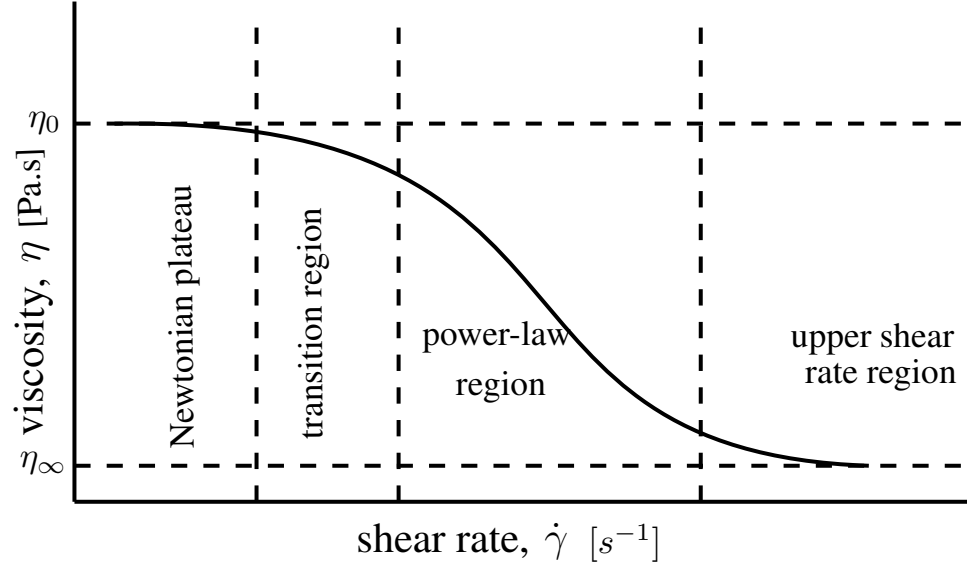


Figure 2.3: Log-log plot of viscosity versus shear rate for shear-thinning fluid.

Figure 2.3 shows the different regions in a viscosity curve for a shear-thinning fluid which are modeled. The η_0 and η_∞ are the limiting viscosity at zero and at infinite shear rate, respectively.

Common viscosity models are given below.

Power-Law Model

The simplest type of model to account for shear-rate-dependent viscosity is the power-law model of Ostwald [19] and de Waele [20]. This model has two parameters and it is given by

$$\eta(\dot{\gamma}) = m\dot{\gamma}^{n-1}, \quad (2.12)$$

where m represents the consistency index describing the vertical shift of the power-law region and has units of $Pa.s^n$, and n is the power-law exponent which describes the slope of viscosity curve $\eta(\dot{\gamma})$ in the power-law region and it is dimensionless.

Note that $n = 1$ corresponds to Newtonian fluids. With $0 < n < 1$, the fluid is shear-thinning or pseudo-plastic and with $1 < n < \infty$, the fluid is shear-thickening or dilatant. Note that the power-law model only describes the viscosity curve in the power-law region.

According to Bird et al. [21], this model is widely-used and very popular among many engineers because the analytical solutions are available for many problems and it can be used to get a rough estimate of the effect of the shear-rate-dependent viscosity. However, this model cannot describe behavior outside the power-law region, i.e., for small or large values of shear rate. Therefore, its use in CFD programs can lead to large computational errors.

An improvement to the power-law model can be achieved by describing the behavior of the viscosity outside the region of power law. This can be seen in the following model.

Carreau-Yasuda Model

This model is given by the expression (refer to Carreau [22] and Yasuda [23])

$$\frac{\eta - \eta_{\infty}}{\eta_0 - \eta_{\infty}} = (1 + (k\dot{\gamma})^a)^{(n-1)/a}, \quad (2.13)$$

where k is a time constant with units of seconds, whose reciprocal gives the shear rate at which the fluid changes from the constant viscosity behavior to the power-law behavior, a is a dimensionless parameter describing the transition between the zero-shear-rate-dependent viscosity and the power-law region, and the dimensionless power law exponent n describes the slope of $(\eta - \eta_\infty)/(\eta_0 - \eta_\infty)$ in the power-law region (see Fig. 2.3).

For many shear-thinning fluids, $a \approx 2$, and so as a special case of this model, the Bird-Carreau model or simply the Carreau model is given by taking a to be 2 as follows

$$\frac{\eta - \eta_\infty}{\eta_0 - \eta_\infty} = (1 + (k\dot{\gamma})^2)^{(n-1)/2}. \quad (2.14)$$

For more details about this model refer to Bird et al. [21].

Yield-Stress Models

A fluid has a yield-value (yield-stress), if it can sustain an applied (nonzero) stress without flowing. The yield-value (yield-stress) is the stress below which there is no relative flow. Common yield stress fluids include blood, toothpaste, and paints.

If we denote the yield stress by τ_0 , then for the yield-stress fluid

$$\begin{cases} \tau_{ij} = \eta(\dot{\gamma})\dot{\gamma}_{ij} & : |\boldsymbol{\tau}|^2 \geq \tau_0^2; \\ \dot{\gamma}_{ij} = 0 & : |\boldsymbol{\tau}|^2 < \tau_0^2. \end{cases}$$

where

$$|\boldsymbol{\tau}|^2 = 1/2(\boldsymbol{\tau} : \boldsymbol{\tau}).$$

$\eta(\dot{\gamma})$ is the apparent viscosity of the material beyond the yield point

$$\eta(\dot{\gamma}) = \frac{\tau_0}{\dot{\gamma}} + \hat{\eta}(\dot{\gamma}), \quad (2.15)$$

where $\hat{\eta}(\dot{\gamma})$ is the constitutive equation for the fluid after the yield stress is reached.

For yield-value fluids, the viscosity approaches infinity at small shear rate. Examples of yield-value models are the Bingham model (refer to Bingham [24]) in which the fluid behaves like a Newtonian fluid after the yield stress has been reached, i.e., $\hat{\eta}(\dot{\gamma}) = \mu =$ constant, and the Herschel-Bulkley model in which the fluid behaves like a power-law fluid after the yield stress has been reached, i.e., $\hat{\eta}(\dot{\gamma}) = m\dot{\gamma}^{n-1}$.

One common method of obtaining a yield stress value is to extrapolate the stress versus shear rate curve back to the stress intercept at zero shear rate. For more details about these models refer to Bird et al. [21] and Steffe [25].

2.3 Relevant Dimensionless Numbers

This section discusses two dimensionless numbers that are relevant in our study: the Reynolds number and the Deborah number.

The dimensionless Reynolds number Re , expresses the ratio of the inertial forces to the viscous forces, and it is defined by

$$Re = \frac{\rho u_0 L_0}{\eta^0}, \quad (2.16)$$

where u_0 is a characteristic velocity, L_0 is a characteristic length of the geometry and η^0 is the characteristic viscosity (which is μ for a Newtonian fluid).

A flow having a very small Re is called a Stokes flow, or a creeping flow. A flow is called a laminar flow if the Re falls below some critical value which depends on the considered geometry. Otherwise, it is called a turbulent flow. The flows in this thesis are restricted to the laminar regime. For example, if we consider the flow inside a tube with a fixed diameter, the flow is Stokes if $Re \ll 1$, the flow is laminar if $Re < 2100$, and turbulent if $Re > 4000$ and the range $2100 < Re < 4000$ represents the transition range.

Using Eq. (2.9) in the momentum equation of incompressible flow, Eq. (2.7), yields the Navier-Stokes equations

$$\rho \left(\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} \right) = -\nabla p + \mu \nabla^2 \mathbf{u} + \rho \mathbf{g}. \quad (2.17)$$

On the other hand, substituting Eq. (2.11) into Eq. (2.7) yields the momentum equation for a generalized Newtonian fluid as follows:

$$\rho \left(\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} \right) = -\nabla p + \nabla \cdot (\eta (\dot{\gamma}) \dot{\gamma}) + \rho \mathbf{g}. \quad (2.18)$$

In order to get dimensionless forms of Eqs. (2.17 and 2.18), let us define the dimensionless quantities as follows

$$\tilde{\mathbf{u}} = \frac{\mathbf{u}}{u_0}, \quad \tilde{\nabla} = L_0 \nabla, \quad \tilde{p} = \left(\frac{L_0}{\eta^0 u_0} \right) p, \quad \tilde{\eta} = \frac{\eta}{\eta^0},$$

Using the above dimensionless quantities, Eqs. (2.17 and 2.18) can be rewritten in the dimensionless form as (after dropping the tildes):

$$Re \left(\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} \right) = -\nabla p + \nabla^2 \mathbf{u} + \left(\frac{Re}{Fr} \right) \frac{\mathbf{g}}{g} \quad (2.19)$$

$$Re \left(\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} \right) = -\nabla p + \nabla \cdot (\eta \dot{\gamma}) + \left(\frac{Re}{Fr} \right) \frac{\mathbf{g}}{g}, \quad (2.20)$$

where g is the gravitational constant and $Fr = \frac{u_0^2}{gL_0}$ is the Froude number.

The dimensionless Deborah number De , has been proposed by Marcus Reiner [26] as a means of distinguishing between solids and liquids. This number is important to understand the behavior of viscoelastic materials and it becomes the fundamental number of rheology.

Reiner defined the Deborah number De in [26] as

$$De = \frac{t_{\text{fluid}}}{t_{\text{flow}}}, \quad (2.21)$$

where the numerator is time scale of the material's response, and the denominator is time scale of the flow process. If $De < De_{\text{critical}}$, the elastic effects can be neglected and we can treat the non-Newtonian fluid as a pure viscous material. Otherwise, the material has elastic effects that must be accounted for in the constitutive equation. The critical value of the Deborah number De_{critical} , depends on the flow and the geometry of the problem. This thesis considers fluids whose $De < De_{\text{critical}}$, i.e., no elastic effect will be studied and a generalized Newtonian constitutive equation Eq. (2.11), can be used.

2.4 Numerical Methods

2.4.1 Finite Volume Method

A discretization method is a method of approximating the partial differential equations by a system of algebraic equations for the variables at some set of discrete locations in space and time. In this subsection, we briefly recall the main explanations given by Jasak [27] about the description of the discretization process in Finite Volume Method (FVM) with the following properties in mind:

- The method is based on discretizing the integral form of the conservation equations over each control volume of the discrete domain. The basic quantities, such as mass and momentum, will therefore be conserved at the discrete levels.
- The method is applicable to both steady-state and transient calculations.
- The control volumes can be of any shape.
- System of partial differential equations is treated in a segregated way, meaning that they are solved one at a time in a sequential manner.

Polyhedral FVM discretizes the computational domain by splitting it into convex polyhedra bounded by convex polygons. These polyhedra, which are called control volumes (CVs), do not overlap and completely cover the computational domain.

The computational point P is located at the centroid of the control volume CV, such that

$$\int_{V_P} (\mathbf{x} - \mathbf{x}_P) dV = 0, \quad (2.22)$$

where V_P stands for the volume of the CV with centroid \mathbf{x}_P . The topology of the control volume is not important in this thesis. A typical polyhedral control volume is shown in Fig. 2.4, where S_f is the face area, \mathbf{n}_f is the face unit normal vector, N is the computational point of a neighboring control volume, \mathbf{d}_f is the vector between the computational points P and N , and \mathbf{r}_P is the vector between the origin and P . The control volume is bounded by a set of flat faces and each face is shared with only one neighboring control volume. All

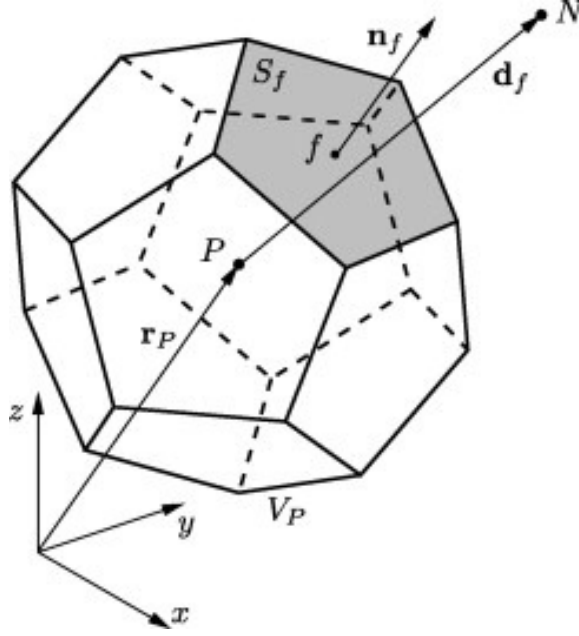


Figure 2.4: Arbitrary polyhedral control volume. Source: Jasak and Tukovic [1].

faces of the control volume will be marked with f , which also represents the centroid of the face.

The conservation equations described in Section 2.1 can be written in integral form for a tensorial quantity ϕ over a given control volume as follows

$$\int_{V_P} \underbrace{\frac{\partial \rho \phi}{\partial t}}_{\text{temporal derivative}} dV + \int_{V_P} \underbrace{\nabla \cdot (\rho \mathbf{u} \phi)}_{\text{convective term}} dV = \int_{V_P} \underbrace{\nabla \cdot (\rho \Gamma_\phi \nabla \phi)}_{\text{diffusion term}} dV + \int_{V_P} \underbrace{q_\phi(\phi)}_{\text{source term}} dV, \quad (2.23)$$

where Γ_ϕ is the diffusion coefficient of ϕ . Eq. (2.23) represents the integral of the general time-dependent convection-diffusion equation for a quantity ϕ . Since the diffusion term has a second-order derivative of ϕ in space, the Eq. (2.23) is a second-order equation. To

solve a second-order equation, the discretization method in space and time should be at least of order two.

In fact, the assumption that ϕ has a linear variation in space and time around the computational point P gives a second-order discretization method in space and time. The linear variation of ϕ is represented by

$$\phi(\mathbf{x}) = \phi_P + (\mathbf{x} - \mathbf{x}_P) \cdot (\nabla \phi)_P, \quad (2.24)$$

$$\phi(t + \Delta t) = \phi^t + \Delta t \left(\frac{\partial \phi}{\partial t} \right)^t, \quad (2.25)$$

where $\phi_P = \phi(\mathbf{x}_P)$, $(\nabla \phi)_P = \nabla \phi(\mathbf{x}_P)$, $\phi^t = \phi(t)$ and $\left(\frac{\partial \phi}{\partial t} \right)^t = \frac{\partial \phi}{\partial t}(t)$.

The procedures to derive a second-order discretization method in space and time for the general convection-diffusion equation are given below.

Spatial Discretization

- Discretization of the convection term

Using the divergence theorem on the convection term over a control volume V_P gives

$$\int_{V_P} \nabla \cdot (\rho \mathbf{u} \phi) dV = \oint_{\partial V_P} (\rho \mathbf{u} \phi \cdot \mathbf{n}) dS, \quad (2.26)$$

where ∂V_P is the surface bounding the volume V_P and \mathbf{n} is the unit normal vector on the boundary surface pointing outward.

Since the control volume is bounded by a series of flat faces, the surface integral can be written as

$$\oint_{\partial V_P} (\rho \mathbf{u} \phi \cdot \mathbf{n}) dS = \sum_f \left(\int_f (\rho \mathbf{u} \phi \cdot \mathbf{n}) dS \right). \quad (2.27)$$

Applying the assumption of linear variation simplifies the integral in the right hand side of Eq. (2.27) as follows

$$\int_f (\rho \mathbf{u} \phi \cdot \mathbf{n}) dS \approx (\rho \mathbf{u} \phi)_f \cdot \int_f \mathbf{n} dS + (\nabla(\rho \mathbf{u} \phi))_f : \int_f (\mathbf{x} - \mathbf{x}_f) \mathbf{n} dS, \quad (2.28)$$

where the subscript f implies the value of the variable at the centroid of the face. Notice that $(\nabla(\rho \mathbf{u} \phi))_f \cdot (\mathbf{x} - \mathbf{x}_f) \cdot \mathbf{n} = (\nabla(\rho \mathbf{u} \phi))_f : (\mathbf{x} - \mathbf{x}_f) \mathbf{n}$ and since f represents the centroid of the face (see Fig. 2.4), Eq. (2.28) becomes

$$\int_f (\rho \mathbf{u} \phi \cdot \mathbf{n}) dS \approx (\rho \mathbf{u} \phi)_f \cdot \int_f \mathbf{n} dS = (\rho \mathbf{u} \phi)_f \cdot \mathbf{S}, \quad (2.29)$$

where $\mathbf{S} = S_f \mathbf{n}$ is the outward-pointing face area vector.

The derivation of Eq. (2.29) is equivalent to applying a one-point Gauss quadrature rule to the left-hand side of Eq. (2.28). The one-point Gauss quadrature rule is exact for integrating polynomials of degree one.

Using Eqs. (2.26-2.29) the convection term is discretized as follows:

$$\begin{aligned}
\int_{V_P} \nabla \cdot (\rho \mathbf{u} \phi) dV &\approx \sum_f \mathbf{S} \cdot (\rho \mathbf{u} \phi)_f \\
&= \sum_f \mathbf{S} \cdot (\rho \mathbf{u})_f \phi_f \\
&= \sum_f F \phi_f,
\end{aligned} \tag{2.30}$$

where $F = \mathbf{S} \cdot (\rho \mathbf{u})_f$ is the mass flux through the face. There are several approaches to compute the face value of ϕ in Eq. (2.30). The following is a brief discussion of three of these methods.

The first method is called Central Differencing (CD), where the face value of ϕ is approximated by

$$\phi_f = f_x \phi_P + (1 - f_x) \phi_N, \tag{2.31}$$

where the subscripts of P and N imply the values of the variable at the computational points P and N , respectively. The interpolation factor f_x , is defined as the ratio of the distance between the face and the centroid \mathbf{x}_N , and the distance between the centroids \mathbf{x}_P and \mathbf{x}_N , i.e.,

$$f_x = \frac{\overline{fN}}{\overline{PN}}. \tag{2.32}$$

This method is second-order but unbounded. (Unbounded means that ϕ can take values outside its physically meaningful range.)

The Upwind Differencing (UD) method approximates the face value of ϕ according to the direction of the flow, i.e.,

$$\phi_f = \begin{cases} \phi_P & \text{if } F \geq 0; \\ \phi_N & \text{if } F < 0. \end{cases}$$

This method (which is a full upwinding technique) is bounded, but it is first order accurate and so it violates the order of accuracy of the discretization.

The last method is called Blended Differencing which can be described as a balance (weight) between Central Differencing and Upwind Differencing. In this method the face value of ϕ is given by

$$\phi_f = (1 - \gamma)(\phi_f)_{UD} + \gamma(\phi_f)_{CD}, \quad (2.33)$$

where $(\phi_f)_{UD}$ and $(\phi_f)_{CD}$ are the face values of ϕ computed by the Upwind and Central Differencing, respectively. The blending factor γ , $0 \leq \gamma \leq 1$ controls how much numerical diffusion will be introduced (Jasak [27]).

- Discretization of the diffusion term

By using a similar approach as before, the diffusion term is discretized by

$$\begin{aligned}
\int_{V_P} \nabla \cdot (\rho \Gamma_\phi \nabla \phi) dV &= \oint_{\partial V_P} (\rho \Gamma_\phi \nabla \phi) \cdot \mathbf{n} dS \\
&= \sum_f \int_f (\rho \Gamma_\phi \nabla \phi) \cdot \mathbf{n} dS \\
&\approx \sum_f (\rho \Gamma_\phi \nabla \phi)_f \cdot \int_f \mathbf{n} dS \\
&= \sum_f (\rho \Gamma_\phi \nabla \phi)_f \cdot \mathbf{S} \\
&= \sum_f (\rho \Gamma_\phi)_f \mathbf{S} \cdot (\nabla \phi)_f.
\end{aligned} \tag{2.34}$$

If the mesh is orthogonal, i.e, vectors \mathbf{d}_f and \mathbf{S} in Fig. 2.5 are parallel, then $\mathbf{S} \cdot (\nabla \phi)_f$ can be approximated by

$$\mathbf{S} \cdot (\nabla \phi)_f = |\mathbf{S}| \frac{\phi_N - \phi_P}{|\mathbf{d}_f|}. \tag{2.35}$$

For a non-orthogonal mesh, $\mathbf{S} \cdot (\nabla \phi)_f$ is split in two components as follows

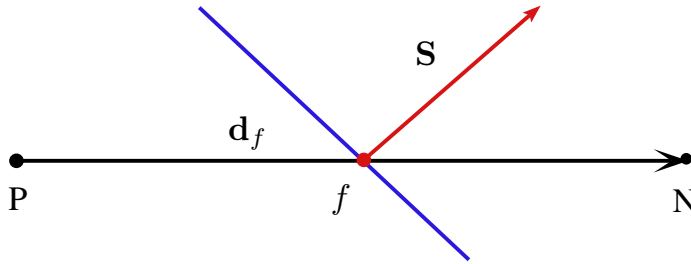


Figure 2.5: Vectors \mathbf{S} and \mathbf{d}_f on a non-orthogonal mesh.

$$\mathbf{S} \cdot (\nabla \phi)_f = \underbrace{\Delta \cdot (\nabla \phi)_f}_{\text{orthogonal contribution}} + \underbrace{\mathbf{k} \cdot (\nabla \phi)_f}_{\text{non-orthogonal contribution}}, \quad (2.36)$$

where Δ and \mathbf{k} need to satisfy $\mathbf{S} = \Delta + \mathbf{k}$.

The most robust, stable and computationally efficient approach to handle the mesh orthogonality decomposition is called over-relaxed approach. In this method, Δ is given by

$$\Delta = \frac{\mathbf{S} \cdot \mathbf{S}}{\mathbf{d}_f \cdot \mathbf{S}} \mathbf{d}_f. \quad (2.37)$$

Substituting Eq. (2.37) in Eq. (2.36) yields

$$\mathbf{S} \cdot (\nabla \phi)_f = |\Delta| \frac{\phi_N - \phi_P}{|\mathbf{d}_f|} + \mathbf{k} \cdot (\nabla \phi)_f. \quad (2.38)$$

Notice that the magnitude of Δ (as defined in Eq. (2.37)) increases with the increase of non-orthogonality (decrease of the denominator), which means that the importance of the term in ϕ_P and ϕ_N is caused to increase with the increase in non-orthogonality.

The decomposition of \mathbf{S} in the over-relaxed method is shown in Fig. 2.6.

For more details about other approaches refer to Jasak [27].

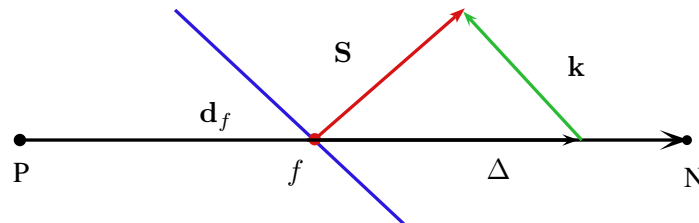


Figure 2.6: Non-orthogonality treatment in the over-relaxed approach.

- Discretization of the source term

The source terms are the terms of the convection-diffusion equation that cannot be written as temporal contribution, convection, and diffusion. These terms need to be linearized as follows

$$q_\phi(\phi) = q_{\mathbf{u}} + q_p\phi, \quad (2.39)$$

where the terms $q_{\mathbf{u}}$ and q_p may also depend on ϕ .

By using Gauss quadrature and the assumption of the linear variation, the volume integral of the source term is given by

$$\begin{aligned} \int_V q_\phi(\phi) dV &\approx (q_\phi(\phi))_P V_P \\ &= q_{\mathbf{u}} V_P + q_p V_P \phi_P. \end{aligned} \quad (2.40)$$

Temporal Discretization

By integrating the Eq. (2.23) in time we get

$$\begin{aligned} \int_t^{t+\Delta t} \left[\int_{V_P} \frac{\partial \rho \phi}{\partial t} dV + \int_{V_P} \nabla \cdot (\rho \mathbf{u} \phi) dV - \int_{V_P} \nabla \cdot (\rho \Gamma_\phi \nabla \phi) dV \right] dt = \\ \int_t^{t+\Delta t} \left(\int_{V_P} q_\phi(\phi) dV \right) dt. \end{aligned} \quad (2.41)$$

Using the spatial discretization results, Eq. (2.41) can be written as:

$$\int_t^{t+\Delta t} \left[\left(\frac{\partial \rho \phi}{\partial t} \right)_P V_P + \sum_f F \phi_f - \sum_f (\rho \Gamma_\phi)_f \mathbf{S} \cdot (\nabla \phi)_f \right] dt = \int_t^{t+\Delta t} (q_u V_P + q_p V_P \phi_P) dt. \quad (2.42)$$

Usually, Eq. (2.42) is called the semi-discretized form of the convection-diffusion equation.

The linear variation of ϕ with respect to time leads to the following approximations

$$\left(\frac{\partial \rho \phi}{\partial t} \right)_P = \frac{\rho_P^n \phi_P^n - \rho_P^o \phi_P^o}{\Delta t} \quad (2.43)$$

$$\int_t^{t+\Delta t} \phi(t) dt = \frac{1}{2} (\phi^o + \phi^n) \Delta t, \quad (2.44)$$

where superscripts o and n represent the values of the variable at time t and $t + \Delta t$, respectively. Using these approximations in Eq. (2.42) and by assuming that ρ and Γ_ϕ do not change with time yield the following second-order Crank-Nicholson method

$$\begin{aligned} \frac{\rho_P \phi_P^n - \rho_P \phi_P^o}{\Delta t} V_P + \frac{1}{2} \sum_f F \phi_f^n - \frac{1}{2} \sum_f (\rho \Gamma_\phi)_f \mathbf{S} \cdot (\nabla \phi)_f^n \\ + \frac{1}{2} \sum_f F \phi_f^o - \frac{1}{2} \sum_f (\rho \Gamma_\phi)_f \mathbf{S} \cdot (\nabla \phi)_f^o = \\ q_u V_P + \frac{1}{2} (q_p V_P \phi_P^n + q_p V_P \phi_P^o), \quad (2.45) \end{aligned}$$

where ϕ_f and $(\nabla\phi)_f$ depend on the values of the surrounding cells. In the Crank-Nicholson method, face and cell values of ϕ and $\nabla\phi$ are required for both old and new time levels, i.e., at time t and $t + \Delta t$. This method is unconditionally stable, but does not guarantee boundedness of the solution.

The explicit and implicit Euler discretization give an approximation of ϕ_f as follows:

$$\phi_f = \begin{cases} f_x \phi_P^o + (1 - f_x) \phi_N^o & \text{explicit Euler discretization ;} \\ f_x \phi_P^n + (1 - f_x) \phi_N^n & \text{implicit Euler discretization .} \end{cases}$$

On the other hand, these two methods approximate $\mathbf{S} \cdot (\nabla\phi)_f$ by

$$\mathbf{S} \cdot (\nabla\phi)_f = \begin{cases} |\Delta| \frac{\phi_N^o - \phi_P^o}{|\mathbf{d}_f|} + \mathbf{k} \cdot (\nabla\phi)_f^o & \text{explicit Euler discretization ;} \\ |\Delta| \frac{\phi_N^n - \phi_P^n}{|\mathbf{d}_f|} + \mathbf{k} \cdot (\nabla\phi)_f^n & \text{implicit Euler discretization .} \end{cases}$$

Both methods are first order, but contrary to the explicit Euler method, the implicit Euler method gives a bounded solution.

For more details about other temporal schemes (e.g. a second-order Backward Differencing) refer to Jasak [27].

For each control volume, Eq. (2.45) gives an algebraic equation:

$$a_P \phi_P^n + \sum_N a_N \phi_N^n = R_P, \quad (2.46)$$

where the coefficient a_P includes the contribution from all terms corresponding to ϕ^n , i.e., the temporal derivative, convection and diffusion terms as well as the linear part of the source term. The coefficient a_N include the corresponding terms for each of the neighboring points. The source term R_P includes the parts of the temporal derivative, convection and diffusion terms corresponding to the old time-level as well as the constant part of the source term.

Assembling the discrete equations, Eq. (2.46), for all control volumes, yields a system of algebraic equations of the form

$$\mathbf{Ax} = \mathbf{b} \tag{2.47}$$

in each time step, where \mathbf{A} is a sparse matrix containing the coefficients a_P and a_N , \mathbf{x} is the vector of unknown ϕ for all control volumes, and \mathbf{b} contains the source terms, R_P . This system is linear, i.e., \mathbf{A} is constant in each time step, if the original continuous convection-diffusion equation is linear in ϕ , or if terms have been linearized during discretization.

Discretization of the Flow Transport Equations

Now let us turn to the discretization of the flow transport equations. We shall start with the continuity and momentum equations for incompressible flow with a non-Newtonian viscosity, Eqs. (2.3 and 2.18):

$$\begin{aligned}\nabla \cdot \mathbf{u} &= 0, \\ \rho \left(\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} \right) &= -\nabla p + \nabla \cdot (\eta(\dot{\gamma}) \dot{\gamma}) + \rho \mathbf{g}.\end{aligned}$$

The momentum equation in the x_i direction can be interpreted as a convection-diffusion equation in which the scalar function ϕ is the velocity component in the x_i direction and the diffusion coefficient is viscosity. On the other hand, notice that the momentum equations contain a contribution from the pressure, which has no analog in the convection-diffusion equation. In fact, the pressure term may be regarded either as a source term or a surface force, but because of the coupling between the transport (continuity and momentum) equations, this term needs a special treatment. Although the viscous stress term in the momentum equation is similar to the diffusive term of the convection-diffusion equation, its contribution is more complex because the momentum equations are vector equations.

The discretized form of the continuity equation for incompressible flow is given by

$$\begin{aligned}
0 &= \int_{V_P} \nabla \cdot \mathbf{u} dV \\
&= \oint_{\partial V_P} \mathbf{u} \cdot \mathbf{n} dS \\
&= \sum_f \left(\int_f \mathbf{u} \cdot \mathbf{n} \right) dS \\
&\approx \sum_f \mathbf{S} \cdot \mathbf{u}_f.
\end{aligned} \tag{2.48}$$

The convection term of the integral form of the momentum equation is discretized as follows

$$\begin{aligned}
\int_{V_P} \nabla \cdot (\rho \mathbf{u} \mathbf{u}) dV &= \oint_{\partial V_P} (\rho \mathbf{u} \mathbf{u}) \cdot \mathbf{n} dS \\
&= \sum_f \int_f (\rho \mathbf{u} \mathbf{u}) \cdot \mathbf{n} dS \\
&\approx \sum_f (\rho \mathbf{u} \mathbf{u})_f \cdot \int_f \mathbf{n} dS \\
&= \sum_f \mathbf{S} \cdot (\rho \mathbf{u} \mathbf{u})_f \\
&= \sum_f F \mathbf{u}_f,
\end{aligned} \tag{2.49}$$

where $F = \mathbf{S} \cdot (\rho \mathbf{u})_f$ should satisfy the continuity equation.

Similar to the procedures of discretization of the convection term, the viscous stress term is discretized as follows

$$\begin{aligned}
\int_{V_P} \nabla \cdot \boldsymbol{\tau} dV &= \int_{V_P} \nabla \cdot (\eta(\dot{\boldsymbol{\gamma}}) \dot{\boldsymbol{\gamma}}) dV \\
&= \oint_{\partial V_P} (\eta(\dot{\boldsymbol{\gamma}}) \dot{\boldsymbol{\gamma}}) \cdot \mathbf{n} dS \\
&\approx \sum_f \mathbf{S} \cdot (\eta(\dot{\boldsymbol{\gamma}}) \dot{\boldsymbol{\gamma}})_f \\
&= \sum_f \mathbf{S} \cdot (\eta(\dot{\boldsymbol{\gamma}}))_f (\dot{\boldsymbol{\gamma}})_f \\
&= \sum_f \mathbf{S} \cdot (\eta(\dot{\boldsymbol{\gamma}}))_f (\nabla \mathbf{u} + \nabla \mathbf{u}^T)_f.
\end{aligned} \tag{2.50}$$

There are different methods to calculate $\mathbf{S} \cdot (\nabla \mathbf{u})_f$ as discussed previously.

Notice that the discretized convection and viscous stress terms can be written as

$$a_P \mathbf{u}_P + \sum_N a_N \mathbf{u}_N, \tag{2.51}$$

where a_P and a_N are functions of \mathbf{u} .

A semi-discretized form of the momentum equation is given by

$$a_P \mathbf{u}_P = \mathbf{H}(\mathbf{u}) - \nabla p. \tag{2.52}$$

At this stage, all terms have been discretized except the pressure gradient term.

The $\mathbf{H}(\mathbf{u})$ term in Eq. (2.52) is given by

$$\mathbf{H}(\mathbf{u}) = - \underbrace{\sum_N a_N \mathbf{u}_N}_{\text{transport part}} + \underbrace{\mathbf{r}}_{\text{source part}}, \quad (2.53)$$

where the transport part includes contributions from the discretization of the unsteady, convection, and viscous stress terms, while the source part \mathbf{r} , includes the source part of the unsteady term (assuming that there is no external forces).

Solving Eq. (2.52) for \mathbf{u}_P yields

$$\mathbf{u}_P = \frac{\mathbf{H}(\mathbf{u})}{a_P} - \frac{\nabla p}{a_P}. \quad (2.54)$$

The face interpolation of Eq. (2.54) is used to express the velocities on the cell face, i.e.,

$$\mathbf{u}_f = \left(\frac{\mathbf{H}(\mathbf{u})}{a_P} \right)_f - \left(\frac{1}{a_P} \right)_f (\nabla p)_f. \quad (2.55)$$

Substituting Eq. (2.55) into Eq. (2.48) gives the discrete pressure equation

$$\sum_f \mathbf{S} \cdot \left(\frac{1}{a_P} \right)_f (\nabla p)_f = \sum_f \mathbf{S} \cdot \left(\frac{\mathbf{H}(\mathbf{u})}{a_P} \right)_f. \quad (2.56)$$

The completely discretized form of the momentum equation is given by

$$a_P \mathbf{u}_P = \mathbf{H}(\mathbf{u}) - \sum_f \mathbf{S} p_f. \quad (2.57)$$

Therefore, the discrete system of equations for an incompressible, generalized Newtonian fluid consists of Eqs. (2.56) and (2.57).

A second-order Crank-Nicholson method for the momentum equation is given by (assuming that there is no external forces)

$$\begin{aligned} \frac{\rho_P \phi_P^n - \rho_P \phi_P^o}{\Delta t} V_P + \frac{1}{2} \sum_f F \phi_f^n + \frac{1}{2} \sum_f p_f^n \mathbf{S} - \frac{1}{2} \sum_f (\rho \mathbf{S} \cdot (\eta(\dot{\gamma}))_f (\nabla \mathbf{u} + \nabla \mathbf{u}^T)^n)_f \\ + \frac{1}{2} \sum_f F \phi_f^o + \frac{1}{2} \sum_f p_f^o \mathbf{S} - \frac{1}{2} \sum_f (\rho \mathbf{S} \cdot (\eta(\dot{\gamma}))_f (\nabla \mathbf{u} + \nabla \mathbf{u}^T)^o)_f = 0, \end{aligned} \quad (2.58)$$

where ϕ is the velocity component in the x_i direction.

For each control volume, Eq. (2.58) gives an algebraic equation

$$a_P \phi_P^n + \sum_N a_N \phi_N^n = R_P, \quad (2.59)$$

where the coefficient a_P includes the contribution from all terms corresponding to ϕ^n , i.e., the temporal derivative, convection and viscosity terms. The coefficient a_N include the corresponding terms for each of the neighboring points. The source term R_P includes the parts of the temporal derivative, convection and viscous stress terms corresponding to the

old time-level as well as the discretized pressure gradient term.

The nonlinearity of the algebraic system is lagged and the coefficients a_P and a_N are computed using the most current guess for velocity field. This approach requires iteration over the non-linear terms. Specifically, the algebraic system is solved several times, with the fluxes being updated each time, until it has converged.

So, after spatial and temporal discretization of the flow transport equations, as well as linearization (as described above), a linear system of algebraic equations of the form

$$\mathbf{Ax} = \mathbf{b} \tag{2.60}$$

is assembled in each time step. The matrix \mathbf{A} is a constant sparse matrix (only a small fraction of its entries are non zero), with coefficients a_P on the diagonal and a_N off the diagonal, \mathbf{x} is the vector of unknowns ϕ for all control volumes and \mathbf{b} is the source term vector.

In simulations with small time-step the linear pressure-velocity coupling is stronger than the non-linear coupling represented in the convection and viscosity terms. There are two different approaches to deal with the pressure-velocity coupling: The first is called simultaneous approach, where the complete system of equations are solved simultaneously over the whole domain. The second approach is called the segregated approach and it is based on solving the equations in sequence. More details about the most popular methods of the segregated approach are given in the next subsection.

2.4.2 Pressure-velocity coupling

The most widely-used pressure-based segregated algorithms are SIMPLE (Patankar [28]) which stands for Semi-Implicit Method for Pressure-Linked Equations and PISO (Issa [29]) which stands for Pressure-Implicit with Splitting of Operators.

SIMPLE

The SIMPLE algorithm is a predictor-corrector procedure for the calculation of pressure on the staggered grid centered around the cell faces.

Before discussing the algorithm, let us start with an important concept. A matrix $\mathbf{A} = (a_{ij})$ is called diagonally equal if and only if $a_{ii} = \sum_{j=1, j \neq i} a_{ij}$, and \mathbf{A} is diagonal dominant if and only if it is diagonally equal and

$$a_{ii} > \sum_{\substack{j=1 \\ j \neq i}} a_{ij} \quad (2.61)$$

for at least one row of \mathbf{A} . The diagonal dominance property is required to guarantee convergence when using an iterative method like SIMPLE.

To increase the diagonal dominance of the matrix resulting from discretizing the momentum conservation equation, we shall use the under-relaxed form which can be

obtained through an artificial term added to both sides of Eq. (2.59)

$$a_P \phi_P^k + \frac{1 - \alpha_u}{\alpha_u} a_P \phi_P^k + \sum_N a_N \phi_N^k = R_P + \frac{1 - \alpha_u}{\alpha_u} a_P \phi_P^{k-1}, \quad (2.62)$$

which can be rewritten as

$$\frac{a_P}{\alpha_u} \phi_P^k + \sum_N a_N \phi_N^k = R_P + \frac{1 - \alpha_u}{\alpha_u} a_P \phi_P^{k-1}, \quad (2.63)$$

where ϕ^{k-1} here represents the velocity component in the x_i direction from the previous iteration or the initial velocity guess (in the first iteration), and α_u is the velocity under-relaxation factor ($0 < \alpha_u \leq 1$).

The SIMPLE algorithm in OpenFOAM is used for steady-state flows and it can be summarized as follows

1. Solve the under-relaxed form of the discretized momentum equation, Eq. (2.63), where the pressure gradient term is calculated using the guessed value p^* which is either the initial pressure guess or the pressure field from the previous iteration. This stage is called the momentum predictor.
2. Define an intermediate velocity field \mathbf{u}^* (which may not satisfy the continuity requirement) by

$$\mathbf{u}^* = \frac{\mathbf{H}(\mathbf{u})}{a_P}, \quad (2.64)$$

and compute the corresponding flux field according to

$$\Phi_f^* = \mathbf{u}_f^* \cdot \mathbf{S}. \quad (2.65)$$

3. Solve the discrete pressure equation

$$\sum_f \left[\left(\frac{1}{a_p} \right)_f (\nabla p^k)_f \right] \cdot \mathbf{S} = \sum_f \Phi_f^*, \quad (2.66)$$

to obtain the new pressure distribution p^k .

4. Correct the mass fluxes at the cell faces to satisfy the continuity requirement:

$$\Phi_f = \Phi_f^* - \left(\frac{1}{a_p} \right)_f (\nabla p^k)_f \cdot \mathbf{S}. \quad (2.67)$$

5. Apply an explicit under-relaxation to the pressure field as follows

$$p^r = p^* + \alpha_p (p^k - p^*), \quad (2.68)$$

where α_p is the pressure under-relaxation factor ($0 < \alpha_p \leq 1$). According to Peric [30], the recommended values for the under-relaxation are $\alpha_p = 0.2$ and $\alpha_u = 0.8$.

6. Correct the velocities on the basis of the relaxed pressure field by using Eq. (2.54).

This step is the explicit velocity correction step.

7. Set $p^* = p^r$ for the next momentum corrector step.
8. Steps 1 to 7 are called the pressure velocity iterations. Repeat these steps until convergence.

The spatial and temporal discretization discussed previously generate a linear system of algebraic equations for velocity in step 1 and for pressure in step 3. These equations will be solved by using a linear solver (which will be discussed further in the next subsection). The initial and final residuals are the calculated residuals before and after the linear system is solved, respectively. The linear solver iterates until one of these two criteria is satisfied:

1. The final residual for the variable (e.g. \mathbf{u} , p , etc) falls below the specified absolute tolerance.
2. The ratio $\frac{\text{final residual}}{\text{initial residual}}$ is less than the specified relative tolerance.

By using the SIMPLE algorithm in OpenFOAM, the pressure-velocity iterations are performed until the initial residual for each variable is less than the corresponding value specified under `residualControl` given in the `<case>/system/fvSolution` file.

PISO

The PISO was developed originally for non-iterative computation of unsteady compressible flows. In this algorithm there is more than one corrector step and so it requires more computational time per velocity-pressure iteration, but can dramatically decrease the number of iterations to convergence, thus reducing overall computational time, and it is specially beneficial for transient problem.

In PISO, several pressure correctors are used with a single momentum equation. So unlike the SIMPLE algorithm, there is no need to under-relax the velocity or the pressure.

The PISO algorithm in OpenFOAM is used for transient flows and it can be summarized as

1. Solve the discretized momentum equation, Eq. (2.59).

2. Define an intermediate velocity field \mathbf{u}^* by

$$\mathbf{u}^* = \frac{\mathbf{H}(\mathbf{u})}{a_P},$$

and compute the corresponding flux field according to

$$\Phi_f^* = \mathbf{u}_f^* \cdot \mathbf{S}.$$

3. Solve the discrete pressure equation

$$\sum_f \left[\left(\frac{1}{a_P} \right)_f (\nabla p^n)_f \right] \cdot \mathbf{S} = \sum_f \Phi_f^*,$$

to obtain the new estimate of the pressure distribution p^n . This step is called the pressure solution.

4. Correct the velocity field according to the new pressure distribution

$$\mathbf{u}_f^n = \mathbf{u}_f^{*n} - \left(\frac{1}{a_P} \nabla p^n \right)_f,$$

where \mathbf{u}_f^{*n} is the current guess for the face value of \mathbf{u}^* . This step is called the explicit velocity correction.

5. Repeat steps 2-4 for a fixed number of times.

6. Increase the time step and repeat from step 1.

At each time step, the pressure-velocity iterations are performed a specified number of times. In OpenFOAM, the parameters that control the PISO algorithm are located in the `<case>/system/fvSolution` file. These parameters specify the number of times that the velocity and the pressure fields are corrected with each other (PISO loop) and the number of times that the pressure equation is solved in each PISO loop.

In every time step, the linear solver iterates at the last PISO loop until the residual of the

variable falls below the specified tolerance `<variable>Final`. Also, in `fvSolution` file the variable `pCorr` is an initial pressure calculation that is done before the first iteration. There will be a `pCorr` loop for each iteration if the mesh is moving.

2.4.3 Linear Solvers

This subsection describes the linear solvers used in the simulations presented in this thesis. As shown in Subsection 2.4.1, the discretization process leads to linear systems of equations in the unknown quantities ϕ . These linear systems are expressed as

$$\mathbf{Ax} = \mathbf{b}, \tag{2.69}$$

where $\mathbf{A} = (a_{ij})$ is a coefficient matrix, \mathbf{x} is a vector of unknowns, and \mathbf{b} is the source term.

The matrix \mathbf{A} is positive definite (respectively, positive semidefinite) if \mathbf{A} is symmetric and $(\mathbf{Ax}, \mathbf{x}) > 0$ (respectively, $(\mathbf{Ax}, \mathbf{x}) \geq 0$) for all non-zero vector \mathbf{x} , where the inner product of two vectors \mathbf{x}, \mathbf{y} is computed by $(\mathbf{x}, \mathbf{y}) = \mathbf{x}^T \mathbf{y}$.

The linear system given in Eq. (2.69) are solved iteratively. The general idea of iterative methods is to start with an initial guess, and improve it systematically in every iteration step until sufficient accuracy is reached.

Jacobi and Gauss-Seidel Iteration

In this iteration method, the matrix \mathbf{A} is decomposed into

$$\mathbf{A} = \mathbf{D} + -\mathbf{E} + -\mathbf{F},$$

where $\mathbf{D} = (d_{ij})$ is a diagonal matrix with $d_{ii} = a_{ii}$, $-E$ and $-F$ are the strictly lower and upper triangular parts of A , respectively.

The Jacobi iteration corrects the i -th component of the current approximate solution \mathbf{x}_k to annihilate the i -th component of the residual vector at step k given by

$$\mathbf{r}_k^i = (\mathbf{b} - \mathbf{A}\mathbf{x}_k)^i$$

This method can be written in vector form as

$$\mathbf{x}_{k+1} = \mathbf{D}^{-1}(\mathbf{E} + \mathbf{F})\mathbf{x}_k + \mathbf{D}^{-1}\mathbf{b}. \quad (2.70)$$

In this method, the components of \mathbf{x}_{k+1} depend only on the components of \mathbf{x}_k .

To speed up the convergence, the approximate solution is updated immediately after the new component is determined. This method is called the Gauss-Seidel iteration which is

given by

$$\mathbf{x}_{k+1} = (\mathbf{D} - \mathbf{E})^{-1} \mathbf{F} \mathbf{x}_k + (\mathbf{D} - \mathbf{E})^{-1} \mathbf{b}. \quad (2.71)$$

Conjugate Gradient Method (CG)

For a given $m \times m$ matrix \mathbf{A} and an m -vector \mathbf{b} , the k^{th} Krylov subspace \mathcal{K}_k generated by \mathbf{A} and \mathbf{b} is defined by

$$\mathcal{K}_k = \text{span} \{ \mathbf{b}, \mathbf{A}\mathbf{b}, \mathbf{A}^2\mathbf{b}, \dots, \mathbf{A}^{k-1}\mathbf{b} \}.$$

Let $\mathbf{x}_* = \mathbf{A}^{-1}\mathbf{b}$ be the exact solution of the system (2.69), where \mathbf{A} is positive definite matrix. The error at each iteration step k is $\boldsymbol{\varepsilon}_k = \mathbf{x}_* - \mathbf{x}_k$. The conjugate gradient algorithm gives a sequence of approximate solutions $\{\mathbf{x}_k \in \mathcal{K}_k\}$ such that the \mathbf{A} -norm of the error at each iteration step, $\|\boldsymbol{\varepsilon}_k\| = \sqrt{\boldsymbol{\varepsilon}_k^T \mathbf{A} \boldsymbol{\varepsilon}_k}$, is minimized.

Note that the residuals \mathbf{r}_k 's are orthogonal and the search directions \mathbf{p}_k are \mathbf{A} -conjugate, i.e., $(\mathbf{A}\mathbf{p}_k, \mathbf{p}_j) = 0$ when $j < k$. The conjugate Gradient algorithm is summarized in Algorithm 1. For more details refer to Trefethen and Bau [31]. The finite volume discretization of the pressure equation leads to a symmetric matrix which can be solved by the CG algorithm. Note that this algorithm is not applicable to the asymmetric matrices generated by the finite volume discretization of the velocity equations, which can be solved

Algorithm 1 The Conjugate Gradient

set $\mathbf{r}_0 = \mathbf{b} - \mathbf{A}\mathbf{x}_0$, $\mathbf{p}_0 = \mathbf{r}_0$	# initialization step
for $k = 0, 1, 2, 3, \dots$	# repeat until convergence is reached
$\alpha_k = (\mathbf{r}_k, \mathbf{r}_k) / (\mathbf{A}\mathbf{p}_k, \mathbf{p}_k)$	# step length
$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k$	# approximate solution
$\mathbf{r}_{k+1} = \mathbf{r}_k - \alpha_k \mathbf{A}\mathbf{p}_k$	# residual
$\beta_k = (\mathbf{r}_{k+1}, \mathbf{r}_{k+1}) / (\mathbf{r}_k, \mathbf{r}_k)$	# improvement this step
$\mathbf{p}_{k+1} = \mathbf{r}_{k+1} + \beta_k \mathbf{p}_k$	# new search direction
end	

by the following algorithm.

Bi-conjugate Gradient Method (BiCG)

In the Bi-conjugate Gradient method, two distinct Krylov subspaces, $\mathcal{K}_k(\mathbf{A}, \mathbf{v}_k)$ and $\mathcal{K}_k(\mathbf{A}^T, \mathbf{w}_1)$, are used for solving linear systems. The approximate solution \mathbf{x}_k is chosen such that the residual \mathbf{r}_k is orthogonal to $\mathcal{K}_k(\mathbf{A}^T, \mathbf{w}_1)$, where \mathbf{w}_1 is any vector satisfies $(\mathbf{w}_1, \mathbf{v}_1) = 1$ with $\mathbf{v}_k \in \mathcal{K}_k(\mathbf{A}, \mathbf{v}_k)$. As an application of the BiCG, one can choose $\mathbf{w}_1 = \mathbf{v}_1 = \mathbf{b} / \|\mathbf{b}\|_2$. As discussed in Trefethen and Bau [31], this choice leads to Algorithm 2.

Note that the inner products $(\mathbf{r}_j, \mathbf{s}_k)$ and $(\mathbf{A}\mathbf{p}_j, \mathbf{q}_k)$ are zero when $j < k$.

Algorithm 2 The Bi-Conjugate Gradient

set $\mathbf{x}_0 = 0, \mathbf{r}_0 = \mathbf{q}_0 = \mathbf{p}_0 = \mathbf{s}_0 = \mathbf{b}/\ \mathbf{b}\ _2$	# initialization step
for $k = 1, 2, 3, \dots$	# repeat until convergence is reached
$\alpha_k = (\mathbf{r}_{k-1}, \mathbf{s}_{k-1}) / (\mathbf{A}\mathbf{p}_{k-1}, \mathbf{q}_{k-1})$	# step length
$\mathbf{x}_k = \mathbf{x}_{k-1} - \alpha_k \mathbf{p}_{k-1}$	# approximate solution
$\mathbf{r}_k = \mathbf{r}_{k-1} - \alpha_k \mathbf{A}\mathbf{p}_{k-1}$	# first residual
$\mathbf{s}_k = \mathbf{s}_{k-1} - \alpha_k \mathbf{A}^T \mathbf{q}_{k-1}$	# second residual
$\beta_k = (\mathbf{r}_k, \mathbf{s}_k) / (\mathbf{r}_{k-1}, \mathbf{s}_{k-1})$	# improvement this step
$\mathbf{p}_k = \mathbf{r}_k + \beta_k \mathbf{p}_{k-1}$	# first new search direction
$\mathbf{q}_k = \mathbf{s}_k + \beta_k \mathbf{q}_{k-1}$	# second new search direction
end	

Generalized Geometric Algebraic Multi-grid Method (GAMG)

The multi-grid method is based on generating a solution for the linear system on a coarser mesh, and take this solution as an initial guess to obtain an accurate solution on the fine mesh. The steps of multi-grid method can be summarized as follows:

- 1) Compute \mathbf{x}_k on the fine mesh by using a few Jacobi or Gauss-Seidel iterations.
- 2) Obtain the residual $\mathbf{r}_k = \mathbf{Ax} - \mathbf{Ax}_k = \mathbf{A}\boldsymbol{\varepsilon}_k$.
- 3) Project \mathbf{r}_k from the fine mesh to the coarse mesh by means of the restriction matrix \mathbf{R} , i.e., $\mathbf{r}_k^H = \mathbf{R}\mathbf{r}_k$, and $\mathbf{A}_k^H = \mathbf{RA}$, to obtain $\mathbf{A}_k^H \boldsymbol{\varepsilon}_k^H = \mathbf{r}_k^H$.
- 4) On the coarse mesh, solve $\mathbf{A}_k^H \boldsymbol{\varepsilon}_k^H = \mathbf{r}_k^H$ for $\boldsymbol{\varepsilon}_k^H$.
- 5) Transferring the correction vector $\boldsymbol{\varepsilon}_k^H$ back to the fine mesh via $\boldsymbol{\varepsilon}_k = \mathbf{I}\boldsymbol{\varepsilon}_k^H$, where \mathbf{I} is the interpolation matrix.
- 6) Update the approximate solution \mathbf{x}_k on the fine mesh via $\mathbf{x}_k^{new} = \mathbf{x}_k + \boldsymbol{\varepsilon}_k$.

For more details about the multi-grid method, refer to Saad [32].

The matrices \mathbf{R} and \mathbf{I} are obtained via the process of agglomeration. In OpenFOAM, the switching from a coarse mesh to the fine mesh is handled by agglomeration of cells, either by a geometric agglomeration where cells are joined together, or by an algebraic agglomeration where matrix coefficients are joined (OpenFOAM [33]).

Note that the Jacobi or Gauss-Seidel iterations are used in multi-grid methods to accelerate the convergence. This is called smoothing and Jacobi or Gauss-Seidel iterations are called smoothers.

Preconditioners

In order to improve the performance of iterative methods for solving the linear system Eq. (2.69), a preconditioner matrix \mathbf{M} is used. If \mathbf{M} is nonsingular then the system

$$\mathbf{M}^{-1}\mathbf{A}\mathbf{x} = \mathbf{M}^{-1}\mathbf{b} \quad (2.72)$$

has the same solution as Eq. (2.69). The matrix \mathbf{M} is chosen such that $\mathbf{M}^{-1}\mathbf{A}$ is close to normal and that the condition number of $\mathbf{M}^{-1}\mathbf{A}$ is smaller than the one of \mathbf{A} . Recall that square matrix \mathbf{B} is normal if $\mathbf{B}^T\mathbf{B} = \mathbf{B}\mathbf{B}^T$ and the condition number of a nonsingular matrix \mathbf{B} is defined by

$$\text{cond}(\mathbf{B}) = \|\mathbf{B}\| \cdot \|\mathbf{B}^{-1}\| \quad (2.73)$$

By convention, $\text{cond}(B) = \infty$ if \mathbf{B} is singular.

The following subsection discusses a preconditioner for positive definite matrices.

Cholesky Factorization

For any $n \times n$ positive definite matrix \mathbf{A} , the Cholesky factorization is given by

$$\mathbf{A} = \mathbf{L}\mathbf{L}^T, \quad (2.74)$$

where $\mathbf{L} = (l_{ij})$ is a lower triangular matrix with $l_{ii} > 0$.

If \mathbf{A} is a sparse matrix, then usually \mathbf{L} is less sparse than \mathbf{A} . The incomplete Cholesky decomposition of a positive definite matrix \mathbf{A} is given by $\tilde{\mathbf{L}}\tilde{\mathbf{L}}^T$ where $\tilde{\mathbf{L}} = (\tilde{l}_{ij})$ is a lower triangular matrix which is more sparse than \mathbf{L} . Algorithm 3 (Datta [34]) computes the elements (\tilde{l}_{ij}) by using the exact Cholesky decomposition, except that $(\tilde{l}_{ij}) = 0$ if the corresponding entry a_{ij} is zero.

The Simplified Diagonal-based Incomplete Cholesky (DIC) is constructed to avoid

Algorithm 3 The Incomplete Cholesky Factorization

```

set  $\tilde{l}_{11} = \sqrt{a_{11}}$ 
for  $i = 1, 2, 3, \dots, m$ 
  for  $j = 1, 2, 3, \dots, i-1$ 
    if  $a_{ij} = 0$ , then  $\tilde{l}_{ij} = 0$  else
       $\tilde{l}_{ij} = \frac{1}{\tilde{l}_{ii}} \left( a_{ij} - \sum_{k=1}^{j-1} \tilde{l}_{ik} \tilde{l}_{jk} \right)$ 
    endif
  end
   $\tilde{l}_{ii} = \sqrt{a_{ii} - \sum_{k=1}^{i-1} \tilde{l}_{ik}^2}$ 
end

```

extracting square roots. This method decomposes the matrix \mathbf{A} into $\tilde{\mathbf{L}}\mathbf{D}\tilde{\mathbf{L}}^T$ where the

diagonal entries of D are computed by

$$d_{ii} = a_{ii} - \sum_{k=1}^{i-1} \tilde{l}_{ik}^2 d_k. \quad (2.75)$$

and the entries of $\tilde{\mathbf{L}}$ ($i > j$) are given by

$$\tilde{l}_{ij} = \begin{cases} \frac{1}{d_{ii}} \left(a_{ij} - \sum_{k=1}^{i-1} \tilde{l}_{ik} \tilde{l}_{jk} d_k \right) & \text{if } a_{ij} \neq 0; \\ 0 & \text{if } a_{ij} = 0. \end{cases}$$

The use of DIC in finding the inverse of a matrix is presented in Krishnamoorthy and Menon [35].

According to Jasak et al. [36], the CG and BiCG methods have a poor convergence rate. This convergence rate can be improved by using a preconditioner. For example, using the CG, and BiCG methods with a preconditioner \mathbf{M} yields the Preconditioned Conjugate Gradient and Preconditioned Bi-Conjugate Gradient methods described in Algorithm 4 and Algorithm 5, respectively. For more details refer to Barrett et al. [37].

Algorithm 4 The Preconditioned Conjugate Gradient

compute $\mathbf{r}_0 = \mathbf{b} - \mathbf{A}\mathbf{x}_0$ and $\mathbf{p}_0 = \mathbf{z}_0 = \mathbf{M}^{-1}\mathbf{r}_0$	# initialization step
for $k = 0, 1, 2, 3, \dots$	# repeat until convergence is reached
$\alpha_k = (\mathbf{r}_k, \mathbf{z}_k) / (\mathbf{A}\mathbf{p}_k, \mathbf{p}_k)$	# step length
$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k$	# approximate solution
$\mathbf{r}_{k+1} = \mathbf{r}_k - \alpha_k \mathbf{A}\mathbf{p}_k$	# original residual
$\mathbf{z}_{k+1} = \mathbf{M}^{-1}\mathbf{r}_{k+1}$	# residual for the preconditioned system
$\beta_k = (\mathbf{r}_{k+1}, \mathbf{z}_{k+1}) / (\mathbf{r}_k, \mathbf{z}_k)$	# improvement this step
$\mathbf{p}_{k+1} = \mathbf{z}_{k+1} + \beta_k \mathbf{p}_k$	# search direction
end	

Algorithm 5 The Preconditioned Bi-Conjugate Gradient

compute $\mathbf{r}_0 = \hat{\mathbf{r}}_0 = \mathbf{b} - \mathbf{A}\mathbf{x}_0$	# initialization step
for $k = 1, 2, 3, \dots$	# repeat until convergence is reached
solve $\mathbf{M}\mathbf{z}_{k-1} = \mathbf{r}_{i-1}$	# first preconditioned residual
solve $\mathbf{M}^T\hat{\mathbf{z}}_{k-1} = \hat{\mathbf{r}}_{k-1}$	# second preconditioned residual
$\xi_{k-1} = \mathbf{z}_k^T \hat{\mathbf{r}}_{k-1}$	
if $\xi_{k-1} = 0$ method fails	
if $k = 1$	
$\mathbf{p}_k = \mathbf{z}_{k-1}$	# first new search direction
$\hat{\mathbf{p}}_k = \hat{\mathbf{z}}_{k-1}$	# second new search direction
else	
$\beta_{k-1} = \xi_{k-1} / \xi_{k-2}$	
$\mathbf{p}_k = \mathbf{z}_{k-1} + \beta_{k-1}\mathbf{p}_{k-1}$	
$\hat{\mathbf{p}}_k = \hat{\mathbf{z}}_{k-1} + \beta_{k-1}\hat{\mathbf{p}}_{k-1}$	
endif	
$\mathbf{q}_k = \mathbf{A}\mathbf{p}_k$	
$\hat{\mathbf{q}}_k = \mathbf{A}^T\hat{\mathbf{p}}_k$	
$\alpha_k = \xi_{k-1} / (\hat{\mathbf{p}}_k^T \mathbf{q}_k)$	# step length
$\mathbf{x}_k = \mathbf{x}_{k-1} + \alpha_k \mathbf{p}_k$	# new approximate solution
$\mathbf{r}_k = \mathbf{r}_{k-1} - \alpha_k \mathbf{q}_k$	# first residual
$\hat{\mathbf{r}}_k = \hat{\mathbf{r}}_{k-1} - \alpha_k \hat{\mathbf{q}}_k$	# second residual
end	

The Incomplete Lower Upper Decomposition (ILU)

For a general matrix \mathbf{A} , one possible conditioner is the incomplete lower upper decomposition, i.e., $\mathbf{M} = \tilde{\mathbf{L}}\tilde{\mathbf{U}}$ where $\tilde{\mathbf{L}}$ and $\tilde{\mathbf{U}}$ are lower and upper triangular matrices, respectively. The entries of $\tilde{\mathbf{L}}$ and $\tilde{\mathbf{U}}$ are chosen so that certain entries of $\tilde{\mathbf{L}}\tilde{\mathbf{U}}$ match the corresponding entries of \mathbf{A} . In particular, the property that is preserved in $\tilde{\mathbf{L}}\tilde{\mathbf{U}}$ is the sparsity of \mathbf{A} . In the diagonal-based Lower upper (DILU) factorization, the preconditioner \mathbf{M} is given by

$$\mathbf{M} = (\mathbf{D} + \mathbf{L})\mathbf{D}^{-1}(\mathbf{D} + \mathbf{U}). \quad (2.76)$$

Algorithm 6 (Barrett et al. [37]) shows how to find the matrix \mathbf{M} .

Algorithm 6 DILU

```
Let  $S$  be the nonempty set  $\{(i, j) : a_{ij} \neq 0\}$ 
for  $k = 1, 2, 3, \dots$ 
     $d_{kk} = a_{kk}$ 
    for  $k = 1, 2, 3, \dots$ 
         $d_{kk} = 1/d_{kk}$ 
        for  $k = k + 1, k + 2, k + 3, \dots$ 
            if  $(k, j) \in S$  and  $(j, k) \in S$  then
                 $d_{jj} = d_{jj} - a_{jk}d_{kk}a_{kj}$ 
            endif
        end
    end
end
```

2.5 Mesh Motion

There are many physical phenomena where the shape of the domain is changing according to a prescribed boundary condition, causing a problem of preserving the quality and validity of the mesh during the simulation.

In FVM terms, preserving the validity of the mesh after deformation means preserving

1. The positivity of cell volumes and face areas.
2. The cell and face convexities.
3. The mesh non-orthogonality bounds.

These conditions prevent the mesh faces and cells from flipping during the deformation.

In this section we discuss an approach to move the internal points inside the domain corresponding to the movement of the boundary.

The integral form of the governing equation for a general property ϕ over an arbitrary moving volume V bounded by a closed surface S is given by (Jasak and Tukovic [38])

$$\frac{d}{dt} \int_{V(t)} \rho \phi dV + \oint_{S(t)} \rho \mathbf{n} \cdot (\mathbf{u} - \mathbf{u}_s) \phi dS - \oint_{S(t)} \rho \Gamma_\phi \mathbf{n} \cdot \nabla \phi dS = \int_{V(t)} q_\phi dV \quad (2.77)$$

where \mathbf{n} is the unit normal vector on the boundary surface pointing outward, \mathbf{u} is the fluid velocity, \mathbf{u}_s is the velocity of the boundary surface, Γ_ϕ is the diffusion coefficient and q_ϕ is

the volume source/sink of ϕ . Notice that Eq. (2.77) is similar to the convection-diffusion equation for ϕ mentioned in Subsection 2.4.1 but with \mathbf{u} replaced with $\mathbf{u} - \mathbf{u}_s$.

The relationship between the rate of change of the volume V and the velocity \mathbf{u}_s is defined by the space conservation law (SCL):

$$\frac{d}{dt} \int_{V(t)} dV - \oint_{S(t)} \mathbf{n} \cdot \mathbf{u}_s dS = 0 \quad (2.78)$$

Jazak and Tukovic [1] mentioned that the mesh validity constraints indicate that a domain could be considered as a solid body under large deformation, governed by the Piola-Kirchhoff stress-strain formulation:

$$\boldsymbol{\Sigma} = 2\mu\mathbf{E} + \lambda \text{tr}(\mathbf{E})\mathbf{I}, \quad (2.79)$$

where $\boldsymbol{\Sigma}$ is the second Piola-Kirchhoff stress tensor, $\mathbf{E} = \frac{1}{2}(\nabla \mathbf{v} + (\nabla \mathbf{v})^T + \nabla \mathbf{v} \cdot (\nabla \mathbf{v})^T)$ is the Green-Lagrangian strain tensor, \mathbf{v} is the displacement, and where μ and λ are the Lamé's coefficients. The resulting equation governing the deformation is non-linear in \mathbf{v} , and very expensive to solve. This mesh motion can be simplified by using the Laplace equation. Specifically, the motion of the internal points are determined by using the boundary motion as a boundary condition and where the mesh motion equation is expressed by Laplace equation with variable diffusivity

$$\nabla \cdot (\gamma \nabla \mathbf{w}) = 0 \quad (2.80)$$

where \mathbf{w} may represent either the velocity or the displacement of a cell, and γ is the variable diffusivity. The reason behind using the variable diffusivity in the Laplacian is to fix a problem of local deterioration in the mesh quality since the movement of points close to the moving boundary are larger than for the other points.

The solution of the Laplace equation, Eq. (2.80), is the motion function \mathbf{w} , which is continuous, smooth, regular and gives non-overlapping streamlines and so it passes the mesh validity constraints. This motion function is then used to determine \mathbf{u}_s in Eq. (2.77).

In OpenFOAM, there are two mesh-manipulation approaches:

1. Automatic mesh motion which is used when the topology of the mesh does not change, but instead only the spacing between nodes changes by stretching or squeezing.
2. Topological changes in the mesh which is used when the topology of the mesh changes during simulation.

In the application of peristaltic motion considered in this thesis the spacing between the nodes are just stretched or squeezed, and so the topology of the mesh is not changing during the simulation.

The approach of the automatic mesh motion needs an extra file called a `dynamicMeshDict` in the `constant` folder of the case. In this file, the user determine the solver and the diffusivity model that will be used for the mesh motion equation.

There are a number of available solvers in OpenFOAM to solve the mesh motion equation. By selecting the solver `velocityLaplacian`, the equations of cell motion are solved based on the Laplacian of the diffusivity times the cell motion velocity, and this solver needs to read an extra file called `pointMotionU` in the starting time folder, which determines the velocity at each boundary point.

Another solver is the `displacementLaplacian` solver which solves the equations of cell motion based on the Laplacian of the diffusivity times the cell displacement. This solver needs an extra file called `pointDisplacement` file in the starting time folder.

The solver `velocityLaplacian` was used in this thesis since it gave better results than `displacementLaplacian` for the simulations in which it was used.

The diffusivity model determines how the points should be moved when solving the equation of cell motion for each time step, for more details refer to Gonzalez [39]. In our simulations, the directional diffusivity model has been used where the deformation of the mesh is done proportional to the direction of the boundary point motion.

In OpenFOAM, the mapping between the old and new meshes using the `dynamicFvMesh` happens behind the scenes, and so the FVM physics solver just has to satisfy the moving mesh terms shown in Eq. (2.77) and it is independent of the mesh.

For more details about moving mesh in OpenFOAM refer to Jasak and Tukovic [1], Kassiotis [40], and Mordnia [41].

Chapter 3

Simulation of Flow in a Collapsed Elastic Tube

In this chapter¹ the flow of a shear-thinning, non-Newtonian fluid through a collapsed tube has been simulated without coupling the flow and the tube deformation. The main purpose of this Chapter is to validate the simulations with experimental data of Nahar et al. [43], and to gain insight into flow and material properties of the fluid which cannot be easily obtained by means of measurements.

¹The material contained in this chapter was previously published in Applied Rheology [42]

3.1 Experimental Details

The experimental study of the flow of a non-Newtonian fluid in a collapsible elastic tube is performed using the Starling Resistor set-up, shown schematically in Fig. 3.1. Here, a silicone elastic tube (20 mm inner diameter, 1 mm thickness and 320 mm long) is suspended in a controllable pressure chamber between two rigid aluminum tubes, and an inelastic shear-thinning aqueous solution is pumped through the tube at a steady flow rate.

The experiment includes the simultaneous measurement of both the deformed tube shape and the corresponding velocity flow field under the influence of compressive transmural pressures (internal minus external). The different characteristic pressures such as inlet (P_i), outlet (P_o) and external (P_e) are measured by the pressure sensors connected in the set-up. The distance between the pressure sensors P_i and P_o is 910 mm. The various states of the tube geometry are achieved by controlling the hydrostatic head connected with the water filled pressure chamber (P_e). The shapes of the tube are analyzed by means of the computer tomography method where several images are taken by rotating a camera at different angles around the pressure chamber. Contrast maximization of the images is applied to identify the grid lines drawn on the tube surface, and tube shapes are then constructed by the obtained projection beam lines. In addition, the pulsed ultrasound Doppler velocimetry technique is applied to monitor velocity profiles of the shear-thinning fluid flowing through the elastic tube of different degree of deformation (under different applied P_e). More details on the

materials, methods used and experimental procedure are described in Nahar et al. [43].

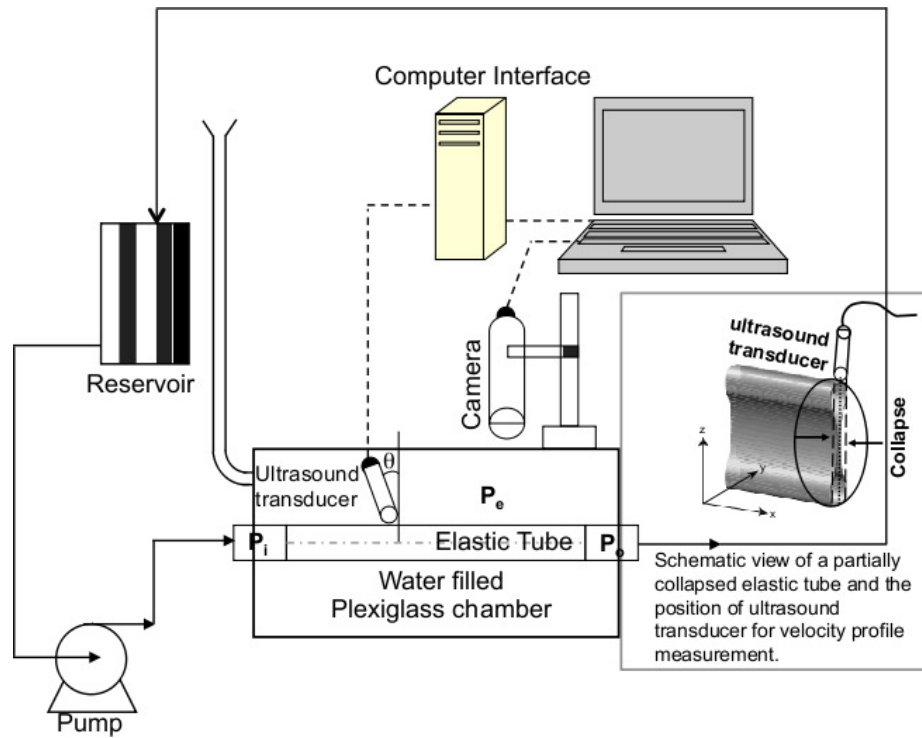


Figure 3.1: Schematic representation of the Starling Resistor for fluid flow behavior study through a collapsible elastic tube. The schematic of collapsed elastic tube and position of ultrasound transducer for velocity profile measurement is also inserted.

3.1.1 Material Properties

The shear-thinning fluid used is a non-Newtonian carboxymethyl-cellulose aqueous solution at 1.5% w/w with 0.1 M NaCl and $M_w = 2.5 \times 10^5$ g/mol (CMC 1.5%). According to Stranzinger [44], the CMC 1.5% solution is inelastic for concentrations up to 2%. The rheological measurements of this solution were carried out using a Physica rheometer (MCR 300, CC27) with cylindrical geometry and gap width = 1.13 mm, as is documented in Nahar et al. [45]. The measured shear rate dependent viscosity showed a shear-thinning behavior and is approximated by the Bird-Carreau equation discussed in Chapter 2

$$\eta - \eta_\infty = (\eta_0 - \eta_\infty)[1 + (k\dot{\gamma})^2]^{(n-1)/2} \quad (3.1)$$

where $\eta_0=0.1452$ Pa s, $\eta_\infty=0$, $k=0.02673$ s and $n=0.7588$. This approximation is illustrated in Fig. 3.2. Note that for $\eta_\infty=0$ and $k\dot{\gamma} \gg 1$, Eq. (3.1) reduces to the power-law

$$\eta = m\dot{\gamma}^{(n-1)}, \quad (3.2)$$

where $m = \eta_0 k^{n-1}$.

The density of this fluid is $\rho = 1000$ kg/m³.

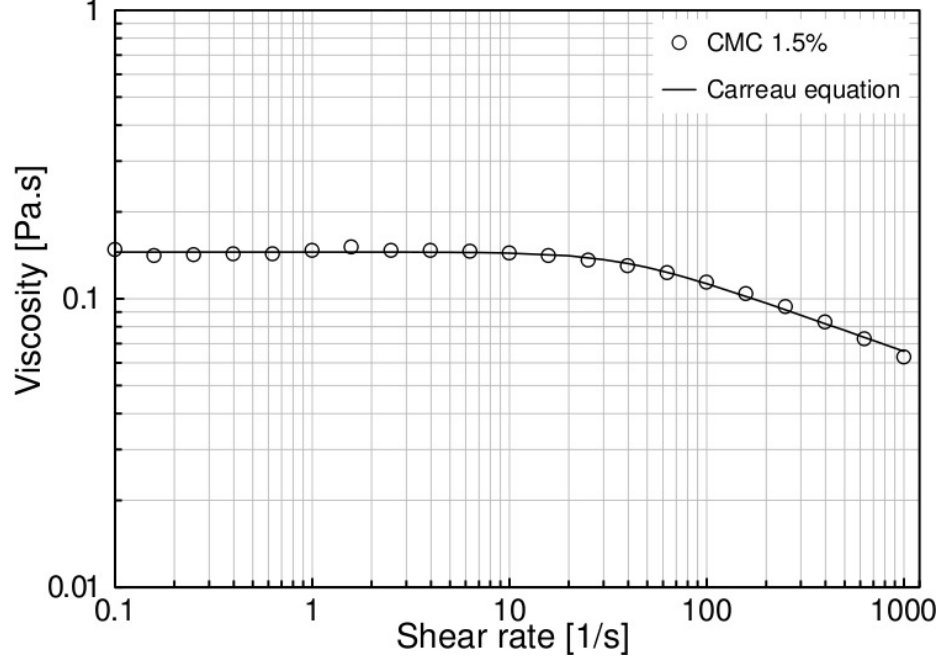


Figure 3.2: Viscosity curve for the non-Newtonian fluid.

3.2 Simulation Details

Computer simulations have been carried out for the non-Newtonian fluid flowing through the collapsed tube described in the previous section. For comparison purposes, the same simulation has been repeated for a Newtonian fluid whose viscosity corresponds to the zero-shear-rate-dependent viscosity $\eta_0 = 0.1452$ Pa s. The simulation details are schematically illustrated in Fig. 3.3. Note that the main flow direction is the negative x-direction and the main collapse occurs in the y-direction. Also, the line-of-measurement makes an angle of 70° with the flow direction. The simulations have been performed with the open source CFD environment OpenFOAM® [17] using *simpleFOAM*, which is a steady-state solver for incompressible flows utilizing the SIMPLE algorithm of Patankar

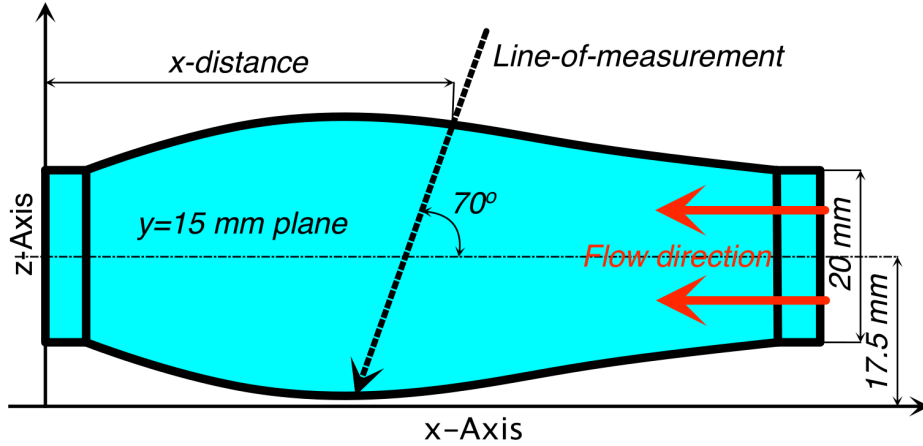


Figure 3.3: Schematic illustration of the flow through the collapsed tube in the $y=15$ mm plane. (The tube is collapsed in the y -direction.)

and Spalding [46] discussed in Chapter 2.

The deformed tube geometry has been obtained from computer tomography image analysis. More precisely, each cross-section has been reconstructed point by point using 31 measurements from the tube periphery. The cross-sections are equally spaced at 20 mm intervals along the x -direction, starting at 10 mm (outlet) and ending at 190 mm (inlet). This resulted in a geometry consisting of nine blocks. Because each block is topologically equivalent to a parallelepiped, the computational mesh obtained in this way is a structured, hexahedral, Cartesian mesh. A three-dimensional view of this mesh is shown in Fig. 3.4. As can be seen, the tube exhibits only small deformation at the inlet and outlet, and large deformation at several cross-sections closer to the outlet, between $x=30$ mm and $x=90$ mm, where the tube is almost fully collapsed.

The fluid flow is in the negative x -direction with the inlet cross-section at $x=190$ mm and the outlet at $x=10$ mm. The inlet boundary velocity was set to the average velocity of

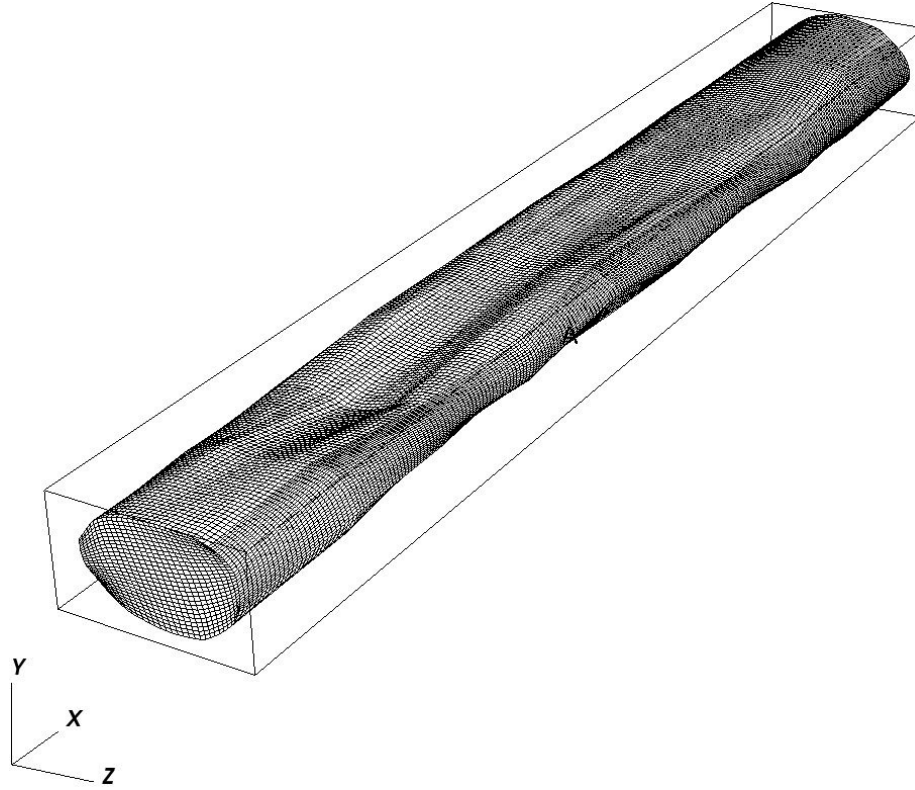


Figure 3.4: Three-dimensional view of the computational mesh.

$u_0=0.05411$ m/s The outlet boundary velocity was set to zero gradient and the velocities at the remaining fixed walls were set to zero. The pressure boundary conditions were zero at the outlet and zero gradient at the inlet and along the walls of the tube.

3.2.1 Mesh Dependence and Convergence

As described above, the computational mesh consists of nine structured, hexahedral, Cartesian blocks, each having a uniform cell distribution. The standard mesh has $27 \times 27 \times 27$ cells per block, which gives a total of 177'147 cells. The coarse and fine

meshes were obtained from the standard mesh by reducing, respectively increasing, the number of cells in each direction by a factor of 1.5. This led to the coarse mesh with $18 \times 18 \times 18$ cells per block, resulting in a total of 52'488 cells. Similarly, the fine mesh was refined to $40 \times 40 \times 40$ cells per block, resulting in a total of 576'000 cells.

The results of this mesh refinement study are shown in Fig. 3.5 for the non-Newtonian velocity profiles at one of the most deformed cross-section, namely at $x=70$ mm. This figure shows that the values of the standard mesh are almost identical to the ones of the fine mesh, but that there is a considerable discrepancy to the ones of the coarse mesh. Therefore, it can be concluded that the standard mesh exhibits sufficient mesh resolution. Note that the double peak in the velocity profile is the result of the almost collapsed tube shape, as is discussed in more detail below.

The convergence of the computation to steady-state is controlled by means of the residuals for the pressure and velocity equations in the SIMPLE algorithm. In order to assure sufficient convergence, the residuals for pressure and velocity have been increased, respectively decreased, by one order of magnitude from their standard values of 10^{-3} for pressure and 10^{-4} for velocity. This led to the smaller residuals of 10^{-4} and 10^{-5} , and to the respective larger residuals of 10^{-2} and 10^{-3} . The results of these variations are illustrated in Fig. 3.6 for the non-Newtonian velocity profiles at $x=70$ mm. As is seen, the results of the standard and small residual simulations are identical, which demonstrates that the standard residuals are sufficient to reach convergence.

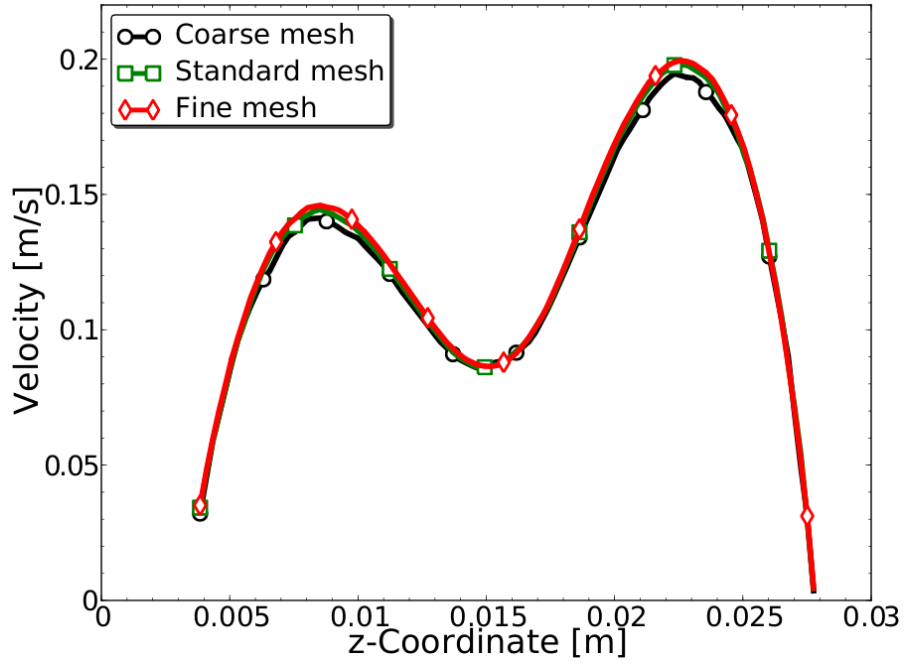


Figure 3.5: Mesh dependence study for the non-Newtonian velocity profile at $x=70$ mm.

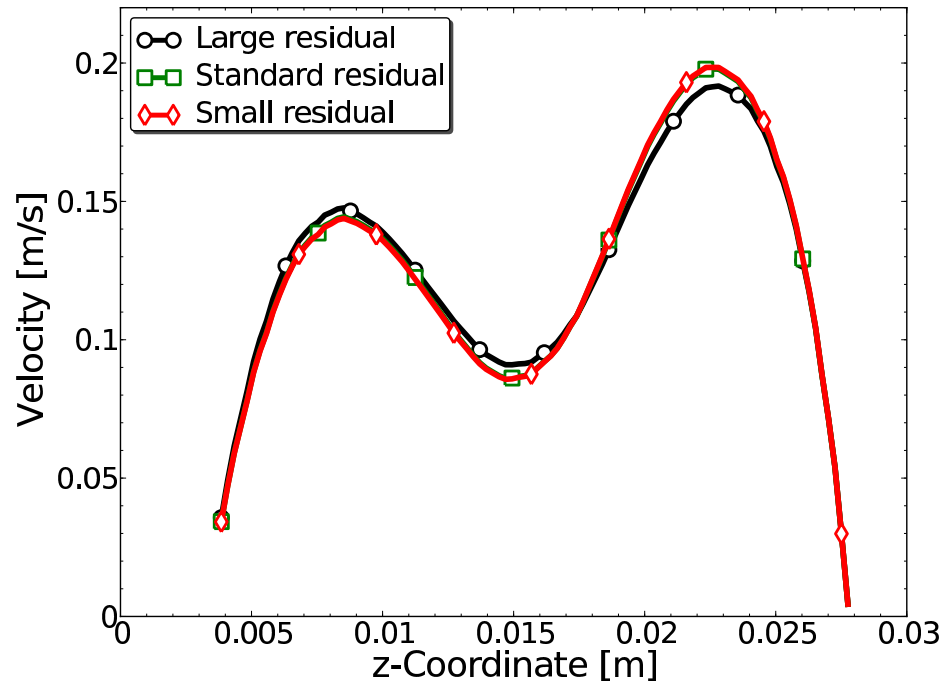


Figure 3.6: Convergence study for the non-Newtonian velocity profile at $x=70$ mm.

3.3 Results and Discussion

3.3.1 Velocity Comparison

The UVP technique assumes that the fluid velocity is unidirectional, i.e., there is a preferred flow direction (in our case, the x-direction) and the velocity components in the other directions are negligible. To verify this assumption, the simulated velocity magnitudes and velocity x-components have been plotted at various cross-sections throughout the tube. It has been found that the two quantities gave almost identical curves, showing that the UVP assumption was satisfied.

The velocities along the lines-of-measurement (see Fig. 3.3) are compared with the experimental values at several axial positions across the length of the tube. The corresponding velocity profiles are illustrated in Figs. 3.7, 3.9 and 3.11 for $x=70$ mm, $x=90$ mm and $x=150$ mm, respectively. Note that there is considerable uncertainty in determining the exact line-of-measurement and that small changes in the position or direction of this line can lead to relatively large fluctuations in the velocity data. Nevertheless, as can be seen from these figures, there is overall good agreement between simulation and experiment. In particular, in the collapsed part of the tube at $x=70$ mm and $x=90$ mm, the double peaks of the velocities are quite well reproduced by the simulations.

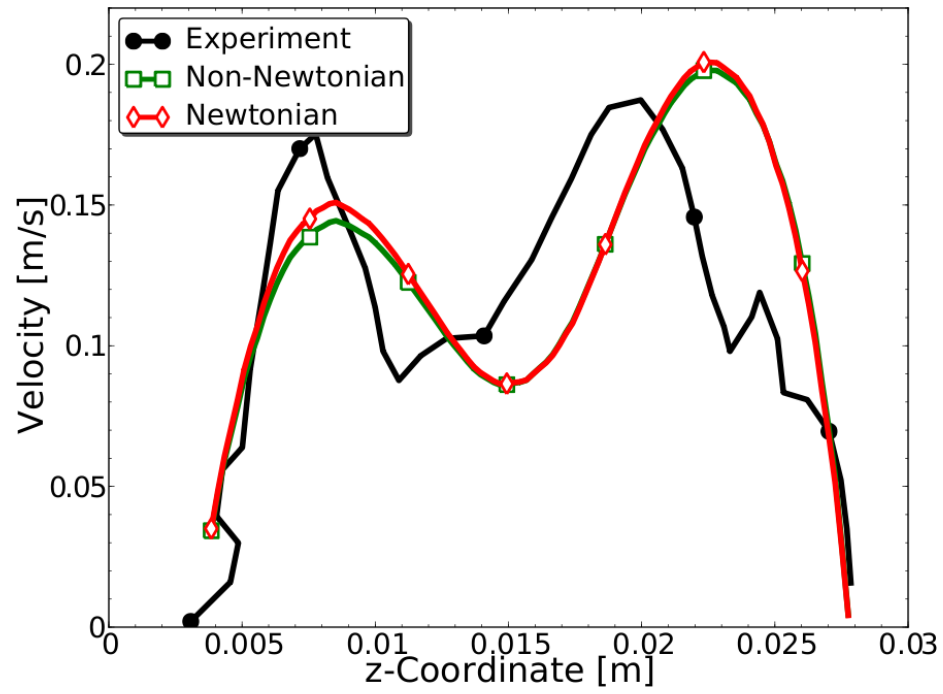


Figure 3.7: Velocity profiles at $x=70$ mm.

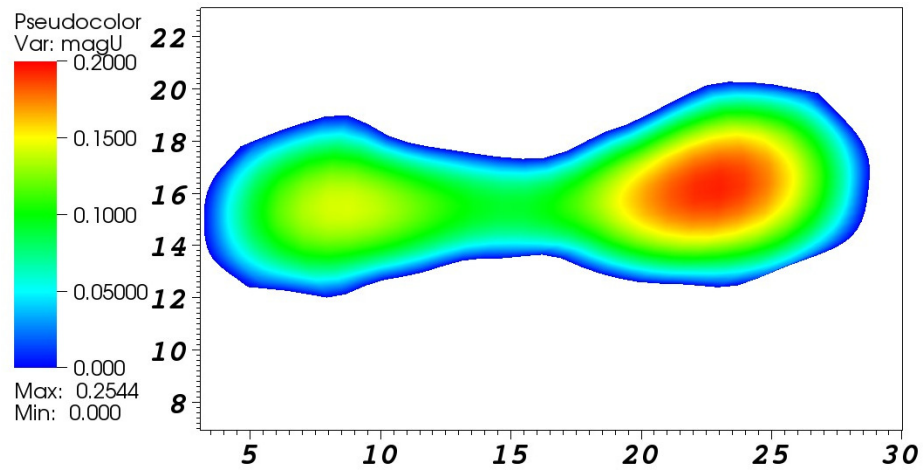


Figure 3.8: Cross-section of the non-Newtonian velocity magnitude at $x=70$ mm. The cross-sectional coordinates are in mm and the velocity is in m/s.

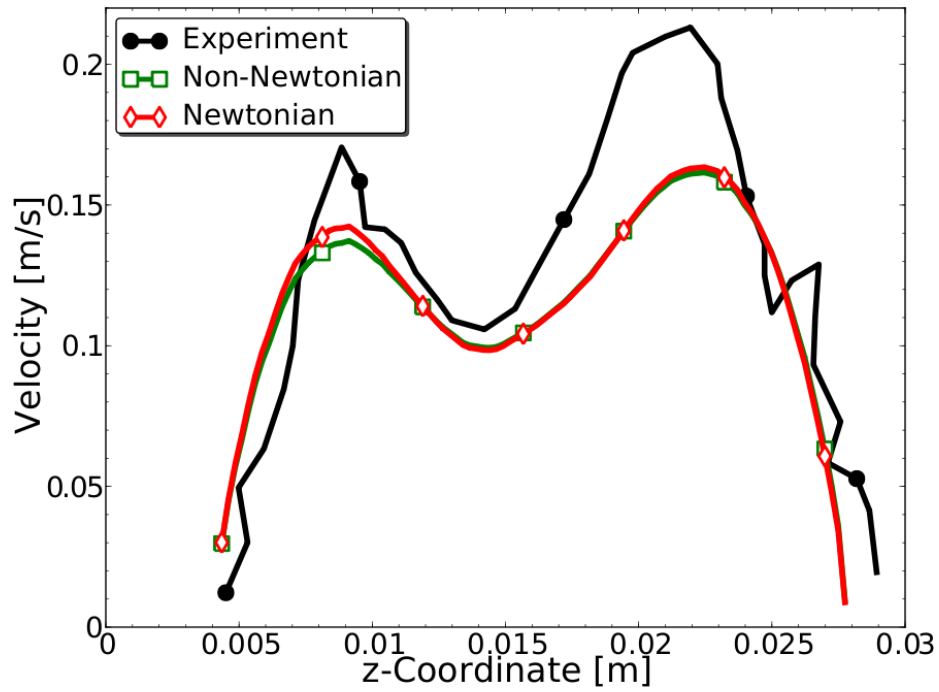


Figure 3.9: Velocity profiles at $x=90$ mm.

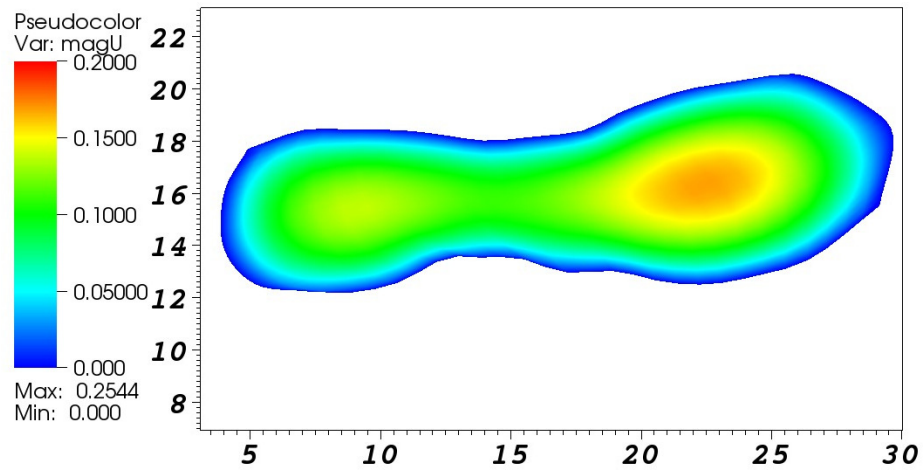


Figure 3.10: Cross-section of the non-Newtonian velocity magnitude at $x=90$ mm. The cross-sectional coordinates are in mm and the velocity is in m/s.

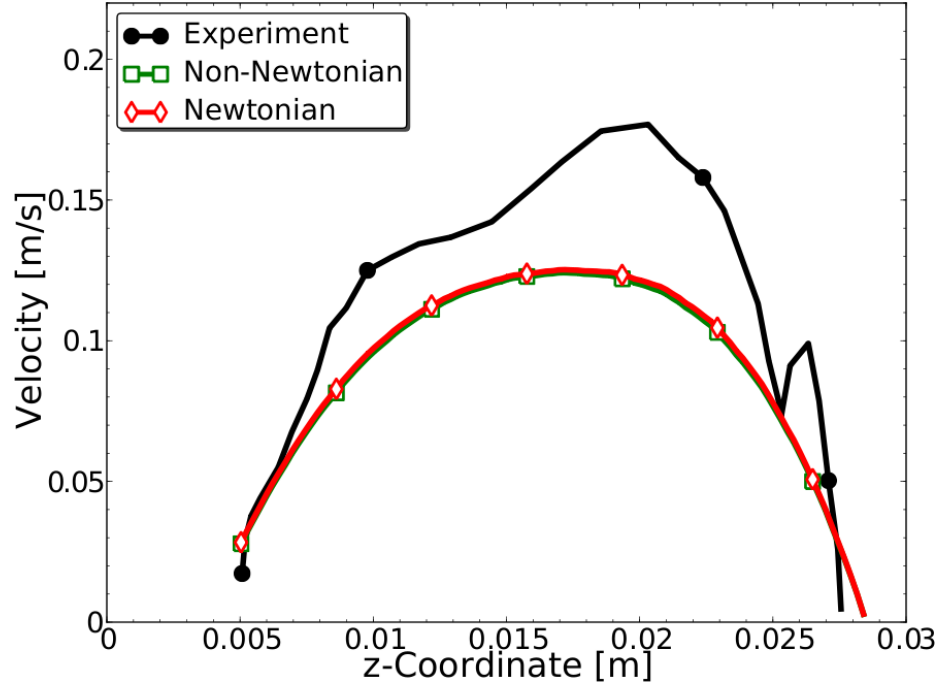


Figure 3.11: Velocity profiles at $x=150$ mm.

The double peaks in the velocity profiles occur because of the bow-tie-shaped tube cross-sections at $x=70$ mm and $x=90$ mm, as is shown in Figs. 3.8 and 3.10. The colors in these pictures represent the velocity magnitudes of the non-Newtonian fluid and is consistent with the double peak behavior shown in the respective velocity profile plots. As is well known, when the velocity at the tube wall is zero, the maximum velocities occur farthest away from the wall. This is consistent with the fact that the fluid takes the path of least resistance, which in this case translates into the double peak behavior in the bow-tie-shaped cross-sections. In contrast, as is illustrated in Fig. 3.11, the velocity profile at $x=150$ mm does not exhibit the double peak, which is consistent with the convex tube cross-section shown in Fig. 3.12.

To illustrate the effect of the non-Newtonian behavior of the fluid, an identical

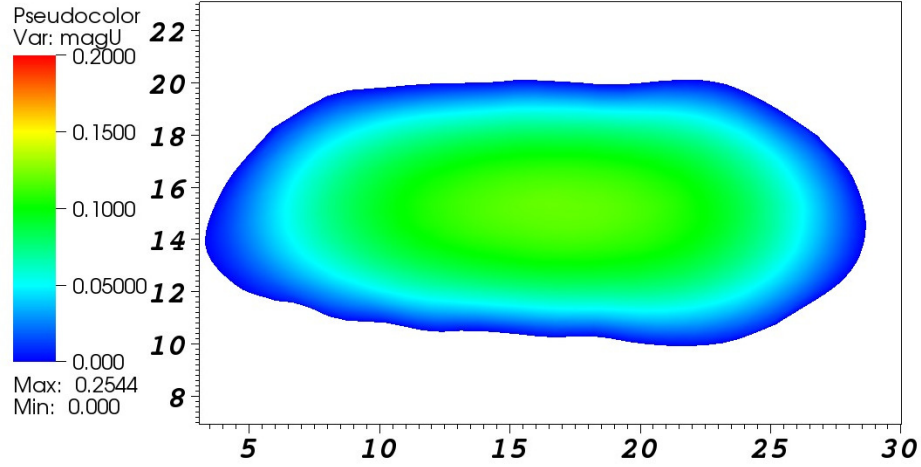


Figure 3.12: Cross-section of the non-Newtonian velocity magnitude at $x=150$ mm. The cross-sectional coordinates are in mm and the velocity is in m/s.

simulation has been carried out for a Newtonian fluid whose viscosity corresponds to the zero-shear-rate-dependent viscosity $\eta_0 = 0.1452$ Pa s of the non-Newtonian fluid. The velocity profiles of this simulation are also presented in Figs. 3.7, 3.9 and 3.11. These figures show that the Newtonian fluid exhibits a small increase in the maximum velocities in the collapsed cross-sections at $x=70$ mm and $x=90$ mm, whereas in the convex part of the tube at $x=150$ mm, the two velocity profiles are almost identical. The maximum velocities for the two computations are shown in Table 3.1, where it is seen that the Newtonian fluid flows faster by about 4%. As is discussed in the next subsection, this is consistent with the corresponding shear rate behavior.

Table 3.1

Comparison between the Newtonian and non-Newtonian simulations.

Case	$ \mathbf{U} _{\max}$ [m/s]	$\dot{\gamma}_{\max}$ [s^{-1}]	$\dot{\gamma}_{\text{avg}}$ [s^{-1}]	η_{\min} [Pa s]	Δp [Pa]
Non-Newtonian	0.2544	733	103.2	0.0715	587
Newtonian	0.2644	647	102.4	0.1452	644

3.3.2 Shear Rate Comparison

Unlike in the experiments, a detailed shear rate field can be determined from the simulation.

The shear rates are computed from the calculated velocity field by

$$\dot{\gamma} = ||\dot{\boldsymbol{\gamma}}|| = \sqrt{\frac{1}{2}(\dot{\boldsymbol{\gamma}} : \dot{\boldsymbol{\gamma}})} = \sqrt{\frac{1}{2} \sum_{i,j} \dot{\gamma}_{ij} \dot{\gamma}_{ji}}, \quad (3.3)$$

where $\dot{\boldsymbol{\gamma}} = \nabla \mathbf{u} + \nabla \mathbf{u}^T$ is the rate-of-strain tensor.

Figure 3.13 shows the shear rates at the tube wall, where the largest shear rates in any axial cross-section exist, and Fig. 3.14 shows the shear rates in the cross-section $x=50$ mm, where the tube is the most deformed. As can be seen from these figures, the largest shear rates for the non-Newtonian fluid have values above 200 s^{-1} with a maximum value of 733 s^{-1} , occurring on the periphery around $x=50$ mm. Moreover, Fig. 3.14 shows that shear rates in the shear-thinning regime, which begins at about $\dot{\gamma}=40 \text{ s}^{-1}$ (see Fig. 3.2), are reached significantly away from the tube wall in this cross-section. Averaging the shear rates in this cross-section gives the average shear rate of $\dot{\gamma}_{avg} = 103.2 \text{ s}^{-1}$ (see Table 3.1).

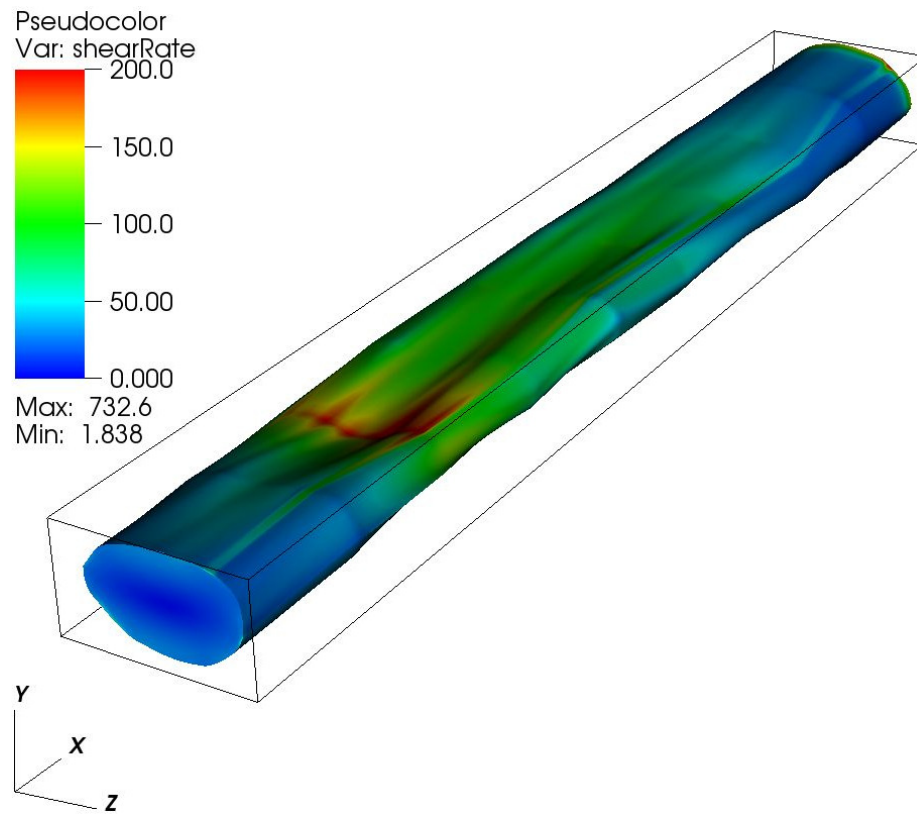


Figure 3.13: Three-dimensional view of the non-Newtonian shear rates on the periphery of the collapsed tube.

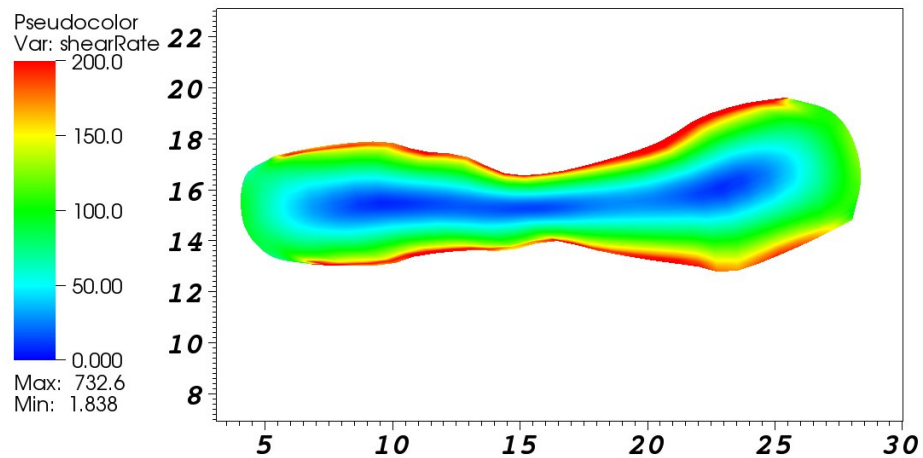


Figure 3.14: Cross-section of the non-Newtonian shear rates at x=50 mm.

The effect of the tube's deformed shape on the shear rates can be determined by comparing the shear rates in the deformed tube to those in the undeformed tube. According to Bird et al. [21, pp. 175–177], the maximum shear rate in an undeformed cylindrical tube, assuming a power-law fluid governed by Eq. (3.2) is given by

$$\dot{\gamma}_{max} = \frac{u_0}{R} \frac{3n+1}{n}. \quad (3.4)$$

where u_0 is the average velocity and R is the tube radius, and the average shear rate is

$$\dot{\gamma}_{avg} = \frac{2u_0}{R} \frac{3n+1}{2n+1}. \quad (3.5)$$

For the flow problem under consideration, using the undeformed tube radius $R=10$ mm and the average velocity $u_0=0.0541$ m/s, together with the fluid properties in Eq. (3.1), it follows that $\dot{\gamma}_{max} = 23.4 \text{ s}^{-1}$ and $\dot{\gamma}_{avg} = 14.1 \text{ s}^{-1}$. Comparison of the maximum shear rate $\dot{\gamma}_{max} = 23.4 \text{ s}^{-1}$ for the undeformed tube with the maximum shear rate $\dot{\gamma}_{max} = 733 \text{ s}^{-1}$ (see Table 3.1) for the deformed tube, shows that the tube deformation results in a significant increase of the maximum shear rate, namely by a factor of 31. As a consequence, the viscosity decreases from its zero shear rate value of $\eta_0=0.1452$ Pa s down to 0.0715 Pa s (cf. Table 3.1), which is a factor of two.

Likewise, the average shear rate increases by a factor of 7.3, from 14.1 s^{-1} in the undeformed tube to 103.2 s^{-1} in the deformed tube. This factor agrees very well with

the factor of 7 found in Nahar et al. [43] for the experiments, which calculated the average shear rate in the deformed tube using Eq. (3.5), in which u_0 and R were determined as follows. First, the cross-sections obtained from the tomography measurements have been approximated numerically, which allowed the determination of the average velocity u_0 from the flow rate. Then, an equivalent tube radius, R , has been determined such that the equivalent circular area is equal to the one of the corresponding deformed cross-section. Using this approach, the area of the cross-section at $x=50$ mm has been estimated to be $A = 105.7 \text{ mm}^2$, which is a factor of 0.34 smaller than the cross-section of the undeformed tube. The excellent agreement in average shear rates between simulation and estimation confirm that the rudimentary method of determining the estimated average shear rates from the experimental data is accurate for this level of tube deformation.

As listed in Table 3.1, the maximum shear rate for the Newtonian case is $\dot{\gamma}_{max}=647 \text{ s}^{-1}$, which is 4% less than the maximum value for the non-Newtonian case. A more detailed comparison between Newtonian and non-Newtonian shear rate behavior along the cross-section $x=50$ mm is shown in Fig. 3.15. As can be seen, the shear rates of the Newtonian fluid are larger throughout most of the cross-section, except possibly at the walls. The largest difference occurs at the z -coordinate $z=9$ mm, where the Newtonian shear rate is approximately 26% higher than the non-Newtonian one. However, as is seen from Fig. 3.2, this relatively large difference in the shear rate leads to an insignificant change in the viscosity, which explains the small differences in the velocity commented on in the previous subsection.

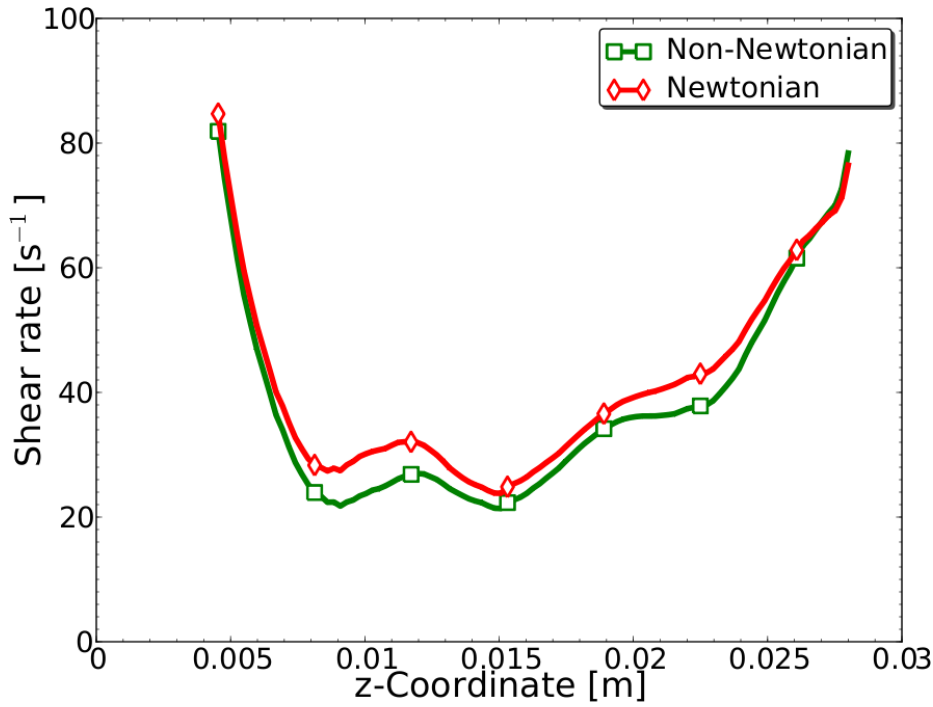


Figure 3.15: Shear rate profile for the non-Newtonian and Newtonian simulations at $x=50$ mm.

3.3.3 Pressure Drop Comparison

In the experiments, the pressure drop in the tube was measured over a distance of 910 mm, of which the middle 320 mm correspond to the deformable elastic tube, and the remaining part had a cylindrical cross-section with a radius of 10 mm. (Recall that only 180 mm of the most deformed part of the elastic tube was simulated.) The total pressure drop was measured to be 1256 Pa. The simulation pressure drop over the entire computational domain of 180 mm was 587 Pa, as can be seen from Fig. 3.16 or Table 3.1. The dashed lines in this figure correspond to the theoretical pressure drop for a power-law fluid in an undeformed tube given in Eq. (3.2). According to Bird et al. [21, p. 176] this pressure drop

is given by

$$\Delta p = \left(\frac{3n+1}{n} \right)^n \frac{2\kappa}{\pi^n R^{3n+1}} Q^n L, \quad (3.6)$$

where L is the length of the tube, Q is the volumetric flow rate. Note that for a Newtonian fluid with $n = 1$ and $\kappa = \eta_o$ Eq. (3.6) reduces to the Hagen-Poiseuille equation $\Delta p = 8\eta_o QL/(\pi R^4)$. Figure 3.16 shows that the slope of the non-Newtonian simulation pressure curve is the steepest at $x=50$ mm where the tube exhibits the largest deformation. The theoretical pressure of the non-Newtonian fluid in an undeformed tube is shown by the dashed lines at the tube inlet and outlet. These lines are tangential to the simulation curve, which is an indication that Eq. (3.6) accurately describes the pressure gradient for the non-deformed portion of the tube. To account for the pressure discrepancy between the experiment and the simulation, the pressure drop in the remaining undeformed part of the pipe of the experiment is taken into consideration via Eq. (3.6). The total length of this undeformed pipe is 730 mm, which leads to a pressure drop of 560 Pa, leading to a total simulation pressure drop of 1147 Pa. Given the fact that in the experiment there are several pipe connections over which the pressure drop is larger than predicted by Eq. (3.6), the agreement between simulation and experiment is excellent.

Also shown in Fig. 3.16 is the pressure curve of the Newtonian calculation. As is seen, the pressure drop is 644 Pa (cf. Table 3.1), which is almost 10% larger than for the non-Newtonian case. Since the non-Newtonian fluid under consideration is shear-thinning,

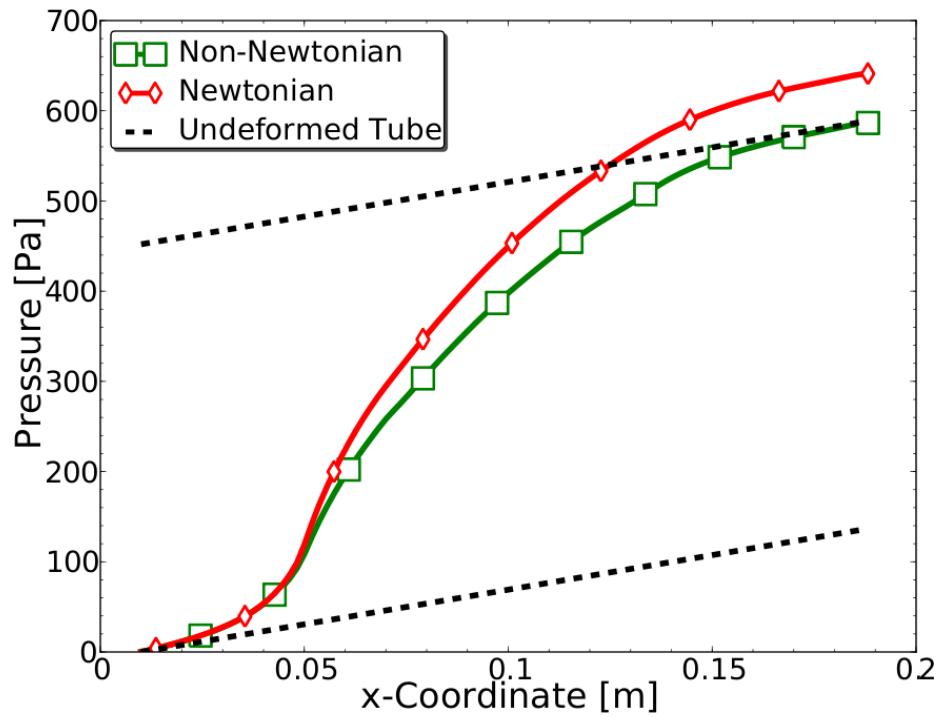


Figure 3.16: Pressure curves for the non-Newtonian and Newtonian simulations. (The flow is from right to left.)

its viscosity is smaller or equal to that of the Newtonian fluid. Thus, the viscous dissipation of the Newtonian fluid is larger, which accounts for the larger pressure drop.

3.4 Summary and Conclusions

The flow of a non-Newtonian fluid through a collapsed tube has been simulated using an open source CFD solver and a deformed tube geometry. The simulation results have been compared with experimental data, and additional insights have been obtained by considering local quantities for shear rates and viscosities. The geometry of the deformed tube has been reconstructed from computer tomography image analysis. The velocity profiles obtained from the simulations have been compared with corresponding ultrasound Doppler velocity profile measurements at various cross-sections. There is generally good agreement between simulation and experiment, especially given the rudimentary approximation to the geometry used in the simulation. The double peaks of the velocity profiles in the collapsed part of the tube were well reproduced. These double peaks are a consequence of the bow-tie-shaped cross-sections where the fluid follows the path of least resistance and flows fastest furthest away from the wall.

The shear-thinning effect of the fluid becomes relevant in the cross-sections with the largest deformation. The maximum shear rate is about a factor of thirty larger than its corresponding maximum value in the undeformed tube, which reduces the viscosity by a factor of two. Similarly, the average shear rate in the most deformed cross-section is a factor of 7.3 larger, which is in good agreement with the estimate derived from the experimental data. Also, the pressure drop across the tube was well predicted by the

simulation after appropriate pressure corrections have been added for the non-deformed portion in the experimental setup.

In order to better assess the non-Newtonian behavior of this fluid the same flow has been simulated for a Newtonian fluid. It was found that there are significant differences in the shear rates at locations where the tube was strongly deformed, These differences are significant enough to cause sufficient shear-thinning which is reflected in the velocity profiles, and leads to a 10% increase in the pressure drop for the Newtonian fluid.

Chapter 4

Simulations of Peristaltic Motion

The simulations presented in this chapter are motivated by the experiments of Nahar [47]. In these experiments a peristaltic motion is induced by means of rollers which squeeze a fluid along a flexible closed tube. Two different frames of reference are considered: the moving frame of reference (wave frame) where the computational domain is fixed and the coordinate system is moving with the roller speed, and the fixed frame of reference (laboratory frame) where the roller motion is represented by a deforming mesh. The transformation between the fixed frame, with coordinates (x, y) and velocity (u_x, u_y) , and the moving frame, with coordinates (\hat{x}, \hat{y}) and velocity (\hat{u}_x, \hat{u}_y) , is given by

$$\hat{x} = x - ct, \quad \hat{y} = y$$

$$\hat{u}_x = u_x - c, \quad \hat{u}_y = u_y,$$

where c is the wave speed and t is time.

The simulations are performed for Newtonian and non-Newtonian fluids for different roller speeds and different gap widths formed by the rollers.

4.1 Moving Frame Simulations

In the moving frame of reference the computational domain is fixed and is moving with the uniform roller speed in the positive x -direction. This results in a Galilean invariant inertial frame in which the governing equations are identical to the ones in a frame at a rest. The two-dimensional, symmetric computational domain reflects the upper half of a deformed tube or channel, as is shown in Fig. 4.1. The tube diameter is 20 mm, the overall length of

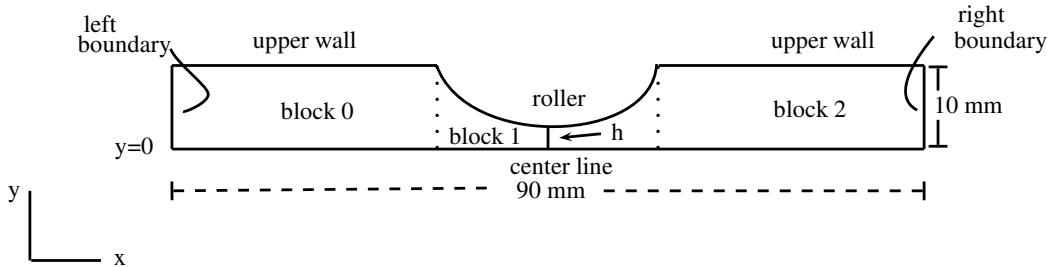


Figure 4.1: Computation domain for moving frame simulations.

the domain is 90 mm and the roller diameter is 30 mm. The tube deformation is represented by an appropriate circular arc which forms a gap of half-width $h=4$ mm for the standard simulation case.

As indicated in Fig. 4.1, the computational mesh consists of three hexahedral blocks, each having a uniform cell distribution. The length of the outer two blocks is 33 mm. In the x-direction the standard mesh has 120 cells for the left and the right blocks, and 180 cells for the middle block. All blocks have 24 cells in the y-direction, and one cell in the z-direction. (For two-dimensional simulations OpenFOAM requires one cell in the third direction.) This gives a total of 10'080 cells for the standard mesh. The smallest cells are located in the gap and are 0.167 mm in height and 0.133 mm in length. The initial and boundary conditions reflect the experimental setup and are described in the next subsection.

4.1.1 Initial and Boundary Conditions

In order to solve the mass and momentum conservation equations introduced in Chapter 2, initial and boundary conditions need to be specified. The initial condition for the internal pressure and velocity fields are set to zero. The pressure boundary conditions on the left and right boundary are set to zero total pressure. This reflects the fact that the experimental system is closed, i.e., the right and left boundaries are connected via a large fluid reservoir which is at constant pressure. The total pressure p_0 is computed by

$$p_0 = p + \frac{1}{2}\rho |\mathbf{u}|^2.$$

The first term on the right hand side stands for the static pressure while the second term expresses the dynamic pressure. The zero total pressure adjusts the pressure p according to the changes in the velocity \mathbf{u} . On the upper wall and on the roller boundary the normal gradient of the pressure is set to zero.

The velocity boundary conditions on the left and right boundary are set to zero gradient. In the moving frame of reference, the velocity on the roller is zero since the coordinate system is moving with the same velocity as the roller. Likewise, the velocity of the undeformed parts of the upper wall is given by the roller speed in the negative x-direction, i.e., velocity in the x- and y-directions are $-c$ and 0, respectively. Finally, a symmetry plane boundary condition has been used for the centerline, which essentially enforces that $\mathbf{u} \cdot \mathbf{n} = 0$ and the tangential component of $\boldsymbol{\sigma} \cdot \mathbf{n}$ is zero. This condition guarantees that there is no flux across the centerline.

4.1.2 Computational Details

For the simulations a steady-state solver for incompressible, turbulent flows using the iterative SIMPLE algorithm, called `simpleFoam`, is used. This solver has been used for the laminar flows in this thesis by deactivating the turbulence models. For the Newtonian simulations the dynamic viscosity is 0.1452 Pa.s, which corresponds to the zero-shear-rate-dependent viscosity of the non-Newtonian cases.

Since the fluid under consideration is incompressible, the mass transport is expressed in

terms of the average speed at the right boundary. Therefore, the transport efficiency can be computed by

$$\text{Transport efficiency} = \frac{\text{average speed at the right boundary}}{\text{roller speed}} \quad (4.1)$$

The discretization schemes for the different operators are chosen in the file `<case>/system/fvSchemes`. In steady state simulations the time derivative scheme is set to `steadyState`. The gradient, Laplacian and divergence terms have been discretized by the standard second order finite volume discretization of Gaussian integration which is based on summing values on cell faces, which must be interpolated from cell centers. The Gaussian integration has been used with a linear (central differencing) interpolation scheme for the gradient and the Laplacian, and an upwind scheme for the divergence terms. For a more detailed discussion see Chapter 2.

The pressure equations were solved by means of the Preconditioned Conjugate Gradient (PCG) method with the Diagonal-based Incomplete Cholesky (DIC) preconditioning for symmetric matrices. The asymmetric velocity equations were solved with the Preconditioned Bi-Conjugate Gradient (PBICG) method using a Diagonal-based Incomplete Lower Upper (DILU) conditioning for the velocity. The solvers and preconditioners are discussed in Chapter 2.

The convergence of the velocity-pressure iterations is controlled by means of the residuals for the pressure and velocity equations in the SIMPLE algorithm. Residuals of 10^{-6} and 10^{-7} were used for the pressure and for the velocities, respectively. In addition,

the respective tolerances for the iterative solutions of the discrete pressure and velocity equations in each SIMPLE step were 10^{-6} and 10^{-5} for the absolute values, and 0.01 and 0.1 for the relative values. As discussed in Chapter 2 for each system of equations, the iteration process is stopped when either the absolute or relative tolerance is met.

4.1.3 Mesh Dependence Study

The mesh dependence study has been performed for a Newtonian fluid for the worst case scenario, that is, for the case with the largest velocity and pressure gradients. Presumably this is the case with the fastest roller speed of 10 mm/s and the smallest half gap width of 2 mm.

A coarser and a finer mesh have been obtained from the standard mesh by reducing, respectively, increasing, the number of cells in the x and y-directions by a factor of 1.5. These meshes are summarized in Table 4.1. The results of this mesh refinement study are

Table 4.1
Number of cells for the different meshes. (Moving frame simulations.)

Mesh	Blocks 0 & 2	Block 1	Total number of cells
Coarse	$80 \times 16 \times 1$	$120 \times 16 \times 1$	4480
Standard	$120 \times 24 \times 1$	$180 \times 24 \times 1$	10080
Fine	$180 \times 36 \times 1$	$270 \times 36 \times 1$	22680

shown in Figs. 4.2-4.4 close to the centerline at $y=0.05$ mm. In these figures, the central

vertical dashed lines indicate the x-coordinate of the center of the roller. The left and right vertical dashed lines represent the end of the roller, that is, the start and end of contact between the roller and the tube.

Figures 4.2 and 4.3 show that the different meshes give identical values of the x-component of the velocity and of the kinematic pressure along the centerline. (The kinematic pressure is the pressure scaled by the density and has units m^2/s^2 .) In order to change the moving frame back to the laboratory frame, the velocity has been corrected by adding 10 mm/s in the x-direction to get Fig. 4.2. This figure shows that the minimum velocity (around 6.5 mm/s) occurs in the narrow gap region. The fact that the peristaltic wave tends to produce a rising pressure in the direction of the wave is shown in Fig. 4.3. This figure also shows that the pressure is symmetric with respect to the center of the roller and takes the extreme values at the ends of the roller. The same is true for the shear rates which have a maximum value of 1.25 s^{-1} at the ends of the roller, as is shown in Fig. 4.4.

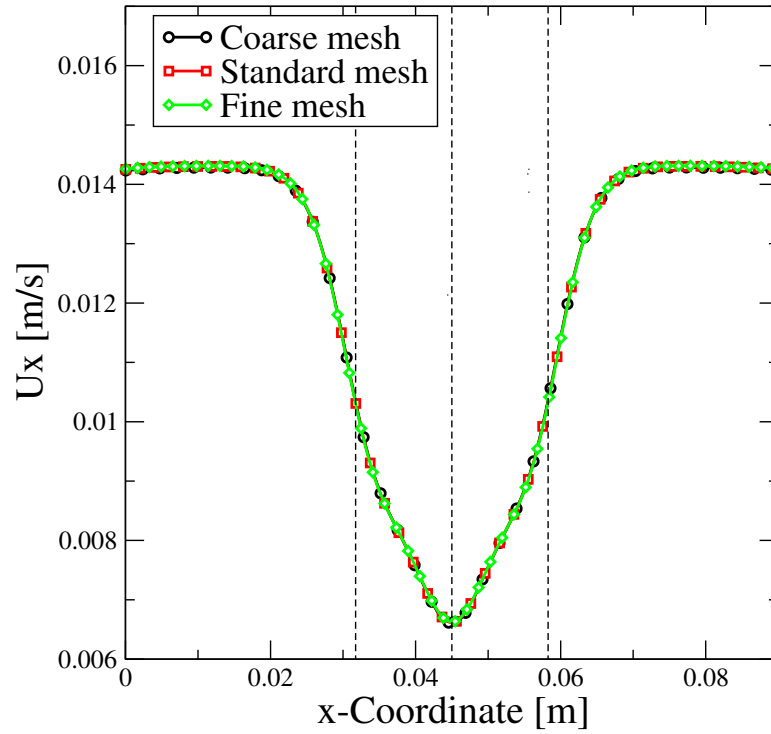


Figure 4.2: Moving frame mesh dependence study for the x-component of the velocity of the Newtonian fluid along the centerline at $y=0.05$ mm. (The left and right vertical dashed lines represent the ends of the roller, the middle vertical dashed line is the center of the roller.)

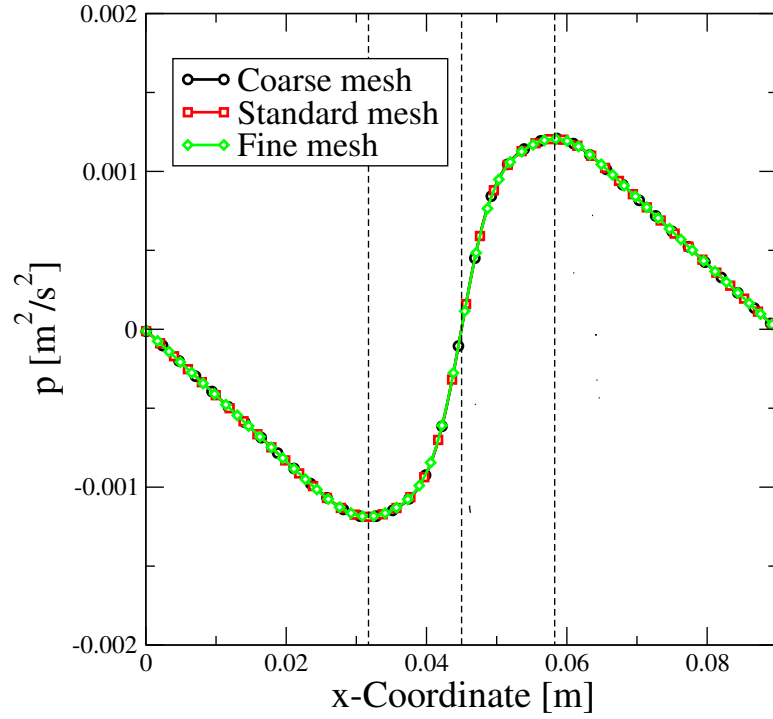


Figure 4.3: Moving frame mesh dependence study for the kinematic pressure of the Newtonian fluid along the centerline at $y=0.05$ mm. (The left and right vertical dashed lines represent the ends of the roller, the middle vertical dashed line is the center of the roller.)

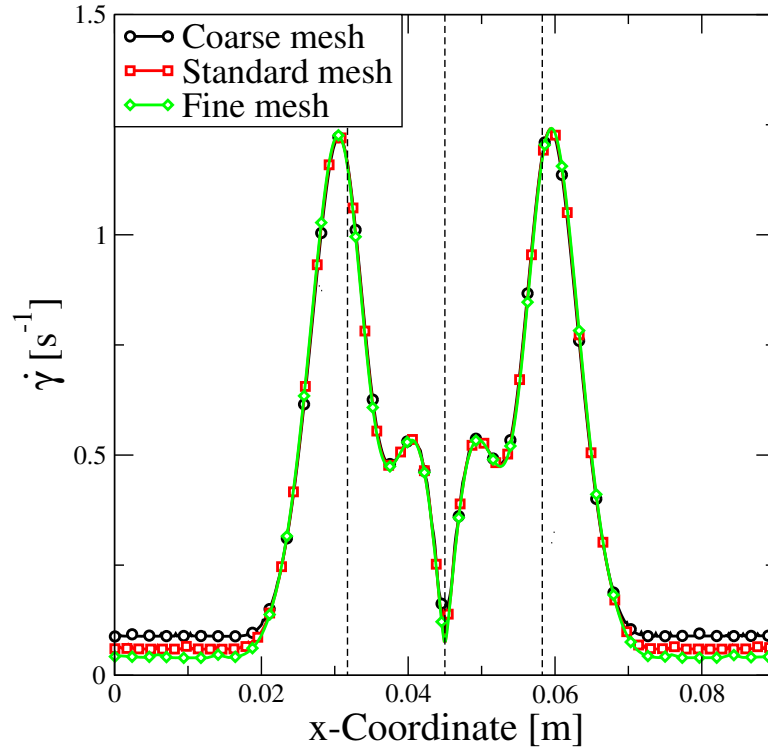


Figure 4.4: Moving frame mesh dependence study for the shear rate of the Newtonian fluid along the centerline at $y=0.05$ mm. (The left and right vertical dashed lines represent the ends of the roller, the middle vertical dashed line is the center of the roller.)

4.1.4 Parameter Study

The main aim of this section is to identify the effect of the shear-thinning, non-Newtonian behavior of the fluid, and to study the material transport efficiency in terms of the roller speed and the gap width.

Newtonian versus Non-Newtonian Fluids

Simulations have been carried out for two non-Newtonian fluids with a shear rate dependent viscosity expressed by the Bird-Carreau equation discussed in Chapter 2

$$\frac{\eta - \eta_{\infty}}{\eta_0 - \eta_{\infty}} = (1 + (k\dot{\gamma})^2)^{(n-1)/2}. \quad (4.2)$$

The fluid parameters for the non-Newtonian fluids used in this comparison are given in Table 4.2, and the viscosity curves are depicted in Fig. 4.5. Observe that the shear-thinning for both fluids starts at a shear rate of $\dot{\gamma} = 1/k = 0.05 \text{ s}^{-1}$, and that two different power-law indices of $n = 0.75$ and $n = 0.5$ were considered, with the latter exhibiting considerably more shear-thinning behavior. Note that the zero-shear-rate viscosity η_0 for the non-Newtonian fluids is the constant viscosity used for the Newtonian case.

Table 4.2
Non-Newtonian fluid parameters

Parameters	Fluid A	Fluid B
$\eta_0[Pa\ s]$	0.1452	0.1452
$\eta_\infty[Pa\ s]$	0	0
$k[s]$	20	20
n	0.75	0.50

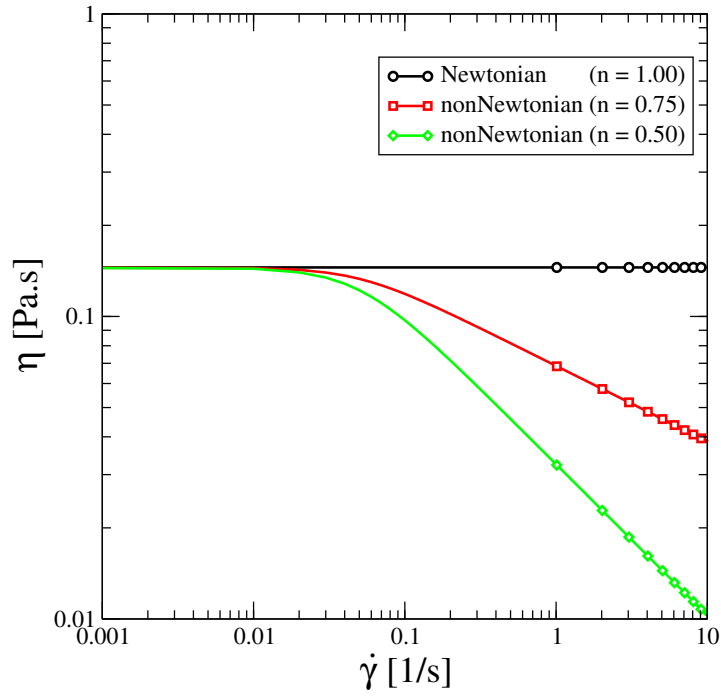


Figure 4.5: Shear rate dependent viscosity curves.

In these simulations the gap half-width is fixed at 4 mm and the roller speed is 5 mm/s. The simulation results are shown in Figs. 4.6 to 4.14 and have been transferred to the fixed/laboratory frame.

Figure 4.6 is a color plot of the x-component of the velocity (transferred to the laboratory frame) of the Newtonian fluid. The figure shows that the maximum velocities are attained in the region near the centerline and away from the roller. These maximum values are close to the roller speed of 5 mm/s. The smallest values of the velocity are in the region under the roller. They are positive which indicates that there is no back-flow. Figure 4.7 shows the kinematic pressure distribution. The negative and positive pressure regions under and near the roller are consistent with the results shown in Fig. 4.3.

The color plot in Fig. 4.8 shows that the maximum values of the shear rates are achieved at the roller boundary and just under the roller. Also, the figure shows small shear rates along the centerline.

Figure 4.9 is a vector plot of the velocity. The direction of these vectors indicates the absence of a back-flow, which is consistent with the positive x-components of the velocity shown in Fig. 4.6.

In Fig. 4.10, the x-components of the velocity (transferred to the laboratory frame) are shown along the centerline at $y=0.05$ mm for the Newtonian and the non-Newtonian fluid simulations. It is seen that the velocities are the lowest in the most narrow part of the channel, that is, just below the roller. It is interesting to note that this velocity component is close to zero but still positive, which indicates that there is no back flow under the roller.

The largest velocities are achieved away from the roller where the velocities are constant. The Newtonian fluid exhibits the largest velocity and the more shear-thinning fluid, the fluid with $n = 0.5$, has the smallest velocity. This is consistent with the vertical velocity profiles at the right boundary shown in Fig. 4.11. The profiles for the Newtonian fluid resembles a parabola, whereas the profiles for the non-Newtonian fluids are more plug-like, with the more shear-thinning fluid being flatter.

The (relative) kinematic pressures along the centerline at $y=0.05$ mm are shown in Fig. 4.12 for the three different fluids. Recall that the fluids are incompressible and, therefore, the pressures are given with respect to the reference total pressure of zero prescribed at the right and left boundaries. As is seen, the pressure curves are symmetric with respect to the center of the roller. As expected, the roller movement induces a larger pressure in front, and an under-pressure region is found behind the roller. The largest pressure variation is observed for the Newtonian fluid and the smallest pressure variation occurs for the more shear-thinning fluid. This is consistent with the theoretical result that the smaller viscosities of the shear-thinning fluid result in a smaller pressure drop in a channel or tube.

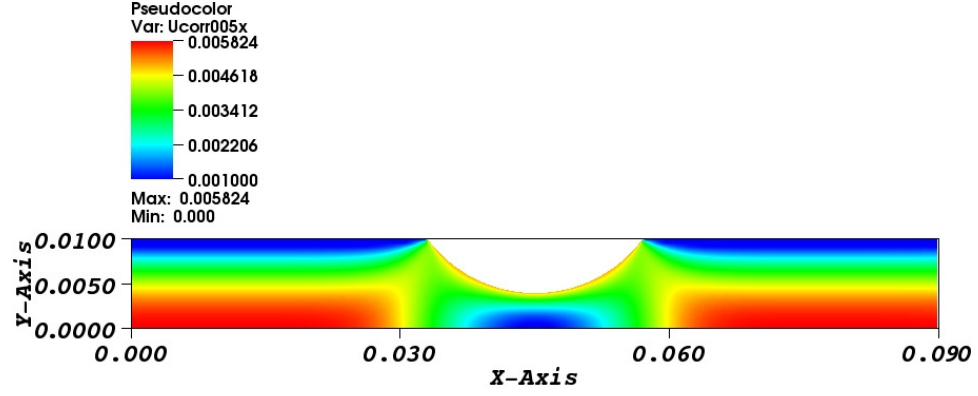


Figure 4.6: Moving frame x-component of the velocity [m/s] of the Newtonian fluid. (The gap half-width is 4 mm, and the roller speed is 5 mm/s.)

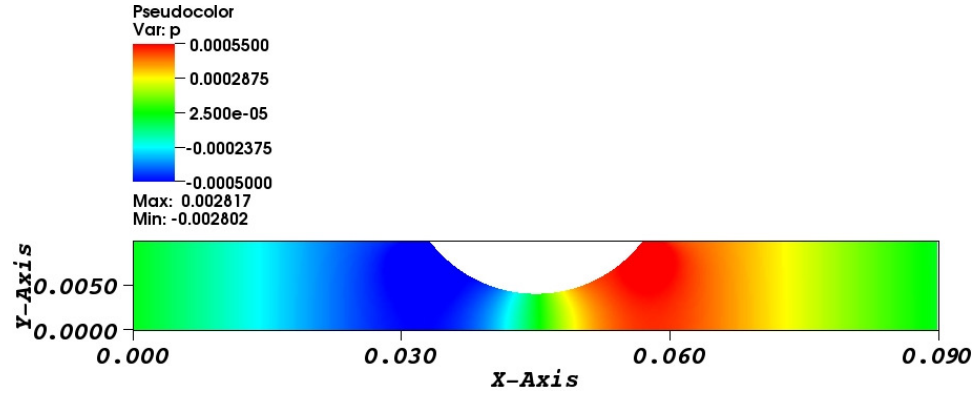


Figure 4.7: Moving frame kinematic pressure [m²/s²] of the Newtonian fluid. (The gap half-width is 4 mm, and the roller speed is 5 mm/s.)

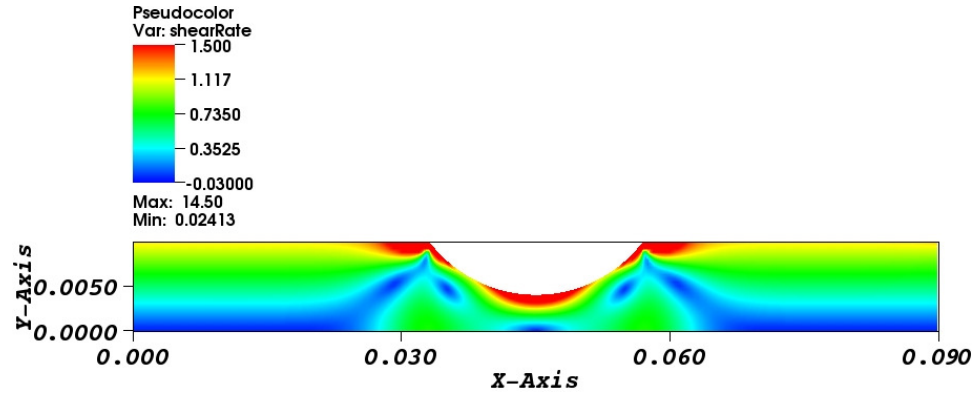


Figure 4.8: Moving frame shear rate [s⁻¹] of the Newtonian fluid. (The gap half-width is 4 mm, and the roller speed is 5 mm/s.)

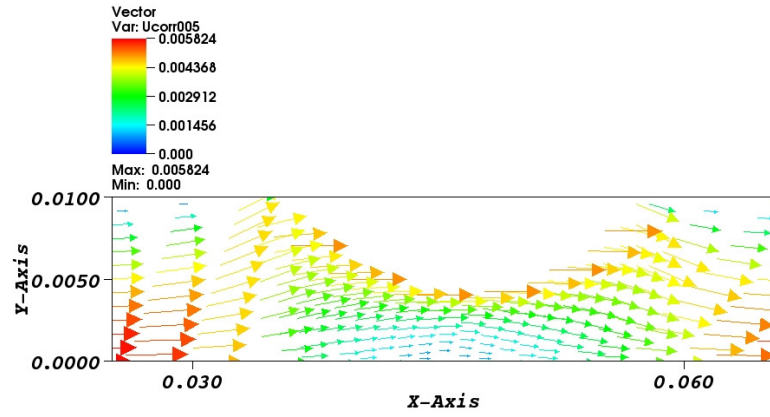


Figure 4.9: Moving frame velocity vectors [m/s] under the roller of the Newtonian fluid. (The gap half-width is 4 mm, the roller speed is 5 mm/s.)

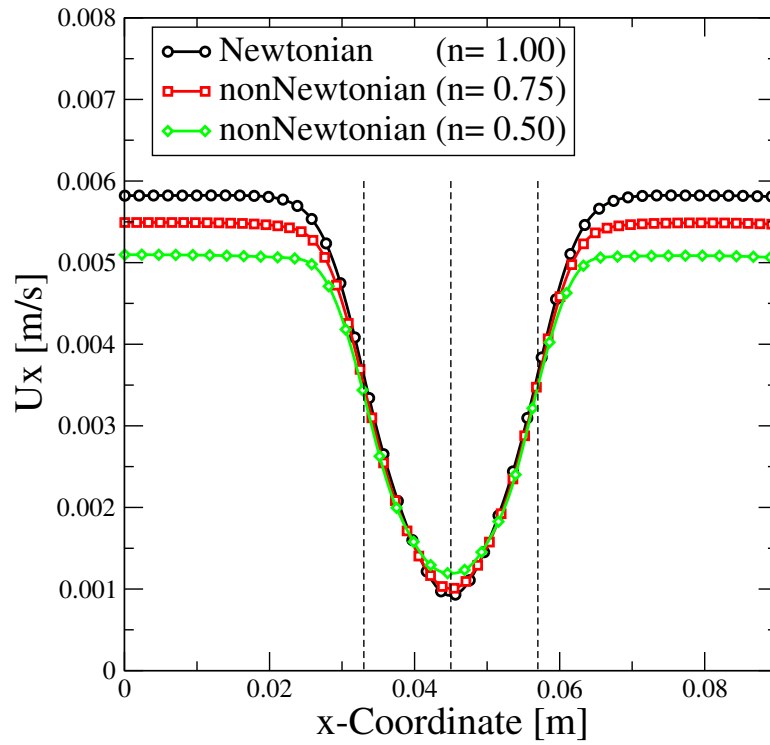


Figure 4.10: Moving frame x-component of the velocity for the Newtonian and non-Newtonian fluids along the centerline at $y=0.05$ mm. (The left and right vertical dashed lines represent the ends of the roller, the middle vertical dashed line is the center of the roller.)

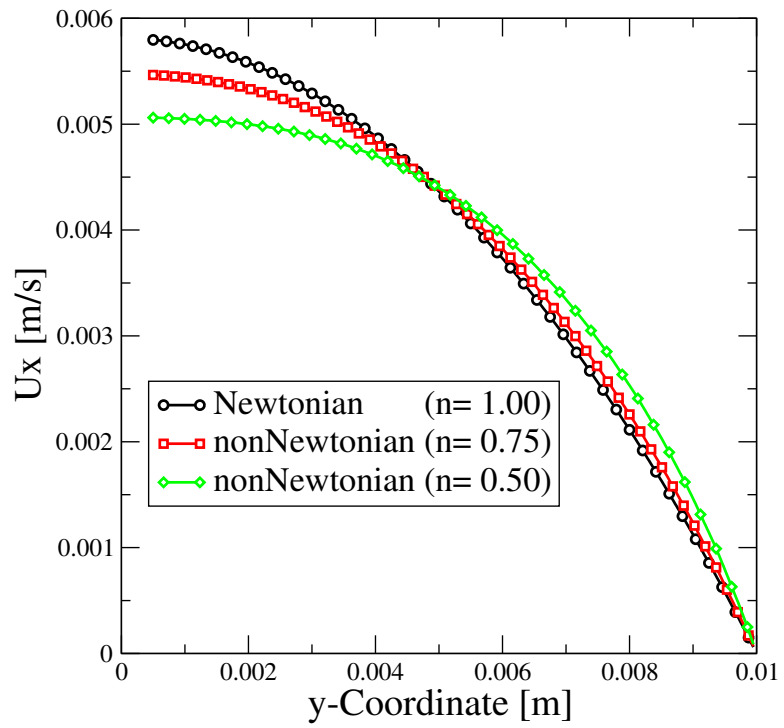


Figure 4.11: Moving frame x-component velocity profile for the Newtonian and non-Newtonian fluids near the right boundary at $x=89$ mm.

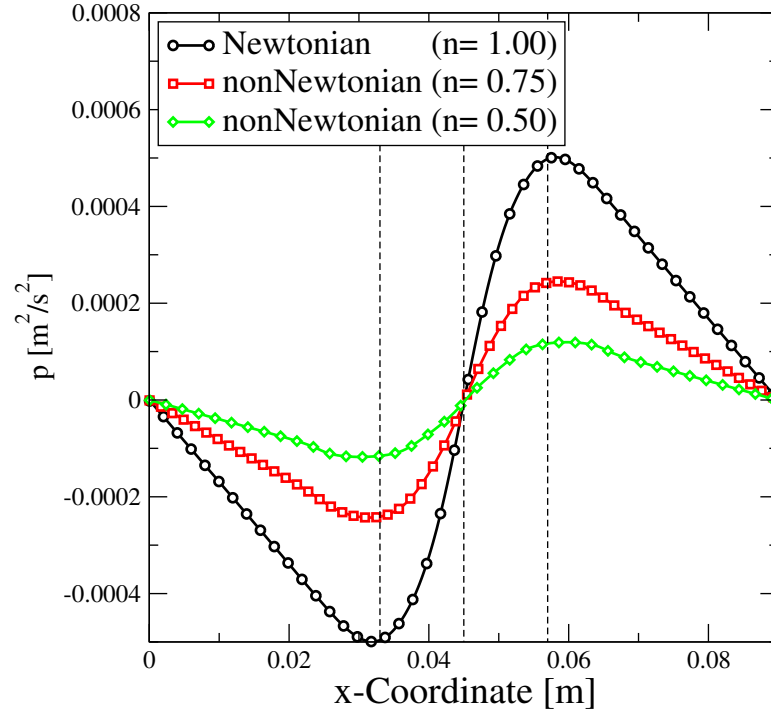


Figure 4.12: Moving frame kinematic pressure for the Newtonian and non-Newtonian simulations along the centerline at $y=0.05$ mm. (The left and right vertical dashed lines represent the ends of the roller, the middle vertical dashed line is the center of the roller.)

The shear rates close to the centerline at $y=0.05$ mm are shown in Fig. 4.13. It can be seen that for all three fluids the shear rates are almost the same under and near the roller, whereas away from the roller noticeable differences occur. This latter observation is also reflected by the slopes of the velocity gradients near the right boundary at $x=89$ mm in Fig. 4.11, which is consistent with the theory. For all three fluids, the shear rates attain their maxima at the beginning and at the end of the roller. This is a typical behavior of shear rates in flows with a cross-sectional change: the first peak is due to the cross-sectional contraction and the second is due to the expansion.

Larger differences in the shear rates between the three fluids are observed at the right boundary, as is shown in Fig. 4.14. As expected, the shear rates of the Newtonian fluid vary linearly, while the nonlinearity of the shear-rate-curves increases with increasing shear-thinning behavior. Also, the variation in the shear rates increases with the amount of shear-thinning.

The transport efficiency for the three fluids, as defined in Eq. (4.1), is shown in Table 4.3. As can be seen, the efficiency is almost the same in all three cases, with only a small decrease with increasing shear-thinning. This is consistent with the averaged velocities, hence volumetric flow rates, computed from the velocity profiles at the right boundary shown in Fig. 4.11. These almost constant flow rates are mainly due to the absence of a back-flow under the roller. As will be discussed in the fixed frame simulations, the effect of the viscosity can be considerable if a back flow is present. The importance of the back-flow is also relevant in the moving frame as can be seen from Table 4.4, where the decrease in

the transport efficiency is reduced significantly more for the gap half-width of 8 mm, which exhibits a back flow. More details are discussed in the next subsection.

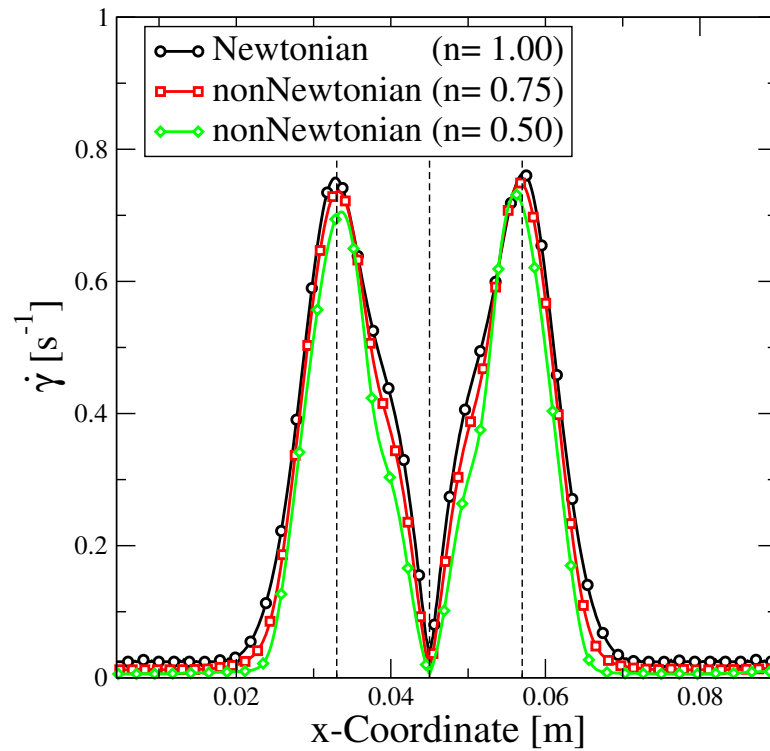


Figure 4.13: Moving frame shear rate for the Newtonian and non-Newtonian simulations along the centerline at $y=0.05$ mm. (The left and right vertical dashed lines represent the ends of the roller, the middle vertical dashed line is the center of the roller.)

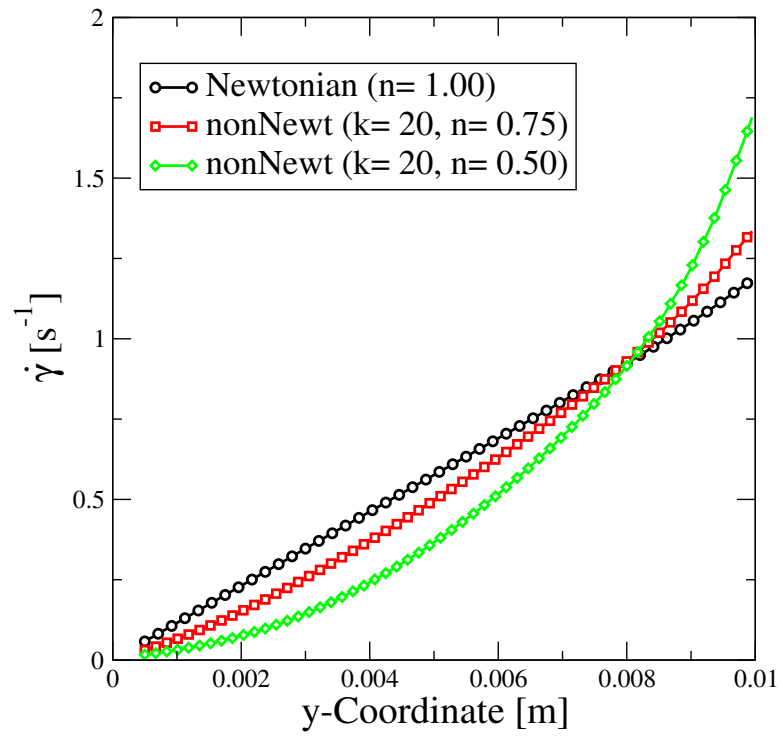


Figure 4.14: Moving frame shear rate profile for the Newtonian and non-Newtonian fluids near the right boundary at $x=89$ mm.

Table 4.3
Moving frame transport efficiency for different fluids

Flow type	Transport efficiency
Newtonian (n= 1.00)	78%
non-Newtonian (n= 0.75)	77%
non-Newtonian (n= 0.50)	76%

Variation of the Gap Width

The normalized gap half-width parameter, h/R_0 , where R_0 is the radius of the tube, is used. Simulations for the Newtonian fluid with the roller speed of 10 mm/s and three different gaps with normalized half-widths of 0.2, 0.4, and 0.8 have been used to study the effect on the transport efficiency. The results of this study are listed in Table 4.4 which shows that the transport efficiency decreases with increasing gap half-width. This is consistent with the velocity profiles at the right boundary shown in Fig. 4.15.

Table 4.4
Moving frame transport efficiency for different normalized gap half-widths

Normalized gap half-width	Transport efficiency
0.2	95%
0.4	78%
0.8	28%

Figure 4.16 shows the x-component of the velocity along the centerline at $y=0.05$ mm for different gap half-widths. The presence of negative velocities when the gap half-width is 8 mm indicates the presence of a back-flow. The other two curves do not show negative values, which means that there is no back-flow in these cases. This back-flow is illustrated in Fig. 4.17 which shows the velocity vectors in the region under the roller for the case with the gap half-width of 8 mm. The recirculation zone shown in the figure explains the significant drop in the transport efficiency by a factor of almost three, as is shown in Table 4.4.

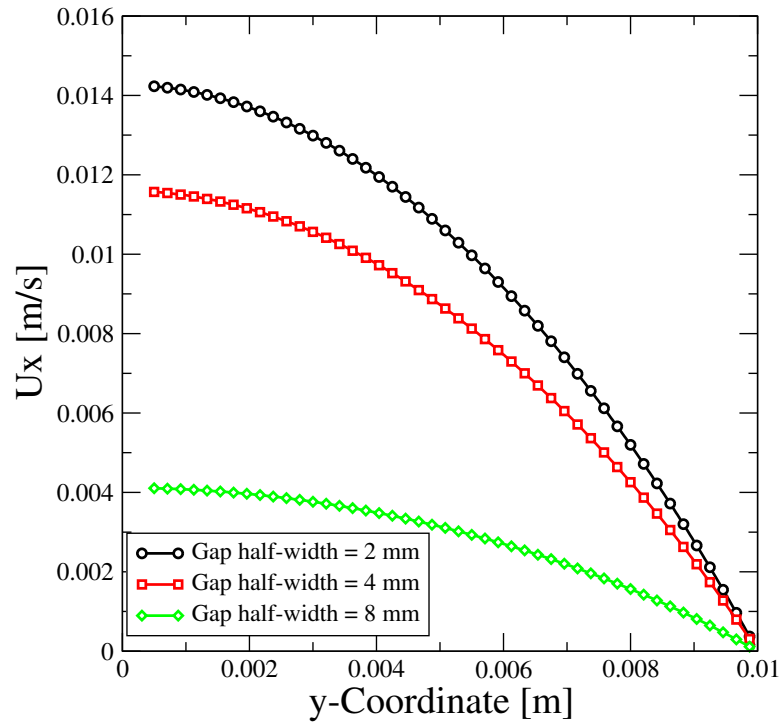


Figure 4.15: Moving frame x-component velocity profile for the Newtonian fluid near the right boundary at $x=89$ mm for different gap half-widths. (The roller speed is 10 mm/s.)

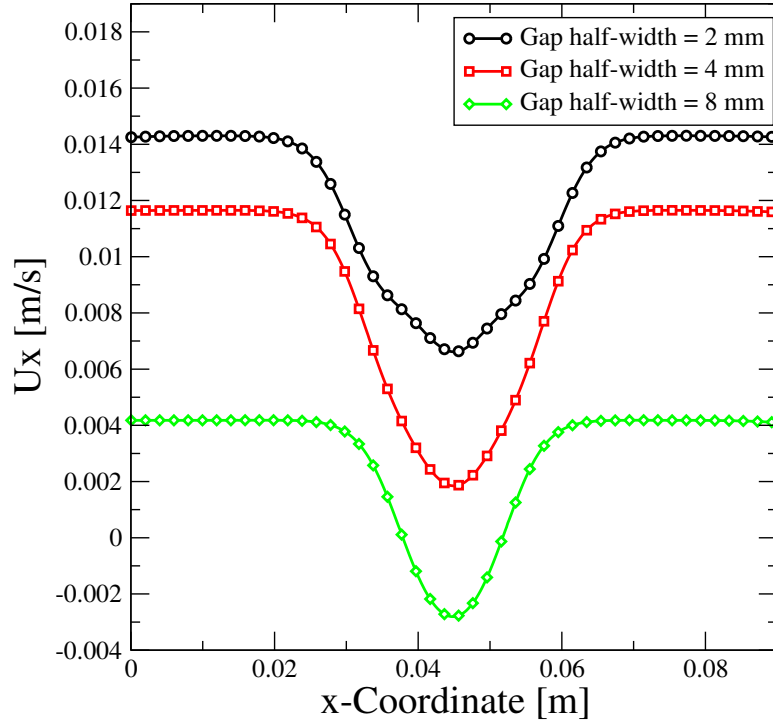


Figure 4.16: Moving frame x-component of the velocity of the Newtonian fluid along the centerline at $y=0.05$ mm for different gap half-widths. (The roller speed is 10 mm/s.)

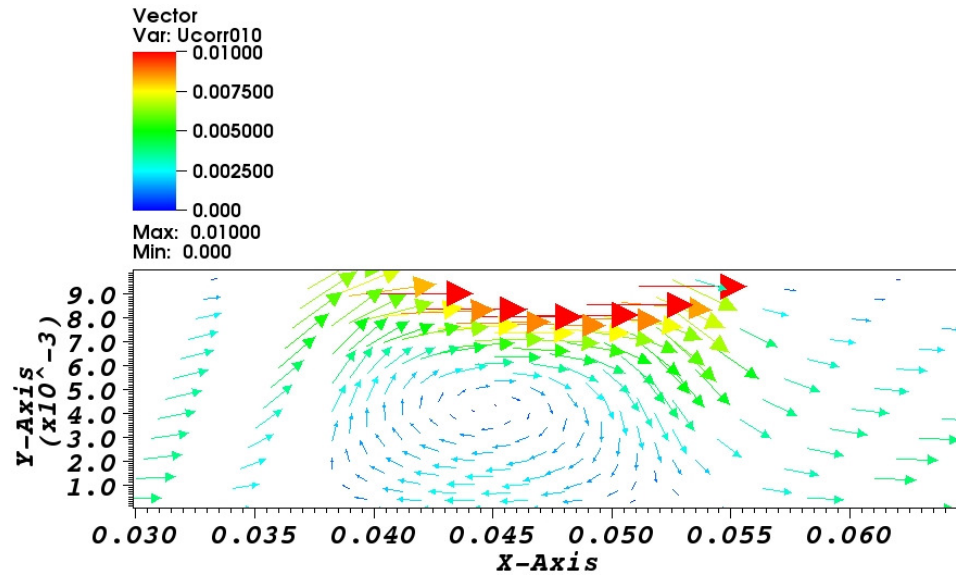


Figure 4.17: Moving frame velocity vectors [m/s] under the roller. (The gap half-width is 8 mm, the roller speed is 10 mm/s.)

Variation of the Roller Speed

To study the effect of the roller speed on the transport efficiency, several cases for a Newtonian fluid with normalized gap half-width of 0.8 and different roller speeds of 2.5, 5, 10 mm/s are simulated. Table 4.5 shows that the transport efficiency is independent of the roller speed. Figure 4.18 shows the x-component of the velocity along the centerline at $y=0.05$ mm for the different roller speeds. The figure shows the presence of a back-flow under the roller for the different speeds and this back-flow increases when the roller speed increases. This increase in the back-flow intensity doesn't affect the transport efficiency, since it is compensated by the roller speed.

Figure 4.19 shows the x-component of the velocity near the right boundary at $x=89$ mm. This figure shows that the maximum values of the curves scale approximately with the roller speeds, which is consistent with the almost constant transport efficiencies listed in Table 4.5.

Table 4.5
Moving frame transport efficiency for different roller speeds

Roller speed [mm/s]	Transport efficiency
2.5	27%
5	28%
10	28%

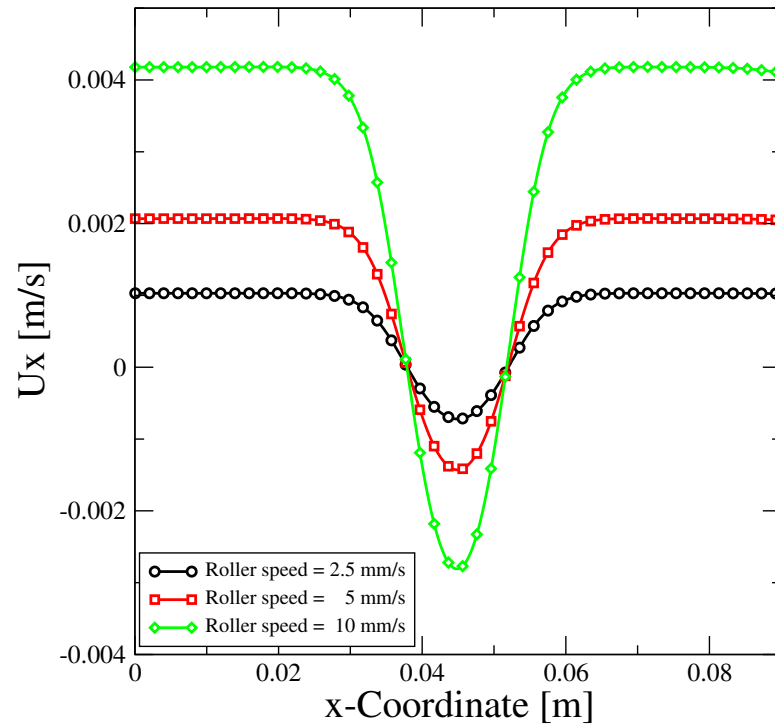


Figure 4.18: Moving frame x-component of the velocity of the Newtonian fluid along the centerline at $y = 0.05$ mm for different roller speeds. (The gap half-width is 8 mm.)

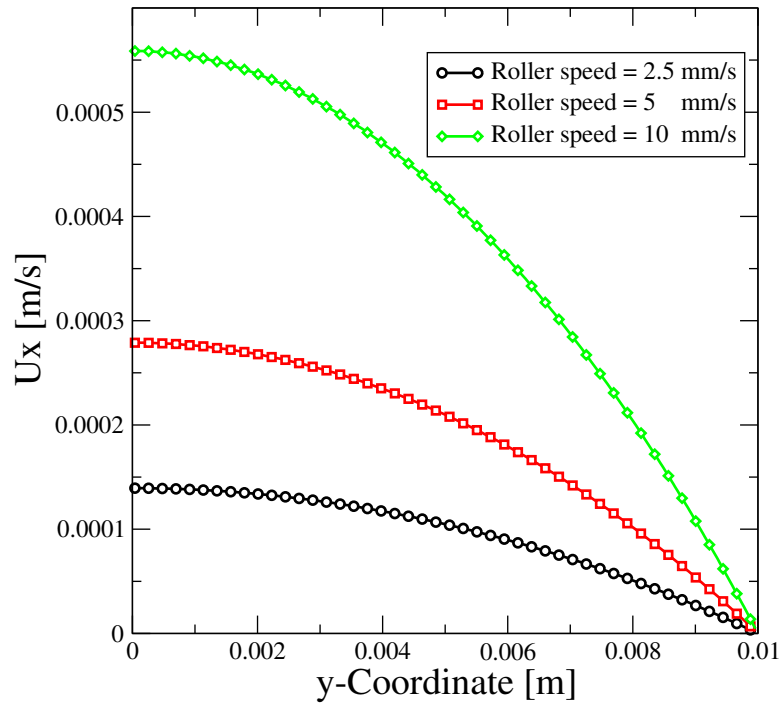


Figure 4.19: Moving frame x-component velocity profile of the Newtonian fluid near the right boundary at $x=89$ mm for different roller speeds. (The gap half-width is 8 mm.)

4.2 Fixed Frame Simulations

In the fixed frame simulations the geometry (computational domain) is changing. The walls are stationary and the roller motion is represented as a moving wave which results in a deforming mesh.

The two-dimensional, symmetric computational domain reflects the upper half of a tube or channel whose length is 180 mm and the diameter is 20 mm. The initial (undeformed) computational mesh consists of one block with a uniform cell distribution. In the x-direction the standard mesh has 810 cells and 23 cells in the y-direction. This gives a total of 18'630 cells for the standard mesh.

The moving wave on the upper wall is generated by moving the mesh points of the upper wall vertically. The vertical movement of the mesh points on the upper wall depends on the horizontal motion of a wave of a given shape and speed. The x-component of the moving vertex has the speed c and the y-component is moving down or up according to the type of mesh deformation. In the standard case, the wave is formed by a circular roller of diameter 30 mm which is moving with the uniform speed of 5 mm/s in the positive x-direction and forms a half gap width of $h=4$ mm. The contact curve between the roller and the tube is a circular arc with a segment length of 24 mm.

The computational domain in the fixed frame simulation is shown in Fig. 4.20. In the undeformed mesh the length of a cell is 0.222 mm, and the height is 0.435 mm. In the deformed mesh, the smallest cells are located under the roller and the cell height is reduced

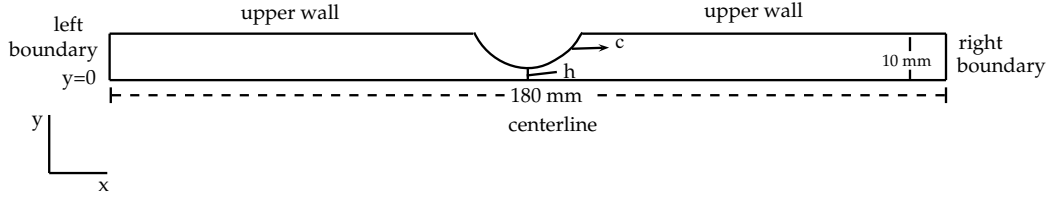


Figure 4.20: Computation domain for fixed frame simulations.

to 0.174 mm.

In comparison to the moving frame geometry, the length of the fixed frame domain is doubled. The initial and boundary conditions reflect the experimental setup and are described in the next subsection.

4.2.1 Initial and Boundary Conditions

The boundary conditions for the pressure and velocity on the left and right boundaries and at the centerline are the same as in the moving frame simulations discussed in the previous section. However, different boundary condition for the velocity is required for the upper wall. This condition is a modification of an existing boundary condition of OpenFOAM, called `movingWallVelocity`, which corrects the flux due to the mesh motion so that the total flux across the wall is zero. The modification enforces that the velocity on the upper wall, including the roller, is zero in the normal direction, that is, on the roller the velocity relative to the velocity of the moving mesh, i.e., $\mathbf{u} - \mathbf{u}_s$, is zero.

To generate the moving wave, an existing boundary condition has been modified to move the mesh points on the upper wall vertically according to a prescribed mathematical formula

that gives the position of the mesh point as a function of time. The circular deformation is controlled by the equation (refer to Fig. 4.21)

$$\alpha(x, t) = y_{\max} - (y_0 - \sqrt{r^2 - (x - x_0(t))^2}), \quad x_1(t) \leq x \leq x_2(t) \quad (4.3)$$

where y_{\max} is the y-component of the points on the undeformed upper wall, $(x_0(t), y_0)$ is

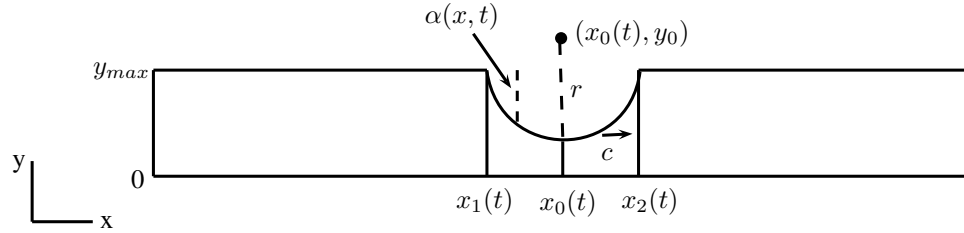


Figure 4.21: The circular wave.

the center of the circle whose radius is r . The x-components of the center and the two ends of the circular arc are moving with the uniform speed of the roller and are given by

$$x_i(t) = x_i + ct, \quad \text{for } i = 0, 1, 2, \quad (4.4)$$

where x_i 's are the initial values at $t = 0$. In fact, the x-component of the center x_0 is determined by the user, while x_1 and x_2 are computed so that $\alpha(x_i, 0) = 0$ for $i = 1, 2$. Notice that the above deformation is valid if and only if $y_0 \geq y_{\max}$.

The parameters for this boundary condition are given in the `<case>/0/pointMotionU` file. These parameters are summarized in Table 4.6. In addition to the modified moving-mesh boundary condition on the upper wall, the zero gradient boundary condition

Table 4.6

Parameters for the circular deformation. (Fixed frame simulations.)

Formula Parameters	OpenFOAM Variable name
r	circleRadius
c	speed
x_0	xCompInitialCenter
y_0	yCompFinalCenter

has been used for the left and right boundaries and a symmetry plane boundary condition has been imposed for the centerline.

4.2.2 Computational Details

For the simulations a transient solver for incompressible, turbulent flows with dynamic mesh capability, called `transientSimpleDyMFoam`, is used. This solver utilizes the segregated SIMPLE-based pressure-velocity coupling algorithm in time-stepping mode, called PISO, discussed in Chapter 2. This solver has been used for the laminar flows in this thesis by deactivating the turbulence models. The temporal loop of the `transientSimpleDyMFoam` solver can be summarized as:

- 1) Update the time t from t^n to t^{n+1} by adding the time step Δt .
- 2) Correct the face fluxes Φ_f to an absolute velocity field \mathbf{u}_f by

$$\Phi_f = \mathbf{u}_f \cdot \mathbf{S}. \quad (4.5)$$

- 3) Use a dynamic mesh solver to move the mesh.
- 4) Correct the fluxes relative to the mesh motion by using

$$\Phi_f = (\mathbf{u} - \mathbf{u}_s)_f \cdot \mathbf{S}, \quad (4.6)$$

where \mathbf{u}_s is the velocity of the boundary surface for the moving control volumes.

For the Newtonian simulations the dynamic viscosity is 0.1452 Pa.s, which corresponds to the zero-shear-rate-dependent viscosity of the non-Newtonian cases.

The transport properties and the discretization schemes are the same as in the moving frame simulations. However, the time derivative in the fixed frame simulation is discretized by the first order, implicit, bounded `Euler` method discussed in Chapter 2. The standard second order finite volume discretization of Gaussian integration with a central differencing scheme has been selected to discretize the gradient, divergence, and Laplacien terms.

For the linear solvers, all pressure equations were solved by the `GAMG` solver with a Gauss-Seidel smoother. A solver for an asymmetric matrix system called `smoothSolver`, is used for the velocity together with a Gauss-Seidel smoother. The PCG solver with DIC preconditioner has been used as the point motion solver.

The time step is very important for stability in transient simulations. A varying time step has been used such that the maximum Courant number Co does not exceed 0.5, where the Courant number is defined by

$$Co = \frac{|\mathbf{u}| \Delta t}{\Delta x}, \quad (4.7)$$

where Δx is the size of the cell.

Convergence Considerations

As discussed in Chapter 2, the solutions of the conservation equations are obtained by means of iterative methods which are implemented as OpenFOAM applications. These applications produce screen output of convergence data which is redirected into a file named `log`. This `log` file contains information such as residuals, number of iterations and Courant number. The data of this file has been extracted using the OpenFOAM utility `foamLog` which generates several files that can be plotted graphically. Each of these files has the name `<variable>_<occurrence inside the time step>`.

Figures 4.22 and 4.23 show the residual as a function of the iteration steps for the discrete x-momentum and pressure equations, respectively, at the beginning of each pressure-velocity iteration in the moving frame simulations discussed in Section 4.1. These figures show that the residuals generally decrease until the residual controls of 10^{-7} and 10^{-6} for velocity and pressure are satisfied. Recall that these simulations were performed with the steady-state solver `simpleFoam` which uses the SIMPLE algorithm.

Figures 4.24 and 4.25 show the residual as a function of time for the discrete x-momentum and pressure equations, respectively, at the beginning of the last PISO iteration in each time step in the fixed frame simulations using the `transientSimpleDyMFoam` solver which is based on the PISO algorithm. The figures show that the residuals oscillating about

a mean level which is acceptable for an unsteady case like this.

In general, the PISO algorithm of OpenFOAM performs a fixed number of pressure-velocity iterations in each time step. In `transientSimpleDyMFoam`, the number of such iterations is controlled by the keyword `nOuterCorrectors` in `<case>/system/fvSolution`. In the fixed frame simulations performed here, `nOuterCorrectors` was set to 20. The absolute tolerances for the linear solvers within each pressure-velocity iterate are shown in Table 4.7 for the moving and fixed frames. The relative tolerances for these solvers in the moving frame were set to 0.1 and 0.01 for the velocity and pressure equations. In the fixed frame the relative tolerance for all variables are set to zero to force the solutions of the system of equations to absolute tolerances in each time step, which is recommended when using the PISO algorithm. For more details refer to Lucchini [48].

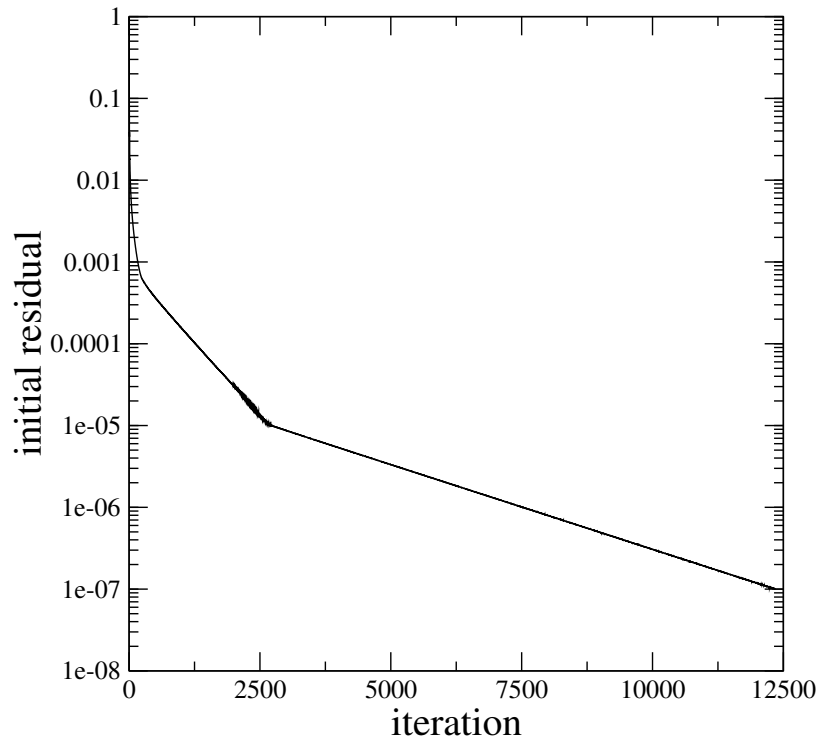


Figure 4.22: Residual of the discrete x-momentum equation at the beginning of each pressure-velocity iteration in the moving frame simulations (using simpleFoam).

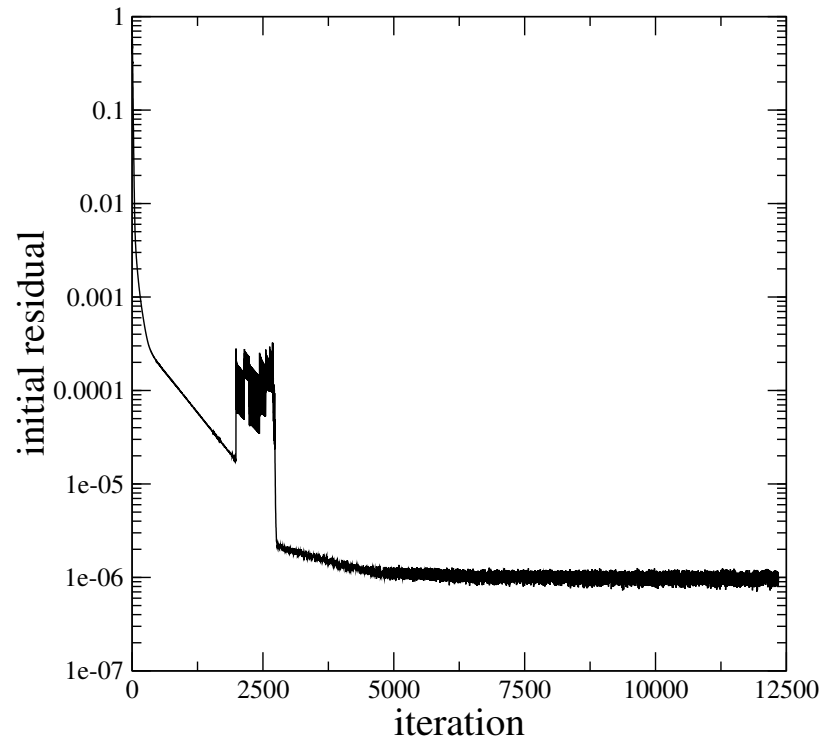


Figure 4.23: Residual of the discrete pressure equation at the beginning of each pressure-velocity iteration in the moving frame simulations (using simpleFoam).

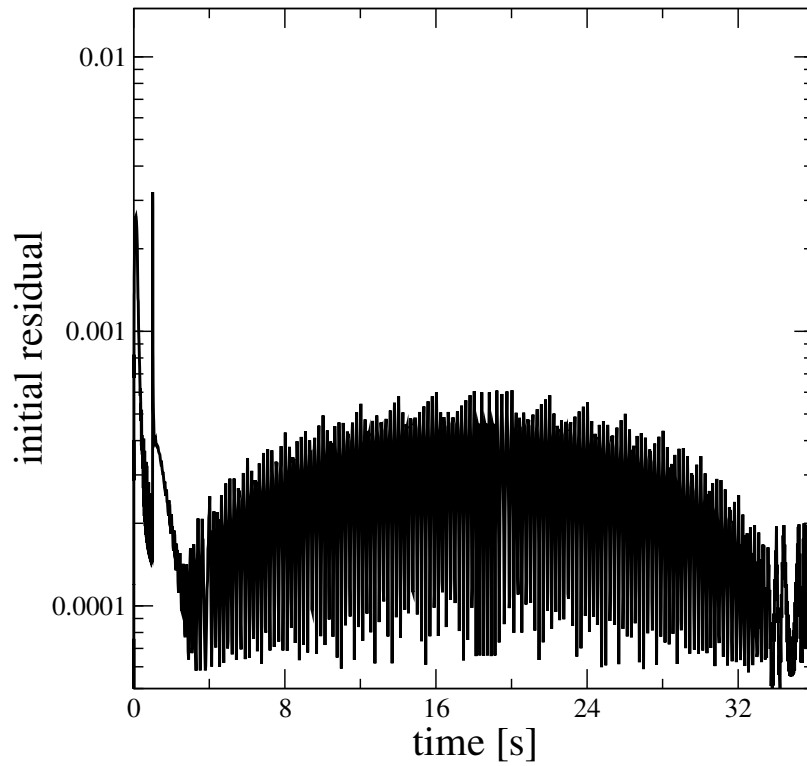


Figure 4.24: Residual of the discrete x-momentum equation at the beginning of the last (i.e. 20th) PISO iteration in each time step in the fixed frame simulations (using transientSimpleDyMFoam).

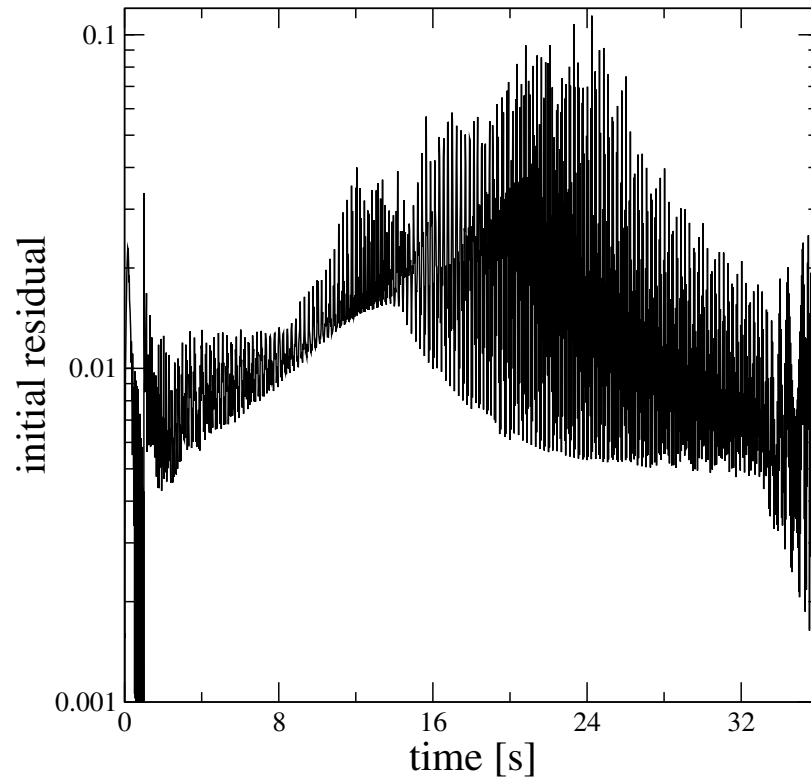


Figure 4.25: Residual of the discrete pressure equation at the beginning of the last (i.e. 20th) PISO iteration in each time step in the fixed frame simulations (using transientSimpleDyMFoam).

Table 4.7
Absolute Tolerances

	Moving Frame	Fixed Frame
p _{corr}	-	10^{-7}
p	10^{-6}	10^{-6}
p _{Final}	-	10^{-7}
U	10^{-5}	10^{-7}
U _{Final}	-	10^{-7}

4.2.3 Mesh Dependence Study

The mesh dependence study has been performed for a Newtonian fluid for the case with the roller speed of 10 mm/s and a half gap width of 2 mm. This is presumably the worst case scenario with the largest gradients for pressure and velocity.

A coarser and a finer mesh have been obtained from the standard mesh by reducing, respectively, increasing, the number of cells in the x and y directions by a factor of 1.5. In the deformed fine mesh the height of the smallest cell is 0.057 mm and in the coarse mesh the height of the smallest cell is 0.133 mm. The details of the three different meshes are shown in Table 4.8.

Table 4.8
Number of cells for the different meshes.(Fixed frame simulations)

Mesh	Number of cells	Total number of cells
Coarse	$540 \times 15 \times 1$	8100
Standard	$810 \times 23 \times 1$	18630
Fine	$1215 \times 35 \times 1$	42525

The results of this mesh refinement study are shown in Figs. 4.26 and 4.27 close to the centerline at $y=0.04$ mm and at time $t=14$ s. In these figures, the central vertical dashed lines indicate the x-coordinate of the center of the roller. The left and right vertical dashed lines represent the end of the roller, that is, the start and end of contact between the roller and the tube.

Figure 4.26 shows that the three different meshes give almost identical values for the x-component of the velocity. This figure shows that the minimum velocity occurs in the gap region. The negative values of the x-component of the velocity indicate the presence of a back-flow.

Figure 4.27 shows that the kinematic pressure values of the standard mesh are closer to the ones of the fine mesh than to the ones of the coarse mesh. Therefore, it can be concluded that the standard mesh exhibits sufficient mesh resolution. This figure shows that the extreme values of the pressure occur in the region under the roller.

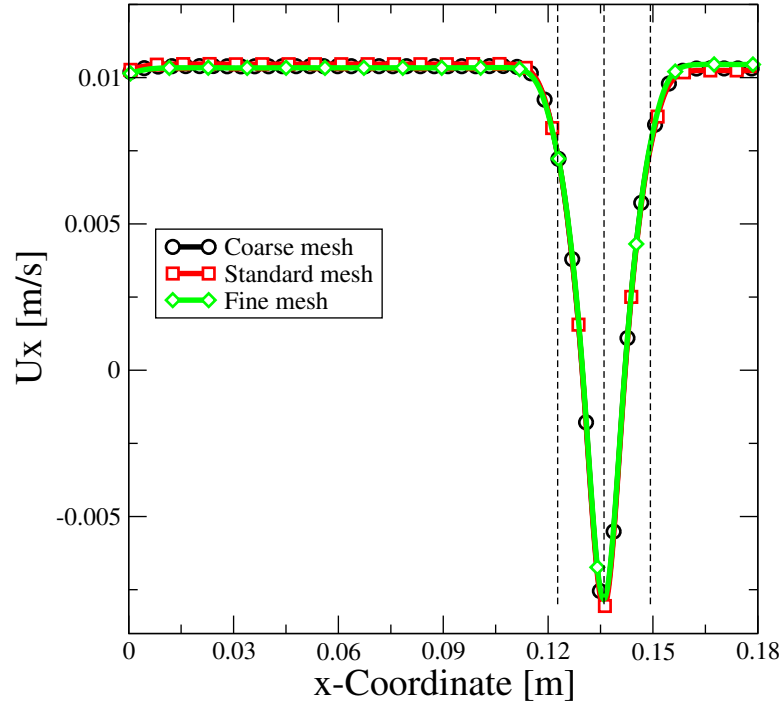


Figure 4.26: Fixed frame mesh dependence study for the x-component of the velocity of the Newtonian fluid along the centerline at $y=0.04$ mm and time $t=14$ s. (The left and right vertical dashed lines represent the ends of the roller, the middle vertical dashed line is the center of the roller.)

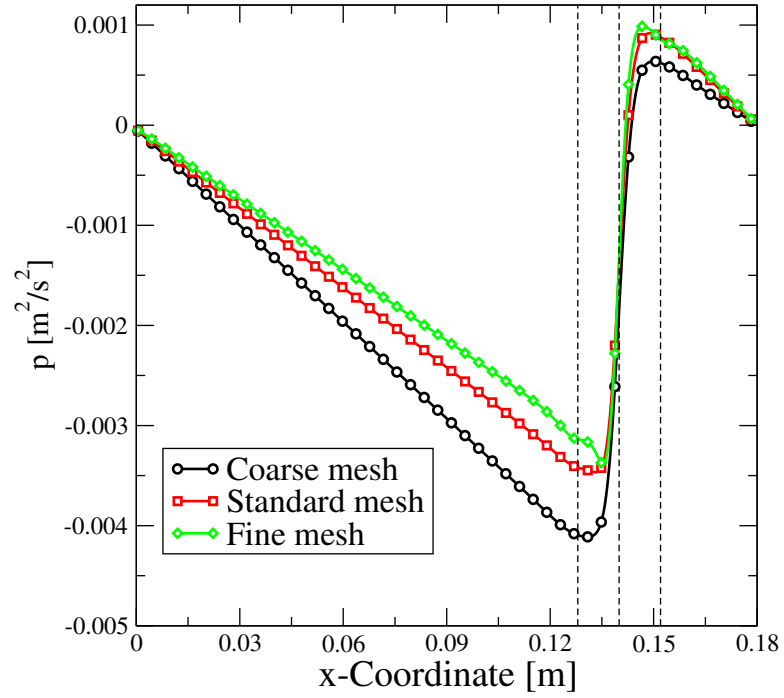


Figure 4.27: Fixed frame mesh dependence study for the kinematic pressure of the Newtonian fluid along the centerline at $y=0.04$ mm and time $t=14$ s. (The left and right vertical dashed lines represent the ends of the roller, the middle vertical dashed line is the center of the roller.)

4.2.4 Parameter Study

The parameter study has been performed to identify the effect of the shear-thinning, non-Newtonian behavior of the fluid, and to examine the material transport efficiency in terms of the roller speed and the gap width.

Newtonian versus Non-Newtonian Fluids

A comparison between the Newtonian and the non-Newtonian simulations has been made. The fluid parameters are the same as in the parameter study for the moving frame given in Table 4.2. Note that the zero-shear-rate viscosity η_0 for the non-Newtonian fluids is the constant viscosity used for the Newtonian case. This study has been performed for the case with the gap half-width of 4 mm, and the roller speed of 5 mm/s.

The results of the simulation of the Newtonian standard case at time $t=28$ s are shown in Figs. 4.28-4.31. Figure 4.28 is a color plot of the x-component of the velocity for the Newtonian fluid. The figure shows that the velocities in the region under the roller are negative which indicates a back-flow.

Figure 4.29 shows the kinematic pressure distribution. The negative and positive pressure regions under and near the roller are consistent with the results shown in Fig. 4.27.

Figure 4.30 is a color plot of the shear rates. The figure shows that the maximum values of

the shear rates are achieved at the roller boundary and just under the roller. Also, the figure shows small shear rates along the centerline.

The velocity vectors are shown in the Fig. 4.31. The direction of these vectors indicates the presence of a back-flow, which is consistent with the negative x-components of the velocity shown in Fig. 4.28 .

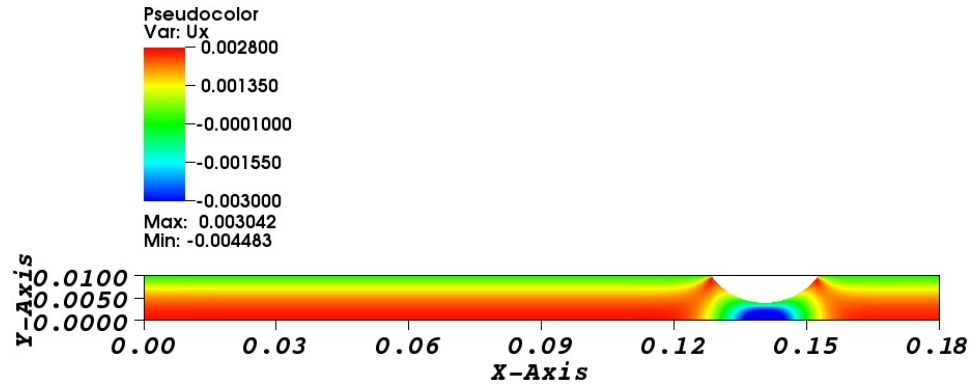


Figure 4.28: Fixed frame x-component of the velocity [m/s] of the Newtonian fluid at time $t=28$ s. (The gap half-width is 4 mm, the roller speed is 5 mm/s.)

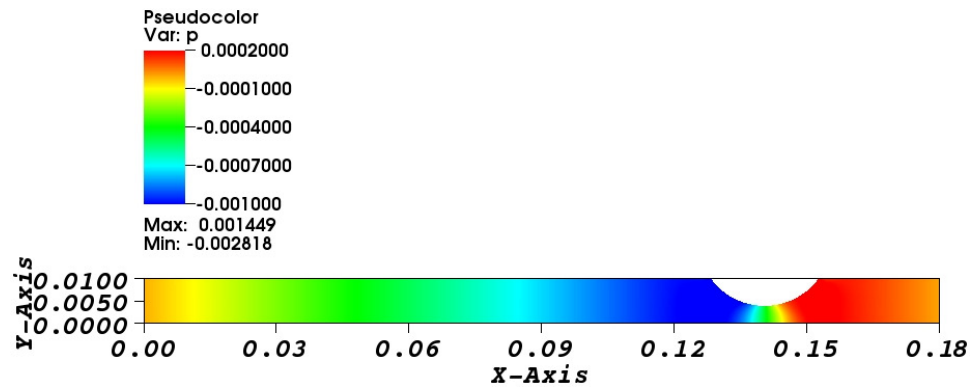


Figure 4.29: Fixed frame kinematic pressure [m^2/s^2] of the Newtonian fluid at time $t=28$ s. (The gap half-width is 4 mm, the roller speed is 5 mm/s.)

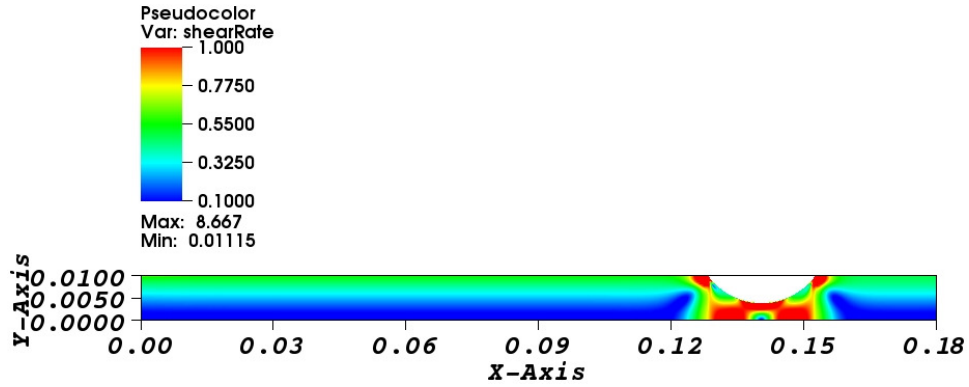


Figure 4.30: Fixed frame shear rate [s^{-1}] of the Newtonian fluid at time $t=28$ s. (The gap half-width is 4 mm, and the roller speed is 5 mm/s.)

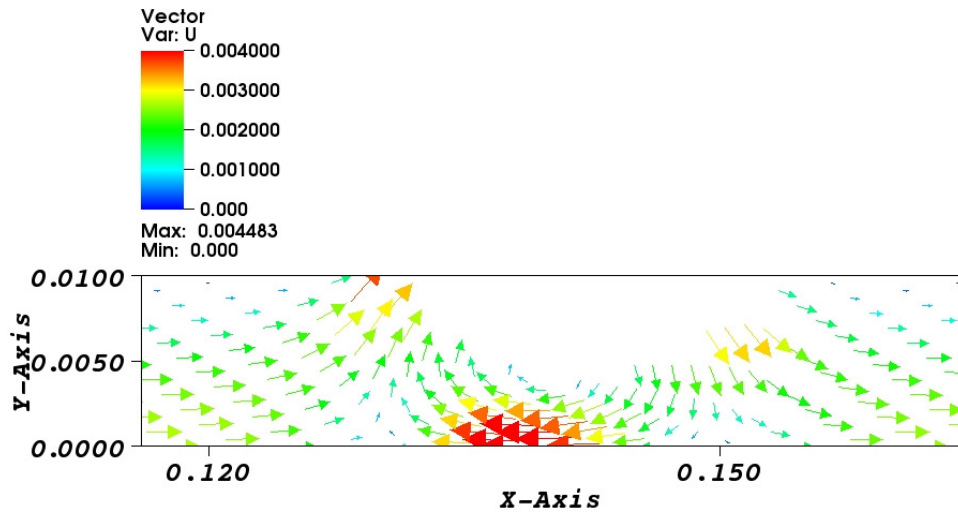


Figure 4.31: Fixed frame velocity vectors [m/s] under the roller of the Newtonian fluid at $t=28$ s. (The gap half-width is 4 mm, and the roller speed is 5 mm/s.)

The simulation results of the x-component of the velocity along the centerline at $y=0.04$ mm and time $t=28$ s for the Newtonian and non-Newtonian fluids are given in Fig. 4.32. It can be seen that the different fluids have a back-flow. This back-flow increases when the shear-thinning behavior increases, with their maximum values occurring in the most narrow part of the channel. This back-flow is consistent with the velocity x-component profiles computed near the right boundary at $y=179$ mm and time $t=28$ s, shown in Fig. 4.33. In this figure the profiles for the Newtonian fluid resembles a parabola, whereas the profiles for the non-Newtonian fluids are more plug-like, with the more shear-thinning fluid being flatter. The gap between the two upper curves is twice the gap between the two lower curves, which is consistent with the transport efficiency results given in Table 4.9. The table shows that the efficiency decreases with more shear-thinning.

Figure 4.34 shows the pressure along the centerline at $y=0.04$ mm and time $t=28$ s for the three different fluids. In this figure, the smallest pressure variation is observed for the more shear-thinning fluid. The pressure curves in Fig. 4.34 intersect near the most narrow part of the channel. As discussed for the corresponding case of the moving frame simulation in Fig. 4.12, the pressure variation is decreasing with increasing shear-thinning.

Figure 4.35 presents the shear rates near the centerline at $y=0.04$ mm and time $t=28$ s for the different fluids. The figure shows that the maximum shear rate under the roller increases with the amount of shear-thinning, and reaches the maximum value of $1.7s^{-1}$ for the most shear-thinning fluid.

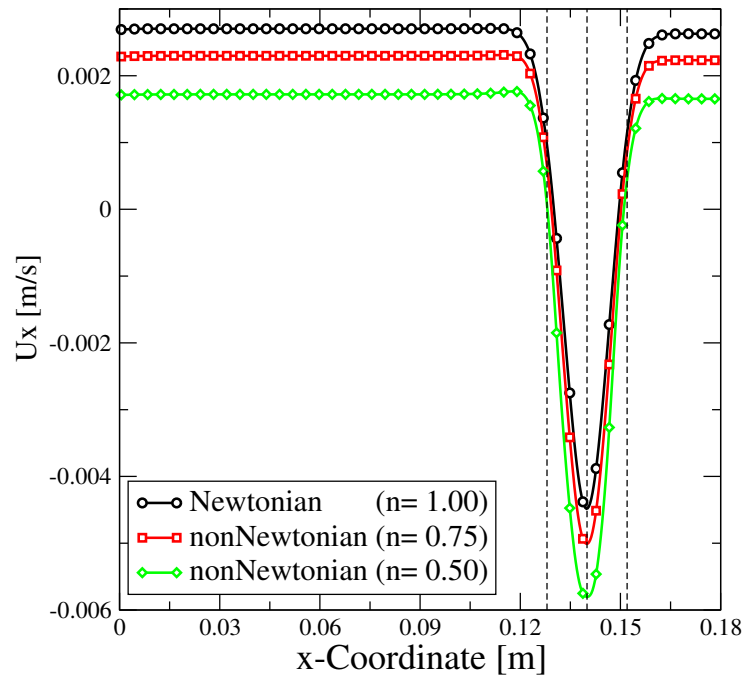


Figure 4.32: Fixed frame x-component of the velocity for the Newtonian and non-Newtonian fluids along the centerline at $y=0.04$ mm and time $t=28$ s. (The left and right vertical dashed lines represent the ends of the roller, the middle vertical dashed line is the center of the roller.)

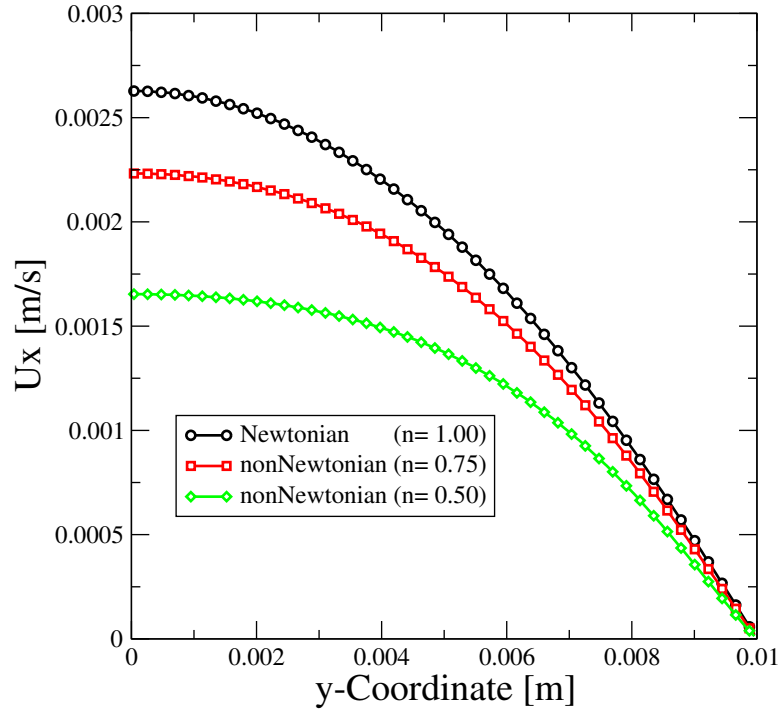


Figure 4.33: Fixed frame x-component velocity profile for the Newtonian and non-Newtonian fluids near the right boundary at $x=179$ mm and time $t=28$ s.

Table 4.9

Fixed frame transport efficiency for different fluids

Flow type	Transport efficiency
Newtonian (n= 1.00)	35%
non-Newtonian (n= 0.75)	31%
non-Newtonian (n= 0.50)	24%

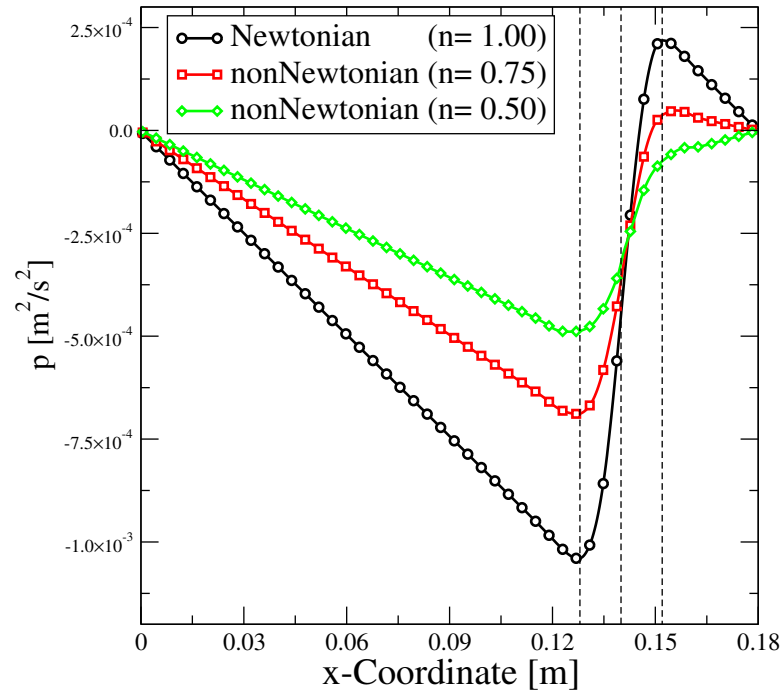


Figure 4.34: Fixed frame kinematic pressure for the Newtonian and non-Newtonian fluids along the centerline at $y=0.04$ mm and time $t=28$ s. (The left and right vertical dashed lines represent the ends of the roller, the middle vertical dashed line is the center of the roller.)

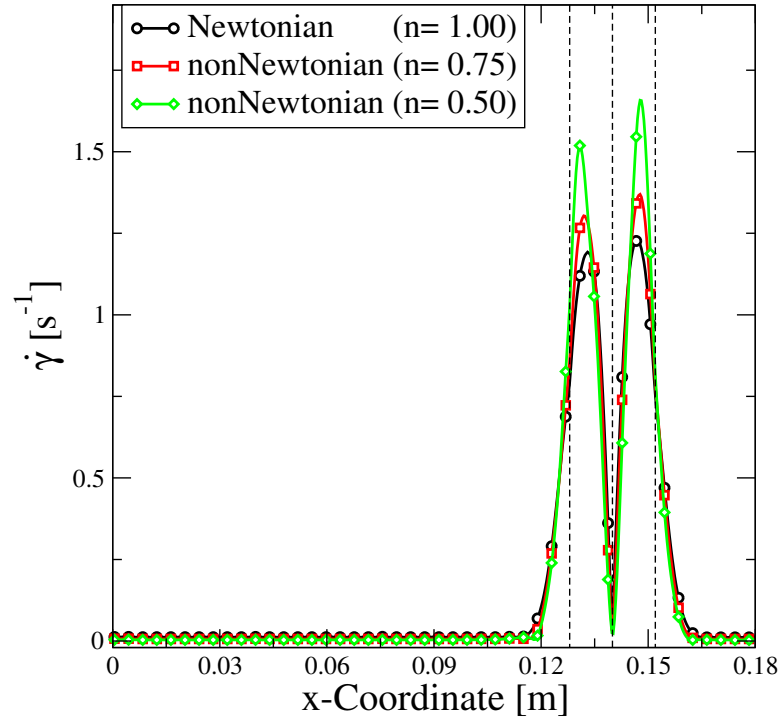


Figure 4.35: Fixed frame shear rate for the Newtonian and non-Newtonian fluids along the centerline at $y=0.04$ mm and time $t=28$ s. (The left and right vertical dashed lines represent the ends of the roller, the middle vertical dashed line is the center of the roller.)

Variation of the Gap Width and the Roller Speed

Several simulations for Newtonian fluid have been carried out to study the transport efficiency in terms of the normalized gap width and the roller speed. The first group of simulations have been performed with a fixed roller speed of 10 mm/s and several normalized gap half-widths of 0.2, 0.4, 0.8. Recall that the normalized gap half-width is h/R_0 . In order to examine the influence of the roller speed on the transport efficiency, another group of simulations has been performed with a fixed normalized gap half-width of 0.8 and different roller speeds of 2.5, 5 and 10 mm/s.

The results of these simulations are shown in Table 4.10. These results show that the transport efficiency is independent of the roller speed and decrease with increasing gap half-width. In fact, the presence of the back-flow in the fixed frame leads to a sharper decrease of the efficiency when the gap half-width increase.

Table 4.10
Fixed frame transport efficiencies

normalized half-width gap	roller speed (mm/s)		
	2.5	5	10
0.2	68%	69%	69%
0.4	35%	35%	35%
0.8	3.7%	3.7%	3.7%

Figure 4.36 shows the x-component velocity profile near the right boundary at $x=179$ mm and time $t=14$ s for different gap half widths with a roller speed of 10 mm/s. It can be seen from the figure that the velocity decreases when the gap half-width increases. Also, the velocity values are very small when the gap half-width is 8 mm. This is consistent with the results shown in Table 4.10.

Figure 4.37 shows the x-component of the velocity for the Newtonian fluid along the centerline at $y=0.04$ mm and time $t=14$ s for different gap half-widths and a roller speed of 10 mm/s. The figure shows that there is a back-flow for all the different gap half-widths. It is interesting to see that the back flow is increasing when the gap half-width increases from 2 mm to 4 mm, but that the back flow decreases when the gap half-width is further increased to 8 mm.

Figure 4.38 shows the x-component velocity profile near the right boundary at $x=179$ mm and time $t=14$ s for different roller speeds and gap half-width of 8 mm. It is obvious from the figure that the increase of the roller speed gives higher velocities. At $y=0$ the distance between the curves corresponding to the roller speed of 10 mm/s and 5 mm/s is almost double the distance between the curves corresponding to the roller speed of 5 mm/s and 2.5 mm/s. This is consistent with the constant transport efficiencies shown in Table 4.10.

Also, the previous results are consistent with the results in Fig. 4.39 which shows the x-component of the velocity along the centerline at $y=0.04$ mm and time $t=14$ s for different roller speeds and gap half-width of 8 mm. The minimum velocity of the case with the roller speed of 10 mm/s is double the one of the case with the roller speed of 5 mm/s, the latter is

also double the minimum velocity of the case with the roller speed of 2.5 mm/s.

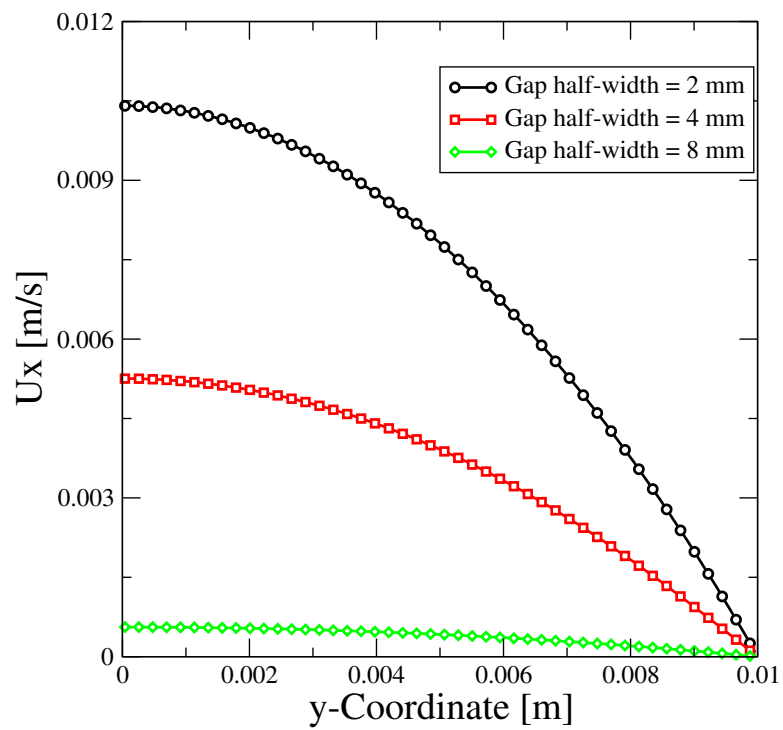


Figure 4.36: Fixed frame x-component velocity profile of the Newtonian fluid near the right boundary at $x=179$ mm and time $t=14$ s for different gap half-widths. (The roller speed is 10 mm/s.)

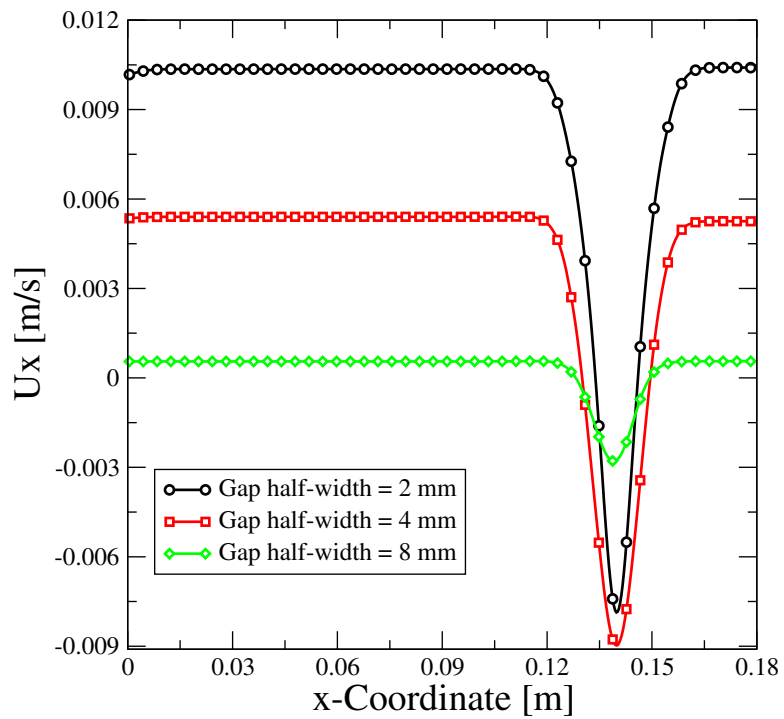


Figure 4.37: Fixed frame x-component of the velocity of the Newtonian fluid along the centerline at $y=0.04$ mm and time $t=14$ s for different gap half-widths. (The roller speed is 10 mm/s.)

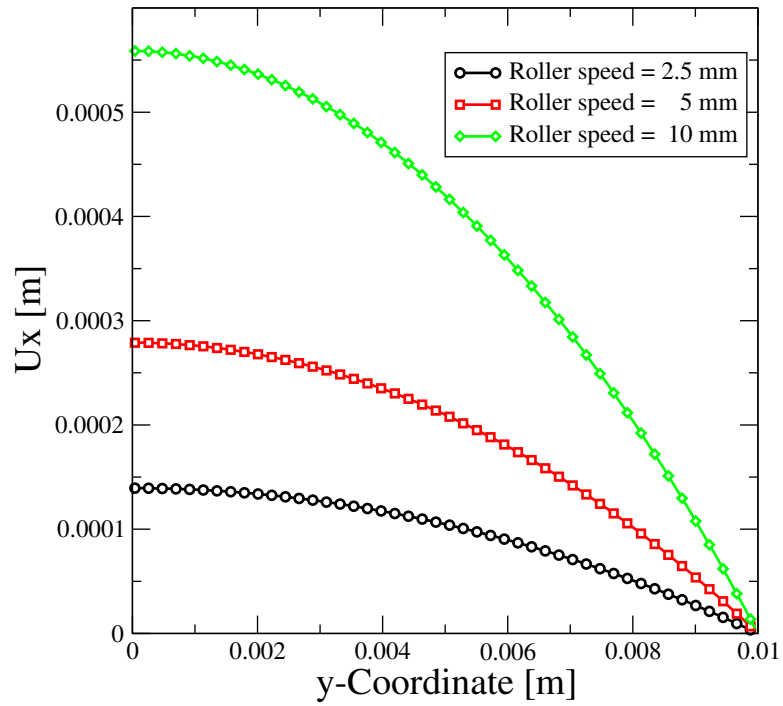


Figure 4.38: Fixed frame x-component velocity profile for the Newtonian fluid near the right boundary at $x=179$ mm and time $t=14$ s for different roller speed. (The gap half-width is 8 mm.)

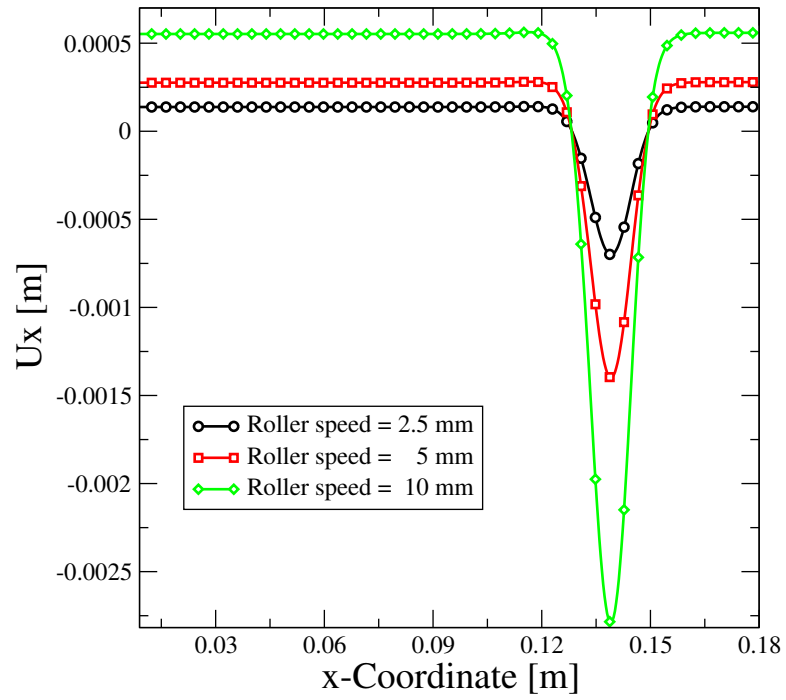


Figure 4.39: Fixed frame x-component of the velocity of the Newtonian fluid along the centerline at $y=0.04$ mm and time $t=14$ s for different roller speeds. (The gap half-width is 8 mm.)

4.2.5 Comparison between the Moving and Fixed Simulations

In the moving and fixed simulations, the minimum velocities are attained in the region under the roller, the peristaltic wave tends to produce a rising pressure in the direction of the wave. Also, the transport efficiency is found to be independent of the roller speed and decreases when the gap half-width increases.

In the moving frame, a zero velocity boundary condition is imposed on the roller boundary. In the fixed frame the velocity boundary condition on the roller is modified such that the normal velocity is zero. Due to these different velocity boundary conditions, the effect of the shear-thinning viscosity and the gap half-width on the transport efficiency is more significant in the fixed frame simulations, as is shown in Tables 4.11 and 4.12, respectively.

The standard case in the moving frame simulation needs around 12350 iterations to

Table 4.11
Transport efficiency for different fluids

Flow type	Moving frame	Fixed frame
Newtonian ($n= 1$)	78%	35%
non-Newtonian ($n= 0.75$)	77%	31%
non-Newtonian ($n= 0.50$)	76%	24%

converge, and the CPU time is about 10 minutes, while the standard case in the fixed frame simulation needs about 100 minutes for the roller to be out of the domain (at $t=38.4$ s). The CPU time for the fixed frame simulations is larger due to the following reasons:

Table 4.12

Transport efficiency for different normalized gap half-widths.

normalized gap half-width	Moving frame	Fixed frame
0.2	95%	69%
0.4	78%	35%
0.8	28%	3.7%

1. The large number of PISO loops performed per time steps coupled with the total number of time steps.
2. The computational domain in the fixed frame is twice as long as in the moving frame.
3. The mesh motion solver must be performed in the fixed frame.

Although, the moving frame simulation takes less CPU time, the fixed frame approach more accurately represents reality and it is preferable to use. Specifically, in the fixed frame simulation, the velocity boundary condition on the wave guarantees that the velocity is zero in the normal direction which reflects the real boundary condition of the peristaltic wave.

4.2.6 Comparison with Experiment

The experimental setup is shown in Fig. 4.40 (see Nahar [47]). As can be seen, three pairs of rollers are used to induce the peristaltic flow from left to right. The velocity measurements have been obtained by means of the pulsed ultrasound Doppler velocimetry technique of Takeda [49]. The location of the velocity measurements is between the first and second pair of rollers where the tube height is the largest. The ratio of the gap width to the tube height is 4/11 which is very close to the ratio of 8/20 encountered in the standard simulations. The same non-Newtonian fluid and the same elastic tube whose characteristics are described in Chapter 3 are used.

Simulations have been carried out for the non-Newtonian fluid used in the experiments. Recall from Chapter 3 that this fluid has a shear rate dependent viscosity expressed by the Bird-Carreau equation with parameters $\eta_0 = 0.1452$ Pa.s, $\eta_\infty = 0$ Pa.s, $k = 0.02673$ s and $n = 0.7588$. Note that the shear-thinning of this fluid starts at approximately $\dot{\gamma} = 1/k \approx 37.4 \text{ s}^{-1}$ which is a factor of almost 750 larger than for the non-Newtonian fluids used in the parameter studies of Sections 4.1 and 4.2. The results of these simulations are shown in Fig. 4.41 for the roller speeds $c = 10$ mm/s, 5 mm/s, 3 mm/s. As can be seen from Fig. 4.41, the simulation results show excellent agreement with the experimental values.

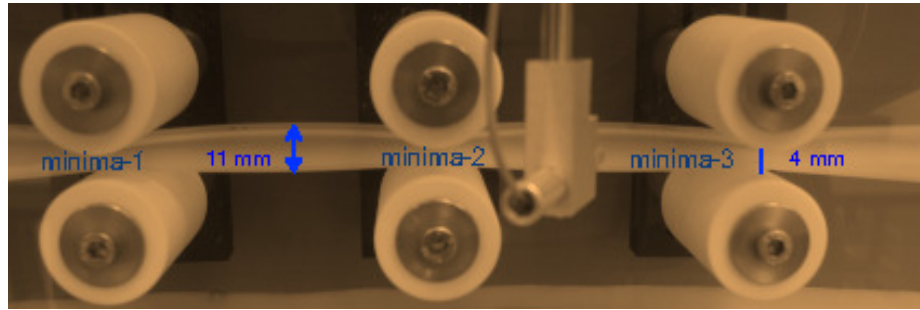


Figure 4.40: Experimental setup of the peristaltic motion.

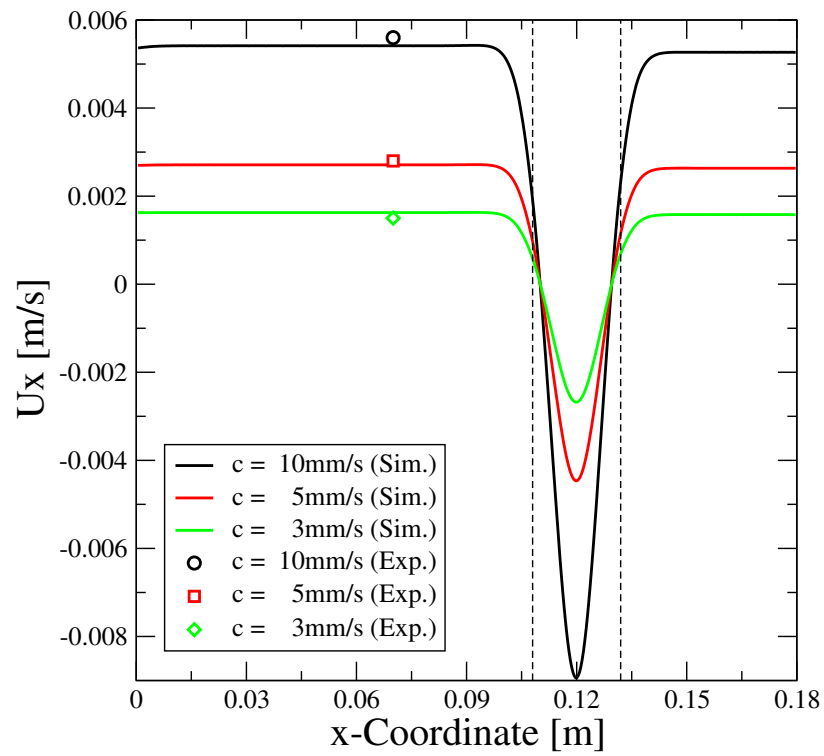


Figure 4.41: Fixed frame x-component of the velocity of the non-Newtonian fluid along the centerline at $y=0.04$ mm. (The gap half-width is 4 mm.)

4.2.7 Three Dimensional Channel

In this subsection, the two-dimensional computational model of peristaltic flow in the fixed frame of reference is extended to three dimensions specifically, the peristaltic flow of a Newtonian fluid in a three-dimensional channel is simulated. The dimensions of the channel are same as in two dimensional case except that the width is 20 mm. The symmetry plane has been used in the middle of the height to save computation time. The initial undeformed mesh consists of one block with $810 \times 23 \times 40$ cells which gives a total of 745'200 cells. As discussed before in this section, the roller motion is presented by a moving wave on the upper wall of the channel. In this simulation the gap half-width is 4 mm, the roller speed is 5 mm/s and the roller diameter is 30 mm.

A three-dimensional view of the computational mesh after deformation is shown in Fig. 4.42.

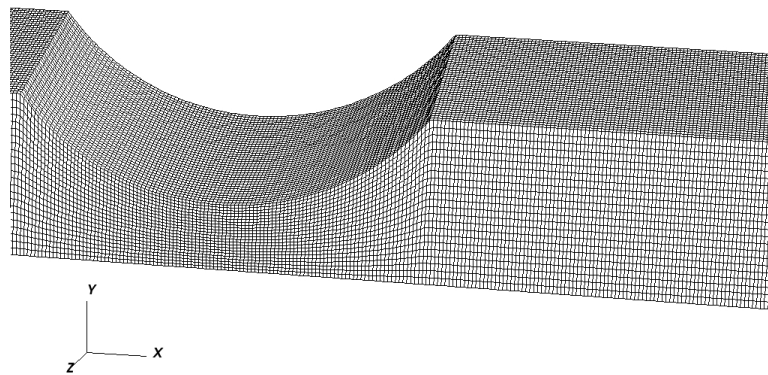


Figure 4.42: Three-dimensional view of the deformed computational mesh.

The boundary conditions are same as in the two dimensional case except that for the two new boundaries normal to the third dimension a fixed value of zero has been imposed for the velocity. In addition, the zero gradient boundary condition was set for the kinematic pressure and point motion.

The remaining simulation setup is same as the two dimensional case. The results of this simulation are shown in Figs. (4.43-4.46).

Figure 4.43 gives a color plot of the x-component of velocity at time $t=28$ s, while Fig. 4.44 gives the corresponding velocity vector plot in the plane bisecting the domain in the z-direction. These figures show that negative velocities are reached in the region under the roller, which indicates that there is a back-flow similar to the two-dimensional case. Moreover, the largest velocity magnitudes are reached near the centerplane underneath the roller. The maximum magnitude of the velocity is almost the same as in the two dimensional case.

Figure 4.45 shows the kinematic pressure distribution. The negative and positive pressure regions under and near the roller are consistent with the fact that the peristaltic wave tends to produce a rising pressure in the direction of the wave. The pressure difference in three dimensional channel is ≈ 2.6 times the one of the two dimensional channel.

Figure 4.46 is a color plot of the shear rates, where the maximum values of the shear rate are in the region of direct contact with the roller and in the narrow gap. The maximum value of the shear rate in three dimensional channel is ≈ 1.7 times the one of the two dimensional channel.

The transport efficiency in the three dimensional channel is 26% compared to 35% in the two dimensional case. The CPU time for the three dimensional case is about 4900 minutes compared to 100 minutes (both CPU times are computed at the first time that the roller is completely out of domain, i.e., at $t=38.4$ s).

In conclusion, although there are some quantitative differences between the two dimensional and three dimensional simulation results, qualitatively the flow behavior is the same. That is, the two dimensional simulation sufficiently reflects the three dimensional flow. This, together with the considerably smaller CPU times of the two dimensional simulation, make it an attractive choice over the three dimensional simulation.

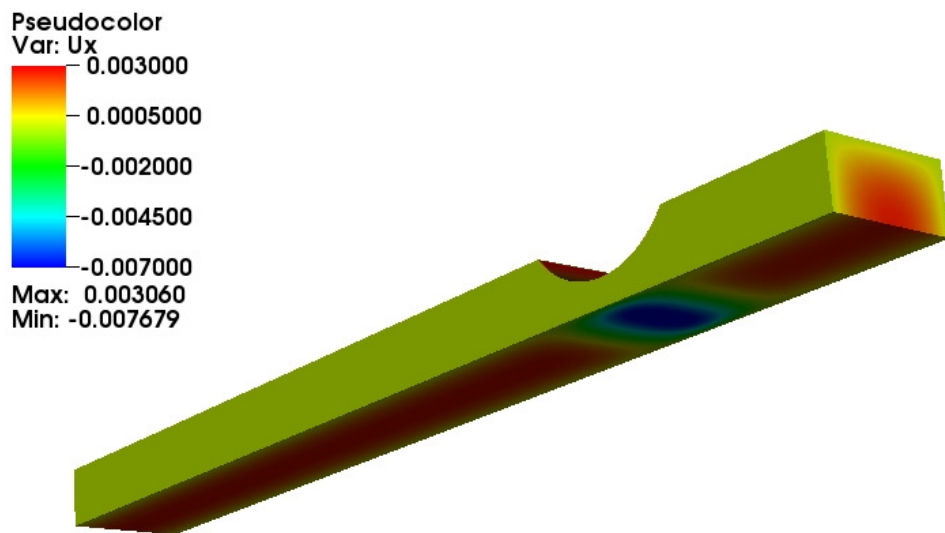


Figure 4.43: Fixed frame x-component of the velocity [m/s] in the three dimensional channel simulation at $t = 28$ s. (The gap half-width is 4 mm, the roller speed is 5 mm/s.)

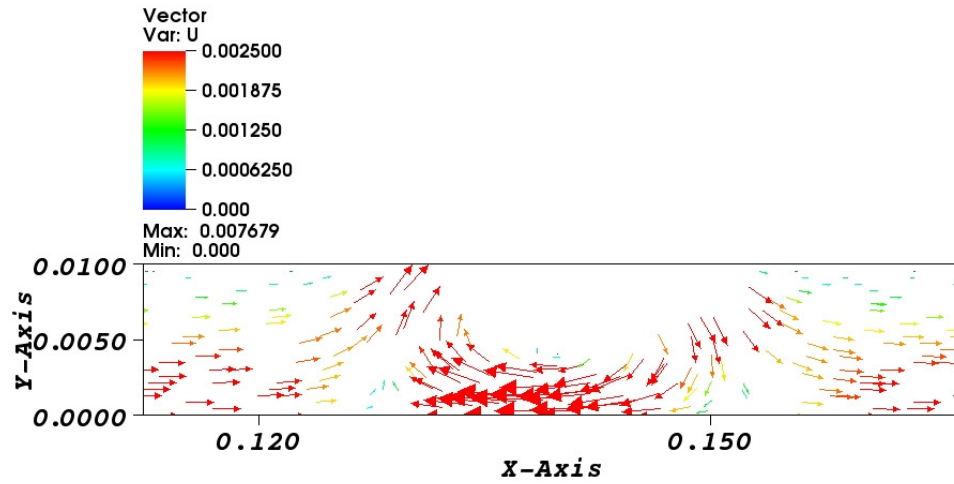


Figure 4.44: Fixed frame velocity vectors [m/s] under the roller in the three dimensional channel simulation at $t = 28$ s. (The gap half-width is 4 mm, the roller speed is 5 mm/s.)

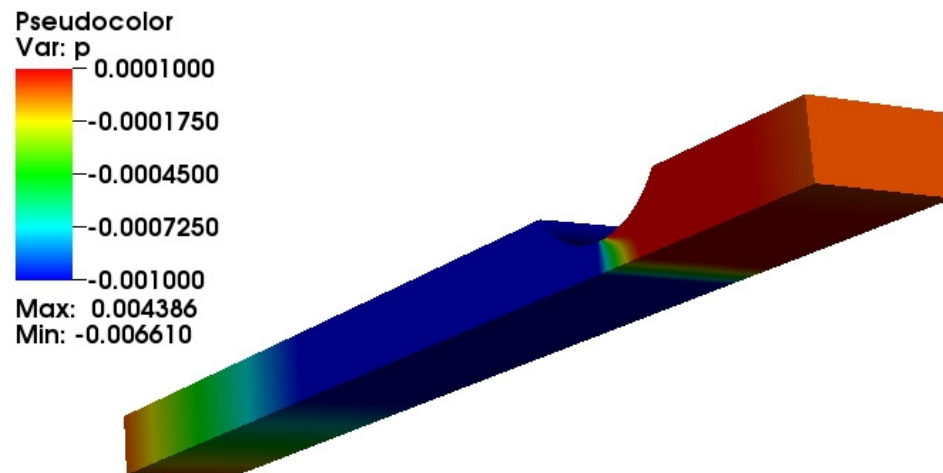


Figure 4.45: Fixed frame kinematic pressure [m^2/s^2] in the three dimensional channel simulation at $t = 28$ s. (The gap half-width is 4 mm, the roller speed is 5 mm/s.)

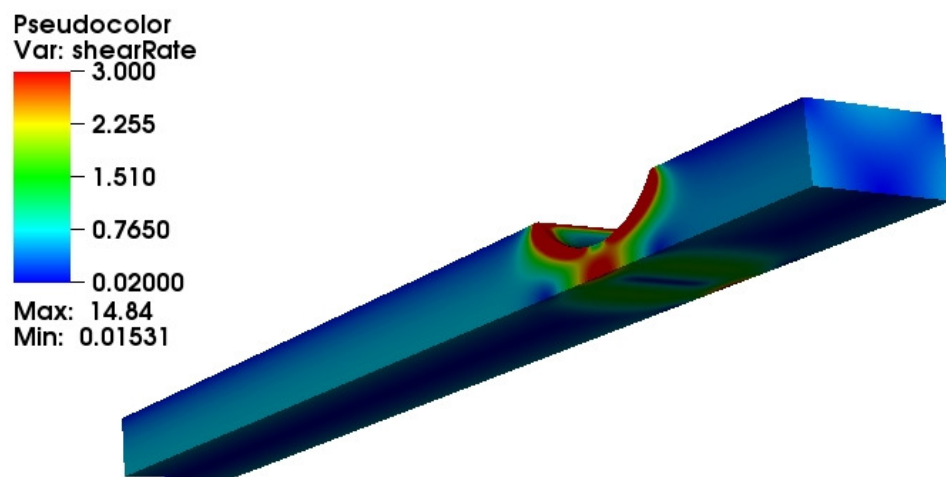


Figure 4.46: Fixed frame shear rate [s^{-1}] in the three dimensional channel simulation at $t = 28$ s. (The gap half-width is 4 mm/s, the roller speed is 5 mm/s.)

4.2.8 Other Types of Moving Waves

In order to check the flexibility of the modified moving-mesh boundary condition, two different types of moving wave have been tested. The upper wall of the two dimensional domain is deformed according to a parabolic and sinusoidal wave.

In the case where the parabolic wave is used, the parabolic deformation is controlled by

$$\alpha(x, t) = y_{\max} - (y_0 + A(x - x_0(t))^2), \quad x_1(t) \leq x \leq x_2(t) \quad (4.8)$$

where y_{\max} is the y-component of the points on the undeformed upper wall, A is positive, and $(x_0(t), y_0)$ is the vertex of the parabola where $x_0(t) = x_0 + ct$. Moreover, $x_1(t) = x_1 + ct$ and $x_2(t) = x_2 + ct$. The initial x-component of the vertex x_0 , is determined by the user, while x_1 and x_2 are computed such that $\alpha(x, t) = 0$. Notice that the parabolic deformation is valid if and only if $y_0 < y_{\max}$. The parameters for this boundary motion are summarized in Table 4.13 whereas the tube geometry is shown in Fig. 4.47.

Table 4.13
Parameters for the parabolic deformation.

Formula Parameters	OpenFOAM Variable name
x_0	xCompInitialVertex
A	CoeffA
c	speed
y_0	yCompFinalVertex

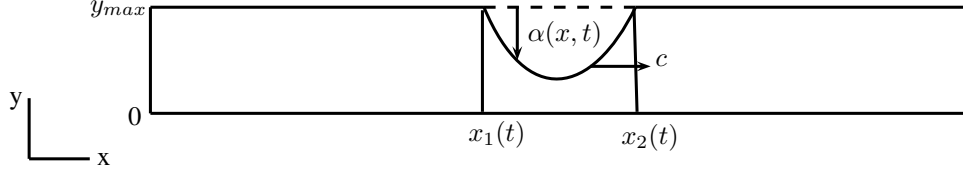


Figure 4.47: The parabolic wave.

On the other hand, in the case where the sinusoidal wave is used, the sinusoidal deformation is controlled by

$$\alpha(x, t) = -A \cos\left(\frac{2\pi}{\lambda}(x - ct)\right), \quad x_1(t) \leq x \leq x_2(t) \quad (4.9)$$

where A is the amplitude of the peristaltic wave, λ is the wavelength, and $x_1(t) = x_1 + ct$, $x_2(t) = x_2 + ct$ where

$$x_1 = \frac{1}{4}\lambda, \quad x_2 = \frac{3}{4}\lambda.$$

Notice that the sinusoidal deformation is valid if and only if $A < y_{\max}$. The parameters for this boundary motion are summarized in Table 4.14 and the tube geometry is shown in Fig. 4.48.

Table 4.14
Parameters for the parabolic deformation.

Formula Parameters	OpenFOAM Variable name
A	amplitude
λ	waveLength
c	speed

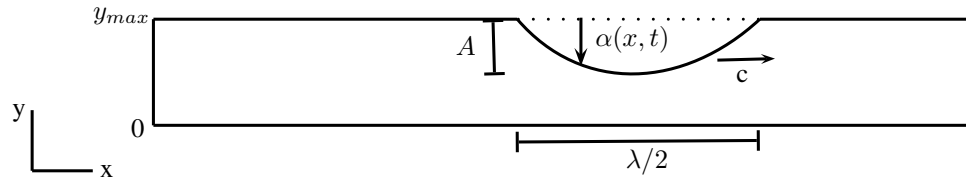


Figure 4.48: The sinusoidal wave.

Figures 4.49 and 4.50 show a zoomed view of the computational mesh for the parabolic and sinusoidal wave simulation, respectively.

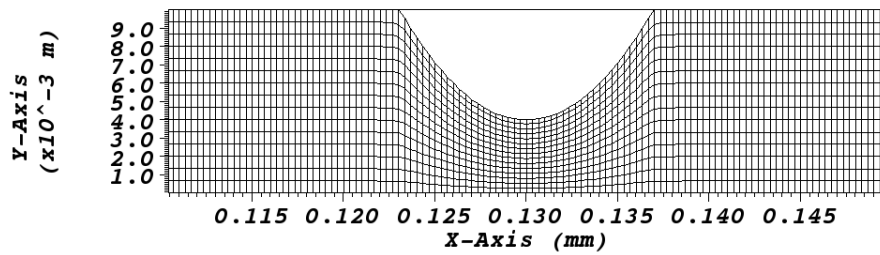


Figure 4.49: View of the computational mesh using parabolic wave, with $A = 41.6667$ mm, $x_0 = 0$ mm, $c = 5$ mm/s, and $y_0 = 4$ mm.

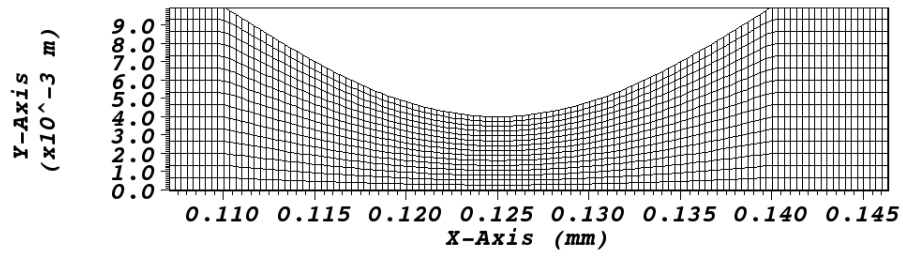


Figure 4.50: View of the computational mesh using sinusoidal wave, with $A = 6$ mm, $\lambda = 60$ mm, and $c = 8$ mm/s.

4.3 Conclusions

The most relevant conclusions about the simulation of peristaltic flow in the moving and fixed frame of reference are summarized below.

1. The simulations results in the moving and fixed frames show that the minimum velocities are attained in the region under the roller. Moreover, they agreed with the fact that the peristaltic wave tends to produce a rising pressure in the direction of the wave.
2. The transport efficiency is independent of the roller speed and decreases when the gap half-width increases.
3. The transport efficiency decreases with more shear-thinning behavior.
4. In comparison with the moving frame, the fixed frame approach more accurately represents reality and it is preferable to use.
5. The x-component of the velocity for the three-dimensional channel agreed with the ones of the two-dimensional channel.
6. In the three-dimensional channel, the pressure difference and maximum shear rate are ≈ 2.6 and ≈ 1.7 the ones of the corresponding case in the two dimensional channel, respectively.

7. The two-dimensional simulation sufficiently reflects the three-dimensional simulation and it is considerably faster.

Chapter 5

Summary and Future Work

Different types of flow in deformable geometries have been presented in this thesis. The flow of a non-Newtonian fluid through a collapsed tube has been simulated using a deformed tube geometry. These simulations were designed to reflect an experimental setup. The geometry of the deformed tube used in the simulations has been reconstructed from the experiments using computer tomography image analysis. The simulation results have been compared with experimental data. There is generally good agreement between simulation and experiment, especially given the rudimentary approximation to the geometry used in the simulation. In order to better assess the non-Newtonian behavior of this fluid the same flow has been simulated for a Newtonian fluid. Conclusions from the simulations in the collapsed tube are given in Section 3.4.

Simulations of peristaltic flow have also been performed. These simulations were designed

to reflect an experimental setup in which the peristaltic flow was induced by deforming a tube using rollers that moved along the tube wall. In the simulations, two frames of reference were used: the moving frame of reference (wave frame) where the computational domain is fixed and the coordinate system is moving with the roller speed, and the fixed frame of reference (laboratory frame) where the roller motion is represented by a deforming mesh. In either case, good agreement was found with experimental data. Simulations have been performed for Newtonian and non-Newtonian fluids, and a parameter study has been performed to determine the effect of shear-thinning behavior, roller speed, and gap width on the transport efficiency. Conclusions from the peristaltic flow simulations are given in Section 4.3.

Future Work

To gain a better understanding of the peristaltic motion, further work is still required.

Future simulation work includes the following:

- Modify the moving-mesh boundary condition to generate more general types of waves.
- Extend the two-dimensional numerical model to axisymmetric geometry.
- Generalize the deformation method by means of rollers to a three dimensional tube.

- Develop a fully three-dimensional model of traveling wave deformation in complex geometries such as the stomach.

References

- [1] H. Jasak and Z. Tuković, “Dynamic mesh handling in openfoam applied to fluid-structure interaction simulations,” in *Proceedings of the V European Conference on Computational Fluid Dynamics (ECCOMAS CFD 2010)(Lisbon, Portugal, 14-17 June 2010)*, JCF Pereira AS, Pereira JMC,(Eds.).(27), 2010.
- [2] M. Jaffrin and A. Shapiro, “Peristaltic pumping,” *Annual Review of Fluid Mechanics*, vol. 3, no. 1, pp. 13–37, 1971.
- [3] M. Heil, “Stokes flow in collapsible tubes: computation and experiment,” *Journal of Fluid Mechanics*, vol. 353, no. 1, pp. 285–312, 1997.
- [4] M. Heil, “Stokes flow in an elastic tube—a large-displacement fluid-structure interaction problem,” *International journal for numerical methods in fluids*, vol. 28, no. 2, pp. 243–265, 1998.
- [5] A. L. Hazel and M. Heil, “Steady finite-reynolds-number flows in three-dimensional collapsible tubes,” *Journal of Fluid Mechanics*, vol. 486, pp. 79–103, 2003.
- [6] J. B. Grotberg and O. E. Jensen, “Biofluid mechanics in flexible tubes,” *Annu. Rev. Fluid Mech.*, vol. 36, pp. 121–147, 2004.
- [7] T. W. Latham, *Fluid Motions in a Peristaltic Pump*. PhD thesis, Massachusetts Institute of Technology, 1966.
- [8] A. H. Shapiro, M. Y. Jaffrin, and S. L. Weinberg, “Peristaltic pumping with long wavelengths at low reynolds number,” *J. Fluid Mech*, vol. 37, no. 4, pp. 799–825, 1969.
- [9] C. Kleinstreuer, *Biofluid dynamics: Principles and selected applications*. CRC, 2006.
- [10] C. Barton and S. Raynor, “Peristaltic flow in tubes,” *Bull Math Biophys*, vol. 30, pp. 663–680, 1968.
- [11] K. Raju and R. Devanathan, “Peristaltic motion of a non-newtonian fluid,” *Rheologica Acta*, vol. 11, no. 2, pp. 170–178, 1972.

- [12] L. Srivastava and V. Srivastava, "Peristaltic transport of a non-newtonian fluid: applications to the vas deferens and small intestine," *Annals of biomedical engineering*, vol. 13, no. 2, pp. 137–153, 1985.
- [13] S. Nadeem, N. S. Akbar, M. Malik, and C. Lee, "Peristaltic flow of a jeffrey-six constant fluid in a uniform inclined tube," *International Journal for Numerical Methods in Fluids*, vol. 69, no. 9, pp. 1550–1565, 2011.
- [14] S. Nadeem and N. S. Akbar, "Influence of heat transfer on a peristaltic transport of herschel–bulkley fluid in a non-uniform inclined tube," *Communications in Nonlinear Science and Numerical Simulation*, vol. 14, no. 12, pp. 4100–4113, 2009.
- [15] T. Hayat and N. Ali, "A mathematical description of peristaltic hydromagnetic flow in a tube," *Applied mathematics and computation*, vol. 188, no. 2, pp. 1491–1502, 2007.
- [16] E. W. Merrill, "Rheology of blood," *Physiol Rev*, vol. 49, no. 4, pp. 863–88, 1969.
- [17] OpenFOAM, "The open source CFD toolbox." OpenCFD Ltd., 2004-2013. <http://www.openfoam.org>.
- [18] K. Hutter and K. Jöhnk, *Continuum methods of physical modeling: continuum mechanics, dimensional analysis, turbulence*. Springer Verlag, 2004.
- [19] W. Ostwald, "About the rate function of the viscosity of dispersed systems," *Kolloid Z*, vol. 36, pp. 99–117, 1925.
- [20] A. De Waele, "Viscometry and plastometry," *Oil Color Chem Assoc J*, vol. 6, pp. 33–88, 1923.
- [21] R. B. Bird, R. C. Armstrong, and O. Hassager, *Dynamics of Polymeric Liquids*. New York: John Wiley and Son Inc, 2nd ed., 1987.
- [22] P. J. Carreau, *Rheological equations from molecular network theories*. University of Wisconsin–Madison, 1968.
- [23] K. Yasuda, *Investigation of the analogies between viscometric and linear viscoelastic properties of polystyrene fluids*. PhD thesis, Massachusetts Institute of Technology, 1979.
- [24] E. C. Bingham, *Fluidity and Plasticity*. New York: McGraw-Hill, 1922.
- [25] J. F. Steffe, *Rheological methods in food process engineering*. Freeman Press, 1996.
- [26] M. Reiner, "The deborah number," *Physics Today*, vol. 17, p. 62, 1964.
- [27] H. Jasak, *Error analysis and estimation in the Finite Volume method with applications to fluid flows*. PhD thesis, Imperial College, University of London, 1996.

- [28] S. Patankar, *Numerical heat transfer and fluid flow*. Taylor & Francis, 1980.
- [29] R. I. Issa, “Solution of the implicitly discretised fluid flow equations by operator-splitting,” *Journal of Computational Physics*, vol. 62, pp. 40–65, 1986.
- [30] M. Peric, *A finite volume method for the prediction of three-dimensional fluid flow in complex ducts*. PhD thesis, Imperial College London (University of London), 1985.
- [31] L. N. Trefethen and D. Bau III, *Numerical linear algebra*. No. 50, Society for Industrial and Applied Mathematics, 1997.
- [32] Y. Saad, *Iterative methods for sparse linear systems*. Society for Industrial and Applied Mathematics, 2003.
- [33] *OpenFOAM User Guide*, 2.1.0 ed., December 2011. foam.sourceforge.net/docs/Guides-a4/UserGuide.pdf.
- [34] B. N. Datta, *Numerical linear algebra and applications*, vol. 116. Society for Industrial and Applied Mathematics, 2010.
- [35] A. Krishnamoorthy and D. Menon, “Matrix inversion using cholesky decomposition,” *arXiv preprint arXiv:1111.4144*, 2011.
- [36] H. Jasak, A. Jemcov, and J. Maruszewski, “Preconditioned linear solvers for large eddy simulation,” in *CFD 2007 Conference, CFD Society of Canada*, 2007.
- [37] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst, *Templates for the solution of linear systems: building blocks for iterative methods*. No. 43, Society for Industrial and Applied Mathematics, 1987.
- [38] H. Jasak and Z. Tukovic, “Automatic mesh motion for the unstructured finite volume method,” *Transactions of FAMENA*, vol. 30, no. 2, pp. 1–20, 2006.
- [39] A. O. González, A. Vallier, and H. Nilsson, “Mesh motion alternatives in openfoam,” http://www.tfd.chalmers.se/~hani/kurser/OS_CFD_2009/AndreuOliverGonzalez/ProjectReport_Corrected.pdf.
- [40] C. Kassiotis, “Which strategy to move the mesh in the computational fluid dynamic code openfoam,” *Report École Normale Supérieure de Cachan*. Available online: <http://perso.crans.org/kassiotis/openfoam/movingmesh.pdf>, 2008.
- [41] P. Moradnia, “A tutorial on how to use dynamic mesh solver icodymfoam,” http://www.tfd.chalmers.se/~hani/kurser/OS_CFD_2007/PiroozMoradnia/OpenFOAM-rapport.pdf.

- [42] F. X. Tanner, A. A. Al-Hababbeh, K. A. Feigl, S. Nahar, S. A. Jeelani, W. R. Case, and E. J. Windhab, "Numerical and experimental investigation of a non-newtonian flow in a collapsed elastic tube," *Applied Rheology*, vol. 22, no. 6, p. 63910, 2012.
- [43] S. Nahar, S. Jeelani, and E. Windhab, "Influence of elastic tube deformation on flow behavior of a shear thinning fluid," *Chemical Engineering Science*, 2012.
- [44] M. Stranzinger, *Numerical and Experimental Investigations of Newtonian and Non-Newtonian Flow in Annular Gaps with Scraper Blades*. PhD thesis, Swiss Federal Institute of Technology (ETH), 1999.
- [45] S. Nahar, S. Jeelani, and E. Windhab, "Steady and unsteady flow characteristics of a shear thinning fluid through a collapsed elastic tube," in *7th International Symposium on Ultrasonic Doppler Methods for Fluid mechanics and Fluid Engineering*, Chalmers University of Technology, Gothenburg, Sweden, pp. 7–8, 2010.
- [46] S. V. Patankar and D. B. Spalding, "A calculation procedure for heat, mass and momentum transfer in three-dimensional parabolic flows," *International Journal of Heat and Mass Transfer*, vol. 15, no. 10, pp. 1787–1806, 1972.
- [47] S. Nahar, *Steady and Unsteady Flow Characteristics of non-Newtonian Fluids in Deformed Elastic Tubes*. PhD thesis, Swiss Federal Institute of Technology (ETH), 2012.
- [48] T. Lucchini, "Running openfoam tutorials." http://web.student.chalmers.se/groups/ofw5/Basic_Training/runningTutorialsLucchini.pdf.
- [49] Y. Takeda, "Velocity profile measurement by ultrasound doppler shift method," *International journal of heat and fluid flow*, vol. 7, no. 4, pp. 313–318, 1986.
- [50] M. Auvinen, J. Ala-Juusela, N. Pedersen, and T. Siikonen, "Time-accurate turbomachinery simulations with open-source cfd; flow analysis of a single-channel pump with openfoam," in *Proceedings of the V European Conference on Computational Fluid Dynamics, ECCOMAS CFD*, (Lisbon, Portugal), June 2010.
- [51] F. J. Blom, "Considerations on the spring analogy," *International Journal for Numerical Methods in Fluids*, vol. 32, no. 6, pp. 647–668, 2000.
- [52] G. Böhme and R. Friedrich, "Peristaltic flow of viscoelastic liquids," *Journal of Fluid Mechanics*, vol. 128, no. 1, pp. 109–122, 1983.
- [53] I. Demirdžić and M. Perić, "Space conservation law in finite volume calculations of fluid flow," *International journal for numerical methods in fluids*, vol. 8, no. 9, pp. 1037–1050, 1988.
- [54] J. H. Ferziger and M. Perić, *Computational methods for fluid dynamics*, vol. 3. Springer Berlin etc, 2001.

- [55] M. Hanin, “The flow through a channel due to transversely oscillating walls,” *Isr. J. Technol.*, vol. 6, pp. 67–71, 1968.
- [56] L. Hogben, *Handbook of linear algebra*. Chapman & Hall, 2007.
- [57] H. Jasak and H. Rusche, “Dynamic mesh handling in openfoam,” in *Proceeding of the 47th Aerospace Sciences Meeting Including The New Horizons Forum and Aerospace Exposition, Orlando, Florida*, 2009.
- [58] R. Kverneland, “Cfd-simulations of wave-windinteraction,” Master’s thesis, University of Stravanger, Norway, 2012.
- [59] R. Löhner and C. Yang, “Improved ale mesh velocities for moving bodies,” *Communications in numerical methods in engineering*, vol. 12, no. 10, pp. 599–608, 1998.
- [60] H. Nilsson, “Phd course in cfd with opensource software, 2012,” 2012.
- [61] A. M. Provost and W. Schwarz, “A theoretical study of viscous effects in peristaltic pumping,” *Journal of Fluid Mechanics*, vol. 279, no. 1, pp. 177–195, 1994.
- [62] C. Rhie and W. Chow
- [63] H. K. Versteeg and W. Malalasekera, *An Introduction to Computational Fluid Dynamics*. USA: Pearson Education Limited, 2nd ed., 2007.

Appendix A

Case Files: Elastic Tube Simulations

Initial and boundary conditions are essential in solving any CFD problem. The case is set up to start at time $t = 0$ or in the first iterate, so in OpenFOAM the initial field data is stored in a 0 sub-directory of the case.

<case>/0: File U

This file contains boundary and initial conditions for the velocity.

```
dimensions      [0 1 -1 0 0 0 0];
internalField   uniform (-0.05 0 0);
boundaryField
{
    inlet //channel inlet
    {
        type      fixedValue;
        value      uniform (-0.0541127 0 0);
                  //corresp. to Q = 17 ml/s
    }
    outlet
    {
        type      zeroGradient;
    }
    top
```

```

    {
        type          fixedValue;
        value          uniform (0 0 0);
    }
    bottom
    {
        type          fixedValue;
        value          uniform (0 0 0);
    }
    front
    {
        type          fixedValue;
        value          uniform (0 0 0);
    }
    back
    {
        type          fixedValue;
        value          uniform (0 0 0);
    }
}

```

<case>/0: File p

This file contains boundary and initial conditions for the kinematic pressure.

```

dimensions          [0 2 -2 0 0 0 0];
internalField        uniform 0;
boundaryField
{
    inlet
    {
        type          zeroGradient;
    }
    outlet
    {
        type          fixedValue;
        value          uniform 0;
    }
    top

```

```

    {
        type                zeroGradient;
    }
    bottom
    {
        type                zeroGradient;
    }
    front
    {
        type                zeroGradient;
    }
    back
    {
        type                zeroGradient;
    }
}

```

<case>/constant: File transportProperties

This file contains the material properties for the fluid.

```

transportModel  BirdCarreau; // Newtonian;

nu              nu [ 0 2 -1 0 0 0 0 ] 0.1452e-03;

BirdCarreauCoeffs
{
    nu0          nu0 [ 0 2 -1 0 0 0 0 ] 0.1452e-03;
    nuInf        nuInf [ 0 2 -1 0 0 0 0 ] 0.0;
    k            k [ 0 0 1 0 0 0 0 ] 0.02673;
    n            n [ 0 0 0 0 0 0 0 ] 0.7588;
}

```

<case>/constant: File RASProperties

This file contains the choice of RAS (Reynolds-averaged stress) modelling


```

RASModel      laminar;
turbulence    off;
printCoeffs   off;

```

<case>/constant/polyMesh: File blockMeshDict

This file contains input for the generation of the mesh.

```

convertToMeters 0.01;  // measurements are in cm
vertices
(
    (19      1.694444444      0.620370370) //vertex #0
    (19      1.150000000      0.800000000) //vertex #1
    (19      1.210000000      2.700000000) //vertex #2
    (19      1.950000000      2.420000000) //vertex #3
    (17      1.768500000      0.590000000) //vertex #4
    (17      1.180000000      0.700000000) //vertex #5
    (17      1.194000000      2.700000000) //vertex #6
    (17      2.061000000      2.450000000) //vertex #7
    (15      1.825000000      0.590000000) //vertex #8
    (15      1.164000000      0.600000000) //vertex #9
    (15      1.074074074      2.518518519) //vertex #10
    (15      1.796296296      2.675925926) //vertex #11
    (13      1.731481481      0.518518519) //vertex #12
    (13      1.250000000      0.700000000) //vertex #13
    (13      1.326000000      2.660000000) //vertex #14
    (13      1.828000000      2.550000000) //vertex #15
    (11      1.768518519      0.481481481) //vertex #16
    (11      1.220000000      0.590000000) //vertex #17
    (11      1.320000000      2.740000000) //vertex #18
    (11      1.950000000      2.520000000) //vertex #19
    (9       1.768518519      0.490740741) //vertex #20
    (9       1.222222222      0.722222222) //vertex #21
    (9       1.550000000      2.910000000) //vertex #22
    (9       2.050000000      2.500000000) //vertex #23
    (7       1.777777778      0.462962963) //vertex #24
    (7       1.240000000      0.490000000) //vertex #25
    (7       1.300000000      2.510000000) //vertex #26

```

```

(7      1.981481481      2.675925926)//vertex #27
(5      1.731481481      0.537037037)//vertex #28
(5      1.305555556      0.685185185)//vertex #29
(5      1.480000000      2.800000000)//vertex #30
(5      1.960000000      2.540000000)//vertex #31
(3      1.705000000      0.542000000)//vertex #32
(3      1.255000000      0.768000000)//vertex #33
(3      1.666666667      2.990740741)//vertex #34
(3      2.110000000      2.361111111)//vertex #35
(1      1.750000000      0.740000000)//vertex #36
(1      1.200000000      1.055555556)//vertex #37
(1      1.470000000      2.842592593)//vertex #38
(1      2.166666667      2.509259259)//vertex #39
);

```

blocks

```

(
  hex ( 5      1      0      4      6      2      3      7) (27 27 27)
    simpleGrading (1 1 1)      // block #0
  hex ( 9      5      4      8     10      6      7     11) (27 27 27)
    simpleGrading (1 1 1)      // block #1
  hex (13      9      8     12     14     10     11     15) (27 27 27)
    simpleGrading (1 1 1)      // block #2
  hex (17     13     12     16     18     14     15     19) (27 27 27)
    simpleGrading (1 1 1)      // block #3
  hex (21     17     16     20     22     18     19     23) (27 27 27)
    simpleGrading (1 1 1)      // block #4
  hex (25     21     20     24     26     22     23     27) (27 27 27)
    simpleGrading (1 1 1)      // block #5
  hex (29     25     24     28     30     26     27     31) (27 27 27)
    simpleGrading (1 1 1)      // block #6
  hex (33     29     28     32     34     30     31     35) (27 27 27)
    simpleGrading (1 1 1)      // block #7
  hex (37     33     32     36     38     34     35     39) (27 27 27)
    simpleGrading (1 1 1)      // block #8
);

```

edges

```

(
  spline 0 1 (
    (19      1.550000000      0.600000000)

```

```

(19      1.356000000      0.651000000)
(19      1.349000000      0.648000000)
)
spline 1 2 (
(19      1.009259259      0.944444444)
(19      0.920000000      1.115000000)
(19      0.854000000      1.250000000)
(19      0.800000000      1.420000000)
(19      0.753000000      1.564814815)
(19      0.750000000      1.700000000)
(19      0.740740741      1.851851852)
(19      0.760000000      2.050000000)
(19      0.824074074      2.240740741)
(19      0.950000000      2.420000000)
(19      1.111111111      2.601851852)
)
spline 2 3 (
(19      1.379629630      2.777777778)
(19      1.600000000      2.730000000)
(19      1.768518519      2.620370370)
)
spline 3 0 (
(19      2.083333333      2.240740741)
(19      2.142000000      2.044000000)
(19      2.175925926      1.870370370)
(19      2.200000000      1.750000000)
(19      2.219000000      1.518518519)
(19      2.200000000      1.350000000)
(19      2.153000000      1.138888889)
(19      2.100000000      1.000000000)
(19      1.981481481      0.814814815)
(19      1.880000000      0.700000000)
)
spline 4 5 (
(17      1.600000000      0.538000000)
(17      1.450000000      0.550000000)
(17      1.324074074      0.592592593)
)
spline 5 6 (
(17      1.037037037      0.870370370)
(17      0.946000000      1.020000000)
(17      0.898148148      1.157407407)

```

```

(17      0.873000000      1.320000000)
(17      0.867000000      1.490740741)
(17      0.861000000      1.700000000)
(17      0.873000000      1.851851852)
(17      0.884000000      2.050000000)
(17      0.907407407      2.222222222)
(17      0.935000000      2.380000000)
(17      1.009259259      2.527777778)
)
spline 6 7 (
(17      1.447000000      2.842592593)
(17      1.700000000      2.820000000)
(17      1.916666667      2.656000000)
)

spline 7 4 (
(17      2.111000000      2.268518519)
(17      2.120000000      2.100000000)
(17      2.140000000      1.861111111)
(17      2.151000000      1.750000000)
(17      2.157000000      1.544000000)
(17      2.140000000      1.400000000)
(17      2.083333333      1.138888889)
(17      2.050000000      1.050000000)
(17      1.962962963      0.796296296)
(17      1.900000000      0.700000000)
)

spline 12 13 (
(13      1.600000000      0.480000000)
(13      1.462900000      0.500000000)
(13      1.360000000      0.580000000)
)

spline 13 14 (
(13      1.186000000      0.787037037)
(13      1.120000000      0.980000000)
(13      1.092592593      1.129629630)
(13      1.120000000      1.320000000)
(13      1.148148148      1.472222222)
(13      1.164000000      1.640000000)
(13      1.157407407      1.805555556)
(13      1.154000000      2.000000000)
(13      1.137000000      2.185185185)

```

```

(13      1.200000000      2.406000000)
(13      1.242000000      2.509259259)
)
spline 14 15 (
(13      1.462962963      2.824074074)
(13      1.650000000      2.840000000)
(13      1.781481481      2.733333333)
)
spline 15 12 (
(13      1.852000000      2.400000000)
(13      1.905000000      2.220000000)
(13      1.948000000      2.006000000)
(13      1.958000000      1.769000000)
(13      1.941000000      1.564814815)
(13      1.938000000      1.450000000)
(13      1.925925926      1.185185185)
(13      1.950000000      1.020000000)
(13      1.912000000      0.814814815)
(13      1.850000000      0.700000000)
)
spline 8 9 (
(15      1.610000000      0.410000000)
(15      1.377550000      0.331500000)
(15      1.223000000      0.449000000)
)
spline 9 10 (
(15      1.150000000      0.700000000)
(15      1.092592593      0.842592593)
(15      1.080000000      1.020000000)
(15      1.046296296      1.194444444)
(15      1.050000000      1.350000000)
(15      1.037037037      1.518518519)
(15      1.024000000      1.660000000)
(15      1.027777778      1.824074074)
(15      1.000000000      2.000000000)
(15      0.990740741      2.185185185)
(15      1.010000000      2.350000000)
)
spline 10 11(
(15      1.250000000      2.780000000)
(15      1.435185185      2.861111111)
(15      1.640000000      2.810000000)

```

```

)
spline 11 8 (
(15    1.900000000    2.520000000)
(15    1.990740741    2.333333333)
(15    2.010000000    2.160000000)
(15    1.990740741    1.907407407)
(15    2.000000000    1.750000000)
(15    2.009259259    1.546296296)
(15    2.000000000    1.420000000)
(15    1.994000000    1.185185185)
(15    1.980000000    1.020000000)
(15    1.953703704    0.814814815)
)
spline 16 17 (
(11    1.650000000    0.424000000)
(11    1.527700000    0.415000000)
(11    1.370370370    0.444444444)
)
spline 17 18 (
(11    1.166666667    0.768518519)
(11    1.150000000    0.950000000)
(11    1.175925926    1.101851852)
(11    1.220000000    1.250000000)
(11    1.250000000    1.435185185)
(11    1.240000000    1.560000000)
(11    1.222222222    1.750000000)
(11    1.200000000    1.940000000)
(11    1.175925926    2.111111111)
(11    1.150000000    2.290000000)
(11    1.203703704    2.472222222)
)
spline 18 19 (
(11    1.527777778    2.879629630)
(11    1.748000000    2.780000000)
(11    1.907407407    2.638888889)
)
spline 19 16 (
(11    1.981481481    2.333333333)
(11    1.900000000    2.150000000)
(11    1.861111111    1.888888889)
(11    1.842000000    1.750000000)
(11    1.831000000    1.583333333)

```

```

(11      1.870000000    1.360000000)
(11      1.898148148    1.185185185)
(11      1.880000000    1.020000000)
(11      1.916666667    0.833333333)
(11      1.880000000    0.650000000)
)
spline 20 21 (
(9      1.600000000    0.400000000)
(9      1.386900000    0.399074074)
(9      1.250000000    0.520000000)
)
spline 21 22 (
(9      1.220000000    0.900000000)
(9      1.259259259    1.074074074)
(9      1.350000000    1.250000000)
(9      1.351851852    1.398148148)
(9      1.350000000    1.530000000)
(9      1.296296296    1.703703704)
(9      1.300000000    1.900000000)
(9      1.259259259    2.083333333)
(9      1.250000000    2.250000000)
(9      1.277777778    2.416666667)
(9      1.350000000    2.650000000)
)
spline 22 23 (
(9      1.814814815    2.962962963)
(9      1.940000000    2.850000000)
(9      2.046296296    2.629629630)
)
spline 23 20 (
(9      2.018518519    2.305555556)
(9      1.990000000    2.150000000)
(9      1.916666667    1.935185185)
(9      1.840000000    1.780000000)
(9      1.814814815    1.583333333)
(9      1.800000000    1.400000000)
(9      1.828000000    1.194444444)
(9      1.840000000    1.060000000)
(9      1.842592593    0.879629630)
(9      1.840000000    0.700000000)
)
spline 24 25 (

```

```

(7      1.638800000    0.350000000)
(7      1.500000000    0.323000000)
(7      1.355000000    0.351851852)
)
spline 25 26 (
(7      1.222222222    0.666666667)
(7      1.200000000    0.820000000)
(7      1.259259259    0.981481481)
(7      1.290000000    1.140000000)
(7      1.342592593    1.324074074)
(7      1.350000000    1.480000000)
(7      1.361111111    1.648148148)
(7      1.300000000    1.810000000)
(7      1.259259259    1.981481481)
(7      1.250000000    2.160000000)
(7      1.240740741    2.342592593)
)
spline 26 27 (
(7      1.430000000    2.800000000)
(7      1.638888889    2.870370370)
(7      1.800000000    2.840000000)
)
spline 27 24 (
(7      2.020000000    2.460000000)
(7      2.018518519    2.275000000)
(7      1.960000000    2.140000000)
(7      1.870370370    1.962962963)
(7      1.830000000    1.850000000)
(7      1.731481481    1.611111111)
(7      1.768518519    1.240740741)
(7      1.810000000    1.050000000)
(7      1.898148148    0.861111111)
(7      1.850000000    0.650000000)
)
spline 28 29 (
(5      1.648100000    0.420000000)
(5      1.462962963    0.416666667)
(5      1.340000000    0.520000000)
)
spline 29 30 (
(5      1.300000000    0.800000000)
(5      1.305555556    0.981481481)

```



```

(5      1.340000000      1.100000000)
(5      1.361111111      1.296296296)
(5      1.360000000      1.450000000)
(5      1.398148148      1.629629630)
(5      1.360000000      1.800000000)
(5      1.324074074      2.000000000)
(5      1.300000000      2.180000000)
(5      1.277777778      2.333333333)
(5      1.380000000      2.600000000)
)
spline 30 31 (
(5      1.648148148      2.833333333)
(5      1.800000000      2.800000000)
(5      1.907407407      2.712962963)
)
spline 31 28 (
(5      1.935185185      2.370370370)
(5      1.880000000      2.200000000)
(5      1.787037037      2.046296296)
(5      1.720000000      1.850000000)
(5      1.666666667      1.629629630)
(5      1.660000000      1.450000000)
(5      1.731481481      1.277777778)
(5      1.750000000      1.110000000)
(5      1.787037037      0.962962963)
(5      1.770000000      0.730000000)
)
spline 32 33 (
(3      1.580000000      0.489000000)
(3      1.443000000      0.529000000)
(3      1.342592593      0.594000000)
)
spline 33 34 (
(3      1.192000000      0.972222222)
(3      1.154000000      1.205888888)
(3      1.153000000      1.450000000)
(3      1.162000000      1.601851852)
(3      1.175000000      1.769000000)
(3      1.185185185      1.935185185)
(3      1.162000000      2.123000000)
(3      1.157407407      2.305555556)
(3      1.206000000      2.523000000)

```

```

(3      1.305555556      2.740740741)
(3      1.460000000      2.898000000)
)
spline 34 35 (
(3      1.915000000      2.881000000)
(3      2.074074074      2.731481481)
(3      2.120000000      2.558000000)
)
spline 35 32 (
(3      2.097000000      2.166000000)
(3      2.064814815      1.962962963)
(3      2.026000000      1.791000000)
(3      1.990740741      1.592592593)
(3      1.999000000      1.441000000)
(3      2.027777778      1.194444444)
(3      2.008000000      1.075000000)
(3      1.962962963      0.953703704)
(3      1.897000000      0.789000000)
(3      1.796296296      0.611111111)
)
spline 36 37 (
(1      1.600000000      0.743000000)
(1      1.444444444      0.796296296)
(1      1.313000000      0.936000000)
)
spline 37 38 (
(1      1.113000000      1.182000000)
(1      1.020000000      1.333333333)
(1      0.973000000      1.529000000)
(1      0.940000000      1.703703704)
(1      0.927000000      1.850000000)
(1      0.925925926      1.990740741)
(1      0.947000000      2.198000000)
(1      0.981481481      2.351851852)
(1      1.080000000      2.508000000)
(1      1.190000000      2.646000000)
(1      1.313000000      2.765000000)
)
spline 38 39 (
(1      1.653000000      2.866000000)
(1      1.916666667      2.824074074)
(1      2.080000000      2.679000000)

```

```

)
  spline 39 36 (
    (1      2.220000000    2.289000000)
    (1      2.226000000    2.064814815)
    (1      2.240000000    1.845000000)
    (1      2.226000000    1.657407407)
    (1      2.207000000    1.460000000)
    (1      2.166000000    1.250000000)
    (1      2.127000000    1.118000000)
    (1      2.043000000    0.966000000)
    (1      1.913000000    0.861000000)
  )
);

```

```

boundary
(
  inlet
  {type patch;
    faces
    (
      ( 1    0    3    2)
    );
  }

```

```

outlet
{type patch;
  faces
  (
    (37  36  39  38)
  );
}

```

```

top
{type wall;
  faces
  (
    ( 0    4    7    3)
    ( 4    8   11    7)
    ( 8   12   15   11)
    (12   16   19   15)
    (16   20   23   19)
    (20   24   27   23)
  )

```

```

        (24  28  31  27)
        (28  32  35  31)
        (32  36  39  35)
    );
}

bottom
{type wall;
  faces
  (
    ( 1   2   6   5)
    ( 5   6  10   9)
    ( 9  10  14  13)
    (13  14  18  17)
    (17  18  22  21)
    (21  22  26  25)
    (25  26  30  29)
    (29  30  34  33)
    (33  34  38  37)
  );
}

front
{type wall;
  faces
  (
    ( 1   5   4   0)
    ( 5   9   8   4)
    ( 9  13  12   8)
    (13  17  16  12)
    (17  21  20  16)
    (21  25  24  20)
    (25  29  28  24)
    (29  33  32  28)
    (33  37  36  32)
  );
}

back
{type wall;
  faces
  (

```

```

( 2    3    7    6)
( 6    7   11   10)
(10   11   15   14)
(14   15   19   18)
(18   19   23   22)
(22   23   27   26)
(26   27   31   30)
(30   31   35   34)
(34   35   39   38)
);
}
); //end boundary

```

```

mergePatchPairs
(
);

```

<case>/system: File controlDict

This dictionary sets input parameters essential for the creation of the database.

```

application      simpleFoam;
startFrom         startTime;
startTime         0;
stopAt           endTime;
endTime          50000;
deltaT           1;
writeControl      timeStep;
writeInterval     1000;
purgeWrite       0;
writeFormat       ascii;
writePrecision    6;
writeCompression  uncompressed;
timeFormat        general;
timePrecision     6;
runTimeModifiable yes;

```

<case>/system: File fvSolution

This file controls the equation solvers, tolerances and algorithms.

```
solvers
{
    p
    {
        solver            GAMG;
        // Geometric-algebraic multi-grid solver
        tolerance          1e-06;
        // Stop if the residual is below this solver
        // tolerance
        relTol             0.00;
        // Or, stop if the ratio of curent residual
        // to initial residual falls below this solver
        // relTol
        smoother           GaussSeidel;
        nPreSweeps          0;
        nPostSweeps         2;
        cacheAgglomeration true;
        nCellsInCoarsestLevel 10;
        agglomerator         faceAreaPair;
        mergeLevels         1;
    }

    U
    {
        solver            smoothSolver;
        smoother           GaussSeidel;
        nSweeps            2;
        tolerance          1e-08;
        relTol             0.00;
    }
}

SIMPLE
{
    nNonOrthogonalCorrectors 0;
    pRefCell                0;
```

```

    pRefValue      0;

    residualControl
    {
        p          1e-3;
        U          1e-4;
    }
}

relaxationFactors
{
    default        0;
    p              0.3;
    U              0.7;
    nuTilda        0.7;
}

```

<case>/system: File fvSchemes

This file sets the numerical schemes for terms., such as derivatives in equations.

```

ddtSchemes
{
    default        steadyState;
}
gradSchemes
{
    default        Gauss linear;
    grad(p)        Gauss linear;
    grad(U)        Gauss linear;
}
divSchemes
{
    default        none;
    div(phi,U)     Gauss linearUpwind grad(U);
    div(phi,nuTilda)
    Gauss linearUpwind grad(nuTilda);
    div((nuEff*dev(T(grad(U)))) Gauss linear;
}

```

```

laplacianSchemes
{
    default          none;
    laplacian(nuEff,U) Gauss linear corrected;
    laplacian((1|A(U)),p) Gauss linear corrected;
    laplacian(DnuTildaEff,nuTilda)
    Gauss linear corrected;
    laplacian(1,p) Gauss linear corrected;
}
interpolationSchemes
{
    default          linear;
    interpolate(U) linear;
}
snGradSchemes
{
    default          corrected;
}
fluxRequired
{
    default          no;
    p                ;
}

```


Appendix B

Case Files: Peristaltic Motion Simulations

B.1 Moving Frame

The following are the files for the standard case

<case>/0: File U

```
dimensions      [0 1 -1 0 0 0 0];
internalField   uniform (0 0 0);
boundaryField
{
    upperWall
    {
        type      fixedValue;
        value      uniform (-0.005 0 0);
    }
    roller
    {
        type      fixedValue;
        value      uniform (0 0 0);
    }
}
```

```

leftBoundary
{
    type                zeroGradient;
}
rightBoundary
{
    type                zeroGradient;
}
centerLine
{
    type                symmetryPlane;
}
frontAndBack
{
    type                empty;
}
}

```

<case>/0: File p

```

dimensions      [0 2 -2 0 0 0 0];
internalField    uniform 0;
boundaryField
{
    upperWall
    {
        type                zeroGradient;
    }
    roller
    {
        type                zeroGradient;
    }
    rightBoundary
    {
        type                totalPressure;
        p0                  uniform 0;
        gamma                1;
        value                uniform 0;
    }
    leftBoundary

```

```

    {
        type            totalPressure;
        p0              uniform 0;
        gamma           1;
        value            uniform 0;
    }
    centerLine
    {
        type            symmetryPlane;
    }
    frontAndBack
    {
        type            empty;
    }
}

```

<case>/constant: File transportProperties

```

transportModel  Newtonian;
nu              nu [ 0 2 -1 0 0 0 0 ] 0.1452e-03;

```

<case>/constant: File RASProperties

```

RASModel        laminar;\ \ uses no turbulence models
turbulence       off;
printCoeffs      off;

```

<case>/constant/polyMesh: File blockMeshDict

```
convertToMeters 1.0e-3;
vertices
(
    //back side
    (      0.00      10.00      -0.10)
// Vertex A0 = 0
    (      0.00      0.00      -0.10)
// Vertex A2 = 1
    (     33.00      0.00      -0.10)
// Vertex B2 = 2
    (     33.00     10.00      -0.10)
// Vertex B0 = 3
    (     57.00     10.00      -0.10)
// Vertex C0 = 4
    (     57.00      0.00      -0.10)
// Vertex C2 = 5
    (     90.00      0.00      -0.10)
// Vertex D2 = 6
    (     90.00     10.00      -0.10)
// Vertex D0 = 7

    //front side
    (      0.00     10.00      0.10)
// Vertex A1 = 8
    (      0.00      0.00      0.10)
// Vertex A3 = 9
    (     33.00      0.00      0.10)
// Vertex B3 = 10
    (     33.00     10.00      0.10)
// Vertex B1 = 11
    (     57.00     10.00      0.10)
// Vertex C1 = 12
    (     57.00      0.00      0.10)
// Vertex C3 = 13
    (     90.00      0.00      0.10)
// Vertex D3 = 14
    (     90.00     10.00      0.10)
// Vertex D1 = 15
```

```

);

blocks
(
    hex(1 2 3 0 9 10 11 8)(120 24 1)
    simpleGrading (1 1 1) //blk 0
    hex(2 5 4 3 10 13 12 11)(180 24 1)
    simpleGrading (1 1 1) //blk 1
    hex(5 6 7 4 13 14 15 12)(120 24 1)
    simpleGrading (1 1 1) //blk 2
);
edges
(
    arc 3 4 (45 4.00 -0.10)
    arc 11 12 (45 4.00 0.10)
);
boundary
(
    leftBoundary
    { type patch;
      faces
      (
          (0 1 9 8) //on blk0
      );
    }
    rightBoundary
    { type patch;
      faces
      (
          (15 14 6 7) //on blk2
      );
    }
    centerLine
    { type symmetryPlane;
      faces
      (
          (1 2 10 9) //on blk0
          (2 5 13 10) //on blk1
          (5 6 14 13) //on blk2
      );
    }
    upperWall

```

```

    { type wall;
      faces
      (
        (0 3 11 8) //on blk0
        (4 7 15 12) //on blk1
      );
    }
    roller
    { type wall;
      faces
      (
        (3 4 12 11) //on blk1
      );
    }

    frontAndBack //for 2D case
    { type empty;
      faces
      (
        ( 3 2 1 0) //on blk0 back
        ( 8 9 10 11) //on blk0 front
        ( 4 5 2 3) //on blk1 back
        (11 10 13 12) //on blk1 front
        ( 7 6 5 4) //On blk2 back
        (12 13 14 15) //on blk2 front
      );
    }
  );

  mergePatchPairs
  (
  );

```

<case>/system: File controlDict

```

application      simpleFoam;
startFrom        startTime;
startTime        0;
stopAt           endTime;
endTime          20000;
deltaT           1;

```

```

writeControl    timeStep;
writeInterval   500;
purgeWrite      0;
writeFormat     ascii;
writePrecision  6;
writeCompression off;
timeFormat      general;
timePrecision   6;
runTimeModifiable true;

```

<case>/system: File fvSolution

```

solvers
{
    p
    {
        solver          PCG;
        preconditioner   DIC;
        tolerance        1e-06;
        relTol           0.01;
    }
    U
    {
        solver          PBiCG;
        preconditioner   DILU;
        tolerance        1e-05;
        relTol           0.1;
    }
    // The following parameters are not used for laminar
    k
    {
        solver          PBiCG;
        preconditioner   DILU;
        tolerance        1e-05;
        relTol           0.1;
    }
    epsilon
    {
        solver          PBiCG;
        preconditioner   DILU;
    }
}

```



```

        tolerance      1e-05;
        relTol         0.1;
    }
    R
    {
        solver          PBiCG;
        preconditioner   DILU;
        tolerance        1e-05;
        relTol           0.1;
    }
    nuTilda
    {
        solver          PBiCG;
        preconditioner   DILU;
        tolerance        1e-05;
        relTol           0.1;
    }
}
SIMPLE
{
    nNonOrthogonalCorrectors 0;
    residualControl
    {
        p                1e-6;
        U                1e-7;
        "(k|epsilon|omega)" 1e-3;
    }
}
relaxationFactors
{
    p                0.3;
    U                0.7;
    k                0.7;
    epsilon          0.7;
    R                0.7;
    nuTilda          0.7;
}

```

<case>/system: File fvSchemes

```
ddtSchemes
{
    default          steadyState;
}
gradSchemes
{
    default          Gauss linear;
    grad(p)          Gauss linear;
    grad(U)          Gauss linear;
}
divSchemes
{
    default          none;
    div(phi,U)       Gauss upwind;
    div(phi,k)       Gauss upwind;
    div(phi,epsilon) Gauss upwind;
    div(phi,R)       Gauss upwind;
    div(R)           Gauss linear;
    div(phi,nuTilda) Gauss upwind;
    div((nuEff*dev(T(grad(U)))) Gauss linear;
}
laplacianSchemes
{
    default          none;
    laplacian(nuEff,U) Gauss linear corrected;
    laplacian((1|A(U)),p) Gauss linear corrected;
    laplacian(DkEff,k) Gauss linear corrected;
    laplacian(DepsilonEff,epsilon)
    Gauss linear corrected;
    laplacian(DREff,R) Gauss linear corrected;
    laplacian(DnuTildaEff,nuTilda)
    Gauss linear corrected;
}
interpolationSchemes
{
    default          linear;
    interpolate(U)   linear;
}
snGradSchemes
```

```

{
    default          corrected;
}
fluxRequired
{
    default          no;
    p                ;
}

```

B.2 Fixed Frame

<case>/0: File U

```

dimensions          [0 1 -1 0 0 0 0];
internalField        uniform (0 0 0);
boundaryField
{
    leftBoundary
    {
        type          zeroGradient;
    }
    rightBoundary
    {
type                zeroGradient;
    }
    centerLine
    {
        type          symmetryPlane;
    }
    upperWall
    {
        type          movingWallNormalVel;
        value          uniform (0 0 0);
    }
    frontAndBack
    {
        type          empty;
    }
}

```

```
}
```

<case>/0: File p

```
dimensions      [0 2 -2 0 0 0 0];
internalField    uniform 0;
boundaryField
{
    leftBoundary
    {
        type      totalPressure;
        p0         uniform 0;
        gamma      1;
        value       uniform 0;
    }
    rightBoundary
    {
        type      totalPressure;
        p0         uniform 0;
        gamma      1;
        value       uniform 0;
    }
    centerLine
    {
        type      symmetryPlane;
    }
    upperWall
    {
        type      zeroGradient;
    }
    frontAndBack
    {
        type      empty;
    }
}
```

<case>/0: File pointMotionU

This file contains some input values that control the movement of the upper wall.

```
dimensions      [0 1 -1 0 0 0 0];
internalField    uniform (0 0 0);
boundaryField
{
    leftBoundary
    {
        type      zeroGradient;
    }
    rightBoundary
    {
        type      zeroGradient;
    }
    centerLine
    {
        type      symmetryPlane;
    }
    upperWall
    {
        type      dynPerCircle;
        circleRadius    0.01500;
        xCompInitialCenter    0.0;
        speed            0.00500;
        yCompFinalCenter    0.01900;
        value            uniform (0 0 0);
    }
    frontAndBack
    {
        type      empty;
    }
}
```

<case>/constant: File dynamicMeshDict

```
dynamicFvMesh    dynamicMotionSolverFvMesh;
```

```

motionSolverLibs    ("libfvMotionSolvers.dylib");
solver              velocityLaplacian;
diffusivity          directional (1 1200 0);

```

<case>/constant: File turbulenceProperties

```

simulationType    laminar;

```

<case>/constant/polyMesh: File blockMeshDict

```

convertToMeters 1.0e-03;
vertices
(
    ( 0.00 0.00 -0.10) // vertex#0
    (180.00 0.00 -0.10) // vertex#1
    (180.00 10.00 -0.10) // vertex#2
    ( 0.00 10.00 -0.10) // vertex#3
    ( 0.00 0.00 0.10) // vertex#4
    (180.00 0.00 0.10) // vertex#5
    (180.00 10.00 0.10) // vertex#6
    ( 0.00 10.00 0.10) // vertex#7
);
blocks
(
    hex (0 1 2 3 4 5 6 7) (810 23 1)
    simpleGrading (1 1 1)
// block #0
);
edges
(
);
boundary
(
    leftBoundary
    {
        type patch;
        faces

```

```

(
    (0 3 7 4)
);
}
rightBoundary
{
    type patch;
    faces
    (
        (5 6 2 1)
    );
}
centerLine
{
    type symmetryPlane;
    faces
    (
        (1 0 4 5)
    );
}
upperWall
{
    type wall;
    faces
    (
        (2 3 7 6)
    );
}
frontAndBack
{
    type empty;
    faces
    (
        (0 1 2 3)
        (5 4 7 6)
    );
}
);
mergePatchPairs
(
);

```

<case>/system: File controlDict

```
application      transientSimpleDyMFoam;
startFrom        startTime;
startTime        0;
stopAt           endTime;
endTime          48;
deltaT           0.00005;
writeControl      adjustableRunTime;
writeInterval     2;
purgeWrite       0;
writeFormat       ascii;
writePrecision    6;
writeCompression off;
timeFormat        general;
timePrecision     6;
runTimeModifiable true;
adjustTimeStep    yes;
maxCo             0.5;
maxDeltaT 1; // Maximum deltaT in seconds
libs
(
    "dynPerCircle.dylib"
    "movingWallNormalVel.dylib"
);
```

<case>/system: File fvSolution

```
solvers
{
    pcorr
    {
        solver          GAMG;
        tolerance        1e-7;
        relTol           0;
        smoother         GaussSeidel;
        nPreSweeps        0;
        nPostSweeps       2;
```



```

        cacheAgglomeration off;
        agglomerator      faceAreaPair;
        nCellsInCoarsestLevel 20;
        mergeLevels      1;
// controls the speed at which coarsening
// or refinement levels is performed.
        maxIter          100;
        minIter          1;
    }
    p
    {
        $pcorr;
        tolerance         1e-6;
        relTol            0;
    }
    pFinal
    {
        $p;
        tolerance         1e-7;
        relTol            0;
    }
    "(U|k|epsilon|omega|nuTilda) "
    {
        solver            smoothSolver;
        smoother          GaussSeidel;
        nSweeps           1;
        tolerance         1e-07;
        relTol            0;
        maxIter           100;
        minIter           1;
    };
    "(U|k|epsilon|omega|nuTilda)Final"
    {
        solver            smoothSolver;
        smoother          GaussSeidel;
        nSweeps           2;
        tolerance         1e-07;
        relTol            0;
        maxIter           100;
        minIter           1;
    }
    cellMotionU

```

```

        {
            solver          PCG;
            preconditioner   DIC;
            tolerance        1e-08;
            relTol           0;
        }
    }
PISO
{
    nCorrectors             2;
    nOuterCorrectors        20;
    nNonOrthogonalCorrectors 0;
    correctPhi              true;
}

```

<case>/system: File fvSchemes

```

ddtSchemes
{
    default          Euler;
}
gradSchemes
{
    default          Gauss linear;
    grad(p)          Gauss linear;
}
divSchemes
{
    default          none;
    div(phi,U)       Gauss linear;
    div((nuEff*dev(T(grad(U)))) Gauss linear;
}
laplacianSchemes
{
    default          none;
    laplacian(nu,U)  Gauss linear corrected;
    laplacian(rAU,pcorr) Gauss linear corrected;
    laplacian(rAU,p) Gauss linear corrected;
    laplacian(diffusivity,cellMotionU)
    Gauss linear uncorrected;
}

```

```

        laplacian(nuEff,U) Gauss linear uncorrected;
    }
interpolationSchemes
{
    default          linear;
    interpolate(HbyA) linear;
}
snGradSchemes
{
    default          corrected;
}
fluxRequired
{
    default          no;
    pcorr            ;
    p                 ;
}

```

Appendix C

OpenFOAM Codes

C.1 movingWallNormalVel boundary condition

movingWallNormalVelFvPatchVectorField.H

```
SourceFiles
    movingWallNormalVelFvPatchVectorField.C
/*-----*/

#ifndef movingWallNormalVelFvPatchVectorField_H
#define movingWallNormalVelFvPatchVectorField_H

#include "fvPatchFields.H"
#include "fixedValueFvPatchFields.H"

// * * * * *

namespace Foam
{

/*-----*\
    Class movingWallNormalVelFvPatch Declaration
\*-----*/
```

```

class movingWallNormalVelFvPatchVectorField:
    public fixedValueFvPatchVectorField
{
    // Private data

    //- Name of velocity field
    word UName_;

public:

    //- Runtime type information
    // aaalhaba110 08-15-12 (changed the type name
    // from movingWallVelocity to:
    TypeName("movingWallNormalVel");
    // aaalhaba111

    // Constructors

    //- Construct from patch and internal field
    movingWallNormalVelFvPatchVectorField
    (
        const fvPatch&,
        const DimensionedField<vector, volMesh>&
    );

    //- Construct from patch, internal field and
    // dictionary
    movingWallNormalVelFvPatchVectorField
    (
        const fvPatch&,
        const DimensionedField<vector, volMesh>&,
        const dictionary&
    );

    //- Construct by mapping given
    // movingWallNormalVelFvPatchVectorField
    // onto a new patch
    movingWallNormalVelFvPatchVectorField
    (
        const movingWallNormalVelFvPatchVectorField&,

```

```

const fvPatch&,
const DimensionedField<vector, volMesh>&,
const fvPatchFieldMapper&
    );

    //- Construct as copy
    movingWallNormalVelFvPatchVectorField
    (
const movingWallNormalVelFvPatchVectorField&
    );

    //- Construct and return a clone
    virtual tmp<fvPatchVectorField> clone() const
    {
        return tmp<fvPatchVectorField>
        (
new movingWallNormalVelFvPatchVectorField(*this)
        );
    }

    //- Construct as copy setting internal field
    //- reference
    movingWallNormalVelFvPatchVectorField
    (
const movingWallNormalVelFvPatchVectorField&,
const DimensionedField<vector, volMesh>&
    );

    //- Construct and return a clone setting
    //- internal field reference
    virtual tmp<fvPatchVectorField> clone
    (
const DimensionedField<vector, volMesh>& iF
    ) const
    {
        return tmp<fvPatchVectorField>
        (
new
movingWallNormalVelFvPatchVectorField(*this,iF)
        );
    }

```

```

        // Member functions

    //- Update the coefficients associated with the
    // patch field
        virtual void updateCoeffs();

        //- Write
        virtual void write(Ostream&) const;
};

// * * * * *
} // End namespace Foam

// * * * * *

#endif

// *****

```

movingWallNormalVelFvPatchVectorField.C

```

// aaalhaba110 08-15-2012
#include
"movingWallNormalVelFvPatchVectorField.H"
// aaalhaba111
// Note: replacing all movingWallVelocity to the
// new class movingWallNormalVel
#include "addToRunTimeSelectionTable.H"
#include "volFields.H"
#include "surfaceFields.H"
#include "fvcMeshPhi.H"

// ***** Constructors *****

Foam::movingWallNormalVelFvPatchVectorField::

```

```

movingWallNormalVelFvPatchVectorField
(
    const fvPatch& p,
    const DimensionedField<vector, volMesh>& iF
)
:
    fixedValueFvPatchVectorField(p, iF),
    UName_("U")
{}

Foam::movingWallNormalVelFvPatchVectorField::
movingWallNormalVelFvPatchVectorField
(
    const movingWallNormalVelFvPatchVectorField& ptf,
    const fvPatch& p,
    const DimensionedField<vector, volMesh>& iF,
    const fvPatchFieldMapper& mapper
)
:
    fixedValueFvPatchVectorField(ptf, p, iF, mapper),
    UName_(ptf.UName_)
{}

Foam::movingWallNormalVelFvPatchVectorField::
movingWallNormalVelFvPatchVectorField
(
    const fvPatch& p,
    const DimensionedField<vector, volMesh>& iF,
    const dictionary& dict
)
:
    fixedValueFvPatchVectorField(p, iF),
    UName_(dict.lookupOrDefault<word>("U", "U"))
{
    fvPatchVectorField::operator=
        (vectorField("value", dict, p.size()));
}

Foam::movingWallNormalVelFvPatchVectorField::

```



```

movingWallNormalVelFvPatchVectorField
(
    const
movingWallNormalVelFvPatchVectorField& mwvpvf
)
:
    fixedValueFvPatchVectorField(mwvpvf),
    UName_(mwvpvf.UName_)
{}

Foam::movingWallNormalVelFvPatchVectorField::
movingWallNormalVelFvPatchVectorField
(
    const
movingWallNormalVelFvPatchVectorField& mwvpvf,
    const DimensionedField<vector, volMesh>& iF
)
:
    fixedValueFvPatchVectorField(mwvpvf, iF),
    UName_(mwvpvf.UName_)
{}

// ***** Member Functions ***** //

void Foam::
movingWallNormalVelFvPatchVectorField::
updateCoeffs()
{
    if (updated())
    {
        return;
    }

    const fvPatch& p = patch();
    const polyPatch& pp = p.patch();
    const
fvMesh& mesh=dimensionedInternalField().mesh();
    const pointField& oldPoints =
    mesh.oldPoints();

```

```

    vectorField oldFc(pp.size());

    forAll(oldFc, i)
    {
        oldFc[i] = pp[i].centre(oldPoints);
    }

    const vectorField
Up((pp.faceCentres() - oldFc)/
    mesh.time().deltaTValue());

    const volVectorField&
U = db().lookupObject<volVectorField>(UName_);
    scalarField phip
    (
        p.patchField
<surfaceScalarField, scalar>(fvc::meshPhi(U))
    );

    const vectorField n(p.nf());
    const scalarField& magSf = p.magSf();
    tmp<scalarField> Un = phip/(magSf + VSMALL);

/* aaalhaba010 08-15-12
(commented out the old operator)
    vectorField::
operator=(Up + n*(Un - (n & Up)));
aaalhaba011 */

// aaalhaba120 08-15-12 (project the
// movingWallVelocity
// onto normal direction)
// Note: (a & b) is for the dot product between
// vectors
// a and b
    vectorField::
operator=((n & (Up + n*(Un - (n & Up))))*n);
// aaalhaba121
    fixedValueFvPatchVectorField::updateCoeffs();
}

```

```

void Foam::
movingWallNormalVelFvPatchVectorField::
write(Ostream& os) const
{
    fvPatchVectorField::write(os);
    writeEntryIfDifferent<word>(os,"U", "U", UName_);
    writeEntry("value", os);
}

// *****

namespace Foam
{
    makePatchTypeField
    (
        fvPatchVectorField,
        movingWallNormalVelFvPatchVectorField
    );
}

// *****

```

C.2 dynPerCircle boundary condition

dynPerCircle.H

```

Class
    Foam::dynPerCircle

Description
    Foam::dynPerCircle

SourceFiles dynPerCircle.C

\*-----*/

```

```

#ifndef dynPerCircle_H
#define dynPerCircle_H

#include "fixedValuePointPatchField.H"

// * * * * *

namespace Foam
{
    /*-----*\
    Class dynPerCircle
    Declaration
    \*-----*/
    class
    dynPerCircle
    :
    public fixedValuePointPatchField<vector>
    {
        // Private data
        scalar circleRadius_;
        scalar xCompInitialCenter_;
        scalar speed_;
        scalar yCompFinalCenter_;
        pointField p0_;

    public:

        //- Runtime type information
        TypeName("dynPerCircle");

        // Constructors

        //- Construct from patch and internal field
        dynPerCircle
        (
            const pointPatch&,
            const DimensionedField<vector, pointMesh>&
        );

        //- Construct from patch, internal field and

```

```

    // dictionary
    dynPerCircle
    (
        const pointPatch&,
        const DimensionedField<vector, pointMesh>&,
        const dictionary&
    );

    //- Construct by mapping given
    // patchField<vector> onto
    // a new patch
    dynPerCircle
    (
        const dynPerCircle&,
        const pointPatch&,
        const DimensionedField<vector, pointMesh>&,
        const pointPatchFieldMapper&
    );

    //- Construct and return a clone
    virtual
    autoPtr<pointPatchField<vector> > clone() const
    {
        return autoPtr<pointPatchField<vector> >
            (
                new dynPerCircle
                (
                    *this
                )
            );
    }

    //- Construct as copy setting internal field
    // reference
    dynPerCircle
    (
        const dynPerCircle&,
        const DimensionedField<vector, pointMesh>&
    );

    //- Construct and return a clone setting
    // internal field

```

```

        // reference
virtual autoPtr<pointPatchField<vector> > clone
(
    const DimensionedField<vector, pointMesh>& iF
) const
{
    return autoPtr<pointPatchField<vector> >
        (
            new dynPerCircle
                (
                    *this,
                    iF
                )
        );
}
// Member functions

// Mapping functions

//- Map (and resize as needed) from self given
// a mapping
// object
virtual void autoMap
(
    const pointPatchFieldMapper&
);

//- Reverse map the given pointPatchField onto
// this
// pointPatchField
virtual void rmap
(
    const pointPatchField<vector>&,
    const labelList&
);

// Evaluation functions

//- Update the coefficients associated with the
// patch
// field

```

```

        virtual void updateCoeffs();

        //- Write
        virtual void write(Ostream&) const;
};

// * * * * *

} // End namespace Foam

// * * * * *

#endif

// *****

```

dynPerCircle.C

```

#include "dynPerCircle.H"
#include "pointPatchFields.H"
#include "addToRunTimeSelectionTable.H"
#include "Time.H"
#include "polyMesh.H"
#include "mathematicalConstants.H"

// * * * * *

namespace Foam
{
// * * * * * Constructors * * * * *

dynPerCircle::
dynPerCircle
(
    const pointPatch& p,
    const DimensionedField<vector, pointMesh>& iF
)

```

```

:
    fixedValuePointPatchField<vector>(p, iF),
    circleRadius_(0.0),
    xCompInitialCenter_(0.0),
    speed_(0.0),
    yCompFinalCenter_(0.0),
    p0_(p.localPoints())
{}

dynPerCircle::
dynPerCircle
(
    const pointPatch& p,
    const DimensionedField<vector, pointMesh>& iF,
    const dictionary& dict
)
:
    fixedValuePointPatchField<vector>(p, iF, dict),
    circleRadius_(readScalar
        (dict.lookup("circleRadius"))),
    xCompInitialCenter_(readScalar
        (dict.lookup("xCompInitialCenter"))),
    speed_(readScalar(dict.lookup("speed"))),
    yCompFinalCenter_(readScalar
        (dict.lookup("yCompFinalCenter")))
{
    if (!dict.found("value"))
    {
        updateCoeffs();
    }

    if (dict.found("p0"))
    {
        p0_ = vectorField("p0", dict, p.size());
    }
    else
    {
        p0_ = p.localPoints();
    }
}

dynPerCircle::

```



```

dynPerCircle
(
    const dynPerCircle& ptf,
    const pointPatch& p,
    const DimensionedField<vector, pointMesh>& iF,
    const pointPatchFieldMapper& mapper
)
:
fixedValuePointPatchField<vector>
(ptf, p, iF, mapper),
    circleRadius_(ptf.circleRadius_),
    xCompInitialCenter_(ptf.xCompInitialCenter_),
    speed_(ptf.speed_),
    yCompFinalCenter_(ptf.yCompFinalCenter_),
    p0_(ptf.p0_)
{}

dynPerCircle::
dynPerCircle
(
    const dynPerCircle& ptf,
    const DimensionedField<vector, pointMesh>& iF
)
:
    fixedValuePointPatchField<vector>(ptf, iF),
    circleRadius_(ptf.circleRadius_),
    xCompInitialCenter_(ptf.xCompInitialCenter_),
    speed_(ptf.speed_),
    yCompFinalCenter_(ptf.yCompFinalCenter_),
    p0_(ptf.p0_)
{}

// * * * * * Member Functions * * * * * //

void dynPerCircle::autoMap
(
    const pointPatchFieldMapper& m
)
{

```

```

        fixedValuePointPatchField<vector>::autoMap(m);

        p0_.autoMap(m);
    }
    void dynPerCircle::rmap
    (
        const pointPatchField<vector>& ptf,
        const labelList& addr
    )
    {
        const
        dynPerCircle& aOVptf =
        refCast
        <const dynPerCircle>(ptf);

        fixedValuePointPatchField<vector>::
        rmap(aOVptf, addr);

        p0_.rmap(aOVptf.p0_, addr);
    }
    void
    dynPerCircle::updateCoeffs()
    {
        if (this->updated())
        {
            return;
        }

        const polyMesh& mesh =
        this->dimensionedInternalField().mesh();
        const Time& t = mesh.time();
        const pointPatch& p = this->patch();

        scalar yMax
        (
            max(p0_.component(vector::Y))
        );

        scalar yCompInitialCenter
        (
            yMax+circleRadius_
        );

```

```

scalar varCenterXcomp
(
    xCompInitialCenter_+speed_*t.value()
);

scalar varCenterYcomp = 0.0;
if
(
    (t.value() > 0)
    &&
    (t.value() < 1)
)
{
    varCenterYcomp =
    yCompInitialCenter-
    t.value()*
    (yCompInitialCenter-yCompFinalCenter_);
}
else
{
    varCenterYcomp = yCompFinalCenter_;
}

scalar yDiff
(
    yMax-varCenterYcomp
);

scalar yDiffSquared
(
    yDiff*yDiff
);

scalar yRadicand
(
    circleRadius_*circleRadius_-yDiffSquared
);

scalar yRadicandSqrt
(
    sqrt(yRadicand)
);

```

```

    scalar lowerBound
    (
        varCenterXcomp-yRadicandSqrt
    );

    scalar upperBound
    (
        varCenterXcomp+yRadicandSqrt
    );

    scalar b = -1.0;

pointField
yCenterShift (p0_.size(), point (0.0, 0.0, 0.0));

pointField
velocity (p0_.size(), point (0.0, 0.0, 0.0));

    forAll (p0_, pointI)
    {

        scalar xRadicandSqrt = 0.0;

        if
        (
            (p0_.component (vector::X) () [pointI] > lowerBound)
            &&
            (p0_.component (vector::X) () [pointI] < upperBound)
        )
        { //major

            scalar xDiff
            (
                p0_.component (vector::X) () [pointI] - varCenterXcomp
            );

            scalar xDiffSquared
            (
                xDiff*xDiff
            );

```

```

        scalar xRadicand
        (
            circleRadius_*circleRadius_-xDiffSquared
        );

        xRadicandSqrt = sqrt(xRadicand);

        yCenterShift[pointI]=
            point(0.0,varCenterYcomp-
1*p0_.component(vector::Y)()[pointI],0.0);
        }
        else
        {
            xRadicandSqrt = 0.0;
            yCenterShift[pointI]=point(0.0,0.0,0.0);
        }
        velocity[pointI]
        =yCenterShift[pointI]+
b*point(0.0,xRadicandSqrt,0.0);

    }

    pointField::operator=
    (
        (p0_
        +velocity
        -p.localPoints()
        )/t.deltaT().value()

    );

    fixedValuePointPatchField<vector>::
updateCoeffs();
}

void
dynPerCircle::
write
(
    Ostream& os
) const

```

```

{
    pointPatchField<vector>::write(os);
    os.writeKeyword("circleRadius")
    << circleRadius_ << token::END_STATEMENT << nl;
    os.writeKeyword("xCompInitialCenter")
    << xCompInitialCenter_ << token::END_STATEMENT << nl;
    os.writeKeyword("speed")
    << speed_ << token::END_STATEMENT << nl;
    os.writeKeyword("yCompFinalCenter")
    << yCompFinalCenter_ << token::END_STATEMENT << nl;
    p0_.writeEntry("p0", os);
    writeEntry("value", os);
}

// * * * * *

makePointPatchTypeField
(
    pointPatchVectorField,
    dynPerCircle
);

// * * * * *

} // End namespace Foam

// *****

```

C.3 Parabolic wave

0/pointMotionU

```

FoamFile
{
    version      2.0;

```

```

        format      ascii;
        class       pointVectorField;
        object       pointMotionU;
    }
    // * * * * * * * * * * * * * * * * * * * * //

    dimensions      [0 1 -1 0 0 0 0];

    internalField    uniform (0 0 0);

    boundaryField
    {
        leftBoundary
        {
            type      zeroGradient;
        }

        rightBoundary
        {
            type      zeroGradient;
        }

        centerLine
        {
            type      symmetryPlane;
        }

        upperWall
        {
            type      dynPerParabolic;
            coeffA     41.66667;
            xCompInitVertex  0.000;
            speed       0.005;
            yCompFinVertex  0.004;
            value       uniform (0 0 0);
        }

        frontAndBack
        {
            type      empty;
        }
    }

```

```
}
```

dynPerParabolic.H

```
Class
    Foam::dynPerParabolic

Description
    Foam::dynPerParabolic

SourceFiles
    dynPerParabolic.C

/*-----*/

#ifndef dynPerParabolic_H
#define dynPerParabolic_H

#include "fixedValuePointPatchField.H"

// * * * * *

namespace Foam
{
/*-----*\
    Class dynPerParabolic Declaration
/*-----*/
class dynPerParabolic
:
    public fixedValuePointPatchField<vector>
{
    // Private data
    scalar xCompInitVertex_;
    scalar coeffA_;
    scalar speed_;
    scalar yCompFinVertex_;
    pointField p0_;
```



```

public:

    //- Runtime type information
    TypeName("dynPerParabolic");

    // Constructors

    //- Construct from patch and
    // internal field
    dynPerParabolic
    (
        const pointPatch&,
const DimensionedField<vector, pointMesh>&
    );

    //- Construct from patch, internal
    // field and dictionary
    dynPerParabolic
    (
        const pointPatch&,
const DimensionedField<vector, pointMesh>&,
        const dictionary&
    );

    //- Construct by mapping given
    // patchField<vector> onto a new patch
    dynPerParabolic
    (
        const dynPerParabolic&,
        const pointPatch&,
const DimensionedField<vector, pointMesh>&,
        const pointPatchFieldMapper&
    );

    //- Construct and return a clone
    virtual autoPtr<pointPatchField<vector> > <brk>
clone() const
    {
        return autoPtr<pointPatchField<vector> >
        (
            new dynPerParabolic

```

```

        (
            *this
        )
    );
}

//- Construct as copy setting internal
// field reference
dynPerParabolic
(
    const dynPerParabolic&,
const DimensionedField<vector, pointMesh>&
);

//- Construct and return a clone setting
// internal field reference
virtual autoPtr<pointPatchField<vector> > clone
(
const DimensionedField<vector, pointMesh>& iF
) const
{
    return autoPtr<pointPatchField<vector> >
        (
            new dynPerParabolic
            (
                *this,
                iF
            )
        );
}

// Member functions

// Mapping functions

//- Map (and resize as needed) from
// self given a mapping object
virtual void autoMap
(
    const pointPatchFieldMapper&
);

//- Reverse map the given

```

```

        // pointPatchField onto this
        // pointPatchField
        virtual void rmap
        (
            const pointPatchField<vector>&,
            const labelList&
        );

// Evaluation functions

        //- Update the coefficients associated
        // with the patch field
        virtual void updateCoeffs();

        //- Write
        virtual void write(Ostream&) const;
};

// * * * * *

} // End namespace Foam

// * * * * *

#endif

// ***** //
```

dynPerParabolic.C

```

#include "dynPerParabolic.H"
#include "pointPatchFields.H"
#include "addToRunTimeSelectionTable.H"
#include "Time.H"
#include "polyMesh.H"
#include "mathematicalConstants.H"
```

```

// * * * * *
namespace Foam
{
// * * * * * Constructors * * * * * //

dynPerParabolic::
dynPerParabolic
(
    const pointPatch& p,
    const DimensionedField<vector, pointMesh>& iF
)
:
    fixedValuePointPatchField<vector>(p, iF),
    xCompInitVertex_(0.0),
    coeffA_(0.0),
    speed_(0.0),
    yCompFinVertex_(0.0),
    p0_(p.localPoints())
{}

dynPerParabolic::
dynPerParabolic
(
    const pointPatch& p,
    const DimensionedField<vector, pointMesh>& iF,
    const dictionary& dict
)
:
    fixedValuePointPatchField<vector>(p, iF, dict),
    xCompInitVertex_(<<brk>>
(readScalar(dict.lookup("xCompInitVertex"))),
    coeffA_(readScalar(dict.lookup("coeffA"))),
    speed_(readScalar(dict.lookup("speed"))),
    yCompFinVertex_(<<brk>>
(readScalar(dict.lookup("yCompFinVertex"))))
{
    if (!dict.found("value"))
    {
        updateCoeffs();
    }
}

```

```

        if (dict.find("p0"))
        {
p0_ = vectorField("p0", dict , p.size());
        }
        else
        {
            p0_ = p.localPoints();
        }
    }
}

```

```

dynPerParabolic::
dynPerParabolic
(
    const dynPerParabolic& ptf,
    const pointPatch& p,
const DimensionedField<vector, pointMesh>& iF,
    const pointPatchFieldMapper& mapper
)
:
    fixedValuePointPatchField<vector> <<brk>>
    (ptf, p, iF, mapper),
    xCompInitVertex_(ptf.xCompInitVertex_),
    coeffA_(ptf.coeffA_),
    speed_(ptf.speed_),
    yCompFinVertex_(ptf.yCompFinVertex_),
    p0_(ptf.p0_)
{}

```

```

dynPerParabolic::
dynPerParabolic
(
    const dynPerParabolic& ptf,
const DimensionedField<vector, pointMesh>& iF
)
:
    fixedValuePointPatchField<vector>(ptf, iF),
    xCompInitVertex_(ptf.xCompInitVertex_),
    coeffA_(ptf.coeffA_),
    speed_(ptf.speed_),
    yCompFinVertex_(ptf.yCompFinVertex_),

```

```

        p0_(ptf.p0_)

    {}

// * * * * * Member Functions * * * * * //

void dynPerParabolic::autoMap
(
    const pointPatchFieldMapper& m
)
{
    fixedValuePointPatchField<vector>::autoMap(m);

    p0_.autoMap(m);
}

void dynPerParabolic::rmap
(
    const pointPatchField<vector>& ptf,
    const labelList& addr
)
{
    const dynPerParabolic& aOVptf =
        refCast<const dynPerParabolic>(ptf);

    fixedValuePointPatchField<vector>:: <<brk>>
    rmap(aOVptf, addr);

    p0_.rmap(aOVptf.p0_, addr);
}

void dynPerParabolic::updateCoeffs()
{
    if (this->updated())
    {
        return;
    }

    const polyMesh& mesh = this-> <<brk>>
    dimensionedInternalField().mesh();
    const Time& t = mesh.time();
    const pointPatch& p = this->patch();

```

```

    scalar yMax
    (
        max(p0_.component(vector::Y)())
    );

    scalar varVertexXcomp
    (
        xCompInitVertex_+speed_*t.value()
    );

    scalar yDiff
    (
        yMax-yCompFinVertex_
    );

    scalar ratio
    (
        yDiff/coeffA_
    );
    scalar sqrtRatio
    (
        sqrt(ratio)
    );

    scalar lowerBound
    (
        varVertexXcomp-sqrtRatio
    );

    scalar upperBound
    (
        varVertexXcomp+sqrtRatio
    );
    Info <<"lower"<<lowerBound;
    Info <<"upper"<<upperBound;
    pointField <<brk>>
    yVertexShift(p0_.size(),point(0.0,0.0,0.0));

    pointField <<brk>>
    velocity(p0_.size(),point(0.0,0.0,0.0));

    forAll(p0_,pointI)

```

```

        {

            scalar yParabola=yMax;
            if
            (
                (p0_.component(vector::X)()[pointI]>lowerBound)
                &&
                (p0_.component(vector::X)()[pointI]<upperBound)
            )
                { //major

                    scalar xDiff
                    (
                        p0_.component(vector::X)()[pointI]- <<brk>>
                        varVertexXcomp
                    );

                    scalar xDiffSqu
                    (
                        xDiff*xDiff
                    );

                    yParabola= coeffA_*xDiffSqu+yCompFinVertex_;
                }
            else
            {
                yParabola= yMax;
            }

            velocity[pointI]
            =point(0.0,yParabola,0.0)- <<brk>>
            point(0.0,yMax,0.0);

        }

        pointField::operator=
        (
            (p0_
             +velocity
             -p.localPoints())

```



```

        )/t.deltaT().value()

    );

    fixedValuePointPatchField<vector>:: <<brk>>
    updateCoeffs();
}

void dynPerParabolic::write
(
    Ostream& os
) const
{
    pointPatchField<vector>::write(os);
    os.writeKeyword("coeffA")
    << coeffA_ << token::END_STATEMENT << nl;
    os.writeKeyword("xCompInitVertex")
    << xCompInitVertex_ <<brk>>
    << token::END_STATEMENT << nl;
    os.writeKeyword("speed")
    << speed_ << token::END_STATEMENT << nl;
    os.writeKeyword("yCompFinVertex")
    << yCompFinVertex_ <<brk>>
    << token::END_STATEMENT << nl;
    p0_.writeEntry("p0", os);
    writeEntry("value", os);
}

// * * * * *

makePointPatchTypeField
(
    pointPatchVectorField,
    dynPerParabolic
);

// * * * * *

} // End namespace Foam

```

```
// ***** //
```

C.4 Sinusoidal wave

0/pointMotionU

```
FoamFile
{
    version      2.0;
    format       ascii;
    class        pointVectorField;
    object       pointMotionU;
}
// * * * * * * * * * * * * * * * * * * * * //

dimensions      [0 1 -1 0 0 0 0];

internalField   uniform (0 0 0);

boundaryField
{
    leftBoundary
    {
        type          zeroGradient;
    }

    rightBoundary
    {
        type          zeroGradient;
    }

    centerLine
    {
        type          symmetryPlane;
    }
}
```

```

    upperWall
    {
        type            dynPerSinusoidal;
        amplitude        0.006;
        waveLength        0.06;
        speed            0.005;
        value            uniform (0 0 0);
    }

    frontAndBack
    {
        type            empty;
    }

}

// ***** //

```

dynPerSinusoidal.H

Class

Foam::dynPerSinusoidal

Description

Foam::dynPerSinusoidal

SourceFiles dynPerSinusoidal.C

-----/

#ifndef dynPerSinusoidal_H

#define dynPerSinusoidal_H

#include "fixedValuePointPatchField.H"

// * * * * * //

namespace Foam

{

```

/*-----*\
Class dynPerSinusoidal Declaration
\*-----*/
class dynPerSinusoidal
:
    public fixedValuePointPatchField<vector>
{
    // Private data
    scalar amplitude_;
    scalar waveLength_;
    scalar speed_;
    pointField p0_;

public:

    //- Runtime type information
    TypeName("dynPerSinusoidal");

    // Constructors

    //- Construct from patch and internal field
    dynPerSinusoidal
    (
        const pointPatch&,
        const DimensionedField<vector, pointMesh>&
    );

    //- Construct from patch, internal field and
    // dictionary
    dynPerSinusoidal
    (
        const pointPatch&,
        const DimensionedField<vector, pointMesh>&,
        const dictionary&
    );

    //- Construct by mapping given patchField<vector>
    // onto
    // a new patch
    dynPerSinusoidal
    (

```

```

        const
dynPerSinusoidal&,
        const pointPatch&,
        const DimensionedField<vector, pointMesh>&,
        const pointPatchFieldMapper&
    );

    //- Construct and return a clone
    virtual
autoPtr<pointPatchField<vector> > clone() const
    {
        return autoPtr<pointPatchField<vector> >
            (
                new
dynPerSinusoidal
                (
                    *this
                )
            );
    }

    //- Construct as copy setting internal field
    //- reference
    dynPerSinusoidal
        (const
dynPerSinusoidal&,
        const DimensionedField<vector, pointMesh>&
        );

    //- Construct and return a clone setting internal
    //- field
    //- reference
    virtual autoPtr<pointPatchField<vector> > clone
    (
        const DimensionedField<vector, pointMesh>& iF
    ) const
    {
        return autoPtr<pointPatchField<vector> >
            (
                new dynPerSinusoidal
                (
                    *this,

```

```

        iF
    )
    );
}
// Member functions

    // Mapping functions
//- Map (and resize as needed) from self given a
// mapping
    // object
    virtual void autoMap
    (
        const pointPatchFieldMapper&
    );

//- Reverse map the given pointPatchField onto
// this
    // pointPatchField
    virtual void rmap
    (
        const pointPatchField<vector>&,
        const labelList&
    );

    // Evaluation functions

//- Update the coefficients associated with the
// patch field
    virtual void updateCoeffs();

    //- Write
    virtual void write(Ostream&) const;
};

// * * * * *
} // End namespace Foam

// * * * * *

```

// ***** //

```
#include "dynPerSinusoidal.H"
#include "pointPatchFields.H"
#include "addToRunTimeSelectionTable.H"
#include "Time.H"
#include "polyMesh.H"
#include "mathematicalConstants.H"
```

// * * * * * * * * * * * * * * * * * * //

```
// * * * * * Constructors * * * * *
```

```
dynPerSinusoidal::dynPerSinusoidal
(
    const pointPatch& p,
    const DimensionedField<vector, pointMesh>& iF,
    const dictionary& dict
)
```

```

:
    fixedValuePointPatchField<vector>(p, iF, dict),
    amplitude_(readScalar
(dict.lookup("amplitude"))),
    waveLength_
(readScalar(dict.lookup("waveLength"))),
    speed_(readScalar
(dict.lookup("speed")))
{
    if (!dict.found("value"))
    {
        updateCoeffs();
    }

    if (dict.found("p0"))
    {
        p0_ = vectorField("p0", dict , p.size());
    }
    else
    {
        p0_ = p.localPoints();
    }
}

dynPerSinusoidal::dynPerSinusoidal
(
    const dynPerSinusoidal& ptf,
    const pointPatch& p,
    const DimensionedField<vector, pointMesh>& iF,
    const pointPatchFieldMapper& mapper
)
:
fixedValuePointPatchField<vector>(ptf, p, iF, mapper),
    amplitude_(ptf.amplitude_),
    waveLength_(ptf.waveLength_),
    speed_(ptf.speed_),
    p0_(ptf.p0_)
{}

dynPerSinusoidal::dynPerSinusoidal
(

```



```

const
dynPerSinusoidal& ptf,
    const DimensionedField<vector, pointMesh>& iF
)
:
    fixedValuePointPatchField<vector>(ptf, iF),
    amplitude_(ptf.amplitude_),
    waveLength_(ptf.waveLength_),
    speed_(ptf.speed_),
    p0_(ptf.p0_)

{}

// * * * * * Member Functions * * * * //

void dynPerSinusoidal::autoMap
(
    const pointPatchFieldMapper& m
)
{
    fixedValuePointPatchField<vector>::autoMap(m);

    p0_.autoMap(m);
}

void dynPerSinusoidal::rmap
(
    const pointPatchField<vector>& ptf,
    const labelList& addr
)
{
    {
        // The following is a long line, so will break
        // into three lines
        const dynPerSinusoidal& aOVptf = //<brk>
        refCast<const dynPerSinusoidalPoint //<brk>
        PatchVectorField>(ptf);

        fixedValuePointPatchField<vector>::rmap(aOVptf, addr);

        p0_.rmap(aOVptf.p0_, addr);
    }

    void dynPerSinusoidal::updateCoeffs()

```

```

{
    if (this->updated())
    {
        return;
    }

    const polyMesh& mesh =
this->dimensionedInternalField().mesh();
    const Time& t = mesh.time();
    const pointPatch& p = this->patch();

    scalar yMax
        (
            max(p0_.component(vector::Y)())
        );

    scalar waveNumber
        (2*constant::mathematical::pi/waveLength_
        );

    scalar lowerBound
        (
            0.25*waveLength_+speed_*t.value()
        );

    scalar upperBound
        (
            0.75*waveLength_+speed_*t.value()
        );

    pointField yShift(p0_.size(),point(0.0,0.0,0.0));

    pointField
velocity(p0_.size(),point(0.0,0.0,0.0));

    forAll(p0_,pointI)
    {
        scalar ySinusoidal = 0.0;
        if
        (
            (p0_.component(vector::X)()[pointI]>lowerBound)
            &&

```

```

(p0_.component(vector::X)()[pointI]<upperBound)
    )
    { //major

        scalar xDifference
        (
            p0_.component(vector::X)()[pointI]-
            speed_*t.value()
        );

        scalar angle
        (
            waveNumber*xDifference
        );

        ySinusoidal
        = amplitude_*cos(angle);
        yShift[pointI]
        =point(0.0,0.0,0.0);
    }
    else
    {
        yShift[pointI]=
        point(0.0,-1*p0_.component(vector::Y)()[pointI]+
        yMax,0.0);
    }

    velocity[pointI]=
    yShift[pointI]+point(0.0,ySinusoidal,0.0);

}

pointField::operator=
(
    (p0_
    +velocity
    -p.localPoints()
    )/t.deltaT().value()

);

```

```

fixedValuePointPatchField<vector>::updateCoeffs();
}

void dynPerSinusoidal::write
(
    Ostream& os
) const
{
    pointPatchField<vector>::write(os);
    os.writeKeyword("amplitude")
    << amplitude_ << token::END_STATEMENT << nl;
    os.writeKeyword("waveLength")
    << waveLength_ << token::END_STATEMENT << nl;
    os.writeKeyword("speed")
    << speed_ << token::END_STATEMENT << nl;
    p0_.writeEntry("p0", os);
    writeEntry("value", os);
}

// * * * * *

makePointPatchTypeField
(
    pointPatchVectorField,
    dynPerSinusoidal
);

// * * * * *

} // End namespace Foam

// *****

```

C.5 transientSimpleDyMFoam

checkTotalVolume.H

```
scalar newTotalVolume =
    sum(mesh.cellVolumes());

Info<< "Volume: new = "
<< newTotalVolume << " old = " << totalVolume
    << " change = "
<< Foam::mag(newTotalVolume - totalVolume)<<endl;

    totalVolume = newTotalVolume;
```

correctPhi.H

```
{
wordList pcorrTypes(p.boundaryField().types());
for (label i=0; i<p.boundaryField().size(); i++)
{
    if (p.boundaryField()[i].fixesValue())
    {
        pcorrTypes[i] =
fixedValueFvPatchScalarField::typeName;
    }
}

volScalarField pcorr
(
    IOobject
    (
        "pcorr",
        runTime.timeName(),
        mesh,
        IOobject::NO_READ,
        IOobject::NO_WRITE
    )
);
```

```

        ),
        mesh,
        dimensionedScalar("pcorr", p.dimensions(), 0.0),
        pcorrTypes
    );

#    include "continuityErrs.H"

    // Flux predictor
    phi = (fvc::interpolate(U) & mesh.Sf());
    rAU == runTime.deltaT();

    for(int nonOrth=0; nonOrth<=nNonOrthCorr; nonOrth++)
    {
        fvScalarMatrix pcorrEqn
        (
            fvm::laplacian(rAU, pcorr) == fvc::div(phi)
        );

        pcorrEqn.setReference(pRefCell, pRefValue);
        pcorrEqn.solve();

        if (nonOrth == nNonOrthCorr)
        {
            phi -= pcorrEqn.flux();
        }
    }
    // Fluxes are corrected to absolute velocity and
    // further corrected
    // later.  HJ, 6/Feb/2009
}

```

createFields.H

```

    Info<< "Reading field p\n" << endl;
    volScalarField p
    (
        IOobject
        (
            "p",

```

```

        runTime.timeName(),
        mesh,
        IOobject::MUST_READ,
        IOobject::AUTO_WRITE
    ),
    mesh
);

Info<< "Reading field U\n" << endl;
volVectorField U
(
    IOobject
    (
        "U",
        runTime.timeName(),
        mesh,
        IOobject::MUST_READ,
        IOobject::AUTO_WRITE
    ),
    mesh
);

#   include "createPhi.H"

label pRefCell = 0;
scalar pRefValue = 0.0;
setRefCell
(p, mesh.solutionDict().subDict("PISO"), //<brk>
 pRefCell, pRefValue);

scalar totalVolume = sum(mesh.V()).value();

volScalarField rAU
(
    IOobject
    (
        "rAU",
        runTime.timeName(),
        mesh
    ),

```

```

        mesh,
        runTime.deltaT(),
        zeroGradientFvPatchScalarField::typeName
    );

    singlePhaseTransportModel
    laminarTransport(U, phi);

    autoPtr<incompressible::RASModel> turbulence
    (
        incompressible::RASModel::
        New(U, phi, laminarTransport)
    );

```

readControls.H

```

#   include "readTimeControls.H"
#   include "readPISOControls.H"

    bool correctPhi = false;
    if (piso.found("correctPhi"))
    {
        correctPhi = Switch(piso.lookup("correctPhi"));
    }

    bool checkMeshCourantNo = false;
    if (piso.found("checkMeshCourantNo"))
    {
        checkMeshCourantNo =
        Switch(piso.lookup("checkMeshCourantNo"));
    }

```

transientSimpleDyMFoam.C

Application

transientSimpleDyMFoam

Description

Transient solver for incompressible, turbulent
flow of Newtonian
fluids with dynamic mesh. Solver implements a
SIMPLE-based
algorithm in time-stepping mode.

Author

Hrvoje Jasak, Wikki Ltd. All rights reserved.

Modification

Evaluation of turbulence model moved inside
the
SIMPLE loop.
- Mikko Auvinen, Aalto University

-----/

```
#include "fvCFD.H"
// The following is a long line, so will break
// into two
#include //<brk>
"incompressible/singlePhaseTransportModel/
//<brk>
singlePhaseTransportModel.H"
#include "incompressible/RASModel/RASModel.H"
#include "dynamicFvMesh.H"
```

```
// * * * * * * * * * * * * * * * * * * * * //
```

```
int main(int argc, char *argv[])
{
#   include "setRootCase.H"
#   include "createTime.H"
#   include "createDynamicFvMesh.H"
#   include "initContinuityErrs.H"
#   include "createFields.H"
```

```
// * * * * * * * * * * * * * * * * * * * * //
```

```

Info<< "\nStarting time loop\n" << endl;

while (runTime.run())
{
#       include "readControls.H"
#       include "checkTotalVolume.H"
#       include "CourantNo.H"

        // Make the fluxes absolute
        fvc::makeAbsolute(phi, U);

#       include "setDeltaT.H"

        runTime++;

Info<< "Time = "<<runTime.timeName()<< nl << endl;

        bool meshChanged = mesh.update();

#       include "volContinuity.H"

        if (correctPhi && meshChanged)
        {
// Fluxes will be corrected to absolute velocity
// HJ, 6/Feb/2009
#       include "correctPhi.H"
        }

        // Make the fluxes relative to the mesh motion
        fvc::makeRelative(phi, U);

        if (checkMeshCourantNo)
        {
#       include "meshCourantNo.H"
        }

        // --- SIMPLE loop

        for (int ocorr = 0; ocorr < nOuterCorr; ocorr++)
        {
// #       include "CourantNo.H"    -- mikko

```

```

#             include "UEqn.H"

            rAU = 1.0/UEqn.A();

            U = rAU*UEqn.H();
            phi = (fvc::interpolate(U) & mesh.Sf());
                //+ fvc::ddtPhiCorr(rAU, U, phi);

            adjustPhi(phi, U, p);

            p.storePrevIter();

        for
        (int nonOrth=0; nonOrth<=nNonOrthCorr; nonOrth++)
        {
            fvScalarMatrix pEqn
            (
                fvm::laplacian(rAU, p) == fvc::div(phi)
            );

            pEqn.setReference(pRefCell, pRefValue);

        if
        (ocorr == nOuterCorr - 1 && nonOrth == nNonOrthCorr)
        {
            pEqn.solve(mesh.solver(p.name() + "Final"));
        }
        else
        {
            pEqn.solve(mesh.solver(p.name()));

            if (nonOrth == nNonOrthCorr)
            {
                phi -= pEqn.flux();
            }
        }

#             include "continuityErrs.H"

//Explicitly relax pressure for momentum corrector
            p.relax();

```

```

        // Make the fluxes relative to the mesh motion
        fvc::makeRelative(phi, U);

        U -= rAU*fvc::grad(p);
        U.correctBoundaryConditions();

// The turbulence model evaluation is necessary
// within
        // the SIMPLE loop. -- mikko
        turbulence->correct();

    }

    runTime.write();

Info<< "ExecutionTime = "
<< runTime.elapsedCpuTime() << " s"
<< "   ClockTime = "
<< runTime.elapsedClockTime() << " s"
        << nl << endl;
    }

    Info<< "End\n" << endl;

    return(0);
}

// *****

```

UEqn.H

```

fvVectorMatrix UEqn
(
    fvm::ddt(U)
    + fvm::div(phi, U)
    + turbulence->divDevReff(U)
);

```

```
UEqn.relax();

// Solve the momentum equation
solve(UEqn == -fvc::grad(p));
```

Make/files File

```
transientSimpleDyMFoam.C

EXE = $(FOAM_USER_APPBIN)/transientSimpleDyMFoam
```

Make/options File

```
EXE_INC = \
    -I$(LIB_SRC)/dynamicFvMesh/lnInclude \
    -I$(LIB_SRC)/dynamicMesh/lnInclude \
    -I$(LIB_SRC)/meshTools/lnInclude \
    -I$(LIB_SRC)/finiteVolume/lnInclude \
    -I$(LIB_SRC)/turbulenceModels/RAS \
    -I$(LIB_SRC)/transportModels

EXE_LIBS = \
    -ldynamicFvMesh \
    -ldynamicMesh \
    -lengine \
    -lmeshTools \
    -lincompressibleRASModels \
    -lincompressibleTransportModels \
    -lfiniteVolume \
    -llduSolvers
```

C.6 shearRate

```
Application
    shearRate
Description
    For each time: calculate the shear rate.
\*-----*/

#include "fvCFD.H"

// * * * * *

int main(int argc, char *argv[])
{
    timeSelector::addOptions();

    # include "setRootCase.H"
    # include "createTime.H"

    instantList timeDirs =
        timeSelector::select0(runTime, args);

    # include "createMesh.H"

    forAll(timeDirs, timeI)
    {
        runTime.setTime(timeDirs[timeI], timeI);

        Info<< "Time = " << runTime.timeName() << endl;

        IOobject Uheader
        (
            "U",
            runTime.timeName(),
            mesh,
            IOobject::MUST_READ
        );

        // Check U exists
        if (Uheader.headerOk())
```

```

{
    mesh.readUpdate();

    Info<< "    Reading U" << endl;
    volVectorField U(Uheader, mesh);

    Info<< "    Calculating shearRate" << endl;
    if (U.dimensions() ==
        dimensionSet(0, 1, -1, 0, 0))
    {
        volScalarField shearRate
        (
            IOobject
            (
                "shearRate",
                runTime.timeName(),
                mesh,
                IOobject::NO_READ
            ),
            sqrt(0.5*(2*symm(fvc::grad(U))&&
                (2*symm(fvc::grad(U))))
            );
        shearRate.write();
    }
    else
    {
        Info<< "    No U" << endl;
    }

    Info<< endl;
}
return 0;
}

// *****

```