Genetic Algorithm Based Optimization of Baffle Positions in a Forward Osmosis Draw Channel

James Koch
Washington University in St Louis

Follow this and additional works at: https://openscholarship.wustl.edu/eng_etds

Part of the Engineering Commons

Recommended Citation

This Thesis is brought to you for free and open access by the McKelvey School of Engineering at Washington University Open Scholarship. It has been accepted for inclusion in McKelvey School of Engineering Theses & Dissertations by an authorized administrator of Washington University Open Scholarship. For more information, please contact digital@wumail.wustl.edu.
Genetic Algorithm Based Optimization of Baffle Positions in a Forward Osmosis Draw Channel

by

James Robert Lee Koch

A thesis presented to the School of Engineering of Washington University in St. Louis in partial fulfillment of the requirements for the degree of Master of Science

May 2015

Saint Louis, Missouri
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Figures</td>
<td>v</td>
</tr>
<tr>
<td>List of Tables</td>
<td>vi</td>
</tr>
<tr>
<td>List of Abbreviations</td>
<td>vii</td>
</tr>
<tr>
<td>Nomenclature</td>
<td>viii</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>xii</td>
</tr>
<tr>
<td>Dedication</td>
<td>xiii</td>
</tr>
<tr>
<td>Abstract</td>
<td>xiv</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2 Background Theory</td>
<td>4</td>
</tr>
<tr>
<td>2.1 Membrane Filtration</td>
<td>4</td>
</tr>
<tr>
<td>2.1.1 Osmotic Separation Process</td>
<td>5</td>
</tr>
<tr>
<td>2.1.2 Concentration Polarization</td>
<td>7</td>
</tr>
<tr>
<td>2.1.3 Analytic Water and Solute Flux Models</td>
<td>9</td>
</tr>
<tr>
<td>2.1.4 Analytic Forward Osmosis Model</td>
<td>11</td>
</tr>
<tr>
<td>2.2 Computational Fluid Dynamics: Mathematical Flow Modeling of Governing Equations</td>
<td>14</td>
</tr>
<tr>
<td>2.2.1 The Continuity Equation</td>
<td>15</td>
</tr>
<tr>
<td>2.2.2 The Momentum Equation</td>
<td>16</td>
</tr>
<tr>
<td>2.2.3 The Energy Equation</td>
<td>18</td>
</tr>
<tr>
<td>2.2.4 Incompressible Newtonian Fluids</td>
<td>18</td>
</tr>
<tr>
<td>2.2.5 The Navier-Stokes Equation</td>
<td>19</td>
</tr>
<tr>
<td>2.2.6 The Diffusion-Convection Equation</td>
<td>20</td>
</tr>
<tr>
<td>2.3 Spacer Implementation</td>
<td>20</td>
</tr>
<tr>
<td>2.4 Genetic Algorithm Optimization</td>
<td>24</td>
</tr>
<tr>
<td>3 OpenFOAM 2.3.0 CFD Solution Implementation</td>
<td>26</td>
</tr>
<tr>
<td>3.1 Solution Domain</td>
<td>27</td>
</tr>
<tr>
<td>3.1.1 Diffusion-Convection Equation Discretization</td>
<td>28</td>
</tr>
<tr>
<td>3.1.1.1 Convection Term</td>
<td>29</td>
</tr>
<tr>
<td>3.1.1.2 Diffusion Term</td>
<td>30</td>
</tr>
<tr>
<td>3.1.1.3 Source Term</td>
<td>31</td>
</tr>
<tr>
<td>3.1.1.4 Temporal Term</td>
<td>32</td>
</tr>
<tr>
<td>3.1.2 General Boundary Conditions</td>
<td>33</td>
</tr>
<tr>
<td>3.1.2.1 Dirichlet Boundary Condition</td>
<td>34</td>
</tr>
<tr>
<td>3.1.2.2 von Neumann Boundary Condition</td>
<td>35</td>
</tr>
<tr>
<td>3.2 Forward Osmosis Asymmetric Membrane Implementation</td>
<td>35</td>
</tr>
</tbody>
</table>
## Appendix B

Appendix A

8

### Future Research

Appendix B

OpenFOAM Asymmetric Membrane Source Code

OpenFOAM Solver Source Code

### Model and Case Assessment

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2.1</td>
<td>Weakly Compressible Condition</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Membrane Boundary Conditions</td>
</tr>
<tr>
<td>3.2.2.1</td>
<td>Velocity Boundary Condition</td>
</tr>
<tr>
<td>3.2.2.2</td>
<td>Solute Boundary Condition</td>
</tr>
<tr>
<td>3.3</td>
<td>Weakly Compressible Solver Implementation</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Pressure-Velocity Coupling between the Governing Equations</td>
</tr>
<tr>
<td>3.3.2</td>
<td>The PISO Algorithm</td>
</tr>
<tr>
<td>3.3.3</td>
<td>The PIMPLE Algorithm</td>
</tr>
<tr>
<td>3.3.4</td>
<td>The SaltPimpleFoam Solver</td>
</tr>
<tr>
<td>3.4</td>
<td>OpenFOAM File System</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Basic Case Folders and Files</td>
</tr>
<tr>
<td>3.4.1.1</td>
<td>System Folder</td>
</tr>
<tr>
<td>3.4.1.2</td>
<td>Constant Folder</td>
</tr>
<tr>
<td>3.4.1.3</td>
<td>0 and 0.org Folders</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Two Dimensional Geometric Model</td>
</tr>
<tr>
<td>3.4.2.1</td>
<td>Dual-Channel Validation Geometry</td>
</tr>
<tr>
<td>3.4.2.2</td>
<td>Triple-Channel Baffled Geometry</td>
</tr>
<tr>
<td>3.4.3</td>
<td>Case Post-Processing</td>
</tr>
<tr>
<td>3.4.3.1</td>
<td>Pressure Loss Measurement</td>
</tr>
<tr>
<td>3.4.3.2</td>
<td>Water Flux Calculation</td>
</tr>
<tr>
<td>3.4.3.3</td>
<td>Objective Function Calculation</td>
</tr>
</tbody>
</table>

### Results and Discussion

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Model and Case Assessment</td>
</tr>
<tr>
<td>4.1</td>
<td>Sources of Error and Case Verification</td>
</tr>
<tr>
<td>4.2</td>
<td>Validation Experiments</td>
</tr>
<tr>
<td>4.3</td>
<td>Objective Function Dependence on Baffle Inclusion Study</td>
</tr>
<tr>
<td>4.4</td>
<td>Grid Convergence Study</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Triple-Channel Case Meshing Issues</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Baffle Inclusion Meshing Issues</td>
</tr>
<tr>
<td>4.4.3</td>
<td>GCS Results and Discussion</td>
</tr>
<tr>
<td>4.5</td>
<td>Pre-Conditioner and Solver Associated Study</td>
</tr>
</tbody>
</table>

### DAKOTA Genetic Algorithm Implementation

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>DAKOTA Genetic Algorithm Implementation</td>
</tr>
<tr>
<td>5.1</td>
<td>Relevant File System and Basic Solution Process</td>
</tr>
<tr>
<td>5.2</td>
<td>Design of the Genetic Algorithm</td>
</tr>
</tbody>
</table>

### Results and Discussion

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>Results and Discussion</td>
</tr>
<tr>
<td>6.1</td>
<td>Genetic Algorithm Test Run</td>
</tr>
<tr>
<td>6.2</td>
<td>Final Genetic Algorithm Run</td>
</tr>
</tbody>
</table>

### Conclusion

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Conclusion</td>
</tr>
</tbody>
</table>

### Future Research

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>Future Research</td>
</tr>
</tbody>
</table>

Appendix A

OpenFOAM Asymmetric Membrane Source Code

Appendix B

OpenFOAM Solver Source Code
<table>
<thead>
<tr>
<th>Appendix C</th>
<th>OpenFOAM Basic Case File System Source Code</th>
<th>132</th>
</tr>
</thead>
<tbody>
<tr>
<td>Appendix D</td>
<td>DAKOTA Source Code</td>
<td>141</td>
</tr>
<tr>
<td>References</td>
<td></td>
<td>152</td>
</tr>
</tbody>
</table>
List of Figures

Figure 2.1: Forms of basic filtration ................................................................. 5
Figure 2.2: Membrane desalination method based on trans-membrane their osmotic
and hydraulic pressure relationships ............................................................. 6
Figure 2.3: ICP and ECP effects on the effective osmotic pressure across an asymmetric
FO membrane ............................................................................................... 8
Figure 2.4: Osmotically and hydraulically pressurized membrane separation in equilibrium .......... 9
Figure 2.5: AL-FS flux vector relations .................................................................. 11
Figure 2.6: Submerged and zigzag in-channel spacer orientations ................................. 21
Figure 3.1: Internal face relation of neighboring CVs within a non-orthogonally discretized field .......................... 31
Figure 3.2: Boundary face of a CV within a discretized field ....................................... 34
Figure 3.3: CV face-match pairings across membrane boundary .................................. 38
Figure 3.4: Organization of the PIMPLE algorithm .................................................. 50
Figure 3.5: Validation and baffled mesh sketches ..................................................... 59
Figure 4.1: Steady-state flux values for various $\Delta m_A$ obtained using the SaltPimpleFoam solver ..... 71
Figure 4.2: Comparison of SaltPimpleFoam's steady-state flux development profiles ............. 72
Figure 4.3: Comparison of SaltPimpleFoam's steady-state solute profiles ....................... 73
Figure 4.4: Objective function dependence on baffle inclusion ..................................... 74
Figure 4.5: Velocity magnitude fields for the 5, 15 and 40 evenly spaced baffle cases .......... 75
Figure 4.6: Down-stream baffle wake for Figure 4.5’s cases ........................................ 76
Figure 4.7: Basic geometries of the initial cases that form the final triple-channel baffle case .. 78
Figure 4.8: Visual representation of mergeMesh and stitchMesh OpenFOAM utilities’
mesing processes ......................................................................................... 79
Figure 4.9: Samples of the five GCS cases' meshes .................................................... 81
Figure 4.10: Cell counts for all GCS cases ............................................................. 82
Figure 4.11: Water fluxes through the top and bottom membranes of all ten GCS case ......... 83
Figure 4.12: Water flux percentage increase of a GCS mesh due a maximum number of baffles ..... 83
Figure 4.13: Normalized (with respect to the 150 x-cell case) draw channel pressure loss
of all ten GCS cases ...................................................................................... 85
Figure 4.14: Final baffle’s steady state wake velocity profile of the GCS’s 44 baffle cases .... 86
Figure 4.15: Final baffle’s initial and steady state wake ECP profile of the GCS’ 44 baffle cases..... 87
Figure 4.16: Continuity and Residual plots for the new and old $fvSolution$ settings in 60 x-cell cases .91
Figure 4.17: Continuity and Residual plots for the new $fvSolution$ settings in 200 x-cell case .... 92
Figure 5.1: The basic solution procedure of the DAKOTA GA ..................................... 95
Figure 6.1: DAKOTA’s graphical output for the test GA ............................................... 100
Figure 6.2: Fitness mapping of the test GA’s population evolution ................................. 101
Figure 6.3: Complete fitness values for population 30 of the test GA ............................ 102
Figure 6.4: The flow variable profiles for the test GA's high-fitness chromosome ............... 102
Figure 6.5: Pressure loss and water flux for the test GA’s ten best chromosomes .......... 103
Figure 6.6: Baffle positions for each chromosome of the test GA's final population ............ 104
Figure 6.7: DAKOTA’s graphical output for the final GA ............................................. 106
Figure 6.8: Fitness mapping of the final GA’s population evolution ................................. 106
Figure 6.9: Pressure loss and water flux for the final GA’s fifteen best chromosomes ......... 107
Figure 6.10: Baffle positions for each chromosome of the test GA’s final population ......... 108
Figure 6.11: ‘Best’ final GA chromosome’s mesh and relevant flow variable fields .......... 110
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Optimal GA case's <code>fvSolution</code> file settings for the OpenFOAM 1.7.1 framework</td>
<td>89</td>
</tr>
<tr>
<td>4.2</td>
<td>Optimal GA case's <code>fvSolution</code> file settings for the OpenFOAM 2.3.0 framework</td>
<td>90</td>
</tr>
<tr>
<td>5.1</td>
<td>Baffle position constraints available in the DAKOTA <code>.in</code> file.</td>
<td>96</td>
</tr>
<tr>
<td>5.2</td>
<td>DAKOTA test GA settings</td>
<td>97</td>
</tr>
<tr>
<td>6.1</td>
<td>Test GA's chromosome case parameters</td>
<td>99</td>
</tr>
<tr>
<td>6.2</td>
<td>Final DAKOTA GA Settings</td>
<td>105</td>
</tr>
<tr>
<td>6.3</td>
<td>Final GA's chromosome case parameters</td>
<td>105</td>
</tr>
</tbody>
</table>
# List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ND</td>
<td>N-dimensional (where N = 1, 2 or 3)</td>
</tr>
<tr>
<td>AL-FS</td>
<td>Active Layer Facing Feed Solution</td>
</tr>
<tr>
<td>AR</td>
<td>Aspect Ratio</td>
</tr>
<tr>
<td>BC</td>
<td>Boundary Conditions</td>
</tr>
<tr>
<td>CFD</td>
<td>Computational Fluid Dynamics</td>
</tr>
<tr>
<td>CP</td>
<td>Concentration Polarization</td>
</tr>
<tr>
<td>CV</td>
<td>Control Volume</td>
</tr>
<tr>
<td>D-C</td>
<td>Diffusion-Convection equation</td>
</tr>
<tr>
<td>DO</td>
<td>Direct Osmosis</td>
</tr>
<tr>
<td>ECP</td>
<td>External Concentration Polarization</td>
</tr>
<tr>
<td>FO</td>
<td>Forward Osmosis</td>
</tr>
<tr>
<td>FVM</td>
<td>Finite Volume Method</td>
</tr>
<tr>
<td>GA</td>
<td>Genetic Algorithm</td>
</tr>
<tr>
<td>GBC</td>
<td>Gradient Boundary Coefficients</td>
</tr>
<tr>
<td>GIC</td>
<td>Gradient Internal Coefficients</td>
</tr>
<tr>
<td>ICP</td>
<td>Internal Concentration Polarization</td>
</tr>
<tr>
<td>LHS</td>
<td>Left-Hand-Side (of and equation)</td>
</tr>
<tr>
<td>MOGA</td>
<td>Multi-Objective Genetic Algorithm</td>
</tr>
<tr>
<td>N-S</td>
<td>Navier-Stokes equation</td>
</tr>
<tr>
<td>PIMPLE</td>
<td>Pressure-Implicit Method for Pressure-Linked Equations</td>
</tr>
<tr>
<td>PISO</td>
<td>Pressure Implicit Splitting of Operators</td>
</tr>
<tr>
<td>PPM</td>
<td>Parts Per Million</td>
</tr>
<tr>
<td>PRO</td>
<td>Pressure Retarded Osmosis</td>
</tr>
<tr>
<td>RHS</td>
<td>Right-Hand-Side (of and equation)</td>
</tr>
<tr>
<td>RO</td>
<td>Reverse Osmosis</td>
</tr>
<tr>
<td>SIMPLE</td>
<td>Semi-Implicit Method for Pressure-Linked Equations</td>
</tr>
<tr>
<td>SOGA</td>
<td>Single-Objective Genetic Algorithm</td>
</tr>
<tr>
<td>VBC</td>
<td>Value Boundary Coefficients</td>
</tr>
<tr>
<td>VIC</td>
<td>Value Internal Coefficients</td>
</tr>
</tbody>
</table>
### Nomenclature

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Units</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Latin Symbols:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>([\frac{m}{sPa}])</td>
<td>Membrane's resistance to water transport</td>
</tr>
<tr>
<td>a</td>
<td>[-]</td>
<td>Discretization information coefficients</td>
</tr>
<tr>
<td>B</td>
<td>([\frac{m}{s}])</td>
<td>Membrane's resistance to solute permeation</td>
</tr>
<tr>
<td>C%</td>
<td>[-]</td>
<td>Percentage chance of a crossover in a genetic algorithm</td>
</tr>
<tr>
<td>c</td>
<td>([\frac{kg}{m^3}])</td>
<td>Concentration /</td>
</tr>
<tr>
<td></td>
<td>[-]</td>
<td>Chromosome of a genetic algorithm</td>
</tr>
<tr>
<td>D</td>
<td>([\frac{m^2}{s}])</td>
<td>Diffusion coefficient</td>
</tr>
<tr>
<td>d</td>
<td>[m]</td>
<td>Mid-channel baffle diameter</td>
</tr>
<tr>
<td>dh</td>
<td>[m]</td>
<td>Hydraulic diameter</td>
</tr>
<tr>
<td>F</td>
<td>[-]</td>
<td>Fitness function set of a genetic algorithm</td>
</tr>
<tr>
<td>f</td>
<td>[-]</td>
<td>Fitness function of a genetic algorithm</td>
</tr>
<tr>
<td>h</td>
<td>[m]</td>
<td>Channel height</td>
</tr>
<tr>
<td>K</td>
<td>([\frac{s}{m}])</td>
<td>Porous membrane support layer diffusion resistivity /</td>
</tr>
<tr>
<td></td>
<td>([\frac{s}{m}])</td>
<td>Experimentally determined mass transfer coefficient of the membrane</td>
</tr>
<tr>
<td>l</td>
<td>[m]</td>
<td>Physical thickness of the membrane's support layer /</td>
</tr>
<tr>
<td></td>
<td>[m]</td>
<td>Center-to-center spacing between mid-channel baffles</td>
</tr>
<tr>
<td>M%</td>
<td>[-]</td>
<td>Percentage chance of a mutation in a genetic algorithm</td>
</tr>
<tr>
<td>m</td>
<td>[kg]</td>
<td>Mass or mass fraction</td>
</tr>
<tr>
<td>Pop</td>
<td>[-]</td>
<td>Population of a genetic algorithm</td>
</tr>
<tr>
<td>ΔP</td>
<td>[Pa]</td>
<td>Hydraulic pressure along the length of a channel</td>
</tr>
<tr>
<td>p</td>
<td>[Pa]</td>
<td>Pressure</td>
</tr>
<tr>
<td>Δp</td>
<td>[Pa]</td>
<td>Hydraulic pressure across membrane</td>
</tr>
<tr>
<td>Rg</td>
<td>([\frac{T}{Kmol}])</td>
<td>Universal gas constant</td>
</tr>
<tr>
<td>Re</td>
<td>[-]</td>
<td>Reynolds number</td>
</tr>
</tbody>
</table>
\( \text{Re}_{\text{ch}} \) \([-\)] Reynolds number of a baffled channel

\( \text{Re}_w \) \([-\)] Reynolds number on the channel wall

\( S \) \([-\)] Source term linearization coefficient /

\( [m] \) Membrane Structural Parameter

\( T \) \([K]\) Solution temperature

\( t \) \([s]\) Time

\( V_M \) \([m^3/\text{mol}]\) Molar volume of pure solvent

Vector or Matrix Symbols (\textbf{bolded} in text):

\( A \) \([-\)] Matrix containing discretization information coefficients

\( a \) \([m/s^2]\) Acceleration vector

\( d \) \([m]\) Vector from a control volume centroid to the center of one of its faces

\( F \) \([N]\) Force vector

\( f \) \([N/kg]\) Body force per unit mass vector

\( g \) \([m/s^2]\) Gravitational acceleration

\( H \) \([-\)] Matrix containing coefficients of neighboring control volumes

\( J \) \([m/s]\) Water flux through the membrane /

\( [kg/m^2 s]\) Solute flux through the membrane

\( S \) \([m^2]\) Outward-pointing (normal) vector from a control volume surface

\( U \) \([m/s]\) Velocity

\( u \) \([m/s]\) x-direction velocity

\( v \) \([m/s]\) y-direction velocity

\( w \) \([m/s]\) z-direction velocity
Greek Symbols:

\( \epsilon \) \([\text{-}]\) Porosity of the membrane

\( \lambda \) \([\text{Pas}]\) Second viscosity coefficient

\( \mu \) \([\frac{L}{mol}]\) Chemical potential / [Pas] Dynamic viscosity /

\( \mu \) \([\text{Pas}]\) Molecular viscosity coefficient

\( \xi \) \([\frac{Pa m^3}{kg}]\) Concentration to osmotic pressure linear constant of proportionality

\( \nu \) \([\frac{m^2}{s}]\) Kinematic viscosity

\( \Delta \pi \) \([\text{Pa}]\) Osmotic pressure across membrane

\( \pi \) \([\text{Pa}]\) Osmotic pressure

\( \rho \) \([\frac{kg}{m^3}]\) Density

\( \tau \) \([\text{-}]\) Tortuosity of the membrane's support layer

\( \tau_{ij} \) \([\frac{N}{m^2}]\) Shear stress in \(j\)-direction on the plane perpendicular to \(i\)-axis

\( \Phi \) \([\frac{kg}{s}]\) Mass flux

\( \phi \) \([\text{-}]\) Flow variable place-holding value

\( \chi \) \([\frac{mol}{mol}]\) Molar fraction

\( \Psi, \Upsilon, \text{ and } \Lambda \) \([\text{-}]\) Known values defining the coefficient fields VIC, VBC, GIC and GBC
Superscripts Symbols:

n  New time step
o  Old time step

Subscript Symbols:

A  Solute
AB Solute-solvent interaction
ave Average
B  Solvent
b  Body (of a CV) /
    Control volume boundary face
D  Draw side surface of the membrane
eff Effective
F  Feed side surface of the membrane
f  Control volume face
I  Interface between membrane active and support layers
m  Membrane's surface-to-free-solution interface
N  Neighboring control volume centroid
n  Normal vector
P  Control volume centroid
S  Surface (of a CV)
tot  Total
x  Unit x-direction or component
y  Unit y-direction or component
z  Unit z-direction or component
Acknowledgments

First and foremost, I would like to give my deepest thanks to both Professor Ramesh Agarwal and my friend Tim Wray for their unending and fruitful help and support. Without them, I would not have had the foresight and willpower to complete the research described within this work. I am grateful also to Mathias Gruber for selflessly answering my plethora of questions at the beginning of this project. Finally, my thanks to my Alma Maters Grinnell College and Washington University and their multitude of professors who honed my intellect and desire for the pursuit of knowledge.

Without each and every one of you I would not be here.

Thank you.

James Robert Lee Koch

Washington University in St. Louis

May 2015
Dedicated to.

My family: intimate and otherwise.

You have birthed, built and guided me throughout my entire life.

I love and respect each and every one of you.
ABSTRACT

Genetic Algorithm Based Optimization of Baffle Positions in a Forward Osmosis Draw Channel

by

James Robert Lee Koch

Master of Science in Mechanical Engineering

Washington University in St. Louis, 2015

Research Advisor: Dr. Ramesh K. Agarwal

Forward Osmosis (FO) driven asymmetric membrane filtration is a developing technology which shows promise for seawater desalination and wastewater treatment. Due to the fact that asymmetric membranes are widely used in conjunction with this technology, internal concentration polarization (ICP), a flow-entrainment effect occurring within such membranes, is a significant if not dominant source of overall osmotic pressure loss across the membrane. Accurate modeling of ICP effects is therefore very critical for accurate Computational Fluid Dynamic (CFD) modeling of asymmetric membranes. A related, dilutive effect known as external concentration polarization (ECP) also develops on both the rejection and draw sides of the membrane, further contributing to osmotic pressure loss. In order to increase the overall water flux, circular spacers known as baffles can be implemented in the draw channel of FO cross-flow membrane exchange units to decrease the effects of ICP and draw ECP. The drawback of baffle inclusions is an increased pressure loss across the length of the draw channel. The system efficiency gained by the decrease in ECP must therefore be weighed against the energy cost of hydraulically making up lost channel pressure. To model the geometry of a FO cross-flow channel, the open source CFD package OpenFOAM is used. A compressible
flow model with explicit boundary conditions is developed to simulate the flux transfer, ICP and ECP effects present within an asymmetric membrane when exposed to a NaCl solution. Results are validated by comparison with the numerical data generated by earlier models of asymmetric membranes implemented by other investigators using similar simulation conditions. Mesh convergence and OpenFOAM flow-variable solver studies are then conducted to minimize solution inaccuracy and computation time. Finally, a genetic algorithm is implemented via the open source optimization toolkit DAKOTA which allows optimized draw-channel baffle placement such that the pressure loss along the draw channel is minimized and the water flux through the Asymmetric, FO membrane is maximized. The results of this optimization are then analyzed and discussed.
Chapter 1

Introduction

Membrane-desalination technology is an emerging field of science which has become significantly important over the past half century. Development and research in the membrane field can be traced to the early eighteenth century, though most past applications of membranes were related to the theoretical development of physics and chemistry [1,2]. The early 1960's saw the advent of the industrial process via the development of the Loeb-Sourirajam process which allowed the production of defect-free, high flux, asymmetric membranes [3]. The decades following the discovery of this process saw significant scientific progress in the field of membrane desalination, including the ability to efficiently package membranes within specialized systems and the development of the pressure-driven membrane separation process. By the 1980's, this process had been implemented in large-scale plants worldwide [1,2].

Membrane processes are primarily distinguished by their form of driving force (hydraulic pressure, osmotic pressure, thermal gradients, etc.), separation phases (e.g. gas or liquid), and the relative sizes the molecules in the desired permeate and re-buffed material (as this will determine the filtration scale, e.g. micro or nanofiltration). This thesis only considers the Forward Osmosis (FO) process of separating aqueous solutions via an osmotic pressure gradient. FO has recently emerged as an energy-efficient alternative to pressure-driven processes such as Reverse Osmosis (RO) since it establishes a chemical trans-membrane pressure gradient via a concentrated draw solution (draw) and diluted feed solution (feed) which drives water flux (referring to the flux of water through the membrane separating the two solutions membrane) from the feed solution to the draw solution [4].

By 2013, RO plants’ energy use was close to the pressure-related thermodynamic energy consumption limit of $1.06 \frac{kW\text{hr}}{m^3}$ with plant usages of $3.4 \frac{kW\text{hr}}{m^3}$. FO plants of proportional sizes can perform membrane separation at $\approx 0.5 \frac{kW\text{hr}}{m^3}$. Additionally, FO has been found to have a lower propensity for membrane fouling, although this effect is not the focus of this work and will therefore not be discussed [5-9].

Recent studies have found simulated FO water flux values which are lower than comparable experimental results [10,11]. This error between observed and simulated values is due to the
employment of a model which did not accurately describe solution concentration and dilution
effects which occur on the surfaces and within the supporting layer of FO membranes, as they
diminish the overall osmotic pressure seen across the membrane [12]. Until recently, these effects
were modeled with one-dimensional (1D) analytic and semi-analytic models derived from [12,13], which
more accurately predicted experimental values, though they did not take the effects of the support-
layer's dilution into account [7,12,14-16]. Despite these advances, developing a model to encompass
all aspects of membrane filtration defined within complex geometries is not possible with an
analytic model. Thus, the last 20 years have seen the employment of Computational Fluid Dynamics
(CFD) to model the complex flow patterns present in membrane systems as it takes a robust,
numerical approach that can handle the inclusion of many parameters. A model was recently
proposed by Gruber et al. [17] which accurately predicted all aforementioned effects on the trans-
asymmetric-membrane osmotic pressure, as well as the slip-velocity condition which can occur
within the membrane's support layer. The model developed in this thesis is influenced by Gruber's
model.

Industrially, membrane systems are generally implemented in spiral-wound models in order to
maximize the total membrane surface area per membrane-unit installed. The spiral-wound setup
easily allows for the inclusion of spacer-filament sheets within the feed or draw channels. The
introduction of spacer filaments (or 'baffles' as they are interchangeably referred to in this work)
into a membrane flow channel can, depending on the Reynolds number and spacer-to-spacer
locations, create unsteady regimes in the flow field which can reduce undesirable solute
concentration or dilution surrounding or in the channel's membrane through vortex shedding
[18,19]. These effects can be included in the CFD model of a membrane system in order to study
their effects.

The goal of this thesis is to form a Genetic Algorithm (GA) in such a way that future
researchers can easily manipulate and alter it in order to explore the many implications of
baffle placement optimization within FO asymmetric-membrane channels in crossflow.
This will be accomplished by simulating and analyzing a FO system with draw-channel baffles using
a rigorous CFD model inspired by previous models developed using commercial and open-source
software [17,19,20]. This simulation model will act as a chromosome for the GA. The model
developed in this thesis cannot be found in commercial software, and is therefore implemented
using the open-source toolkit OpenFOAM® (Open Field Operation And Manipulation: a registered
trade mark of OpenCFD Limited, the producer of the OpenFOAM software), which provides a free and extensive range of features and solution algorithms. The GA is then employed through the coupling of the OpenFOAM toolkit with the open-source optimization toolkit DAKOTA (the Design Analysis Kit for Optimization and Terascale Applications developed by Sandia National Laboratories which provides a flexible, extensible interface between analysis codes and iterative systems analysis methods), in order to find draw-channel baffle locations which maximize the water flux of the modeled system and minimize the pressure loss which results from baffle inclusion.

This thesis first presents the applicable theory for the following: osmotic membrane separation processes, the fundamentals of CFD (focusing on its relation to OpenFOAM), baffle-induced flow and the basic genetic algorithm (focusing on its relation to DAKOTA). Next, this formulated theory is used to structure a model capable of simulating a FO, asymmetric membrane system within the OpenFOAM toolkit. Model verification and validation is presented in defense of this model. A geometry capable of being modified by an external GA is then presented and analyzed. Finally, the developed GA is applied to this geometry via DAKOTA, and the results are discussed.
Chapter 2

Background Theory

2.1 Membrane Filtration

A membrane is a physical, selective barrier which separates two phases, remaining both impermeable to those solutes within each phase considered undesirable and permeable to specific, desired solutes [1,2]. The solution containing particles the membrane is designed to reject to is generally labeled as the feed, which is partially transported through the membrane by a driving force acting across the membrane. The source of this forcing gradient can differ physically depending on the filtration process being considered. In the case of FO this driving force is an osmotic pressure gradient, though temperature, electric and hydraulic pressure gradients are all in common use. Membrane processes are also characterized by the phases they separate (gas or liquid) and the sizes of the particles they retain, e.g. for hydraulic pressure-driven separation of aqueous solutions there are roughly four size categories: microfiltration, (0.1 – 5 μm), ultrafiltration (1 – 100 nm), nanofiltration (0.5 – 10 nm), and RO (< 0.5 nm) [2, 21].

Membrane modules are commonly distinguished by two modes of operation; dead-end or cross-flow filtration. Dead-end filtration, seen in Figure 2.1.a on the following page, is used primarily in pressure-driven filtration such as RO. The feed is pneumatically forced against the membrane, which develops a water flux from the feed to the draw due to the relatively large Brownian motion of the rejected particles on the feed side of the membrane. Over time, the rejected particles accumulate, causing resistance to the passage of water molecules [1,2]. Thus, unless the initial feed is very clean, this process becomes inefficient due to membrane fouling and the time-dependent decrease in water flux [21]. In cross-flow filtration, seen in Figure 2.1.b the feed and draw solutions are contained in channels running in parallel connection to each other through a series of walls and membranes. The tangential flow of the streams helps to sweep out the accumulation of rejected molecules from the surface of the membrane over time, generally decreasing membrane fouling and increasing overall permeate flux through the membrane [2,21]. Although these results usually make cross-flow the filtration style of choice, both forms discussed above can be implemented in a spiral-wound module form. All geometries used for verification and experimentation in this work
implement cross-flow filtration.

![Diagram of filtration types](image)

Figure 2.1: Forms of basic filtration. a) Dead-end filtration in which pneumatic gradients create permeate flux and the accumulation of rejected molecules. b) Cross-flow filtration in which tangential-to-membrane flow helps reduce rejected particle buildup.

### 2.1.1 Osmotic Separation Process

The driving gradient used in this work to develop permeate (water) flux across a membrane is osmotic pressure. The analytic meaning of osmotic pressure is discussed in Section 2.1.3, though it should be stated at this point that the osmotic pressure will increase for larger ratios between the average PPM of the draw and feed solutions [22]. This pressure leads to the effect of osmosis: 'the net movement of water across a selective permeable membrane driven by a difference in osmotic pressure across the membrane' [2,4]. When the primary driving force in a membrane separation process is osmosis, the process is known generally as Direct Osmosis (DO) [4,12]. If a DO system has no induced pressure other than an osmotic pressure driving permeate flow from the feed to the draw, it is known as a FO system. If a DO system has a draw solution with high osmotic pressure conjoined to a feed solution with low osmotic pressure, and hydraulic pressure is added to the draw side of the membrane until it equates the net osmotic pressure of the system (in effect limiting the magnitude of induced water flux to zero), it is known as a Pressure Retarded Osmosis (PRO) system. PRO systems in effect convert osmotic pressure into hydraulic pressure, and have therefore shown potential in the field of electricity generation [12,23]. FO and PRO can be easily and graphically
related to RO if the water flux through the membrane if linearly proportional to the net driving force across it (this linear assumption is due to some approximations which are discussed in Section 2.1.3). This relation is seen in Figure 2.2, in which $\Delta \pi$ and $\Delta p$ respectively represent the osmotic and hydraulic pressure differences across the membrane.

From an industrial desalination standpoint, FO processes have a unique energy advantage over commonly used pressure-based processes such as RO, as FO precludes the need for the energy-intensive generation of hydraulic pressure required for RO since its process theoretically operates at $0 \Delta p$. This, along with other unique advantages, potentially makes FO more cost-effective than processes like RO [4,7,9]. However, several technical barriers still limit the field of FO [25]. One such issue is that the osmotic potential needed to efficiently filter the feed solution requires the implementation of chemical compounds in the draw, lending it an artificially high osmotic potential. Such artificial solutes may react negatively with the membrane if poorly designed, and must be processed out of the diluted draw solution later [4,25]. Draw solution additives usually take the form of an ionic compound suspended within a gas matrix which can be removed thermally or chemically with few waste reactants. One example of this form of compound is a combination of ammonia and carbon dioxide gases: the solution can achieve up to 250atm in osmotic pressures and the ammonia salt is later removed thermally [4,26,27]. This is the portion of a FO plant's water-production process which requires a majority of the energy supplied to it, to the point where the
total energy consumption is over that of RO. However, since the energy required for this process almost never needs to be clean, waste, geothermal or solar generated heat can be cheaply used. This leniency in energy source is what makes FO cost-effective and energy-efficient compared to RO. Another issue is that of membrane development. Pressure driven filtration systems use asymmetric membranes with a thin, dense rejection layer facing the feed and a porous support layer facing the draw (although this orientation can be reversed) [1]. Depending on the membrane orientation, the porous support layer will see either a dilution or concentration of solute due to permeate flux, effectively reducing the osmotic pressure across the active layer of the membrane (this phenomena is analyzed in Section 2.1.2). Until recently, this effect had not been accurately modeled, making simulation testing of FO membranes questionable [17].

### 2.1.2 Concentration Polarization

One of the main influences on performance in FO systems is the overall 'seen' osmotic pressure difference across the rejection layer of the asymmetric membrane (notice that this does not include the support layer). The main inhibitor of this 'seen' osmotic pressure is an effect called Concentration Polarization (CP), described as 'the induced accumulation or dilution of solute concentration in or at the surfaces of the membrane due to its ability to transport certain components more readily than others' [1,2,21]. Due to the effect of CP, the effective, or 'seen', osmotic pressure is not the bulk osmotic pressure defined by the average solute concentration difference between the feed and draw solutions.

Dilutive or concentrative CP which occurs on the surfaces of an FO membrane is known as External Concentration Polarization (ECP), while CP which occurs within the porous support-layer of the FO membrane is known as Internal Concentration Polarization (ICP). In FO systems, the ICP effect is dilutive, as permeate water will dilute the draw solution within the support-layer, lowering the concentration and thus the osmotic pressure across the rejection portion of the membrane. This effect is definitively non-negligible (if it was, the effective osmotic pressure would be that of the membrane osmotic pressure, \( \pi_m \), as seen in Figure 2.3 on the following page), and must be included in analytic models if accuracy is to be achieved. Water flux through the membrane will dually create a concentrative ECP on the feed side of the membrane as solute present in the local free-stream is forced against the rejection layer. When the now diluted draw solution within the support-layer is
convected out into the draw's free-stream, it is at a lower solute concentration and creates dilutive ECP on the draw side of the membrane. These effects are graphically represented in Figure 2.3 below.

![Diagram of ICP and ECP effects on the effective osmotic pressure across an asymmetric FO membrane](image)

Figure 2.3: ICP and ECP effects on the effective osmotic pressure across an asymmetric FO membrane [17].

The channel's mean cross-flow velocity heavily influences the how affective that channel's ECP is on the overall membrane water flux, and an overall or local increase in this velocity due to flow obstructions such as in-channel baffles can help mitigate ECP effects. ICP, however, is in effect decoupled from the macro feed channel flow effects in the FO setup, occurring within the membrane itself as part of a membrane boundary condition. If slip conditions are accounted for in the membrane, the draw channel cross-flow velocity will have a mitigating effect on ICP; an effect that is the focus of the work done in [17]. If this slip condition on the membrane is not introduced, however, neither channel's flow effects will inherently affect the ICP due to the (virtually) zero-gradient velocity boundary condition which occurs on both outer surfaces of the FO membrane.
2.1.3 Analytic Water and Solute Flux Models

The analytical theory behind mass transport within and across membranes is extremely complex, employing non-equilibrium thermodynamics as well as mechanical interactions related to structural membrane characteristics [2]. Thus, only a cursory discussion concerning the driving force of osmotic pressure will be presented in this thesis. Osmotic pressure is essentially a physical system's desire to obtain chemical equilibrium. If two chambers containing a solution and the solution's pure solvent are separated by a membrane, solvent will permeate that membrane until the chemical potentials are equal on either side of it. In such a system, the osmotic pressure is equal to the hydraulic pressure which would be required to be applied on the solution in order to prevent any solvent from crossing the membrane. Ideally, the entirety of the solvent would cross over since no matter how dilute the solution becomes it will still have a higher concentration than the pure solvent. However, physical systems generate force responses such as hydraulic pressure in the solution chamber and (possibly) an induced vacuum in the solvent chamber limit this solvent crossover.

The symbol which will be used to denote osmotic pressure for the remainder of this thesis is \( \pi \), while the symbol for chemical potential will be \( \mu \) and the molar fraction will be \( \chi \). Solvent will be subscripted as \( B \), while solute will be subscripted as \( A \). Consider Figure 2.4 below, which depicts a pressurized system in equilibrium similar to that discussed in the above paragraph.

![Figure 2.4](image.png)  

Figure 2.4: Osmotically and hydraulically pressurized membrane separation in equilibrium.
In this system, the chemical potential equilibrium over the membrane can be written. If \( \mu_B^*(p) \) is the chemical potential for the pure solvent and \( \mu_B(\chi_B, p + \pi) \) is the chemical potential of the solution, then this equilibrium is:

\[
\mu_B^*(p) = \mu_B(\chi_B, p + \pi) \quad (2.1)
\]

If \( R_g \) is the universal gas constant and \( T \) is the solution temperature, the chemical potential for a solvent-solute solution is defined as [22]:

\[
\mu_B(\chi_B, p + \pi) = \mu_B^*(p + \pi) + R_g T \ln(\chi_B) \quad (2.2)
\]

If \( V_M \) is the molar volume of the pure solvent \( B \) and constant temperature is assumed, the pure solvent chemical potential pressure-dependence can be defined as [22]:

\[
\mu_B^*(p + \pi) = \mu_B^*(p) + \int_p^{p+\pi} V_M \, dp \quad (2.3)
\]

The combination of Equations 2.1 and 2.3 leads to an expression for the equilibrium condition:

\[
-R_g T \ln(\chi_B) = \int_p^{p+\pi} V_M \, dp \quad (2.4)
\]

Equation 2.4 leads to the value of \( \pi \) which must be applied in Figure 2.4 to equate the chemical potentials of the solution and pure solvent. Equation 2.5 below shows a common power-series expansion assumption made for strongly diluted solutions which is applicable in the situations being considered:

\[
\ln(\chi_B) \approx \ln(1 - \chi_A) \approx \chi_A \quad (2.5)
\]

Additionally, the assumption that the pressure range of the integration is so small that \( V_M \) is constant can be applied to Equation 2.4. These assumptions lead to the van't Hoff expression for osmotic pressure, which states that the osmotic pressure is linearly proportional to the solute concentration \( c_A \) [1,2,22]:

\[
\pi \approx \frac{R_g T \chi_B}{V_M} \approx c_A R_g T \quad (2.6)
\]

This relationship between osmotic pressure and solute concentration will be assumed for the rest of this work.
2.1.4 Analytic Forward Osmosis Model

The solute-diffusion model is generally approached through the assumption that the chemical potentials of the feed and draw are in equilibrium with their adjacent membrane surfaces. In this way, the expressions for the chemical potential for the in both the fluid and membrane phases can be equated at the solution-membrane interface. When rearranged, they can then be inserted into Fick's law to get the transport equations for each component. When this process is applied to a FO system, ICP must be taken into account within any porous support layers. For the cases considered in this thesis the asymmetric membrane's rejection (or 'active') surface is facing the feed solution. This is known as the Active Layer Facing Feed Solution (AL-FS) orientation, which assumes that solute is rejected solely at the active surface. This orientation, along with all subscripts, vectors and symbols pertaining to it, is exemplified in Figure 2.5.

![Figure 2.5: AL-FS flux vector relations.](image)

The flux transport equations for water and solute are found through employment of a solution-diffusion approach and by assuming that $\Delta p = 0$ across the membrane [7]:
\[ J_w = A(\pi_I - \pi_{F,m})n_D \quad (2.7) \]

\[ J_s = -B(c_I - c_{F,m})n_D \quad (2.8) \]

The coefficient \( A \) describes the membrane's resistance to water transport and \( B \) describes the membrane's resistance to solute permeation. Both of these coefficients are found to be inversely proportional to the thickness of the membrane, and can be estimated experimentally [2]. \( n_D \) is the membrane unit normal vector pointing in the direction of the draw solution. The subscripts \( I, F, D \) and \( m \) denote the internal interface between the active and support layers, the feed side of the membrane, the draw side surface of the membrane, and one of the membrane's surface-to-free-solution interfaces respectively. Equation 2.7 is based on several approximations, including that any solute-solvent couplings are neglected and that the membrane's mechanical properties are assumed to be homogeneous [28,29,30]. It can be seen that while the water flux is dependent on the osmotic potential over the active layer of the AL-FS membrane, the solute flux is dependent only on the solute concentration.

Within the support layer, the transport of solute out of the support layer via convection must be compared to that entering due to diffusion in order to further constrain the overall solute flux \( J_s \):

\[ J_w c - D_{eff} \frac{dc}{dx} n_D = J_s \quad (2.9) \]

Here \( D_{eff} = \epsilon D_{AB} \) (\( D_{AB} \) is the solute diffusion coefficient and \( \epsilon \) is the porosity of the membrane), and is the effective diffusion coefficient of solute within the porous support layer of the AL-FS membrane [7]. \( c \) is the solute concentration at a positive, normal distance from the interface \( I \) within the support layer. Equation 2.9 is subject to the boundary conditions found in Equation 2.10, with \( l \) being the physical thickness, \( l_{eff} \) being the effective thickness and \( \tau \) being the tortuosity of the support layer:

\[ c = \begin{cases} c_I, & x = 0 \\ c_{D,m}, & x = l_{eff} = \tau l \end{cases} \quad (2.10) \]

If \( K \) is the experimentally determined mass transfer coefficient of the AL-FS membrane as coined by [12] and found in Equation 2.12, solving Equations 2.7 to 2.10 leads to the following expression for water flux:
\[ \ln \left( \frac{c_{D,m} + \zeta}{c_2 + \zeta} \right) n_D = K J_w , \quad \text{where } \zeta = \frac{B(c_1 - c_{F,m})}{A(c_1 - c_{F,m})} \quad (2.11) \]

\[ K = \frac{l_{\text{eff}}}{D_{\text{eff}}} = \frac{\tau \ell}{\epsilon_{DAB}} = \frac{S_c}{D_{AB}} \quad (2.12) \]

Equation 2.11 can be simplified into Equation 2.13 by assuming that the osmotic pressure is linearly dependent on the solute concentration through some experimentally determined constant of proportionality this thesis will denote as \( \zeta \). Thus, for an AL-FS oriented asymmetric FO membrane,

\[ J_w = \frac{1}{K} \ln \left( \frac{B + A \pi_{D,m}}{B + |J_w| + A \pi_{F,m}} \right) n_D \quad (2.13) \]

In cases with high solute rejection and water flux (i.e., when \( B \ll A \pi_{D,m} \) and \( B \ll |J_w| \)), this relationship reduces to a ‘simple’ water flux described as:

\[ J_w = A(\pi_{D,m} e^{-|J_w|K} - \pi_{F,m})n_D \quad (2.14) \]

Equating Equations 2.7 and 2.14 leads to the ICP modulus found in Equation 2.15, which only holds for 'perfect', 100% solute rejecting membranes coupled with the linear relationship described with \( \zeta \):

\[ \frac{\pi_I}{\pi_{D,m}} = e^{-|J_w|K} \quad (2.15) \]

Finally, through combination of Equations 2.7 and 2.8 and the employment of the \( \zeta \) linear relationship, the water flux can be directly related to the solute flux:

\[ J_s = -\frac{B}{\zeta A} J_w \quad (2.16) \]

Equations 2.13 and 2.16 are sufficient to describe the AL-FS asymmetric membrane's mass transportation model, since the surface values for the osmotic pressures are directly resolved (a requirement for CFD implementation).
2.2 Computational Fluid Dynamics: Mathematical Flow Modeling of Governing Equations

Computational Fluid Dynamics is a field of computational theory which attempts to solve or analyze fluid mechanic problems through the application of numerical methods and algorithms within a discretized computational domain. The advent of CFD practice can be traced back to the first use of the general-purpose digital computer in the early 1950s, and the field has expanded over the ensuing years in proportion to the development of computing power [31]. CFD has become an important, if not fundamental, tool in investigating experimental flow situations since it can preclude the need for measurement in or development of experimental setups which are too small (nanomaterial development) or too volatile (combustion chamber or turbomachinery assessment) for instrument installation or measurement accuracy to be possible. CFD has also become a useful qualitative tool for the optimization of machinery design or experimental setup geometry, and can reduce the amount of prototype testing required for a product [32]. Application of CFD is limited by several factors. The overall size/intricacy of the problem being considered is hemmed by the computational power available: a case which is too large may take too long to solve, eliminating CFD as a useful analysis tool. Furthermore, CFD cannot (currently) be used for real-time computation or problem-alterations, and requires fundamental knowledge of any numerics pertaining to the problem being assessed [32].

Application of CFD theory to a FO process thus requires first understanding the physical laws governing the fluid used, and then numerically solving them throughout a computational domain which represents such a process. The laws pertaining to the case addressed by this work are the basic governing equations of an aqueous solution, the computational domain solution of which will yield the pressure, velocity and solute concentration information pertaining to at any point in the case geometry in which aqueous flow exists. The governing equations corresponding to aqueous fluid and solute flow are presented in Sections 2.2.1 to 2.2.6 below. Fluid theory states that these equations amount to the conservation of mass, momentum and energy, and are referred to respectively as the continuity, momentum and energy equations [32]. Within a CFD framework, mathematical descriptions of these equations are used to discretize the solution across the individual cells making up the computational domain of the situation being considered. One such description is the Finite Volume Method (FVM), which maps scalar or vectorial values pertaining to
fluid variables onto the centers or faces of the cells making up the computational domain. These
cells may be either stationary or moving within the fluid, and may be treated as either finite *Control
Volumes* (CV) or infinitesimal fluid elements depending on the mathematical descriptions used (a
treatment which can be switched between computation procedures).

### 2.2.1 The Continuity Equation

The continuity equation is the mathematical representation of the principle of mass conservation; a
principle which can be described within a FVM framework as an equality between the ‘net flow out
of a considered CV’ and the 'time-rate of mass decrease inside that CV' [32]. The CV being
considered here has both an arbitrary shape and a finite size. The first portion of the equality can be
described as: (CV surface density)*(CV surface area)*(CV’s surface-normal velocity component).
This is shown mathematically in Equation 2.17:

\[
\int_S \rho \mathbf{U} \cdot d\mathbf{S} \quad (2.17)
\]

Here, \( \mathbf{S} \) is the outward-pointing surface vector of the CV. The second portion of this equality can
be mathematically described as Equation 2.18:

\[
-\frac{\partial}{\partial t} \int_V \rho \, dV \quad (2.18)
\]

The term is negative to account for the fact that \( \mathbf{S} \) is defined as pointing away from the center of the
CV: a directionally-positive net surface mass flow would result in a decrease in the mass contained
within the CV [32]. The continuity equation may thus be described as Equation 2.19:

\[
\int_S \rho \mathbf{U} \, dV = -\frac{\partial}{\partial t} \int_V \rho \, dV \quad (2.19)
\]

Via Gauss’s theorem, this model can be described in differential form as Equation 2.20:

\[
\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{U}) = 0 \quad (2.20)
\]
2.2.2 The Momentum Equation

Due to convenience, the derivation of the momentum equation found in this section is obtained by applying Newton’s second law to a fluid infinitesimal moving in the direction of the flow. The law is vectorial in nature, seen generally in Equation 2.21, and can be considered by Cartesian components for derivation simplicity.

\[ \mathbf{F} = m\mathbf{a} \quad (2.21) \]

The fluid element force vector \( \mathbf{F} \) in Equation 2.21 contains both body forces (which act on the volumetric mass of the element from a distance, e.g. gravity) and surface forces (which act on the surface of the element, e.g. shear stress, normal stress and pressure) [32]. The entire (vectorial) force contribution to the cell will be denoted as ‘\( \mathbf{f} \)', the ‘body force per unit mass’.

The body force of this contribution, \( \mathbf{F}_b \), can be calculated for an element of side lengths \( dx, dy \) and \( dz \) from Equation 2.22.

\[ \mathbf{F}_b = \rho \mathbf{f} (dx \cdot dy \cdot dz) \quad (2.22) \]

The description of element surface forces requires adoption of a shear subscript convention \( \tau_{ij} \) in which pairs ‘ij’ denote a stress on a plane perpendicular to the i-axis, and pointing in the j-unit-direction. Shear stresses on the faces of the infinitesimal volume arise from the surface-imminent flow’s velocity being larger in tangential magnitude than the tangential surface velocity of the volume itself. This creates ‘pulling’ stress if the difference in tangential velocities is positive, and a ‘dragging’ stress if the difference is negative. Pressure must also be accounted for, and acts normal to all surfaces. If, for explanatory purposes, all shear and pressure surface forces on the fluid element pertaining to the x-direction are summed, the relationship of surface forces in the x-direction would be described as Equation 2.23 [32].

\[
\begin{align*}
\mathbf{F}_{xx} &= \left[ p - \left( p + \frac{\partial p}{\partial x} dx \right) \right] dy \, dx + \left[ \left( \tau_{xx} + \frac{\partial \tau_{xx}}{\partial x} dx \right) - \tau_{xx} \right] dy \, dz \\
&\quad + \left[ \left( \tau_{yx} + \frac{\partial \tau_{yx}}{\partial y} dy \right) - \tau_{yx} \right] dx \, dz + \left[ \left( \tau_{zx} + \frac{\partial \tau_{zx}}{\partial z} dz \right) - \tau_{zx} \right] dx \, dy
\end{align*}
\quad (2.23)
\]
By allowing terms in Equation 2.23 to cancel and adding them to the x-direction simplification of Equation 2.22, an expression for the total force on the x-direction of a infinitesimal fluid volume can be found:

\[ F_x = F_{bx} + F_{sx} = \left[ \rho f_x - \frac{\partial p}{\partial x} + \frac{\partial \tau_{xx}}{\partial x} + \frac{\partial \tau_{yx}}{\partial y} + \frac{\partial \tau_{zx}}{\partial z} \right] (dx \cdot dy \cdot dz) \]  \hspace{1cm} (2.24)

Equation 2.24 can be thought of as the Left-Hand-Side (LHS) to Equation 2.21. The Right-Hand-Side (RHS) of the equation can be described with the acceleration being a substantial derivative of the velocity, \( \frac{Du}{Dt} \), and the mass being \( \rho (dx \cdot dy \cdot dz) \). The x-direction portion of the RHS of Equation 2.21 is thus:

\[ ma_x = [\rho (dx \cdot dy \cdot dz)] \left[ \frac{D^2 U}{Dt^2} \right] = [\rho (dx \cdot dy \cdot dz)] \left[ \frac{\partial (u)}{\partial t} + \nabla \cdot (uU) \right] = \left[ \frac{\partial (\rho u)}{\partial t} + \nabla \cdot (\rho uU) \right] (dx \cdot dy \cdot dz) \]  \hspace{1cm} (2.25)

Equating Equations 2.24 and 2.25, the x-component of the momentum equation can be found, and is shown in Equation 2.26. The y and z-components can be derived in a similar manner, and are described by Equations 2.27 and 2.28.

\[ \rho f_x - \frac{\partial p}{\partial x} + \frac{\partial \tau_{xx}}{\partial x} + \frac{\partial \tau_{yx}}{\partial y} + \frac{\partial \tau_{zx}}{\partial z} = \frac{\partial (\rho u)}{\partial t} + \nabla \cdot (\rho uU) \]  \hspace{1cm} (2.26)

\[ \rho f_y - \frac{\partial p}{\partial y} + \frac{\partial \tau_{xy}}{\partial x} + \frac{\partial \tau_{yy}}{\partial y} + \frac{\partial \tau_{zy}}{\partial z} = \frac{\partial (\rho v)}{\partial t} + \nabla \cdot (\rho vU) \]  \hspace{1cm} (2.27)

\[ \rho f_z - \frac{\partial p}{\partial z} + \frac{\partial \tau_{zx}}{\partial x} + \frac{\partial \tau_{zy}}{\partial y} + \frac{\partial \tau_{zz}}{\partial z} = \frac{\partial (\rho w)}{\partial t} + \nabla \cdot (\rho wU) \]  \hspace{1cm} (2.28)
2.2.3 The Energy Equation

The energy equation enforces the mathematical stipulation that the net flux of heat into and the rate of work done on (by both $F_b$ and $F_s$) a fluid CV must equal the rate of change of energy inside that CV [32]. However, since both isothermality and incompressibility apply to all the situations simulated in thesis, there is no need to solve the energy equation since there is no link between the energy and the momentum and continuity equations [31]. Because of this, the energy equation does not require discussion or derivation and will not be presented here.

2.2.4 Incompressible Newtonian Fluids

A Newtonian fluid is a fluid whose stress versus strain rate curve is linear and passes through the origin [33]. The Stokes equations for Newtonian stresses are found in Equations 2.29 to 2.34 below [31,32,33]:

\[
\begin{align*}
\tau_{xx} &= \lambda(\nabla \cdot \mathbf{U}) + 2\mu \frac{\partial u}{\partial x} & (2.29) \\
\tau_{yy} &= \lambda(\nabla \cdot \mathbf{U}) + 2\mu \frac{\partial v}{\partial x} & (2.30) \\
\tau_{zz} &= \lambda(\nabla \cdot \mathbf{U}) + 2\mu \frac{\partial w}{\partial x} & (2.31) \\
\tau_{xy} &= \tau_{yx} = \mu \left[ \frac{\partial v}{\partial x} + \frac{\partial u}{\partial y} \right] & (2.32) \\
\tau_{xz} &= \tau_{zx} = \mu \left[ \frac{\partial u}{\partial z} + \frac{\partial w}{\partial x} \right] & (2.33) \\
\tau_{yz} &= \tau_{zy} = \mu \left[ \frac{\partial w}{\partial y} + \frac{\partial v}{\partial z} \right] & (2.34)
\end{align*}
\]

Here $\mu$ is the molecular viscosity coefficient and $\lambda$ is second viscosity coefficient [34]. Since water is described as a Newtonian fluid due to its nearly incompressible properties, these Newtonian properties can be ascribed to the continuity and momentum equations to find a simplified continuity equation and the Navier-Stokes (N-S) equations [34]. If homogeneous and incompressible
assumption are used, the new continuity equation, Equation 2.35, becomes a simplified version of Equation 2.25:

\[ \nabla \cdot \mathbf{U} = 0 \quad (2.35) \]

### 2.2.5 The Navier-Stokes Equation

The x, y and z components of the isotropic Newtonian fluid N-S equation set can be found by simplification of the equations resulting from the insertion of Equations 2.29 to 2.34 into Equations 2.21 to 2.23. Due to Equation 2.35, the velocity must be non-divergent, allowing a simplification to be applied to Equations 2.29 to 2.31 which removes all terms containing \( \lambda \).

\[
\begin{align*}
\frac{\partial (\rho u)}{\partial t} + \nabla \cdot (\rho u \mathbf{U}) &= \frac{\partial}{\partial x} \left( \mu \left[ \frac{\partial v}{\partial x} + \frac{\partial u}{\partial y} \right] \right) + \frac{\partial}{\partial y} \left( \mu \left[ \frac{\partial u}{\partial y} + \frac{\partial w}{\partial z} \right] \right) + \rho f_x \quad (2.36) \\
\frac{\partial (\rho v)}{\partial t} + \nabla \cdot (\rho v \mathbf{U}) &= \frac{\partial}{\partial x} \left( \mu \left[ \frac{\partial v}{\partial x} + \frac{\partial u}{\partial y} \right] \right) + \frac{\partial}{\partial z} \left( \mu \left[ \frac{\partial w}{\partial y} + \frac{\partial v}{\partial z} \right] \right) + \rho f_y \quad (2.37) \\
\frac{\partial (\rho w)}{\partial t} + \nabla \cdot (\rho w \mathbf{U}) &= \frac{\partial}{\partial x} \left( \mu \left[ \frac{\partial u}{\partial z} + \frac{\partial w}{\partial x} \right] \right) + \frac{\partial}{\partial y} \left( \mu \left[ \frac{\partial w}{\partial y} + \frac{\partial v}{\partial z} \right] \right) + \rho f_z \quad (2.38)
\end{align*}
\]

Equations 2.36 to 2.38 can be transformed into a conservative equation after this simplification to give Equation 2.39 below, which is the N-S equation for incompressible, Newtonian fluids [31]:

\[
\frac{\partial (\rho u)}{\partial t} + \nabla \cdot (\rho \mathbf{U} \mathbf{U}) = -\nabla p + \mu \nabla^2 \mathbf{U} + \rho \mathbf{f} \quad (2.39)
\]

However, if, as is the case when a governing equation is included which models the transport of a density-altering solute mass fraction, the viscosity cannot be assumed constant throughout the flow field the N-S equation, Equation 2.39, is assumed to have the more general form of Equation 2.40 below [29,34,35]:

\[
\frac{\partial (\rho u)}{\partial t} + \nabla \cdot (\rho \mathbf{U} \mathbf{U}) = -\nabla p + \nabla \cdot \left( \mu (\nabla \mathbf{U} + (\nabla \mathbf{U})^T) \right) + \rho \mathbf{f} \quad (2.40)
\]

Where the superscript T signifies taking the transpose of a matrix, and the second term on the LHS and the matrix set in the second term on the RHS of Equation 2.40 each represent a vector [36].
2.2.6 The Diffusion-Convection Equation

An additional equation is required to describe the transport of solute between the CVs of a computational domain. This equation, called the diffusion-convection equation (D-C), is seen in Equation 2.41 below and describes the process in which a concentration value $c$ of solute ‘$i$’ is transferred from one CV to another due to the processes of diffusion and convection [35].

$$\frac{\partial c_i}{\partial t} + \nabla \cdot J_{tot} = S_i \quad (2.41)$$

The second term in the LHS of Equation 2.41 describes the net species flux for a considered CV, and the RHS term describes solute mass fraction generation or destruction within the CV. Since $J_{tot}$, the total flux through the membrane, can be assumed to originate from linearly independent diffusion and convection terms, Equation 2.41 becomes [35]:

$$\frac{\partial c_i}{\partial t} + \nabla \cdot (c_i U - D_{AB} \nabla c_i) = S_i \quad (2.42)$$

Where the first term in the gradient is the convective flux and the second term is determined by Fick’s law applied to inter-diffusion of binary mixtures (thus the $AB$ subscript of the diffusion coefficient) [22]. The assumptions made with Fick’s law above makes Equation 2.42 non-applicable for ionic flows or flows with more than one type of chemical solute, but works well for solutions with a high concentration ratio (the kind modeled in this work) [29].

2.3 Spacer Implementation

The effects and physics of implementing fluid flow spacing elements (generally referred to as baffles throughout this work) in narrow channels is well defined, and has appeared in literature for more than 40 years. The effects most commonly explored are fluid dynamic vortex shedding, wall shear stress, channel pressure loss and, in the case of spiral-wound membrane technology, mitigation of ECP effects [18,19,28,37]. The first two effects are not explored in this work, although further analysis of the simulations presented would allow exploration of (and optimization with respect to) these effects to be accomplished with minimal alteration to the current code. The surrounding and down-stream wall shear and flow mixing effects generated by a spacer in a narrow membrane
channel have two defining effects on the system: an overall increase in the water flux through the channel's membrane due to the breaking of the developed ECP boundary layer, and an overall increase in the pressure loss seen across the length of the membrane channel due to local $Re$ changes, friction losses and down-stream vortex interaction. These effects are almost entirely dependent on two factors: the Reynolds number of the free-stream, and the geometric implementation and shape of the spacers.

Narrow channel spacers are physically implemented in two main ways in a spiral-wound membrane: submerged and zigzag, depicted in Figure 2.6 below [18,19].

![Figure 2.6: Submerged and zigzag in-channel spacer orientations.](image)

Where $l$ is the center-to-center spacing between spacers, $d$ is the spacer diameter and $h$ is the overall channel height. CFD simulations conducted in geometric setups similar to those used in this work have found that the zigzag-type spacer implementation generally performs better than the submerged-type in channel pressure loss vs. free-stream Reynolds number, solute mass fraction reduction vs. free-stream Reynolds number and water permeation velocity vs. free-stream Reynolds number comparisons [19]. However, due to specific OpenFOAM meshing issues discussed in Sections 3.4.2.2 and 4.4 of this thesis, it was not possible to implement a zigzag-type spacer setup in the channel geometries. Thus, only the submerged-type spacers will be discussed in this work.

Furthermore, all sources to use either type of spacer orientation held $l$ constant for all adjacent spacers placed within the system [18,19,28,37,38]. This thesis will use a GA to affect these relative baffle-to-baffle spacings in an attempt to explore the possibility of finding optimal submerged-spacer positioning with respect to channel pressure loss and total water flux.
Although many spacer shapes and sizes have been defined and implemented in CFD simulations with similar geometries to those in this work (see [38]), only circular spacers with a $d/h$ ratio of 0.5 are considered in this thesis. Many of the geometries found in [38] performed better than the circular spacer in many situations. However, since OpenFOAM has a programmed inclusion for circular cylinders within the \texttt{snappyHexMesh} meshing utility used to automatically generate cases for the GA, the circular spacer was the most convenient option. Other spacer geometries can be implemented through the importation of a 3D, CAD-generated \texttt{.stl} file into the \texttt{snappyHexMesh} utility dictionary. The sizing of a $d/h$ value of 0.5 was used both because [38] found this sizing to be consistently near the asymptotic axis of a $d/h$ vs. specific energy consumption graph for several spacings in geometrically-different CFD setups (with smaller $d/h$ values being closer to said axis), and because the relatively large $d/h$ value ensures that the high spacer-local Reynolds numbers will produce significant downstream mixing effects for the low average x-direction velocities used in this work. The alteration of this $d/h$ value in the GA is trivial, allowing the effects of spacer diameter to be easily explored in future research.

One of the most important factors in the generation of down-stream effects is the Reynolds number of a narrow channel ($Re$) found in Equation 2.43. This equation must be altered for the consideration of spacer inclusion, and can be divided into the membrane-wall ($Re_w$) and channel ($Re_{ch}$) formulations found in Equations 2.44 and 2.45 respectively [19,37]. Practical applications of spiral-wound membranes modules can lead to $Re_{ch}$ numbers above 400, which produce non-steady down-stream effects for the type and size of spacer explored in this thesis [18,19].

\begin{align*}
Re & = \frac{4uh}{v} \quad (2.43) \\
Re_w & = \frac{hv}{v} \text{ at } y = 0, h \quad (2.44) \\
Re_{ch} & = \frac{d_hu}{v} \quad (2.45)
\end{align*}

where:

\begin{align*}
d_h & = \frac{2\varepsilon h}{(1-\varepsilon)(\frac{2h}{d})+1} \quad (2.46) \\
\varepsilon & = \frac{V_{tot}-V_{sp}}{V_{tot}} \quad (2.47)
\end{align*}
Complete mathematical descriptions of the effects that a spacer's local and down-stream flow has on the ECP and overall channel pressure loss $\Delta P$ (not to be confused with the trans-membrane pressure in RO, $\Delta \rho$) are exceedingly complicated, and are beyond the scope of this thesis. However, the $\Delta P$ for a channel with membranes as its top and bottom surfaces can be formulated as seen in Equation 2.48 below [37].

$$\Delta P = \left( \frac{\rho u^2}{2} \right) \left( \frac{24}{Re} - \frac{648 Re w}{35 Re} \right) \left( 1 - \frac{2x Re_w}{h Re} \right) \left( \frac{x}{h} \right) \quad (2.48)$$

If the membranes were to be treated as walls, this equation simplifies to the $\Delta P$ of Poiseuille flow. The $\Delta P$ value found from Equation 2.48 will be smaller than the $\Delta P$ found from simulations in which spacers are included in the flow field. Since this $\Delta P$ is directly proportional to the amount of hydraulic power a system would have to apply to maintain a steady state cross-flow within this channel, from an economic standpoint it would be favorable to minimize this $\Delta P$ value while still affecting an increase in overall water flux through the membrane. The relationship desired between $\Delta P$ and water flux with respect to a GA-usable objective function will be case specific, and will have to be assessed on a case-study basis.

Finally, it should be noted that two-dimensional (2D) CFD simulations of membrane-spacer channels such as the ones described in this work tend to find lower overall permeate mass transfer through the membrane and channel pressure loss than their otherwise geometrically-identical three-dimensional (3D) counterparts. Vorticity in the z direction is greater for 2D geometries: the end-wall effects which exist on the lateral sides of 3D geometries slow down the primary span-wise vortex, reducing the velocities at the top and bottom surfaces of the channel [28]. However, the 3D effects of mixing due to stream-wise vortices, the span-wise vortices being open and increased z-direction viscous membrane shear all decrease the boundary layer thickness found on the membrane's surface, resulting in more effective ECP reduction at the cost of a higher channel pressure loss [28].
2.4 Genetic Algorithm Optimization

The genetic algorithm was first introduced as an optimization tool by John Henry Holland in 1975 [39]. A GA is a robust heuristic search method that mimics the processes modeled by the theories of natural selection and evolution. The power of a GA derives from its ability to exploit, in a near-optimal fashion, information about the utility of a large number of structural configurations without the computational burden of explicit calculation and storage [40]. Candidate solutions for a particular optimization task are referred to as n-bit 'chromosomes', labeled \( \epsilon \), and are considered members of finite sets, or 'populations', labeled \( \text{Pop} \). In this work, \( \epsilon \) will always be a set of \( n \) positive decimals, making \( \text{Pop} \) a positive, real set. A GA necessarily contains a set of fitness functions \( F \) which must be maximized (or inversely minimized) through a probabilistic search which evolves \( \text{Pop} \). If \( F \) is a non-singular set, the GA is known as Multi-Objective (MOGA); otherwise it is a Single-Objective GA (SOGA) with a singular fitness function \( f \). A GA's \( \text{Pop} \) is iteratively assessed and changed until an overall convergence condition is met or until the maximum number of iterations is reached. A \( \text{Pop} \) is assessed via evaluations of its \( \epsilon \) through \( F \) or \( f \), and then altered through a set of selective operations in order to create an 'evolved' \( \text{Pop} \), which is then used as the original \( \text{Pop} \) when the above process is repeated.

A GA implements three main types of evolutionary operations before generating a new \( \text{Pop} \): mutation, crossover and fitness selection. The mutation operator, inspired by processes like radiation exposure or chemical miss-matching which alter genetic information in living organisms, will randomly alter the bits of existing \( \epsilon \), and occurs in proportion to a defined percentage mutation occurrence chance \( M\% \). The crossover operator, inspired by the genetic parent-child relationship which results from sexual reproduction, will choose (in a random or elitist fashion in proportion to a defined percentage crossover occurrence \( C\% \)) two \( \epsilon \) and 'mate' them by exchanging bits of each \( \epsilon \)'s bit vector. This will produce two children which will be included within the next population. Finally, \( \epsilon \)'s with a high fitness value are chosen via the fitness selection operator in a fashion similar to the principal of 'survival of the fittest'. The \( \text{Pop} \) for the subsequent iteration of the GA is made up of the \( \epsilon \)'s chosen or generated from the above operations.

As with any optimization method, a GA constraints the values that may vary in its iteration process; in this case the bit values each \( \epsilon \) contains. Without constraints on these variable's, the GA would have no search domain (feasible limits within which it may pull a random \( \epsilon \)) or useful patterns with
which to create $\alpha$ and thereby develop its $Pop$ lineage into a desirable set. The variables used in the GA developed in this thesis are subjected to double-sided inequality constraints. A description of all portions of this GA and the reasoning behind their selection can be found in Chapter 5 of this thesis.
Chapter 3

OpenFOAM 2.3.0 CFD Solution Implementation

This thesis employs the open-source CFD framework OpenFOAM to discretize the solution domain of and iteratively calculate the flow variables of all simulation geometries. OpenFOAM is a general purpose open-source C++ toolbox which is not only robust in its computational ability (it can solve simulations with complex fluid flows involving, but not limited to, chemical reactions, turbulence, heat and mass transfer, solid dynamics and electro-magnetics) but allows user-specified boundary conditions and solution domain solvers to be coded and used directly with its other utilities-a flexibility most commercial CFD frameworks lack [41]. OpenFOAM initially includes a set of solvers which are applicable to a wide range of basic problems and can be used as templates to generate a user-specific solver. Although portions of the work this thesis represents were completed using OpenFOAM version 2.1.1, version 2.3.0 (currently the newest version available) was chosen as the final source code framework due to various meshing utility updates.

Several script formats are used as notational elements throughout this thesis. They are as follows:

- *ItalicWords* 12PointGaramond: names of files or folders for the OpenFOAM or DAKOTA programs.
- **InText**12PointCorbel: names of utilities or programs employed by OpenFOAM or DAKOTA.
- 8 point Corbel outside of text paragraphs: abbreviated code of files used by OpenFOAM or the DAKOTA GA,

For the complete source code of all OpenFOAM files discussed in this section, see Appendixes A to C.
3.1 Solution Domain

The first mathematical step any CFD software must take when initializing a problem is to create a discretized solution domain in which the required governing partial differential equation set may be formulated as algebraic relations, allowing them to be solved in a computationally feasible manner. The equation set relating to this work defines fluid and solute mass fraction flow dynamics, and is presented in Section 2.2. A solution domain (a case’s defined geometry) is discretized by being divided into defined locations in both time (temporal steps which may either be based on the mesh's base Courant number (see Section 4.1) and therefore variable, or fixed if the solver is steady state) and space (a computational grid made up of CVs or cells) in and between which the governing equation set can subsequently be solved. OpenFOAM uses the FVM approach to generate a computational grid that sub-divides the case geometry into a finite number of CVs which completely fill the computational domain and do not overlap. Each CV within this domain is bounded by a set of flat surfaces called faces. Each face of a CV may only contact one neighboring CV face unless its face is on a boundary. Thus, each CV may be described as a polyhedron of any size, allowing both 2D and 3D generation of arbitrarily unstructured meshes to be easily realized. It is also simple to break existing CV cells into smaller cells if local mesh refinement is necessary. The governing equations may then be applied to these cells as algebraic relationships which define and transmit flow variable information between neighboring cells through either body or face values. Body values are stored at the centroid of the cell, known as point $P$ (for the considered cell) or $N$ (for a neighboring cell), and include pressure, mass and velocity vectors. Face, or surface, values are either stored on cell faces, as is the case of flux values, or are body values which have been interpolated through vector relations to the cell face. The generalized relation for the location of the centroid of a CV is found in Equation 3.1 below:

\[ \int_{V_P} (x - x_P) dV = 0 \]  \hspace{1cm} (3.1)

Here, $x$ denotes variable position within the cell, and $x_P$ is the position of the centroid $P$.  

27
3.1.1 Diffusion-Convection Equation Discretization

The discretization of the basic governing equations for fluid dynamics (the continuity and N-S equations) is well known for the FVM, and will therefore not be discussed in this work. However, the D-C equation, which is not included in any original OpenFOAM solver, will be discussed as a theoretical discretization example. For convenience, the D-C equation will be re-written here as Equation 3.2 using the concentration relationship \( c = \rho m_A \) where \( m_A \) is the dimensionless solute mass fraction (a cell-body value). Using this mass fraction description is useful for both describing the discretization procedures of OpenFOAM and therefore developing a weakly-compressible equation set which is usable in OpenFOAM functionality [29,42]. This is due to the fact that OpenFOAM's FVM approach is based on discretizing the integral form of the equations being solved over each cell; a method which conserves basic quantities such as mass and momentum at a discrete level.

\[
\frac{\partial \rho m_A}{\partial t} + \nabla \cdot (\rho \mathbf{U} m_A) - \nabla \cdot (\rho D_{AB} \nabla m_A) = S_{m_A}(m_A) \tag{3.2}
\]

Equation 3.2 must be satisfied over each CV of volume \( V_P \) in the solution domain during each time step \( \Delta t \), transforming it into Equation 3.3 below:

\[
\int_t^{t+\Delta t} \left[ \frac{\partial}{\partial t} \int_{V_P} \rho m_A dV + \int_{V_P} \nabla \cdot (\rho \mathbf{U} m_A) dV + \int_{V_P} \nabla \cdot (\rho D_{AB} \nabla m_A) dV \right] dt = \\
\int_t^{t+\Delta t} \left[ \int_{V_P} S_{m_A}(m_A) dV \right] dt \quad (3.3)
\]

The accuracy of the discretization method used for the terms in Equation 3.3 inherently depends on the assumed variation of the terms' variables. For \( m_A \), which is variable about the cell centroid, second order accuracy achieved by Taylor series expansion is possible only if its variation within the CV is considered linear in both time and space, as seen in Equations 3.4 and 3.5 below [43].

\[
m_A(x) = (m_A)_P + (x - x_P) \cdot (\nabla m_A)_P \tag{3.4}
\]

\[
m_A(t + \Delta t) = m_A(t) + \Delta t \cdot \left( \frac{\partial m_A}{\partial t} \right)_t \tag{3.5}
\]
Equations 3.1 and 3.4 allow a calculation of the $m_A$ volume integral within any CV, shown in Equation 3.6.

$$\int_{V_p} m_A(x) dV = \int_{V_p} (m_A)_p dV + \left[ \int_{V_p} (x - x_p) dV \right] \cdot (\nabla m_A)_p = (m_A)_p V_p \quad (3.6)$$

As a final look into OpenFOAM's FVM, Gauss's theorem will be discussed. Gauss' theorem may be defined in a discretized form which OpenFOAM can use to convert CV body-values to surface-values, and will be used to discretize several terms in the D-C equation [43]. This relationship, found in Equation 3.7, is used by the program during calculation procedures, and will not be explained in a theoretical sense in this work.

$$\int_{V_p} \nabla \cdot adV = \sum_f S_f \cdot a_f \quad (3.7)$$

Here, $a$ is a place-holder variable and subscript $f$ signifies a defined set of CV face values (this could also be stated as: $S_f$ denotes a set of outwardly-facing-normal vectors which have magnitudes equal to their faces) [43]. Each face in the computational grid has an 'owner' and 'neighbor' cell: $S_f$ is defined such that it points from the owner and into the neighbor (the $S_f$ for the neighbor would be a negative outwardly-facing-normal vector).

### 3.1.1.1 Convection Term

Equation 3.7 can be used on the convection term in Equation 3.3 to find its discretized form:

$$\int_{V_p} \nabla \cdot (\rho U m_A) dV = \sum_f S_f \cdot (\rho U m_A)_f = \sum_f \Phi_f \cdot (m_A)_f \quad (3.8)$$

Where $\Phi_f$ is the mass flux through face $f$ as seen in Equation 3.9. Further discussion on this term and how it is used by OpenFOAM can be found in Section 3.3.2.

$$\Phi_f = S_f \cdot (\rho U)_f \quad (3.9)$$

The interpolated CV solute mass fraction face values, $(m_A)_f$, found in Equation 3.8 can be obtained through one of the many differencing schemes OpenFOAM provides. The chosen
schemes for all terms in the governing equations are contained in the *fvSchemes* file of an OpenFOAM case file system: refer to Section 3.4.1. It should be noted that, physically, the convection term may never violate the initial numerical 0 to 1 bounds given by its solute mass fraction term, forcing the differencing scheme to produce values between zero and unity [43].

### 3.1.1.2 Diffusion Term

Using the same Gaussian application, the discretized form of the diffusion term in Equation 3.3 is found to be:

\[
\int_{V_p} \nabla \cdot (\rho D_{AB} \nabla m_A) dV = \sum_f (\rho D_{AB})_f S_f \cdot (\nabla m_A)_f \tag{3.10}
\]

The term \( S_f \cdot (m_A)_f \), can be calculated from the centroid values of the owner and neighbor cells if their shared face plane is orthogonal to the vector \( d \) between them, as seen in Equation 3.11 below.

\[
S_f \cdot (\nabla m_A)_f = |S_f| \frac{(m_A)_N - (m_A)_P}{|d|} \tag{3.11}
\]

If \( d \) and the shared face plane are non-orthogonal, as seen in Figure 3.1 on the following page, this term must instead be decomposed into orthogonal and non-orthogonal contributions. The non-orthogonal contributions must then be corrected via application of various possible correction schemes [43]. Although all the meshes generated in this work are originally orthogonal, non-orthogonal cell relations will develop upon introduction of the mid-channel baffles. However, since it was found that adding non-orthogonal corrections into simulation runs had no positive effects on either the solution obtained or the total runtime, they were omitted. Thus, non-orthogonal correction theory is not relevant to this work and will not be discussed.
3.1.1.3 Source Term

The source term describes the generation or destruction of solute mass fraction values, and must be written as a general function since the causes of such additions or subtractions are case and process specific. The source term from Equation 3.2 may be specified either explicitly or implicitly. For explicit specification, the source term is simply implemented such that only the LHS of Equation 3.2 has to be discretized [42]. In implicit specification, the source term must be first linearized in the form seen in Equation 3.12 [42]:

\[ S_{m_A}(m_A) = S_u + S_p m_A \]  \hspace{1cm} (3.12)

where both \( S_u \) and \( S_p \) are linearized coefficients that may explicitly depend on \( m_A \) [43]. This formulation can be described as a volumetric CV integral through application of Equation 3.6, as seen in Equation 3.13. The implicit method is generally advised for describing introduced source terms as it invites both accuracy and convergence [32]. However, if this method makes \( S_p > 0 \), effectively decreasing the diagonal dominance of the final matrix, the iterative solution may become unstable [42]. If this occurs, a 'mixed-method' is provided in the OpenFOAM framework which uses the explicit specification for the time steps which would otherwise be unstable [41].
\[ \int_{V_P} S_{m_A}(m_A) dV = S_u V_P + S_p V_P m_A \]  

(3.13)

### 3.1.1.4 Temporal Term

Equations 3.5 and 3.6 can be used to discretize the temporal term of Equation 3.3, resulting in Equation 3.14.

\[ \frac{\partial}{\partial t} \int_{V_P} \rho m_A dV = \frac{\partial \rho m_A}{\partial t} \bigg|_{V_P} V_P = \frac{\rho^p(m_A)^n - \rho^p(m_A)^o}{\Delta t} V_P \]  

(3.14)

Where the superscripts \( \sigma \) and \( \sigma_0 \), described in relation to the solute mass fraction via Equations 3.15 and 3.16 below, denote old and new time steps.

\[ (m_A)_P^\sigma = (m_A)_P(t + \Delta t) \]  

(3.15)

\[ (m_A)_P^\sigma_0 = (m_A)_P(t) \]  

(3.16)

Applying the same method used in finding Equation 3.14, the discretized form of the temporal integral found in Equation 3.3 becomes:

\[ \int_{t}^{t+\Delta t} m_A(t) dt = \frac{(m_A)_P^0 + (m_A)_P^n \Delta t}{2} \]  

(3.17)

Finally, the chosen Crank-Nicholson discretization formulation of the D-C equation can be formed by assuming that \( D_{AB} \) and \( \rho \) are temporally constant and combining Equations 3.3, 3.8, 3.10, 3.13, 3.14 and 3.17. This formulation, seen in Equation 3.18, was chosen because unlike many other well-known formulations, it does not ignore the time variation of \( m_A \) and \( \nabla m_A \) face values [43].

\[ \frac{2(\rho^p(m_A)_P^n - \rho^p(m_A)_P^0)}{\Delta t} V_P + \Sigma_f \left[ \Phi_f \left( (m_A)^n_f + (m_A)^o_f \right) - \left( \rho D_{AB} \right)_f S_f \cdot \left( (\nabla m_A)^n_f + (\nabla m_A)^o_f \right) \right] = 2S_u V_P + S_p V_P (m_A)^n_P + S_p V_P (m_A)^o_P \]  

(3.18)

Equation 3.18 requires the computation of the new solute mass fraction values \( m_A^n \) which, when the fact that all face values are determined from neighboring cell values is taken into account, can be found by Equation 3.19:
\[ a_P (m_A)_P^n + \sum_N a_N (m_A)_N^n = R_P \]  

(3.19)

where \( a_P \) and \( a_N \) are coefficients that contain the discretization information of Equation 3.18, and \( R_P \) is the RHS of Equation 3.18. Equation 3.19 is formulated for each CV within the grid, creating a system of algebraic equation described by a matrix of \( a_P \) and \( a_N \) coefficients, called \( A \), and the vectors \( [m_A] \) and \( [R] \): the vectors containing the \( m_A \) and \( R_P \) values corresponding to the 'a' coefficients for each cell. This system of equations, which is used as an information input into any OpenFOAM solver, is described by Equation 3.20 below.

\[ \underline{A}[m_A] = [R] \]  

(3.20)

Equation 3.20 pertains to the solute mass fraction flow variable: each flow variable used by an OpenFOAM solver will have an equation system in the form of Equation 3.20 (see Section 3.3.1).

### 3.1.2 General Boundary Conditions

In order for a set of governing equations to be solved across a computational grid, Boundary Conditions (BC) must be imposed upon all external edges of the grid, as well as any portions of the inner grid which require user specified properties (such as a membrane or internal wall). Due to their direct effect on the computational flow field and the fact that they are usually based upon physical conditions, the BCs of a case effectively dictate what solution is obtained by the case's solver. OpenFOAM BCs are coded directly into the file system of the case prior to it being either discretized or solved. With respect to flow field variables, two main BC formulations exist. The first, Dirichlet, is a fixed-value BC which can be set to either a zero or non-zero value. The second, von Neumann, is a fixed-gradient BC which allows the specification of a zero or non-zero flow variable gradient through the boundary. To discuss how these two main BCs are handled by OpenFOAM, a general, non-orthogonal CV with at least one face belonging to a boundary must first be considered. This cell will be used to demonstrate both BCs with respect to the discretized convection and diffusion terms of the D-C equation.

The general, non-orthogonal cell has centroid \( P \) and boundary face \( b \), as shown in Figure 3.2 on the following page. The normal vector from \( P \) to \( b \) will be defined as \( d_n \) and is parallel to the boundary face's normal vector \( S \). Since the vector from \( P \) to the center of \( b \) (which is from where \( S \) originates: see Section 3.1.1), labeled \( d \), is known, \( d_n \) can be calculated with Equation 3.21.
Figure 3.2: Boundary face of CV within a discretized field.

\[ d_n = \frac{(d \cdot s)S}{|s||s|} \quad (3.21) \]

### 3.1.2.1 Dirichlet Boundary Condition

This BC specifies that the value of the general flow variable \( \phi \) (note that flux is \( \Phi \)) will be \( \phi_b \) on the boundary face \( b \). In terms of \( \phi \), the general face form of the discretized convection term of Equation 3.8 is:

\[
\int_{V_P} \nabla \cdot (\rho \mathbf{U} \phi) dV = \sum_f \Phi_f \cdot \phi_f \quad (3.22)
\]

where the \( \phi \) subscript \( f \) becomes \( b \) for the boundary face. The same form can be found for the diffusion term in Equation 3.10, using \( D \) to denote the general diffusion coefficient for the \( \phi \) quantity.

\[
\int_{V_P} \nabla \cdot (\rho D \nabla \phi) dV = \sum_f (\rho D)_f S_f \cdot (\nabla \phi)_f \quad (3.23)
\]

The gradient of \( \phi \), which must be found in order to compute the RHS of Equation 3.23, is calculated for boundary terms via Equation 3.24 by using the fact that \( S \) and \( d_n \) are parallel.
\[ S \cdot (\nabla \phi)_b = |S| \frac{\phi_b - \phi_P}{|d_n|} \]  

(3.24)

### 3.1.2.2 von Neumann Boundary Condition

This BC specifies that the gradient \( g \) of the general flow variable \( \phi \) on the boundary \( b \) will be specified by Equation 3.25.

\[ g_b = \left( \frac{S}{|S|} \cdot \nabla \phi \right)_b \]  

(3.25)

For the \( \phi \) formulation of the convection term defined by Equation 3.8, this BC stipulates that \( \phi \) at the boundary must be interpolated from the cell centroid, as seen in Equation 3.26 below.

\[ \phi_b = \phi_P + |d_n| g_b \]  

(3.26)

For the \( \phi \) formulation of the diffusion term defined by Equation 3.10, this BC is implemented into the system of discretized governing equations through Equation 3.27 below.

\[ S \cdot (\nabla \phi)_b = |S| g_b \]  

(3.27)

### 3.2 Forward Osmosis Asymmetric Membrane Implementation

#### 3.2.1 Weakly Compressible Condition

When modeling FO systems, the fact that the solution being modeled and constrained by boundary conditions is aqueous in nature must be addressed: the solute mass fraction scalar associated with it that may vary throughout the flow field. The equations governing the flow of the aqueous fluid solution include the mass and momentum conservation equations, seen in Equations 3.28 and 3.29, and a convection-diffusion transport equation, Equation 3.30, which allows transport of the solute mass fraction scalar between the mesh cells used by the OpenFOAM's FVM. The flow is assumed to be isothermal, laminar and incompressible (thus the density and pressure are not coupled). However, several model parameters, including density and viscosity, are dependent on the solute mass fraction \( m_A \). This leads to a semi-compressible (or weakly-compressible) set of flow equations:
\[
\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{U}) = 0 \quad (3.28)
\]

\[
\frac{\partial \rho \mathbf{U}}{\partial t} + \nabla \cdot (\rho \mathbf{U} \mathbf{U}) = \nabla \cdot [\mu (\nabla \mathbf{U} + (\nabla \mathbf{U})^T)] - \nabla p + \rho \mathbf{g} \quad (3.29)
\]

\[
\frac{\partial \rho m_A}{\partial t} + \nabla \cdot (\rho m_A \mathbf{U}) - \nabla \cdot [\rho D_{AB} \nabla m_A] = 0 \quad (3.30)
\]

The model developed by Wiley and Fletcher to study RO membrane flow processes, which was successfully verified by comparison with well-accepted analytical and semi-analytical models, is the main inspiration for the model developed here [19,20]. Their results highlighted the danger of assuming the viscosity and solute diffusivity to be constant, as such simplifications may grossly misrepresent reality near the membrane where the solute concentration is much different from that of the bulk.

The model assumes that the only body force on the fluid is gravity, and that no solute is generated or destroyed (i.e., that the source term of the D-C equation is zero). The density, viscosity and diffusion coefficient terms found in the weakly-compressible model, as well as the osmotic pressure over the membrane, are affected by the solute mass fraction, and will therefore depend upon its value and vary between cells. The empirical equations used to relate these flow variables to the solute mass fraction were taken from [44], and hold for NaCl solutions at 25°C with solute concentrations at or below 1.6M (a mass fraction of 0.09). They can be seen in Equations 3.31 to 3.34 below.

\[
\rho = 997.1 + (694)m_A \quad (3.31)
\]

\[
\mu = 0.89 \times 10^{-3} + (1.45 \times 10^{-3})m_A \quad (3.32)
\]

\[
D_{AB} = \max(1.61 \times 10^{-9}(1 - 14m_A) , 1.45 \times 10^{-9}) \quad (3.33)
\]

\[
\pi = (805.1 \times 10^5)m_A \quad (3.34)
\]

This solute dependency instantiates a coupling between all three governing equations, requiring the
D-C equation to be solved in the PISO loop of the created solver (see Section 3.3.2 for information on OpenFOAM's PISO algorithm). The model described in Equations 3.28 to 3.30 is also dependent on the assumption that the continuum hypothesis, which states that all flow variables are continuous, holds throughout the discretized flow domain [45]. Since the cases run in this work are on the mm scale they are within the limits of this hypothesis since the size of the fluid particles does not approach the size of the computational grid cells, making its assumption is valid [32,45].

3.2.2 Membrane Boundary Conditions

All the asymmetric AL-FS membranes modeled within this Thesis' cases are considered 2D planes, and as such do not attempt to resolve or model any internal membrane effects. For FO, a net pressure of zero can be assumed across the membrane, requiring only velocity and solute concentration BCs to be uniquely specified on the planar surfaces of the membranes. Due to the asymmetry of the membrane, its BCs' interaction with and resulting effect on the surrounding flow field will differ depending on whether the BC in question is on the rejection or support side of the membrane. This requires that a user-specified 'forward-facing' vector relation must be implemented within the membrane. Although the membrane model presented here has the ability to model a slip velocity on the surface of the membrane's support layer as used in the work of Gruber et al. [17], it should be noted that all verification and modeling simulations conducted in this paper use the no-slip version of this velocity BC. In addition, the membrane surface roughness is also neglected as it is conjectured that the effect of membrane roughness on trans-membrane flux would be negligible [17].

The Al-FS membrane file system constructed in this work was implemented by using existing OpenFOAM BC files as templates. The .H and .C files are compiled into a user-specified library, which is then imported into the file system of an OpenFOAM case. As mentioned earlier, in the case's discretized geometry, this boundary must be internal and width-less. OpenFOAM constructs this internal boundary, known as a baffle, through pairing sets of CV faces which share a plane, and then uses the positional information of those sets to place flow-information altering conditions on that baffle-plane. This form of CV set pairing can be seen in Figure 3.3 on the following page.
Thus, each CV on one side of the internal face should have a twin on the opposite side of the boundary with which it can exchange BC-edited information. The information interpolated to the baffle-faces of these CV pairs must be transferred to the membrane BCs so that the flow variables can be appropriately be altered by the mathematical BC assessment and output back to the CV pairs to affect the surrounding flow field. Within the OpenFOAM boundary condition format, this cell pairing is achieved by a simple loop over all the faces assigned to the membrane as follows:

```cpp
// Set up the face-index mapping based on cell centers:
const vectorField& cfvf = patch().Cf();
// Loop through all the membrane faces:
forAll(cfvf, facei)
{
    // For each facei in the prior loop, loop through the other faces again to find the closest:
    for(label i=0; i<cfvf.size(); i++)
    {
        if (facei!=i)
        {
            // Check face-to-face distance to see if it is small enough:
            if (mag(cfvf[facei]-cfvf[i])<1e-9)
            {
                fm_[facei]=i;
                if(debug)
                {
                    // Debug statement here.
                    break;
                }
            }
        }
    }
}
```

Where it should be noted that the `//` symbol found in the above code denotes a user-commented line in C++ syntax and should therefore be ignored as part of the OpenFOAM-interpretable code.
The cell-pair list created by this code is saved in a face matched list 'fm_' from which OpenFOAM can retrieve pairings between which to input and output flow variable information.

3.2.2.1 Velocity Boundary Condition

If the effects of the porous support layer within the membrane are taken into account, the water flux through the membrane can be modeled using the linear relationship between the solute concentration and the osmotic pressure which was discussed in Section 2.1.3 of this work. For convenience, the AL-FS water flux equation previously derived there is repeated here as Equation 3.35 below.

\[ J_w = \frac{1}{K} \ln \left( \frac{B + A \pi_{d,m}}{B + |J_w| + A \pi_{f,m}} \right) n_d \]  

(3.35)

Where \( A \), \( B \) and \( K \) are the pure water permeability, solute permeability and porous support diffusion resistivity membrane parameters which are defined in Equations 3.36 to 3.38. These values, appropriated from [44], are held constant for all cases run in this work and were experimentally verified against models similar to that of this work via physical FO membrane setups [17].

\[ A = 1 \times 10^{-12} \left[ \frac{m}{s \cdot Pa} \right] \]  

(3.36)

\[ B = 1 \times 10^{-7} \left[ \frac{m}{s} \right] \]  

(3.37)

\[ K = 0.5 \left[ \frac{s}{\mu m} \right] \]  

(3.38)

Due to the many required C++ constructors and initializers required by general OpenFOAM BC source code, only the code specific to the implementation of Equation 3.35 will be discussed in this work. If the coefficients of a flow variable matrix must be updated to obey a BC, OpenFOAM will generally call a function called \texttt{updateCoeffs()}, which is a member function of the \texttt{fixedValueFVPatch} OpenFOAM object. This object takes the calculated values for velocity on each face and uses the basic OpenFOAM implementation of the Dirichlet BC to update matrix coefficients according to calculated fixed velocity values. The general code format of the
updateCoeffs() function with respect to the FO membrane velocity BC is as follows:

```cpp
// fixedValueFvPatch object::updateCoeffs()
{
    // Make sure only one update occurs:
    if(updated()) { return; }

    // get the temporary face-normal vector field:
    tmp<vectorField> tvfnf = patch().nf();
    const vectorField& vfnf = tvfnf();

    if(mAName_ == "none")
    {
        // Special case for no solute here.
    }

    else
    {
        // Scalar definitions here.

        // Get the mass fraction field:
        const fvPatchScalarField& mA = patch().lookupPatchField<volScalarField , scalar>(mAName_);
        const scalarField& magSf = patch().magSf();

        // Get the current internal velocity field:
        const fvPatchVectorField& Ufield = patch().lookupPatchField<volVectorField , vector>("U");
        const vectorField& internalU = temp();

        // Get cell-center distances:
        const scalarField deltas = 1.0/patch().deltaCoeffs();

        // Loop over all membrane faces:
        forAll(fs_, facei)
        {
            // Face label definitions here.

            // Use Ridder's method to solve for the flux here.

            // Calculate the velocity through the asymmetric membrane:
            vector v = vfnf[fsi] * flux;

            // Set the feed-side velocity:
            operator[](fsi) = v;

            // Correct the velocity due to density change:
            v *= (1.0 + rho_mACoeff_.value() * feedMem)/(1.0 + rho_mACoeff_.value() * drawMem);

            // Calculate the total flux per time and area:
            totalMassFlux += flux
        }
    }
}
```
* rho0_.value()
* (1.0 + rho_mACoeff_.value() * feedMem)
* magSf[dsi];
Flux = totalMassFlux/(sum(magSf)/2)*3600;

/***********************************************************/
// Slip boundary condition:
if( slipType() == "slip" )
{
    // Special porous support slip condition case here.
}
// Set the draw-side velocity:
operator[](dsi) = v + slipUbounary;
// Information output to terminal and written files here.
} fixedValueFvPatchVectorField::updateCoeffs();
}
The asymmetry of the AL-FS FO membrane requires Equation 3.35 to be solved implicitly. As seen in the above function code, this is achieved through the custom implementation of Ridder's numerical root-finding algorithm, which is applied to each membrane face pair to obtain the membrane permeation velocity. The root-finding method iteratively solves Equation 3.35 for the water flux, and is bounded by a required accuracy and upper and lower flux values. The code pertaining to Ridder's root-finding method can be found in Appendix A. In order to solve the water flux equation, the Ridder method function ridderSolve() calls the fluxEquation() function, which contains equations pertaining to two possible water flux cases: a general, 'advanced-flux' case which uses Equation 3.35, and a 'simple-flux' case which uses a reduced form of Equation 3.35 (See section 2.1.4). The general code format of the fluxEquation() function with respect to the FO membrane velocity BC can be seen on the following page.
3.2.2.2 Solute Boundary Condition

Having formulated the equation for water flux through the AL-FS FO membrane, it is possible to write the solute flux through the membrane as a function of the water flux. This definition was discussed in Section 2.1.4, but is repeated for convenience below as Equation 3.39.

\[
J_s = - \frac{B}{\phi A} J_w \quad (3.39)
\]

\[
\phi = \frac{\pi}{C} = 805 \times 10^2 \left[ \frac{Pa \cdot m^3}{kg} \right] \quad (3.40)
\]

Where \( \phi \) is the linear proportionality factor between the osmotic pressure and the solute concentration within the membrane structure. Knowing the solute flux makes it possible to write a
mixed Dirichlet and von Neumann BC (called a Robin BC) for the solute mass fraction by noting that the convective and diffusive fluxes must be balanced with the solute flux [12]. This BC is stated below in Equation 3.41.

\[
\rho_m \left( -D_{AB} \frac{\partial m_A}{\partial n_d} n_d + m_{A,m} J_w \right) = J_s \quad (3.41)
\]

Where the normal component of the pressure gradient with respect to the membrane is assumed to be zero.

Since the solute balance BC in Equation 3.41 has both Dirichlet and von Neumann requirements, both value and gradient flow field information must be supplied in order for the solute BC to update flow variable matrices. These values can be supplied to the baffle boundary through the use of four coefficient fields: the valueInternalCoeffs (VIC), valueBoundaryCoeffs (VBC), gradientInternalCoeffs (GIC) and gradientBoundaryCoeffs (GBC) fields. These fields effectively work to set the matrix coefficients such that the field values and gradients on the baffle boundary faces become:

\[
\text{Boundary Value} = \text{VIC} \cdot (\phi_P) + \text{VBC} \quad (3.42)
\]

\[
\text{Boundary Gradient} = \text{GIC} \cdot (\phi_P) + \text{GBC} \quad (3.43)
\]

where \( \phi_P \) is the inter field value of the adjacent CV that 'owns' the specific membrane boundary face. All the solute mass fraction BCs given in Equation 3.41 can be written in the general form

\[
Y \frac{\partial m_A}{\partial n} + \Psi m_A = \Lambda \quad (3.44)
\]

where \( Y, \Psi \) and \( \Lambda \) are known values and \( n \) being the outward-surface-normal vector from the boundary-face-owning CV. Equation 3.44 can be approximated and rearranged to find the solute mass fraction on the boundary [17]:

\[
(m_A)_B = \frac{(m_A)_P}{1 + \left| \frac{d_n}{\sqrt{VIC}} \right|} + \frac{\Lambda}{\left[ \left| \frac{d_n}{\sqrt{VIC}} \right| + \Psi \right]} \quad (3.45)
\]
Equation 3.44 can be rearranged into Equation 3.46 upon the boundary surface:

$$\left(\frac{\partial m_A}{\partial n}\right)_b = \frac{\Lambda - \Psi (m_A) n}{Y}$$  \hspace{1cm} (3.46)$$

which, when combined with Equation 3.45, formulates the expression for the solute mass fraction on the surface of the membrane baffle found in Equation 3.47 below [17].

$$\left(\frac{\partial m_A}{\partial n}\right)_b = \frac{(m_A) p}{\left(\frac{\rho}{\psi} + |d_n| \right)} + \frac{\Lambda}{Y + \psi |d_n|}$$  \hspace{1cm} (3.47)

Between Equations 3.45 and 3.47, all four coefficient fields can be calculated, allowing the implementation of Equation 3.41 (in the form of Equation 3.44) within a solute updateCoeffs() function similar to that required by the membrane velocity BC, except that it is a member function of the fvPatchField OpenFOAM object. The abbreviated general code format of this function is as follows (the full file system for all membrane BCs can be found in Appendix A):

```c++
// fvPatchField object::updateCoeffs()
{
    // Make sure only one update occurs here.
    
    // Execute the change to the openFraction only once per time-step:
    if (curTimeIndex_ != this->db().time().timeIndex())
    {
        // Get the surface-Vector geometric field:
        const vectorField& Sf = patch().Sf();

        // Get cell-center distances:
        const scalarField deltas = 1.0/patch().deltaCoeffs();
        tmp<vectorField> tvfnf = patch().nf();
        const vectorField& vfnf = tvfnf();

        // Velocity field along surface normal:
        const fvPatchField< vector >& upvf = patch().lookupPatchField< volVectorField, vector >(UName_);
        scalarField magU = max( mag( cmptMultiply( upvf , vfnf ) ) , VSMALL );

        // BC debugging fields:
        const fvPatchScalarField& mA = patch().lookupPatchField<volScalarField, scalar>("mA");
        tmp<scalarField> temp = mA.patchInternalField();
        const scalarField& mAInternal = temp();

        // Specify equation and debug variables here.
    }
}
```
// Loop over all membrane faces:
forAll( patch(), facei )
{
    // Determine the coefficients for Equation 3.34 (A*(dmA/dn)+B*mA = Js):
    rho = rho0_.value() * ( 1.0 + rho_mACoeff_.value()*operator[](facei) );
    A = rho * max( DabCoeff_ * ( 1.0 - Dab_mACoeff_*operator[](facei) )
                  , DabMin_ ).value();

    //for the case that the flow is from the draw to the feed:
    if ( (upvf[facei]&Sf[facei]) <= 0.0 )
    {
        // Variable B and salt flux:
        B = rho*magU[facei];
        Js = ( B_ * magU[facei] ) / ( pi_mACoeff_.value() / 1000 * A_ );

        // Set coefficients:
        VIC_[facei] = 1.0 / (1.0 + B*deltas[facei]/A);
        VBC_[facei] = Js / (A/deltas[facei] + B);
        GIC_[facei] = -1.0 / (A/B + deltas[facei]);
        GBC_[facei] = Js/( A + B*deltas[facei] );

        // Tests if A*(dmA/dn)+B*mA = Js is satisfied:
        newGrad = GIC_[facei]*mAInternal[facei]+GBC_[facei];
        DrawMassInbalance = A*newGrad+operator[](facei)*B - Js;
    }

    //for the case that the flow is from the feed to the draw:
    else
    {
        // Variable B and salt flux:
        B = -rho*magU[facei];
        Js = ( B_ * magU[facei] ) / ( pi_mACoeff_.value() / 1000 * A_ );

        // Set coefficients:
        VIC_[facei] = 1.0 / (1.0 + B*deltas[facei]/A);
        VBC_[facei] = Js / (A/deltas[facei] + B);
        GIC_[facei] = -1.0 / (A/B + deltas[facei]);
        GBC_[facei] = Js/( A + B*deltas[facei] );

        // Tests if A*(dmA/dn)+B*mA = Js is satisfied:
        newGrad = GIC_[facei]*mAInternal[facei]+GBC_[facei];
        FeedmassInbalance = A*newGrad+operator[](facei)*B - Js;

        // Add to salt flux through membrane:
        totalWeightFlux += patch().magSf()[facei] * Js;
    }
}

// Information output to terminal and written files here.

// Set the time index:
curTimeIndex_ = this->db().time().timeIndex();
3.3 Weakly Compressible Solver Implementation

3.3.1 Pressure-Velocity Coupling between the Governing Equations

The continuity and N-S equations, listed below as Equations 3.48 and 3.49 below for convenience, demonstrate a strong coupling of the velocity and pressure fields. This is because it is necessary to first know the pressure in order to evaluate the velocity, and vice versa.

\[
\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{U}) = 0 \quad (3.48)
\]

\[
\frac{\partial \rho \mathbf{U}}{\partial t} + \nabla \cdot (\rho \mathbf{U} \mathbf{U}) = \nabla \cdot [\mu \nabla \mathbf{U} + \nabla \mathbf{U}^T] - \nabla p + \rho g \quad (3.49)
\]

Solution generation of these terms is further complicated by the non-linear velocity term in Equation 3.49 since discretization of the N-S equation generates quadratic velocity terms that cause the resulting system of algebraic equations to be non-linear. Linearization of this non-linear term is preferable since non-linear solvers generally have heavy computational requirements [46,47]. The matrix coefficients belonging to the velocity components are thus calculated using values from the previous time step, generating a computational lag behind the non-linear term [48]. This lag effect allows iterative calculations of the non-linear terms in each time step, and is used in transient simulations to achieve accuracy if the time step is not small enough.

3.3.2 The PISO Algorithm

The PISO (Pressure Implicit Splitting of Operators) algorithm is one of the most well-accepted transient algorithms found in the OpenFOAM framework that can solve the velocity and pressure coupling found between the continuity and N-S equations. It also is easily able to deal with the non-linearity of the convection term. The algorithm segregates the pressure and velocity solving procedures in an iterative fashion; solving one field, then plugging these calculated values into the equation set to be used as constants in the calculation of the other field. This process is repeated until an acceptable solution tolerance is reached. As with the solver produced by [17], This PISO algorithm forms the fundamental base in the membrane separation model developed in this work.
Equation 3.50 below exemplifies how the N-S equation is discretized into a system of algebraic equations in matrix form. This equation set can be manipulated into the form seen in Equation 3.51, which contains a matrix \( A \) with one value for each CV within the computational domain, a vector of velocity vectors \( [U] \), a source term vector \( [B] \), and a 'matrix part' term called \( H(U) \) which contains the neighboring coefficients of each CV multiplied by the corresponding velocities and added to the source vector.

\[
\begin{bmatrix}
X & \cdots & X \\
\vdots & \ddots & \vdots \\
X & \cdots & X
\end{bmatrix}
\begin{bmatrix}
U
\end{bmatrix}
= 
\begin{bmatrix}
B
\end{bmatrix} 
\tag{3.50}
\]

\[
\begin{bmatrix}
X & \cdots & 0 \\
\vdots & \ddots & \vdots \\
0 & \cdots & X
\end{bmatrix}
\begin{bmatrix}
U
\end{bmatrix}
= 
\begin{bmatrix}
B \\
H(U)
\end{bmatrix} 
\begin{bmatrix}
0 & \cdots & X \\
\vdots & \ddots & \vdots \\
X & \cdots & 0
\end{bmatrix}
\tag{3.51}
\]

If all the terms aside from the pressure gradient and body forces in Equation 3.49 are discretized, it may be written in a manner similar to Equation 3.51, and manipulated as seen below to find Equation 3.52, known as the 'velocity equation'.

\[
\left[ AU - H(U) \right] = -\nabla p + \rho f 
\Rightarrow 
\begin{bmatrix}
U
\end{bmatrix} = \begin{bmatrix}
\frac{H(U)}{A} \\
\frac{\nabla p - \rho f}{A}
\end{bmatrix} 
\tag{3.52}
\]

Equation 3.50 can be used to further manipulate Equation 3.52 into what is known as the 'pressure equation' seen in Equation 3.53 below.

\[
\left[ -\frac{\partial \rho}{\partial t} = \nabla \cdot \left( \frac{\rho H(U)}{A} - \frac{\rho (\nabla p - \rho f)}{A} \right) \right] 
\Rightarrow 
\nabla \cdot \left( \frac{\rho}{A} \nabla p \right) = \nabla \cdot \left( \frac{\rho}{A} [H(U) + \rho f] \right) + \frac{\partial \rho}{\partial t} 
\tag{3.53}
\]

Finally, the mass flux described by Equation 3.9 can be combined with the above formulation to create Equation 3.54, or the 'flux equation'.

\[
\Phi_f = S_f \cdot (\rho U)_f = S_f \cdot \left[ \left( \frac{\rho H(U)}{A} \right)_f - \left( \frac{\rho}{A} \right)_f (\nabla p)_f + \left( \frac{\rho}{A} \right)_f (\rho f)_f \right] 
\tag{3.54}
\]
The velocity, pressure and flux equations described by Equations 3.52, 3.53 and 3.54 constitute the fundamental equations of the PISO algorithm for an incompressible Newtonian fluid. The iterative approach used by the PISO algorithm to calculate the pressure and velocity coupling is as follows [43,48]:

1. **Momentum Predictor Step**: use Equation 3.49 to obtain an approximation (an intermediate field which does not obey the continuity equation) of the velocity field. Equation 3.49 is solved either by using the pressure distribution from the previous iteration step or, in the case of the first iteration, by using an initial guess for the pressure.

2. **Pressure Correction Step**: use the velocities obtained from step 1 to assemble the $H(U)$ operator, allowing the pressure equation (Equation 3.53) to be solved for a new set of pressure values.

3. **Momentum and Flux Corrector Step**: use the pressure field and the $H(U)$ operator found in step 2 to find the corrected velocity and flux equations via solving the velocity and flux equations (Equations 3.52 and 3.54) respectively.

4. **PISO Corrector Iteration Step(s)**: repeat steps 2 and 3 for the user-prescribed number of PISO iterations.

5. Move temporally forward to the next time step and return to step 1.

The momentum found in the momentum corrector step is explicitly calculated, and therefore inaccurate. All error associated with this calculation is assumed to have come from the velocity field due to error in the pressure field used to calculate it, although in actuality errors from the correction of neighboring velocities (found within $H(U)$) are significant as well [43]. The error associated with $H(U)$ and the momentum calculation is mitigated through successive PISO iteration loops, which should occur until a predetermined tolerance is reached [41]. Similarly, the calculation of the $H(U)$ term's coefficients (see Section 3.1.1) depend on the mass flux values, which, when found in the flux corrector step, are not used to immediately correct these values within $H(U)$. The values for $H(U)$ which are calculated in the first iteration of each time step are used for that entire time step, only being altered in the next time step (not the next PISO iteration within it). This process allows the non-linear term in the N-S equation to be effectively linearized since one of the two velocity components of the convection term is held constant at the value of the previous time step [43].
3.3.3 The PIMPLE Algorithm

The PIMPLE (Pressure-Implicit Method for Pressure-Linked Equations) algorithm is an embedding of the PISO algorithm within the iterative SIMPLE (Semi-Implicit Method for Pressure-Linked Equations) algorithm. The SIMPLE algorithm is used to solve steady-state problems where the treatment of the non-linear effects of the velocity during the resolution is more important than the precise determination of the pressure field. As each iteration is equivalent to a pseudo time step, the properties are under-relaxed in order to stabilize the method and improve convergence. Thus, the PIMPLE algorithm is more robust than the PISO algorithm since it is able to apply the PISO algorithm to both steady state and transient situations. Because of this additional functionality, the PIMPLE algorithm has replaced the PISO in recent OpenFOAM versions. Since the SIMPLE portion of the PIMPLE algorithm is simply an outer loop specification, the PIMPLE algorithm may be effectively simplified into the PISO algorithm by stipulating only one 'outer' PIMPLE iteration. The number of 'inner' iterations thus refers to the number of times the PISO algorithm is repeated in each time step. The total iterative cycle of the PIMPLE algorithm is graphically displayed in Figure 3.4 on the following page.
Figure 3.4: Organization of the PIMPLE algorithm.
3.3.4 The SaltPimpleFoam Solver

The OpenFOAM framework requires the implementation of a solver to calculate a case's discretized governing equations throughout the flow field and at the BCs constraining that field [41]. The solver developed for solving the cases in this work is based on the weakly-compressible governing equations defined in Section 2.2. OpenFOAM's top-level C++ code is flexible with respect to solver implementation, allowing users to generate solvers by following existing solvers' file structuring as templates. It is also possible to combine multiple solvers' template structures in order to describe situations containing physical phenomena which the basic solver library of OpenFOAM cannot assess (such as the semi-compressible effect). The solver developed via this template method to describe semi-compressible flow, SaltPimpleFoam, implements the D-C equation in an iterative structure similar to that of the pimpleFoam solver, relying on several mass-fraction-dependent pressure, velocity and flux relations used in the twoLiquidMixingFoam solver to treat portions of these flow variable calculations as compressible. Important portions of this solver's code are discussed within this section, but the entire source code may be found in Appendix B at the end of this thesis.

The SaltPimpleFoam solver consists of a set of seven main files which interact with each other and various lower-level OpenFOAM functionalities. A main SaltPimpleFoam.C file contains the basic iterative procedure structure, and calls other header files for certain calculations. This file begins with a list of included header files which are either lower than the top-level code code of the solver, or are files which are specific to and found only within the file system of the SaltPimpleFoam solver. These files are denoted with the [user-defined] tag in the comment pertaining to their call in the basic format of the SaltPimpleFoam.C file found below.

```
#include "bound.H"
// Value bounding functionality.
#include "fvCFD.H"
// FVM functionality.
#include "pimpleControl.H"
// Pimple functionality.
#include "fvOptionList.H"
// Option to include in-out finite volume source terms.
#include "IOMRFZoneList.H"
// Finite volume check functionalities used to correct velocity.
#include "fixedFluxPressureFvPatchScalarField.H"
// Boundary condition functionality which adjusts the pressure gradient such that the flux on the boundary is that specified by the velocity boundary condition.
```
```c
#include "setRootCase.H"
// Set up case directory.
#include "createTime.H"
// Instantiate time database.
#include "createMesh.H"
// Create mesh object.
#include "readGravitationalAcceleration.H"
// Get gravity constant from case files.
#include "initContinuityErrs.H"
// Find initial pressure and velocity continuity errors.
#include "createFields.H"
// Initialize flow variable fields [user-defined].
#include "createFvOptions.H"
// Create finite volume options.
#include "mAInitialContinuity.H"
// Calculate initial solute mass within solution domain [user-defined].
#include "readTimeControls.H"
// Read case's time controls.
#include "CourantNo.H"
// Calculate the Courant number.
#include "setInitialDeltaT.H"
// Calculate initial temporal step size.

pimpleControl pimple(mesh);
while (runTime.run())
{
    // Read case controls and determine time step size here.

    // Increment temporal counter:
    runTime++;

    // PIMPLE control class to supply convergence information/checks for the PIMPLE loop:
    pimpleCorrector pimple(mesh);
    while (pimpleCorrector.run())
    {
        // Construct the momentum equation (Equation 3.49) [user-defined]:
        #include "UEqn.H"
        // PISO corrector loop (inner loop):
        while (pimpleCorrector.run())
        {
            // Solute mass fraction field calculation (Equation 3.30) [user-defined]:
            #include "mAEqn.H"
            // Correction steps (Equations 3.54, 3.52, 3.53 and 3.54) [user-defined]:
            #include "pEqn.H"
        }
    }

    // Update the density field (Equation 3.31):
    rho = rho0 * ( 1.0 + rho_mACoeff * mA);
```
The \textit{createFields.H} file included within the initial header calls manages the loading and instantiation of the field variables for pressure, velocity, solute mass fraction, density, flux and case-defined constants required by equations within later header file calls. This file will not be further discussed in this section, and can be found in Appendix C at the end of this work.

The basic PIMPLE algorithm format is easily recognizable within the \textit{SaltPimpleFoam.C} code. Three essential files are called within the solver code to enable the solving of the pressure, velocity and solute mass fraction within the PIMPLE iterative structure: the \textit{UEqn.H}, \textit{pEqn.H} and \textit{mAEqn.H} user-defined files. The basic code associated with the first file called, \textit{UEqn.H}, can be seen below:

```c
// Explicitly calculate Equation 3.32 from the solute mass fraction field:
volScalarField mu
{
    "mu",
    mu0*(1.0 + mu_mACoeff*mA)
};

/**********************************************************************************/
// Discretize and assemble those portions of Equation 3.49 not used within the momentum predictor step:
vfVectorMatrix UEqn
{
    fvm::ddt(rho,U)
    + fvm::div(phi,U)
    - fvm::laplacian(mu,U)
    - fvc::div( mu*dev2( fvc::grad(U)().T() ) )
    == fvOptions(rho,U)
};

// Relax the found values with those of the previous temporal step here.

// Constrain any source terms here.

/**********************************************************************************/
// Optionally perform the momentum predictor step iff this is the first time in the time step the file has been called (the non-conservative velocity field is approximated from the balance of the gravitational and pressure body forces present in Equation 3.49):
if (pimple.momentumPredictor())
{
    solve(UEqn == fvc::reconstruct
    {
        ( - ghf*fvc::snGrad(rho)
        - fvc::snGrad(p)
    ) * mesh.magSf()

```
Having assembled the matrix or the N-S equation within the \textit{UEqn.H} file, the PISO loop is entered, and the D-C is solved by calling the \textit{mAEqn.H} file. This file is implemented within the PISO loop to properly resolve the transient nature of the solute mass fraction field [17]. The basic code associated with the first file called, \textit{mAEqn.H}, can be seen below:

// Explicitly calculate $\rho D_{AB}$, where $\rho$, which has been calculated from Equation 3.31 via the current solute mass fraction field, is taken either from the \textit{createFields.H} file (if this is the first time step), or from the previous time step (see code pertaining to \textit{SaltPimpleFoam.C} above) and $D_{AB}$ is calculated from Equation 3.32:

```c
volScalarField rhoDab
{
    "rhoDab",
    rho * max(
        DabCoeff * (1.0 - Dab_mCoeff * mA)
    ,
        DabMin
    )
};
```

Next, all the velocity, pressure and flux correction steps are performed within the PISO loop by calling the \textit{pEqn.H} file. The basic code associated this file called can be seen below:

// Explicitly calculate the density field (Equation 3.31):
```
rho = rho0 * (1.0 + rho_mCoeff * mA);
```

Next, all the velocity, pressure and flux correction steps are performed within the PISO loop by calling the \textit{pEqn.H} file. The basic code associated this file called can be seen below:
// Calculate and store the cell surface velocity flux field within the mesh via Equation 3.54. This is known as the 'first flux divergence argument':
surfaceScalarField phiHbyA
{
    "phiHbyA",
    fvc::interpolate(rho) *
    {
        (fvc::interpolate(HbyA) & mesh.Sf())
        + fvc::interpolate(rho*rAU)*fvc::ddtCorr(rho, U, phi)
        / fvc::interpolate(rho)
    }
};

// Make the flux relative to source terms here.

// Adjust the flux here using pressure and velocity fields here.

// Update the fixedFluxPressure BCs to ensure flux consistency between the actual and temporary velocity fields here.

// Implement the 'second flux divergence argument' here.

// Adjust the flux field according to the 'second flux divergence argument' here.

// Non-orthogonal pressure corrector loop:
while (pimple.correctNonOrthogonal())
{
    // Build the pressure via using the new velocity and flux fields to evaluate Equation 3.53:
    fvScalarMatrix pEqn
    {
        fvm::laplacian(fvc::interpolate(rho)*rAUf, p)
        ==
        fvc::div(phiHbyA)
        + fvc::ddt(rho)
    };

    // If the final orthogonality correction step has been completed, correct and rewrite the new conservative flux field according to the difference between the temporary and the new pressure equation's flux matrix:
    if (pimple.finalNonOrthogonalIter())
    {
        phi = phiHbyA - pEqn.flux();
    }

    // Calculate the new velocity field with the newly corrected pressure and flux fields. This new velocity will be used in the following PISO iteration to find the mA field:
    U =    HbyA
        + rAU
        / rho
In addition to all relevant portions of the PISO correction steps, the \textit{pEqn.H} file has a non-orthogonality loop which corrects for mesh non-orthogonality. The rationale behind its inclusion is that when the Laplacian term is discretized (when Equation 3.53 is assembled) in the above code, the non-orthogonal part of the Laplacian is calculated from the most recent pressure field and thereby must be iteratively corrected for \cite{43}.

Lastly, it should be noted that initial velocity, pressure, density and flux field values are calculated via a set of potential solvers, \textit{potentialSolute1st} and \textit{potentialSolute2nd}. These solvers were created in a manner similar to that of the \textit{SaltPimpleFoam} solver, and are employed in order to decrease the amount of time the main solver must take to generate a physically reasonable flow by allowing a large time step to be obtained early in the solution process. \textit{potentialSolute1st} generates initial compressible values for the velocity, pressure and density fields, but not a flux field (which is required by \textit{SaltPimpleFoam} to initialize a run). This is because to obtain a good set of initial values, two potential solution steps were found to be necessary, and initialization of OpenFOAM potential solvers with an existing flux field file leads to an erratic set of solution fields. Thus, \textit{potentialSolute2nd} solves for a second iteration of compressible velocity, pressure and density fields, then writes a flux field file to the case's file system. Since the \textit{SaltPimpleFoam} solver is effectively incompressible, the units of the pressure and flux flow variable files must divided by the units of density before the \textit{SaltPimpleFoam} solver will accept them as initial field inputs and run the case; a process which was done with the aid of shell-called python files. These potential solvers are not discussed in detail in this thesis, but may be found in full in Appendix B.
3.4 OpenFOAM File System

This section will cover the basics of the OpenFOAM file system and required files. In later sections of this work, files required for case-specific OpenFOAM utilities will be specified which will not be discussed in detail. For a locational mapping of these files in the file system required to run the cases developed in this work, see Appendix C.

3.4.1 Basic Case Folders and Files

An OpenFOAM case directory requires three folders. These folders are the system, constant, and 0 folders. The system folder sets parameters associated with discretization and the numerical solution procedure. The constant folder sets the material properties and contains files which define the mesh. The 0 folder is a time directory, and contains the files defining the boundary and initial conditions of the flow variables. In addition, a 0.org folder is usually included which contains an original copy of the time 0 folder, thereby serving as a backup in case the 0 folder is irreparably altered or deleted.

3.4.1.1 System Folder

The system folder requires three files: the fvSchemes file, and fvSolution file, and the controlDict file. The fvSchemes file contains information pertaining to the discretization method required by the solver being used. The fvSolution file contains the parameters required for the iterative numerical solution procedure used by the flow variable's individual solvers (see Section 4.5). The controlDict file contains information which specifies and controls the SaltPimpleFoam solver. Aside from the three files mentioned above, there is a fourth file included in the system folder which, although not required for the case to run, will enable a case to be solved in a more time-efficient manner through parallel processing. This file, known as the decomposeParDict file, divides the computational domain into a set of blocks which are then solved separately by individual processors. Thus, the number of decomposed domains is limited by the number of processors contained by the computer running the simulation.
3.4.1.2 Constant Folder

The constant directory must contain a polyMesh folder and two additional files called transportProperties and RASProperties. The polyMesh folder contains all the required mesh files for the case, most of which are generated from the initially lone blockMeshDict file upon entry of specific OpenFOAM commands. This blockMeshDict file is discussed later in Section 3.4.2.2 of this thesis. The transportProperties file defines the material properties of the fluid being simulated. They are the required scalar values used in the case's selected solver and turbulence model. For the cases considered in this work, this file contains the coefficient values required for or found in Equations 3.31 to 3.34, 3.36 to 3.38 and 3.40. The RASProperties file is used to select which turbulence model will be used in the simulation. Since turbulence implementation is not required by the Reynolds numbers achieved in this work, it contains null values and will not be discussed.

3.4.1.3 0 and 0.org Folders

These folders initially house the U, p and mA files, and will contain the rho (density) and phi (flux) files generated by the potential and SaltPimpleFoam solvers. The BCs specified in each file will be generally discussed in Section 3.4.2.1 of this thesis.

3.4.2 Two Dimensional Geometric Model

Two main geometries are considered and discussed in this Thesis. The first is a validation case study which directly matches the case implemented in [17]. It employs a single membrane separating a feed and draw channel. The second case study more closely matches the case found in [19]: it has two membranes, each separating a feed channel from a central draw channel. Circular baffles (spacers) are implemented in this central draw channel in order to visually observe the effect they have on draw ECP. For reference, the basic geometric features of each of these cases can be found in Figure 3.5 on the following page.
Figure 3.5: Validation and baffled mesh sketches.
3.4.2.1 Dual-Channel Validation Geometry

The validation case, shown in the top portion of Figure 3.6 above, consists of two, 2D 14cm x 3mm channels implemented in a cross flow exchange relationship across the membrane. The mesh in each channel contains 150 cells along its y-axis which are vertically graded so that grid points appear 5µm from the membrane wall and 50µm from the top and bottom walls. The 280 cells spanning each channel’s x-axis are not graded, making each 50µm in width. All meshing required by this case was accomplished using the blockMesh OpenFOAM meshing utility in conjunction with a blockMeshDict file. For an extended discussion of the blockMesh meshing process, the reader is referred to [41].

Boundary Condition Assignment  The inlets, denoted by parabolas pointing into the channels in Figure 3.6, are assigned uniform solute mass fraction values which could vary between 0.0 and 0.09, a pressure gradient of zero, and the parabolic velocity profile described by Equation 3.55, in which \( \bar{U} \) (the average velocity) is specified as 0.1m s\(^{-1}\). This profile is implemented through a link to the swak4Foam utility via the controlDict [17].

\[
U_x = 6 \bar{U} \frac{y}{h} \left( 1 - \frac{y}{h} \right) \quad (3.55)
\]

The boundary conditions on the exit of both channels are normal mass fraction and velocity gradients of zero and a gauge pressure of zero. All walls are assigned zero-gradient values for the mass fraction and pressure and a no-slip condition for the velocity.

Membrane Boundary Assignment  The membrane's boundary condition, set through use of the topoSet and createBaffles utilities via the topoSetDict and CreateBafflesDict, is given a zero-gradient pressure, mass fraction transport according to Equation 3.41, normal velocity permeation according to Equation 3.35, and a no-slip tangential velocity. As described in [19], the velocity on the draw side of the membrane is corrected for density changes across the membrane.
3.4.2.2 Triple-Channel Baffled Geometry

The baffled case, shown in the bottom portion of Figure 3.6, has a triple channel cross-flow with two membranes separating a central draw channel from two outer feed channels. This case was designed in such a way that any number of circular baffles can be placed on the x-axis-parallel center-line of the draw channel through use of the snappyHexMesh OpenFOAM utility. The spacer locations can be specified by simple manipulation a snappyHexMeshDict. For a discussion of the snappyHexMesh utility, see Section 4.4. All boundary conditions are the same as described in the validation case found in the prior section, with the exception that the membranes must have opposite ‘forward-direction’ orientations with respect to the feed and draw.

There are two main “baffled” meshes used in this work. The first has dimensions of 15cm x 7mm (each feed channel is 2mm in height) and 11cm long membranes and is used to conduct a Grid Convergence Study (GCS) and assess the effects the number of included evenly-spaced baffles have on the draw-channel pressure loss and water flux through the membranes. The number x and y cells within these cases vary, and will be discussed in Sections 4.2 to 4.4.

The second has dimensions of 6cm x 7mm and 4cm long membranes, and is used as the case run by the GA since it is much smaller and allows a reasonable computation time (which is important since several hundred cases are run per GA case assessment). This case was used to run a study to determine what solvers, smoothers, pre-conditioners, tolerances and relative tolerances defined in the fvSolution file for each flow variable would minimize the amount of time each case would take to run. This study is discussed in Section 4.5 of this thesis. Using these new values within the fvSolution file, an initial GA which tested the algorithm’s implementation and settings (see Section 6.1). A final GA was then run using the more computationally-heavy grid found to be required by the GCS for grid independence. The results of this final GA are discussed in Section 6.2.

Mesh and Spacer Generation  The snappyHexMesh utility used to place the circular baffles into the draw channel effectively requires cell aspect ratio of 1 (a uniform mesh) in the region a shape is being specified for a smooth outline to be formed. In order to have both this uniform mesh feature as well as the required grading toward the surface of each membrane, three separate cases (a top and bottom feed and a central draw case) are connected through the use of the mergeMesh and stitchMesh OpenFOAM meshing utilities. This must be done for two reasons: first, because there
should be less x-cells across the membrane than within the draw channel if a reasonable computation time is to be achieved, and second, the blockMesh meshing utility used for general mesh generation does not allow for cell-side discontinuities along the faces shared by defined blocks of mesh.

The top and bottom cases, called topFeed and bottomFeed respectively, have fewer vertically graded cells than the validation case. Each contains a full feed channel which is smaller in the y direction than that of the validation case, containing only 50 y cells graded toward the membrane, and a small portion of the central draw channel which contains 20 y cells graded toward the membrane. The gradings in these cases are such that the cells touching the membrane boundaries are less than 1e-05m away.

The draw channel case is initially generated from a blockMeshDict within the draw case folder that has a uniform grid. The mathematical relation for the number of x-cells required for a desired number of y cells within the draw channel of the 15cm baffled case is given below in Equation 3.56.

\[
\text{[# draw channel } \hat{x} \text{ cells]} = (68.18)[# \text{ draw channel } \hat{y} \text{ cells}] \quad (3.56)
\]

The draw case's blockMesh utility creates a mesh with one cell in the z-direction, effectively rendering it 2D as an OpenFOAM case. However, the snappyHexMesh utility requires an 'extruded' mesh which is technically 2D in an OpenFOAM sense, not just physically 2D as it is within the draw case. A reference mesh is generated from the draw case's mesh within the 2Ddraw case via the extrudeMesh utility, and then converted into a patch usable as a mesh with the createPatch utility (thus the inclusion of the extrudeMeshDict and createPatchDict files in the 2Ddraw's system folder). This mesh is technically 2D, allowing use of the snappyHexMeshDict in the 2Ddraw case in order to place circular baffles into the 2D mesh.

The cylindrical baffles implemented through the snappyHexMesh utility are minimally-defined within the snappyHexMeshDict via a single surface layer and two refinement layers. A higher number of either value results in either an inaccurately small or an impractically higher number of surface cells required to implement a single spacer. Finally, after any desired cylindrical baffles are generated within the 2Ddraw case's draw channel, the two feed cases are connected via the mergeMesh and stitchMesh utilities, and the whole case, which is now contained within the 2Ddraw case's mesh, may be run.
It should be quickly noted here that all the operations required by the above procedure as well as those required to run the case (the altering of the $p$ and $phi$ files' units to be incompressible after the potential solvers' runs and the decomposition of the solution domain for instance) were automated in a single external python Allrun.py file. This file can be minimally altered to change all membrane, velocity, solute mass fraction, cell distribution and the number and position of any draw channel spacers, and then re-write all required case files to account for those changes. This Allrun.py file is not applicable to the GA runs due to the file format required by DAKOTA, and was segregated into several separate files which were called as needed by the DAKOTA functionality during a GA run.

### 3.4.3 Case Post-Processing

Aside from the recombination of the decomposed fields created by the decomposePar utility (which is trivially completed via the reconstructPar utility), two major post-processing measurements must be made in order for the GA objective function to be evaluated (see Section 3.4.3.3 for information on this function): that of the draw channel pressure loss induced by any baffles present in the channel, and the total water flux through the membrane.

#### 3.4.3.1 Pressure Loss Measurement

The pressure boundary conditions for the entrance and exit of each channel are set to a von Neumann value of zero and a Dirichlet value of the internal pressure field respectively. Thus, as a simulation progresses, the pressure will slowly 'back-up' from the exit plane due to pressure losses in the channel, creating a pressure higher than atmospheric at the entrance to compensate for these losses. This method, although a widely-used technique in CFD, creates the counter-intuitive result of the entrance pressure changing to describe the channel pressure loss instead of the exit pressure changing as would be seen in a physical system. However, the resulting loss in pressure across the channel will be the same for both situations if all other portions of the simulation are modeled accurately.

The draw channel's pressure loss was measured through a user-defined utility named pLoss which may be called after a case is run. The pLoss program reads all time files generated by a case during the course of the run, and finds the pressure on both the total entrance and exit boundary patches via an iterative procedure. The difference between these values is found and printed to a text file
which may be read later by the objective function code. The basic code associated with the \texttt{pLoss} utility can be seen below:

```c++
int main(int argc, char *argv[]) {
  // Required lower-level OpenFOAM functionality header files included here.
  /**************************************/

  // Required time and mesh functionality header files included here.
  // For each time file:
  forAll(timeDirs, timeI) {
    runTime.setTime(timeDirs[timeI], timeI);
    Info<< "Time = " << runTime.timeName() << endl;

    // Create a field for the entrance pressure:
    volScalarField pEntrance
      ( // Field object code here.
        );

    // Create a field for the exit pressure:
    volScalarField pExit
      ( // Field object code here.
        );

    // Find the entire pressure field:
    volScalarField p
      ( // Field object code here.
        );

    // Label and ID each patch:
    label entrancePatchID = mesh.boundary().findPatchID("drawEntrance");
    label exitPatchID = mesh.boundary().findPatchID("drawExit");

    // Attach each relevant pressure field value to the patch ID:
    pEntrance.boundaryField()[entrancePatchID] = p.boundaryField()[entrancePatchID];
    pExit.boundaryField()[exitPatchID] = p.boundaryField()[exitPatchID];

    // Convert each field into a float value for pressure loss calculation:
    const scalarField& pEnt = pEntrance.boundaryField()[entrancePatchID];
    const scalarField& pExt = pExit.boundaryField()[exitPatchID];

    // To-terminal information output here.
    // To-File information output here.
  }
  return 0;
}
```
3.4.3.2 Water Flux Calculation

The calculation of the water flux through a case’s membrane BC was implemented within the boundary condition files themselves. The boundary condition writes all water flux values obtained to text files within the case, which may then be graphed to watch for convergence or read by external utilities for flux calculations. The output process becomes complicated once a case's computational domain is decomposed for parallel processing, as was done for all cases in this work. Each processor ends up owning its own membrane portion, yet appends its calculated values for the water flux to the same output file since the file label is contained within the membrane boundary condition source files, and cannot be altered based on processor. The flux value obtained by each processor is written to this output file as soon as it completes its calculation. Thus, the order of output of the processors varies in an unpredictable fashion since different portions of the mesh require more or less computational time during each time step. The flux value output therefore becomes an issue, because depending on the length of the membrane, the length of each of the dividers and the number of times the total length of the channel is divided (the number of processors being used), the fraction that each processor's water flux value must be multiplied by in order to obtain a technically correct value will change, and it is not possible to know which flux value should be multiplied by which scaling fraction. Thus, after each OpenFOAM run is completed in parallel, it is reconstructed and continued for a very short amount of time on one processor with zero relaxation for each of its flow variable matrix calculations. This lack of relaxation forces an initially steady simulation at initialization, allowing an accurate flux output immediately for the entire membrane. Thus, only the last value within the water flux text file must be retrieved. This process may be seen within the objective function evaluation code in the following section.
3.4.3.3 Objective Function Calculation

The objective function used for the DAKOTA GA must be dependent on all major effects that the inclusion of draw channel baffles could have on the results of a case. For this thesis, these effects are the pressure loss along the draw channel and the flux through the membranes. It had been shown that the flux and pressure loss values can be explicitly obtained for any case run, but these values in and of themselves mean nothing if there are not external constants against which to compare and optimize against. Since the pressure loss in the draw channel should be minimized, and is directly dependent on the number of spacers implemented in it, the value for this loss obtained from any given case will be compared to some maximum pressure loss value, \( P_{L,\text{Max}} \). Similarly, since the water flux through the membranes should be maximized, it will be compared to some minimum water flux value, \( J_{w,\text{Min}} \). The Objective function to be maximized is described in Equation 3.57 below

\[
f = \left( \frac{J_w}{J_{w,\text{Min}}} \right) \left( \frac{P_L}{P_{L,\text{Max}}} \right)
\]

(3.57)

The maximum pressure loss used in Equation 3.57 for a case's GA evaluation is calculated from a geometrically identical case with a 'maximum' number of included baffles along the length of the membrane. The number of baffles is stipulated by the minimum distance allowed between baffle centers; that of twice the baffles' diameter. The minimum water flux value used in Equation 3.57 for a case's GA evaluation is calculated from a geometrically identical case with no included baffles along the length of the membrane. The \( P_{L,\text{Max}} \) and \( J_{w,\text{Min}} \) values must be obtained in cases separate to the GA run, and are inputted as constants into the user-defined ObjFnctEval.py python file that calculates the objective function. The basic code associated with the ObjFnctEval.py file for a baffled case can be seen on the following page. For python files, the '#' symbol denotes a user comment and should not be read as part of the code that the file runs.
import os

# case-specific variables:
# ---------------------------------
J_wTotOrig = float(11.6147)  # for the 6cm, 60 x-cell case.
# J_wTotOrig = float(11.8637)  # for the 6cm, 200 x-cell case.

PLossMax = float(132.536)  # for the 6cm, 60 x-cell case.
# PLossMax = float(130.118)  # for the 6cm, 200 x-cell case.
# ---------------------------------

# obtain the flux for the lower membrane:
# ---------------------------------
fL = open('LowerFluxOutput.txt')
lowVals = [float(line) for line in fL]
LL = len(lowVals)
J_wLower = float(lowVals[LL-1])
# ---------------------------------

# obtain the flux of the upper membrane:
# ---------------------------------
fU = open('UpperFluxOutput.txt')
upVals = [float(line) for line in fU]
LU = len(upVals)
J_wUpper = float(upVals[LU-1])
# ---------------------------------

# obtain the values required by the objective function:
# ---------------------------------
J_wTot = float(J_wLower+J_wUpper)
J_wPercInc = 100*(J_wTot-J_wTotOrig)/J_wTotOrig

fP = open('FinalPLoss.txt')
PVals = [float(line) for line in fP]
PL = []
PL.append(PVals[0])
PLoss = float(PL[0])
PLossFracDec = (PLoss/PLossMax)
# ---------------------------------

# calculate the objective function and export it to DAKOTA:
# ---------------------------------
ObjFncVal = float(J_wPercInc/PLossFracDec)
f = open('ObjFncVal.txt','w')
f.write(str(ObjFncVal))
f.flush()
f.close()
# ---------------------------------
Chapter 4

Model and Case Assessment

For a CFD case to be accepted as an accurate model of a physical situation, it must be validated and verified. The concepts behind these two tests, as well as and their relation to the cases of this thesis, are found in this section. Studies concerning the effects that mesh refinement has on case results are discussed and used as a benchmark for the meshes selected for the DAKOTA GA. A study on the effects that different flow variable solvers, pre-conditioners and tolerances have on a case’s solution time is also discussed, as these settings are used to effectively decrease the time required for the overall GA runs with no decrease in result accuracy.

4.1 Sources of Error and Case Verification

Frameworks such as OpenFOAM inherently have several sources of error associated with solutions they obtain. These sources of error must be understood and watched for if verification of a simulated case is to be achieved. Here, verification refers to the utilized mathematics and its solution rather than how a solution matches results previously obtained (a.k.a. validation). Relevant verification errors are listed below. The verification of a case is completed when these possible errors have been assessed and determined insignificant.

Programming errors Using a well-known CFD framework such as OpenFOAM minimizes any basic mathematical programming errors the individual user can make, effectively containing them within the cases or libraries which are user-generated for research purposes. Although it is possible that OpenFOAM itself has some programming errors concerning more theoretical maths, its open-source documentation allows for the CFD community to quickly catch any mistakes and notify OpenCFD Limited. At the time this thesis’ research was conducted, no significant errors in the original OpenFOAM 2.3.0 code were documented. It should also be noted that OpenFOAM is completely accurate with respect to matrix algebra, which encompasses the vast majority of calculations required by the cases run in this thesis.
**Round-off Errors**  
Numerical round-off can be a significant source of error in CFD, as its effects may propagate throughout the solution domain or mask important flow effects within portions of the geometry. Thankfully, this error is easily mitigated through the use of high-precision floating point format in a solver, as is done in most OpenFOAM solvers, as well as the SaltPimpleFoam solver created in this work [46].

**Iterative Solution Divergence (Lack of Sufficient Convergence)**  
This can occur when the solution a solver obtains by solving one of the discretized domain's flow variable matrices has high residual values or continuity error. If, over successive temporal iterations, these errors continue to increase the case is considered divergent and will likely never obtain a steady-state solution. Although convergence was not found to be an issue in any case generated for this work (divergence is more likely for 3D and/or highly non-orthogonal geometries, of which this work contains neither), it can be controlled by increasing flow variables' tolerance values, relaxing those values with values obtained in a previous time step, or increasing the number of iterations run by the solver within each time step (here, that would mean increasing the SIMPLE and PISO loop iterations). For the transient solver developed in this work, complete convergence was achieved for all flow variables when a single iteration of the SIMPLE (outer PIMPLE sub-loop) and two iterations of the PISO (inner PIMPLE sub-loop) were implemented. See Section 3.3 for information on these iterative loops.

**Insufficient Temporal Resolution**  
If the time step of a case is too high, its solution can diverge due to the inability of the case's computational grid to completely transfer variable information to or from neighboring CVs by the time a sampling occurs [46]. This phenomena is known as the Courant-Friedrichs-Lewy condition, and is mathematically described by the Courant number, seen in Equation 4.1 below for the trivial 1D consideration.

\[
Co = \frac{u \Delta t}{\Delta x} \leq C \quad (4.1)
\]

The Courant number is defined by the velocity multiplied by the ratio of temporal step size (defined by the solver) and CV-to-CV distance (defined by the local solution domain). This term is in inequality with a constant \( C \) which will ensure stability in a transient simulation if \( C \) is less than one.
In effect, this relationship states that if the time step is not properly adjusted to account for how fast the information 'wave' containing flow variable information travels between CVs, then a 'reading' of information in a CV will happen either before or after the information arrives, and will thus not be accurate. The Courant number effectively limits the time steps of transient solvers by the smallest cells contained within their discretized domain to prevent information loss. For all cases run in this work, the maximum Courant number was set to 0.7 to ensure convergence.

**Insufficient Spatial Resolution** If a case's computational grid is not dense enough (i.e. it has low spatial resolution), averaging and round-off errors can become significant. Ideally, 'grid-independent' results should be achieved, meaning that further increase in mesh density will not produce a significant change in the results obtained by two otherwise identically run cases. The value used to quantify a Grid Convergence Study (GCS) is a Grid Convergence Index (GCI), seen in a 3D consideration in Equation 4.2 \[ 49 \].

\[
GCI_{\text{coarse}} = \frac{3 |e| R}{R^3 - 1} \leq 0.01 \quad (4.2)
\]

Here, \( R \) is the ratio of cells between the finer and coarser grid and \( e \) is the relative error between the two grids. Usually, this GCI is desired to fall below 1% when a comparison is made between two grids. Due to the meshing issues discussed in Section 4.4, the GCS conducted in this work does not take this error into account, looking only at the changes that occur in the outputs desired for evaluation of the objective function.

**4.2 Validation Experiments**

Unlike verification, discussed in the previous section, validation of a numerical model assesses how accurately a simulated solution represents physical phenomena. Since validating this thesis' work would involve the purchasing of several FO membranes, membrane channels, and a hydraulic pumping system, a direct validation was not conducted. However, Gruber et al. \[17\] performed an extensive and accurate physical validation study on their membrane model (which the model developed in this thesis is based upon) in both RO and FO. Their results mirrored those RO findings of \[20\] and accurately represented physical FO experiments performed with the AL-FS membrane orientation. Because of the accuracy of these findings, it was deemed acceptable to
conduct a validation experiment against the FO simulation results published by [17]. The validation case mesh discussed in Section 3.4.2 was used to generate all graphical results found in this section. All the graphical results reported in [17] were successfully reproduced, but only the draw solute profile development and obtained water fluxes for various mass solute differences over the membrane are discussed here for the sake of brevity and because physical manipulation of these results via draw baffle implementation is the focus of this thesis. Figure 4.1 displays the simple and advanced steady-state flux values obtained from the SaltPimpleFoam solver (for various trans-membrane $\Delta m_A$ values) when using the membrane boundary conditions described in Section 3.2.2. The trends and computed values match those found in [17] within 2.57%.

![Steady-State Simple and Advanced Membrane Water Flux for Cross-flow Velocity 0.1[m/s]](image)

**Figure 4.1: Steady-state flux values for various $\Delta m_A$ obtained using the SaltPimpleFoam solver.**

All flux values obtained in Figure 4.1 reached 97% of their steady-state values within 10 seconds of the simulated time (steady-state values were obtained within 50 seconds of simulated time), with initial trends having a monotonic increase to steady-state instead of the monotonic decrease seen in [17]; a trend that is believed to be more numerically stable. Figure 4.2 shows a comparison (between SaltPimpleFoam and [17]) with respect to this flux development for the ‘Increasing Draw Concentration’ run seen in Figure 4.1 with a $\Delta m_A$ value of 0.06. It should be noted that
comparisons were conducted for each $\Delta m_A$ value in Figure 4.1, but were, again, omitted for brevity. Figure 4.2 demonstrates the convergence trend true to all the runs seen in Figure 4.1: All flux values in these runs reached within 98% of those obtained in [17] by 8 seconds of the simulation time.

![25 Second Example Water Flux Development](image)

*Figure 4.2: Comparison of SaltPimpleFoam's steady-state flux development profile with that of [17].*

Now that the validity of the steady state flux values and the solver’s temporal convergence to those values has been shown, only the draw channel's ECP profile requires attention. On the following page, Figure 4.3 shows a comparison of steady-state developed solute profile near the membrane's draw-side. These results were generated using the same case setup that generated Figure 4.2. The simple flux model's solute mass fraction profile matches within 4% of that in [17], while the advanced flux model's profile matched well within 1%.
Figure 4.3: Comparison of SaltPimpleFoam's steady-state solute profile with that of [17]

The results shown Figures 4.1 to 4.3 demonstrate that the SaltPimpleFoam solver and the asymmetric membrane boundary conditions employed accurately predict the water flux through and solute profile across a physical AL-FS membrane. It may therefore be concluded that, for the purposes of the work done in this thesis, the accuracy of the developed OpenFOAM solvers and membrane boundary conditions is validated.

4.3 Objective Function Dependence on Baffle Inclusion

Due to the fact that the inclusion of baffles in the draw channel will affect both a pressure loss and a decrease in ECP, each baffle implemented will have an effect on the returned objective function value. To see if $f$ was sensitive to one of these induced effects, a study was conducted in which increasing numbers of baffles were placed evenly across the length of the of the 15cm baffled case, discussed in Section 3.4.2.2. This longer form of case was chosen because the overall number of baffles able to be included was larger, allowing dominating trends to become apparent. A membrane x-direction cell count of 300 was chosen due to the large meshing inaccuracy which occurs for the 150 x-cell membrane discussed in Section 4.4. The cross-flow velocity was 0.1 m/s
and the $\Delta m_A$ value across both of the case's membranes was 0.06, with the feed having no solute. Advance flux and no-slip conditions were stipulated for the membranes. The maximum number of baffles allowed in the 15cm case, as determined by the minimum distance between each baffle being fixed at two baffle diameters, was 44. The minimum was 5, as a case with 0 baffles was already explored as a validation case. The number of baffles included was increased by increments of 5 until 40 baffles were reached, then increased by 4 to 44. The results of this set of runs can be seen in Figure 4.4 below.

![Objective Function Deviation for Evenly Spaced Baffles](image)

Figure 4.4: Objective function dependence on baffle inclusion.

Since the 44 baffle case determines the $P_{L,Max}$, its value will be give the objective function the greatest possible decrease with respect to the $P_L$ ratio. However, since baffle inclusion also decreases the ECP in the draw channel, the 44-baffle case should also represent the greatest increase in the objective function with respect to the water flux $J_w$. Thus, it is apparent that $P_L$ is the dominant variable in the objective function. This is not surprising: the $P_{L,Max}$ due to 44 baffles is near 400 Pa, and in the teens for no baffles, while $J_w$ may only increase from $J_{w,Min}$ by a decimal value. Ideally, if the number of baffles is held constant, the $J_w$ variable would be the primary contributor to fluctuations in $f$. This would only be possible if the pressure loss had a very small
variation range with respect to where baffles could be placed relative to each other. The desired $J_w$ dominance becomes apparent as the number of baffles decreases, as seen in Figure 4.4. Thus, for a proper water flux analysis to occur, a small number of baffles must be implemented within a given channel. Figure 4.5 below depicts the velocity magnitude fields of three of the cases from Figure 4.4 which are relevant to the remainder of this section.

The trend in Figure 4.4 flattens out immediately after the number of included baffles reaches 15. This asymptotic trend occurs because, as seen in Figure 4.6 on the following page, the twin sheets of laminar vortices formed by each baffle begins to interact with the next downstream baffle (see Figure 4.6.b) before significantly dissipating (see Figure 4.6.a). Also, if two baffles are close enough to each other, as is the case in the 40 baffle case (Figure 4.6.c), fixed eddies form between them which forces a slip-stream flow to develop against the membrane walls. The shear stress induced on the fluid by the walls has three effects: one, it drastically increases $P_L$ due to drag; two, it creates unnecessary stress on the membrane which could eventually cause it to break; and three, it creates the maximum possible decrease in local ECP (see Figure 4.6). Since the relative increase in local pressure loss far outweighs the effects of the ECP decrease (most of the downstream ECP mitigation of the first baffle is negated by the close proximity of the second), relative baffle distances will be limited by the width between the baffles in the 15 baffle case: 0.00775m (note that this minimum distance to be used in the GA's constraints is CASE SPECIFIC: if the average velocity of the parabolic profiles is changed, the distance of relevant wake effects will alter). Analysis of the resulting shear on the membrane will not be conducted in this work, but may be
easily explored using the defined case framework by linking it to the shear-stress analysis library included within OpenFOAM which can read a wall-boundary’s shear stress.

![Figure 4.6: Down-stream baffle wake for Figure 4.5’s cases.](image)

If the mean cross flow velocity was increased, longer baffle wakes would develop, and the trend seen in Figure 4.4 should begin to flatten out before a baffle number of 15: the curve would flatten asymmetrically to the left. This picture also demonstrates the meshing artifacts (apparent above and below each spacer) which are discussed in Section 4.4. These artifacts have an increasingly negative effect on the accuracy of the objective function as they become more defined, and would be clearer in Figure 4.6 if 150 x-cells were chosen to be along the membrane instead of the 300 instated. The reader will see later that the cases used for the GA are the same height as the 15cm case, but only 40% of its length. This decrease in the case’s aspect ratio (AR) will alter the trend seen in Figure 4.4, lowering the objective function values associated with smaller numbers of baffles. However, as
the basic pressure loss effect is strongly proportional to the percentage of area taken up by the baffles in the channel, if the number of included baffles is small enough the decrease in $f$ due to the increase in the case's AR will be negligible. The number of baffles chosen for the GA was thus held at 3. Combined with the newly defined minimum distance between discussed above, this fixed number of baffles should produce generally good $J_w$ values.

4.4 Grid Convergence Study

This section concerns a mesh analysis of the 15cm triple-channel baffled case described in Section 3.4.2.2. The case's membranes' parameters, solute mass fraction and velocity conditions are the same those found in Section 4.3. The findings of this GCS used to refine the 6cm case for the DAKOTA GA as its results can be directly applied to otherwise identical 'axi-symmetric' (in that the mesh only significantly alters in the y direction) cases of different lengths. As mentioned in Section 4.2, a regular GCS will not be conducted for the cases in this thesis for two main reasons. The first, which concerns meshing artifacts which result in the use of the mergeMesh and stitchMesh OpenFOAM utilities, will be discussed in Section 4.4.1. The second concerns limitations the snappyHexMesh utility imposes on baffle placement, and will be discussed in Section 4.4.2.

Due to the reasoning found in Sections 4.4.1 and 4.4.2, case results will be plotted against the number of cells set along a case's feed channels' x-axis (if the number of cells in the central draw channel changes, such changes will be noted outside of the figures) instead of against a GCI. This parameter is chosen because it is simple to alter, has a significant effect on both the resulting artifacts of the meshing procedure, directly affects the ECP resolution within the draw channel and is result-independent of the y-direction cells on the membranes, which are held constant throughout all cases in this study. The results and analysis of the cases run in this study are found in Section 4.4.3 below.

4.4.1 Triple-Channel Case Meshing Issues

The complete triple-channel case is generated by 'gluing' (Elmer's® consistently performs well) three cases to each other, and then stitching together the cells on the boarders between them so that flow-variable information may be transferred between them. This second step must be taken because the meshes will not recognize each other if they are simply connected through a similar face: the meshes
are still completely discontinuous (they do not share points or centroid relationships in a single, interconnected file set) and therefore cannot communicate information in the OpenFOAM framework. The basic geometry of the three cases to be combined is shown in Figure 4.7 below:

![Figure 4.7: Basic geometries of the initial cases that form the final triple-channel baffle case.](image)

For reasons described in Section 3.4.2.2 and discussed in Section 4.2.2, the draw case must have a uniform mesh. Because this case is very long compared to its height (0.15m x .0022m), i.e. it has a large Aspect Ratio in the x direction (ARx). This means that the draw case will have a majority of the total (2Ddraw) case's cells. For all the cases in this study, the cell count in the draw case was either 750 by 11 or 1500 by 22, as found from Equation 3.56. Since having 750 or 1500 cells along the x-axis of all three cases in Figure 4.7 would create a mesh too dense for a feasible solution time, the topFeed and bottomFeed cases will have fewer x-axis cells. This fact makes generating the entire mesh within one case impossible, since OpenFOAM's normal blockMesh meshing utility will not allow boundary discontinuities between neighboring chunks, or HEXs, of a case's mesh.

To connect the feed meshes to the draw mesh, the boundaries of two cases are connected along a common face boundary with the mergeMesh utility, then broken up and connected geometrically using supermesh theory via the stitchMesh utility. The visual results of these two utilities is shown in Figure 4.8 on the following page, in which mesh A can be compared to the draw case's mesh and mesh B can be compared to either feed case's mesh.
The ARx of the created inter-case-mesh cells in mesh C is extremely under-exaggerated, and would visually be a fraction of the pictured height. This makes the cells connecting the two meshes highly non-orthogonal, and can lead to non-negligible interpolation errors between the connected meshes if the ratio of cells in one mesh is much larger than the other along the direction of the connecting face (in Figure 4.8 this ratio is 3:1). Thus, a mesh ration must be determined that prevents major connection ARx errors, yet allows for a minimal computation time (minimal cell count). This determination is the main purpose of the GCS.

4.4.2 Baffle Inclusion Meshing Issues

The `snappyHexMesh` utility used to generate the mid-channel baffles requires that the ARx of the cells upon which the cylinder is to be implemented should be 1. The mathematical reasoning behind this phenomena is quite complex, and will not be discussed in this thesis. Two numbers of draw case x-cells are used in this GCS: 750 and 1500. Using a count less than 750 requires a very small number of y cells which will force refinement parameters within the `snappyHexMeshDict` to be raised. Such settings will generate far more cells per cylinder than would otherwise be required while also causing a generally inaccurate computational grid in the center of the draw channel. The number of cells required to implement each cylindrical baffle is discussed in Section 4.4.3.

The type and placement of baffles in the draw case was chosen for specific reasons pertaining to the `snappyHexMesh` utility. Cylindrical baffles were selected (aside from the theoretical reasoning discussed in Section 2.3) because they have an automatic call in the `snappyHexMesh` library, making recursive importation of .stl files not required (a process which is difficult, but not impossible to
The baffles were forced to be implemented min-channel instead of in a zigzag orientation, which would require the baffles to be placed touching the membranes of the draw channel, which, as seen in Figure 4.7, are not initially contained in the *draw* case. This fact is relevant because the three cases (for technical reasons not discussed) must be connected after the implementation of the *snappyHexMesh* utility.

### 4.4.3 GCS Results and Discussion

This GCS will assess both the maximum pressure loss (44 baffles) and minimum water flux (0 baffles) cases discussed in Section 4.3. These cases are chosen because they generate results which serve as benchmark values for the objective function evaluation, and must therefore be as accurate as possible for a GA run on a given grid. The trend to be looked for within the study is result consistency between grid refinements. The cases which demonstrate this trend must then be considered with respect to their computation time (cell count).

The five meshes considered in this study are shown on the following page in Figure 4.9. Within this section, these cases will be referred to as their 'Case 1 to 5' labels in this figure.
Each of the above grid's was run with and without baffles, requiring 10 total case runs. The cell counts of each of these cases was been plotted as a function of the x-cells along the case's membranes, and can be seen in Figure 4.10 below.
The relative cell counts (which exclude the extra cells generated by the `stichMesh` utility) of the case pairs reveal that to minimally implement a baffle in a case containing 750 x-cells within the `draw` case requires a net addition of ~860 cells by `snappyHexMesh`. This average number of added cells is increased to ~1525 if the `draw` case's x-cell count is 1500. This cell number is not double the 750 x-cell case's value (i.e. in proportion to the draw case’s unity-cells’ size decrease) because each initial cell is smaller: less initial local mesh refinement is required to create a smooth baffle surface.

Similarly, if the mesh in the `draw` case is made coarser, the number of cells that would need to be added for `snappyHexMesh` to accurately construct a baffle will drastically increase. For cases with several baffles, these additions will likely far exceed the number of cells saved by decreasing the mesh density in the `draw` channel.

The membrane water flux results of the 10 cases can be found in Figures 4.11 and 4.12 on the following page.
Figure 4.11: Water fluxes through the top and bottom membranes of all ten GCS cases.

Figure 4.12: Water flux percentage increase of a GCS mesh due to a maximum number of baffles.

Figure 4.11 displays several important trends. First, and least importantly, the upper membrane of each case has a slightly lower water flux than the lower membrane. This is due to the CV body-centered nature of the solute mass fraction, making it susceptible to gravitational effects and thereby increasing the influence the free stream flow has on breaking up the lower membrane's ECP.
Second, the cases with no baffles see a flux decrease which is far sharper than that of the fully baffled cases as the cell count increases. This is due to an effect that will be dubbed 'push' velocity for the remainder of this thesis. Here, push velocity is the effect which the highly-non-orthogonal case-connection cells have on the velocity of the flow field of the draw channel. If the x-cell ratio between cases is high, the large ARx of these cells creates a slip-stream which propagates a higher-than-average x-velocity along the portions of the draw channel which were initially contained within the feed cases, thereby creating a greater decrease in ECP than would normally be seen. As the ratio of cells between the cases becomes more refined, this effect is decreased. The drastic decrease seen in the 0 baffle Case 5 can be contributed to the fact that there is no longer any AR difference between the cells connecting the cases (see Figure 4.9), making the solution especially accurate. This number of cells, although accurate, is not possible to use within the GA due to the large computation time required, and will not be explored further.

Third, Figure 4.11 also shows the maximizing effect that the 'maximum' 44 baffle case has on the water fluxes. All cases have nearly identical fluxes which are, as seen in Figure 4.12, 110% or more of the corresponding 0 baffle cases. This maximization occurs due to the high velocity seen along the entirety of the membranes due to the baffles' proximity to each other (see Section 4.3), which completely by-passes any push-velocity effects (in a sense, they create a dominant natural push velocity). The drastic +130% value seen by the 44 baffle Case 5 is due to the proportional decrease in flux seen in the 0 baffle Case 5, and will therefore not be discussed.

The draw channel pressure loss results of the 10 cases can be found in Figure 4.13 below. These values have be 'normalized' by dividing all values by the pressure loss of Case 1. This was done so that a high divergence from this initial, unity value will correspond to an increase in accuracy (Case 1 is by far the least accurate).
Figure 4.13: Normalized (with respect to the 150 x-cell case) draw channel pressure loss of all ten GCS cases.

Figure 4.13 depicts an increase in pressure loss for the 0 baffle cases and a decrease for the 44 baffle cases. The increase seen for the 0 baffle cases is caused by the decrease in case connection cell ratio and thus the AR of those cells: the fact that the inner and outer portions of the channel can communicate information more accurately allows for mixing effects to be properly captured (they are not damped by the push velocity stream), the physical result of which is a greater pressure loss. The decrease of pressure loss seen in the 44 baffle cases is also accounted for by the decrease in case connection cell ratio and thus the AR of those cells: the large push velocity stream induced by the final baffle in the cases does not create a large relative velocity at the draw exit patch and there are fewer general losses due to poor interpolation above and below the baffles along the entire channel.

Finally, the effect on the resolution of the physical velocity and solute mass fraction effects within the draw channel due to mesh refinement was explored. Figures 4.14 and 4.15 on the following pages visually describe these resolution effects. Figure 4.14 shows the fully developed velocity profile of the last baffle in the 44 baffle cases, while Figure 4.15 shows initial and fully-developed (steady-state) ECP profiles in the draw channel for these same cases.
Figure 4.14: Final baffle’s steady state wake velocity profile of the GCS’s 44 baffle cases.
Figure 4.15: Final baffle’s initial and steady state wake ECP profile of the GCS's 44 baffles cases.
Figure 4.14 shows a significant mitigation of the push velocity stream (the solid blue line at the top and bottom of the draw channel) by Case 3. Significant resolution of initial ECP convection is also achieved by Case 3. The stead state results of Figure 4.15 are relatively non-conclusive. The achieved resolution seen in Figures 4.14 and 4.15, the relative (local) stability of the pressure loss and water flux in Figures 4.11 to 4.13 and the reasonably low cell count makes Case 3 the best option for a comprehensive GA case.

Two GA cases will therefore be run: an initial, poor resolution GA in which Case 1 will be used to test the GA's parameters and generate a set of solutions against which the results from an accurate, Case 3 GA will be compared.

4.5 Pre-Conditioner and Solver Associated Study

The OpenFOAM framework uses linear, iterative solvers to perform solution operations on a case's flow variable matrices. These solvers are not to be confused with the main solver of the case's governing equations (refer to the SaltPimpleFoam solver in Section 3.3.4), which implements these linear solvers to obtain, keep track of, manipulate and store flow variable information. These linear solvers are paired with either a smoothing or a pre-conditioning function which allows a case to converge to a solution more quickly than would otherwise be possible. This pre-conditioning effect can be described in a more physical sense as allowing faster information propagation through the computational mesh [50]. The solution found by both the linear solver and its smoother or pre-conditioner is constrained by tolerance and relative tolerance values which force a definite solution accuracy for, respectively, a particular calculation and deviance between consecutive (within an iteration set) calculations. The definitions of and information pertaining to these solvers, smoothers and pre-conditioners are set in a case’s \texttt{fvSolution} file (see Appendix C).

Due to the fact that the interactions between the case's linear solvers, governing equation solver and geometry are intimately affected by the way that each flow variable is solved, the settings of the \texttt{fvSolution} file which will achieve an optimal computational time will be case specific. There is no cut and dry process to determine which solvers should be used for the variables of a user-defined case, a study was performed to determine the set of solvers, their pre-conditioner or smoother pairings, and tolerance values which lead to a minimal case solution time while maintaining solution accuracy.
Gruber et. al., [17], found that the following *fvSolution* settings seen in Table 4.1, were optimal for their weakly-compressible system within the OpenFOAM 1.7.1 framework.

<table>
<thead>
<tr>
<th>Flow Variable</th>
<th>Solver</th>
<th>Smoother</th>
<th>Tolerance</th>
<th>Pre-Conditioner</th>
<th>Relative Tolerance</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>PCG</td>
<td>--------</td>
<td>1e-08</td>
<td>DIC</td>
<td>0.075</td>
</tr>
<tr>
<td>U</td>
<td>PBiCG</td>
<td>--------</td>
<td>1e-10</td>
<td>DILU</td>
<td>0.1</td>
</tr>
<tr>
<td>( m_A )</td>
<td>PBiCG</td>
<td>GaussSeidel</td>
<td>1e-12</td>
<td>DILU</td>
<td>0.1</td>
</tr>
<tr>
<td>( \rho )</td>
<td>GAMG</td>
<td>--------</td>
<td>1e-12</td>
<td></td>
<td>0.01</td>
</tr>
</tbody>
</table>

Table 4.1: Optimal GA case's *fvSolution* file settings for the OpenFOAM 1.7.1 framework.

The *PCG* linear solver used to calculate the pressure matrix is a pre-conditioned conjugate gradient solver for symmetric IduMatrices which must use a run-time selectable pre-conditioner. The *PBiCG* linear solver deals with the velocity and solute mass fraction matrices, and is effectively the asymmetric equivalent of the *PCG* [50]. The *DILUPreconditioner* used for the *PBiCG* solver is a simplified diagonal-based incomplete LU pre-conditioner for asymmetric matrices. The *DICPreconditioner*, used for pre-conditioning the pressure matrix, is a simplified diagonal-based incomplete Cholesky pre-conditioner for symmetric matrices; the symmetric equivalent of the DILU pre-conditioner. The reciprocal of the pre-conditioned diagonal is calculated and stored by these pre-conditioning OpenFOAM utilities [50]. The density matrix is calculated by the *GAMG* utility: a geometric agglomerated algebraic multigrid solver which requires a smoother. The *GaussSeidel* smoothing method was therefore employed in order to solver the density's linear equation system: it can be thought of as an improvised version of the Jacobi method [50].

In order to find optimal OpenFOAM 2.3.0 *fvSolution* file settings for the 3-baffle, 6cm long case to be used in the DAKOTA GA (see Section 3.4.2.2), a comprehensive set of case solutions was compared to a geometrically identical case run with the settings described in Table 4.1. Although the final GA run will use a case which has 200 cells along its length (the equivalent of the 500 x-cell case in Section 4.2), the cases in this study had only 60 (the 150 x-cell equivalent) so that the study could be completed in a feasible time span. As this study contained well over 30 runs, only the final solution results, found in Table 4.2 below, will be mentioned here.
Table 4.2: Optimal GA case’s *fvSolution* file settings for the OpenFOAM 2.3.0 framework.

<table>
<thead>
<tr>
<th>Flow Variable</th>
<th>Solver</th>
<th>Smoother</th>
<th>Tolerance</th>
<th>Pre-Conditioner</th>
<th>Relative Tolerance</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p )</td>
<td>PCG</td>
<td>--------</td>
<td>1e-05</td>
<td>DIC</td>
<td>0.075</td>
</tr>
<tr>
<td>( \mathbf{U} )</td>
<td>PBiCG</td>
<td>--------</td>
<td>1e-04</td>
<td>DILU</td>
<td>0.1</td>
</tr>
<tr>
<td>( m_A )</td>
<td>PBiCG</td>
<td>--------</td>
<td>1e-07</td>
<td>DILU</td>
<td>0.01</td>
</tr>
<tr>
<td>( \rho )</td>
<td>SmoothSolver</td>
<td>symGaussSeidel</td>
<td>1e-04</td>
<td>--------</td>
<td>0.0</td>
</tr>
</tbody>
</table>

The solvers for the pressure, velocity and solute fraction matrices were all found to maintain a minimal case solution. The density matrix solution became more efficient if the \textit{SmoothSolver} utility was used in conjunction with the \textit{symGaussSeidel} smoother. The \textit{SmoothSolver} is an iterative solver which uses a smoother for both symmetric and asymmetric matrices, while the \textit{symGaussSeidel} smoother is simply the symmetric version of the more general \textit{GaussSeidel} smoother mentioned before [50]. Aside from this change, the 2.3.0 version of OpenFOAM exhibits a much more stable solution procedure, allowing the solution tolerance of all variables to be significantly relaxed. These changes allowed a decrease in computer-clock (real-world) computation time from 4514s (using the settings found in Table 4.1) to 2069s (using the settings from Table 4.2). The solution hit steady state within 6.5 seconds of in-case (theoretical) computation time. The continuity errors and residuals obtained from both runs does not vary significantly, as can be seen from Figure 4.16 on the following page. An otherwise identical case was run with 200 x-cells to see how long (in theoretical case time) each GA case would have to run using the new *fvSolution* settings. The plots of continuity errors and residuals for this case can be seen in Figure 4.17. As can be seen from Figure 4.17, the case’s solution converges in just under 10 seconds of theoretical time. This time will become the simulation end and solution file writing times stipulated in the DAKOTA GA’s OpenFOAM case’s \textit{controlDict} file.
Figure 4.16: Continuity and Residual plots for the new and old fvSolution settings in 60 x-cell cases.
Figure 4.17: Continuity and Residual plots for the new *feSolution* settings in 200 x-cell case.
Chapter 5

DAKOTA Genetic Algorithm Implementation

DAKOTA is a robust optimization toolkit with an extensible interface which allows for the communication of information between almost any other 'black-box' program. That is, DAKOTA can easily couple with any program that can take in information, perform operations which are not communicated to DAKOTA and thereby generate data of a form which may be retrieved by general file manipulation. OpenFOAM is an ideal black-box for DAKOTA since both the user-defined information which is entered into each case and the relevant post-processed data of those cases is defined and contained in basic text files. The file system within DAKOTA, although mainly controlled by the bash, aprepro and dprepro computing languages, allows for a user to define files in the program's file system and then link them into the optimization procedure. This flexibility allowed portions of the Allrun.py python file used in earlier simulations to be integrated as the interface to OpenFOAM. It also helped overcome one of the main flaws of DAKOTA: the inability to initially define inter-dependent design variable constraints.

The GA which DAKOTA was employed to create allows a dynamic and iterative generation and evaluation of the 6cm triple-channel baffle case discussed in Section 3.4.2.2. The x-axis positions of the mid-channel baffles generated in each case will be controlled and constrained by this GA, and will therefore vary between cases. Each case's BC and membrane settings will be fixed between runs, and are identical to those discussed in Section 4.3.

The basic mapping and relevant source code of the DAKOTA file system used for this GA is showcased in Appendix D.
5.1 Relevant File System and Basic Solution Process

The DAKOTA folder's file system can be generally described (at least in this thesis' case) as follows. A complete OpenFOAM case file system is contained within an original folder, excepting all files which contain information the optimization process must alter on a case-to-case basis. Dummy versions of the files to be altered are held in a separate template folder, and use place-holding symbols within them that DAKOTA recognizes as the design it generates according to the GA's requests. Aside from these folder, several files must exist: a .in file which contains the basic optimization procedure to be followed, including any links to external files which handle user-defined calculations; a dprepro functionality file to map the generated design variables into the template folder's dummy files; files for cleaning and running a case, and any user-defined files which will run those optimization procedurals which are not part of the original DAKOTA file system.

The iterative running of the GA within this DAKOTA file system is as follows:

1. DAKOTA initiates: the baffle_ga.in file (discussed in Section 5.2 below) generates values for all user-defined design variables and links the remainder of the calculation's main process to an external bash file (the simulation file: see Appendix D) which handles the command line that may be fed into the OpenFOAM program.

2. The simulation file calls the dprepro functionality file to write the generated design variables to the relevant template files of the OpenFOAM case, generates a labeled (based on the GA iteration number) copy of that case which includes these written files, then assess if the newly written design variables are within the user-defined constraints.

3. If the constraints from the ConstraintsCheck.py file (see Section 5.2) are violated, a ‘null’ $f$ value (see Equation 3.57) of 0 is written and recorded: proceed to Step 6.

4. If the constraints are not violated, run all code associated with the OpenFOAM case.

5. Read the generated pressure loss and water flux values and evaluate $f$ via the ObjFunctEval.py file.

6. Export the calculated value of $f$ to the DAKOTA system.

7. If the maximum possible number of simulations has been reached, or the defined solution convergence value has been reached, end the GA. Else, if the number of simulations required to evaluate a new population ($Pop$) has been reached, generate that $Pop$ from the prior $Pop$ based on the GA stipulations discussed in Section 5.2 and then return to Step 1. Else, continue evaluating the new $Pop$ by returning to Step 1.

The basics of this process are visualized in Figure 5.1 on the following page.
5.2 Design of the Genetic Algorithm

In the DAKOTA framework, all main portions of the GA aside from the user-specified constraints are defined in an .in file, named `baffle_ga.in` in the cases run in this work. The structure of this file is straightforward, and simply states what settings the GA has been chosen via DAKOTA keywords. For discussion simplicity, these settings have been boiled down into Tables 5.1 and 5.2 on the following pages. The actual code is found in Appendix D.

Table 5.1 represents the upper and lower continuous design variable constraints designated in the `baffle_ga.in`, i.e. the positions of each baffle along the x-axis of the draw channel. The baffles must be further constrained such that they appear consecutively as x increases, and are far enough apart that each baffle's downstream wake does not affect significant pressure losses within the channel. Thus, each baffle must be at least 0.00775m to the right of its preceding baffle (see Section 4.3). The bounds allowed in the `baffle_ga.in` file can only force the baffles to appear along the membranes within the channel, as well as enforce the minimum distance between them at the start and end of the membranes.
<table>
<thead>
<tr>
<th>Design variable (continuous)</th>
<th>Lower bound [m]</th>
<th>Upper bound [m]</th>
</tr>
</thead>
<tbody>
<tr>
<td>x0 (first baffle position)</td>
<td>0.01075</td>
<td>0.03375</td>
</tr>
<tr>
<td>x1 (second baffle position)</td>
<td>0.0185</td>
<td>0.0415</td>
</tr>
<tr>
<td>x2 (third baffle position)</td>
<td>0.02625</td>
<td>0.04925</td>
</tr>
</tbody>
</table>

Table 5.1: Baffle position constraints available in the DAKOTA .in file.

If only the above bounds are used, these baffles may possibly be generated both out of order and/or in close proximity to each other within a certain central portion of the draw channel. A python file was therefore employed in the DAKOTA file system to further constrain the baffles’ placement. This file, called `ConstraintsCheck.py`, is found in the folder housing the original OpenFOAM case and reads in the positions of each baffle. The relevant portions of its source code can be found below (remember, for python '//' denotes a comment):

```python
import os
import sys

# define the number of baffles being implemented in the GA:
N = 3

# Create a place-holding list:
vals = []

# Open the file containing the newly written baffle locations here.
# insert the baffle positions into the list via :
for i in range(N):
    # File terms to orient search here.
    vals.append(float(snappy[point_index+9:end_index]))
    f.flush()
    f.close()

# Define the minimum distance and the null case f value:
Dist = float(0.00775)
errorVal = "0.0"
boolList = [vals[i+1]-vals[i] > Dist for i in range(len(vals) - 1)]

# Create an error-comparison string file used in the simulation file's procedure:
f = open('2Ddraw/NOERRORScript','w')
f.write("NOERROR")
f.flush()
f.close()
```

96
# Check the baffle position list for any constraint violations:
# --------------------------------------------
# Write 'NOERROR' if there are none:
if not False in boolList:
    f = open('2Ddraw/ErrorBool','w')
    f.write("NOERROR")
    f.flush()
    f.close()
# If violations occur, write the null f value and 'ERROR' string to separate files:
else:
    f = open('2Ddraw/ErrorBool','w')
    f.write("ERROR")
    o = open('2Ddraw/ObjFnctVal','w')
    o.write(errorVal)
    f.flush()
    f.close()
    o.flush()
    o.close()
# --------------------------------------------

This code checks to see if the baffles are either disordered or too close. If they are, it writes a text file (containing either the word 'ERROR' or 'NOERROR') telling the bash simulation file's code whether or not to continue with the OpenFOAM case evaluation. If 'ERROR' is written, the objective function will be evaluated as 0 later in the simulation file's process.

It may be obvious from the above information that, at least initially, the number of null chromosomes within a Pop will be high. This comes from the fact that the initial generation of the baffle positions by DAKOTA cannot account for the desired constraints within the center of the channel. The GA must be able to decrease these null chromosomes as quickly as possible while still being robust. Table 5.2, seen below, describes the main GA as defined in the baffle_ga.in file.

<table>
<thead>
<tr>
<th>DAKOTA Setting</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method</td>
<td>SOGA</td>
</tr>
<tr>
<td>Fitness type (f)</td>
<td>Merit function</td>
</tr>
<tr>
<td>Optimization sense</td>
<td>Maximization</td>
</tr>
<tr>
<td>Mutation type</td>
<td>Replacement</td>
</tr>
<tr>
<td>Replacement type</td>
<td>Elitist</td>
</tr>
<tr>
<td>Mutation frequency (M%)</td>
<td>10%</td>
</tr>
<tr>
<td>Crossover type</td>
<td>Two-point binary</td>
</tr>
<tr>
<td>Crossover frequency (C%)</td>
<td>80%</td>
</tr>
<tr>
<td>Population size (Pop)</td>
<td>20</td>
</tr>
<tr>
<td>Maximum OpenFOAM runs</td>
<td>2000</td>
</tr>
<tr>
<td>Convergence criteria (%)</td>
<td>5%</td>
</tr>
<tr>
<td>Number of final solutions</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 5.2: DAKOTA test GA Settings
Due to the fact that all considered aspects of a run evaluation have been contained within Equation 3.57, the type of GA chosen is a SOGA with the desired purpose of maximizing $f$. Each consecutive generation keeps the runs with the best $f$ in an elitist fashion and replaces relatively bad ones with the mutation and crossover alterations discussed in Section 2.4. The $M\%$ is 10\% since mutation can easily cause a constraint fault due to the stringent constraints on baffle position. However, its inclusion is still required for a robust SOGA. The 80\% double binary crossover will, in the case of this particular chromosome design, effectively switch the second and third baffles of the two parent cases. This will achieve two things: first, it acts as a filter in case the elitism of the GA fails since if one of the parents chosen is a null case, the children will likely be null as well. Second, it will help converge the GA if both parents have high $f$ values since the children will likely be non-null and will shorten the case's search for a $Pop$ with few null values. A low population size of twenty is chosen so that new populations are generated frequently, allowing the crossover and elitism effects of the SOGA to influence population development often, forcing a generation of non-null chromosomes and speeding the SOGA’s convergence.

Despite these precautions, null cases will occur periodically throughout the run. Thus, convergence of a solution population within the defined 5\% is highly unlikely. Convergence would be possible if this value was very large (~80\%) since the number of null cases within a $Pop$ will likely remain constant and small, predictably decreasing the overall ‘value’ of a population. However, this form of convergence is undesirable: lack of any convergence would allow a large maximum number of runs and therefore generate a much better $Pop$. This is the reason for the high limit on the number of runs the SOGA is allowed to perform (it effectively limits the GA to 100 generations) and the low convergence tolerance. As this number of generations is very high, the SOGA was simply run until the population contained a large number of chromosomes with a high $f$ value. Were the GA to finish, the best five solutions of the final population were set to be chosen as viable solutions. However, since an ideal solution is not known, and ‘as-required’ comparison and discussion of the final $Pop$’s chromosomes will be conducted.
Chapter 6

Results and Discussion

As discussed in Section 4.4, two GAs were run. The first is a test GA which uses chromosome cases with a mesh equivalent to Case 1 of Section 4.4. This test run will check to see how the GA functions and whether it needs alteration or refinement in certain areas. It will also become a comparison benchmark for the second GA, which will use chromosome cases equivalent to Case 3. This second GA's final population will be the culmination and final result of this thesis.

6.1 Genetic Algorithm Test Run

The specifics of the test GA's 'Case 1' chromosome can be found in Table 6.1 below.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Case length</td>
<td>0.06</td>
<td>[m]</td>
</tr>
<tr>
<td>Membrane length</td>
<td>0.04</td>
<td>[m]</td>
</tr>
<tr>
<td>Membrane start</td>
<td>0.01</td>
<td>[m]</td>
</tr>
<tr>
<td>x-cells along membrane</td>
<td>60</td>
<td>--</td>
</tr>
<tr>
<td>Maximum pressure loss</td>
<td>132.536</td>
<td>[Pa]</td>
</tr>
<tr>
<td>Minimum water flux</td>
<td>11.6147</td>
<td>[kg/h · m²]</td>
</tr>
<tr>
<td>Number of implemented baffles</td>
<td>3</td>
<td>--</td>
</tr>
<tr>
<td>Minimum baffle-to-baffle distance</td>
<td>0.00775</td>
<td>[m]</td>
</tr>
<tr>
<td>Draw free-stream solute mass fraction</td>
<td>0.06</td>
<td>--</td>
</tr>
<tr>
<td>Feed free-stream solute mass fraction</td>
<td>0.00</td>
<td>--</td>
</tr>
<tr>
<td>Average free-stream velocity in all channels</td>
<td>0.1</td>
<td>[m/s]</td>
</tr>
</tbody>
</table>

Table 6.1: Test GA's chromosome case parameters.

The GA was run for 30 successive populations: a total of 500 case runs. The GA was stopped before the prescribed 2000 run limit because a set of reasonable solutions (for the purposes of the test) had been reached. As will be explained, better values would have been achieved if the GA had continued to run, but the time that would have been required to find them was too large to warrant continuing the algorithm. DAKOTA's graphical output for each chromosome's objective function value and corresponding baffle positions during the test GA run can be seen in Figure 6.1.
The three right three graphs in Figure 6.1 show the positions of each baffle per generated and evaluated chromosome. The left-most of this set of graphs showcases the search method that DAKOTA will use if given three design variables: it will repetitively increase the first chromosome variables while oscillating the remaining two to convergence. This process allows DAKOTA to look for trends predictably within a GA calculation. The position graphs of the last two baffles show that the GA began to converge by 300 chromosome evaluations.

The left-most graph of Figure 6.1 shows that the GA both finds good solutions and improves its set of best solutions (populations) between successive runs. It also demonstrates the negative interaction that mutation and crossover have on the GA when combined with the strict constraints: null chromosomes are generated with regularity despite the fact that the current population is entirely made up of chromosomes with good fitness values. To see the evolving effect of the constraints, it helps to map out each population's chromosome fitness. This evolutionary population comparison can be found in Figure 6.2 on the following page.
Figure 6.2 begins with Pop 6 simply for convenience: Populations 1 to 5 had null chromosomes in them, and would have skewed the graphical results. Populations 6 to 13 have large negative slopes, meaning that the best chromosomes in these populations have a much better fitness value than the rest. This trend is undesirable: similar, high fitness values throughout an entire population show a converged solution. These populations also show a large average fitness improvement between populations, which is good since it proves that the GA develops well initially. It is not until Pop 14 that the constraints actually begin to control the GA. This control delay is likely because of a high number of null chromosomes are present in the earlier populations: the overall population must acquire generally good chromosomes before crossover has an overall beneficial effect on the GA’s convergence. Populations 14 to 30 demonstrate a convergent trend: their slopes flatten out, and successive populations show less relative improvement.

The best chromosome of each population in Figure 6.2 has been removed (the chromosomes’ ordered ranking for each Pop begins at 2). This was done so that the trends of Figure 6.2 could be easily seen: from population 14 onward, an extremely high chromosome was found. This can be seen in Figure 6.1 as the fitness spike that occurs around chromosome 230. The fitness of this chromosome was over 21, and overshadows all other values found before or after. The complete
final population can be seen below in Figure 6.3, while the steady-state pressure, velocity and solute profiles for this high-fitness chromosome can be seen below in Figure 6.4.

Figure 6.3: Complete fitness values for population 30 of the test GA.

Figure 6.4: The flow variable profiles for the test GA's high-fitness chromosome.
This high-fitness chromosome could be the result of three possibilities: first, a sampling error occurred in OpenFOAM, leading to an artificially low pressure loss. Second, the GA has randomly found a very good solution toward which the population should eventually trend. Third, the GA has randomly found a solution which looks good due to the pressure-term dominance of \( f \), but is bad for the purposes of optimizing the water flux. As can be seen from Figure 6.5 below, both the pressure loss and water flux of this high-fitness chromosome was well below that of the final population's other top chromosomes.

There are several points to take from this result. First, that the pressure loss ratio can be confirmed as a dominant term in \( f \). Second, that if the third baffle is located at the far right portion of the membrane, some of its wake's mixing effect are not fully accounted for within the channel (they are 'cut off' by the end of the geometry), making the overall pressure loss lower. Finally, that this last chromosome is undesirable; both because of this pressure-decreasing geometric cut-off and because its water flux value is lower than the others. It will therefore be discarded. This kind of result makes it apparent that the best chromosomes of the final GA \( Pop \) will have to be individually assessed against the whole population. For a more visually-informative representation of the best solutions found by the test GA, the baffle positions of the final \( Pop \)'s top ten chromosomes are graphed (relative to the beginning and end of a case's membranes) in Figure 6.6 on the following page.
Unlike the top nine chromosomes preceding it, the best chromosome's first baffle is much farther to the right. Thus, the overall distance between each of the baffles is smaller, reducing the total wake-mixing effects along the membranes. This means that the membrane-portion of the draw channel sees a lower ECP reduction and pressure loss.

It is possible that the small size of the test GA's populations has induced an 'in-breeding' effect that the current level of mutation cannot overcome. If a GA's population is small enough, and enough solutions are possible, a small set of chromosomes which have both a good fitness and similar bit values can over-take the population as it evolves. Crossover breeding between these chromosomes becomes increasingly likely, and, since their children will necessarily also have a high fitness, the population will soon fill up with these high valued, similar chromosomes. Mutation usually helps prevent this. However, the low mutation rate and stringent GA constraints dampen the mutation operation's beneficial effects, as it is likely that a mutation will generate a null chromosome. To prevent this damping effect, the final GA's the mutation rate will be increased to 20%, its population size will be increased to 30, and its crossover rate will be reduced to 70%.
6.2 Final Genetic Algorithm Run

The updated specifics of both the final GA’s settings and its ‘Case 3’ chromosome’s constants can be found, respectively, in Tables 6.2 and 6.3 below.

<table>
<thead>
<tr>
<th><strong>DAKOTA Setting</strong></th>
<th><strong>Value</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Method</td>
<td>SOGA</td>
</tr>
<tr>
<td>Fitness type ((f))</td>
<td>Merit function</td>
</tr>
<tr>
<td>Optimization sense</td>
<td>Maximization</td>
</tr>
<tr>
<td>Mutation type</td>
<td>Replacement</td>
</tr>
<tr>
<td>Replacement type</td>
<td>Elitist</td>
</tr>
<tr>
<td>Mutation frequency ((M%))</td>
<td>20%</td>
</tr>
<tr>
<td>Crossover type</td>
<td>Two-point binary</td>
</tr>
<tr>
<td>Crossover frequency ((C%))</td>
<td>70%</td>
</tr>
<tr>
<td>Population size ((Pop))</td>
<td>30</td>
</tr>
<tr>
<td>Maximum OpenFOAM runs</td>
<td>2000</td>
</tr>
<tr>
<td>Convergence criteria (percentage difference)</td>
<td>5%</td>
</tr>
<tr>
<td>Number of final solutions</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 6.2: Final DAKOTA GA Settings.

<table>
<thead>
<tr>
<th><strong>Parameter</strong></th>
<th><strong>Value</strong></th>
<th><strong>Unit</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Case length</td>
<td>0.06</td>
<td>[m]</td>
</tr>
<tr>
<td>Membrane length</td>
<td>0.04</td>
<td>[m]</td>
</tr>
<tr>
<td>Membrane start</td>
<td>0.01</td>
<td>[m]</td>
</tr>
<tr>
<td>x-cells along membrane</td>
<td>200</td>
<td>--</td>
</tr>
<tr>
<td>Maximum pressure loss</td>
<td>130.118</td>
<td>[Pa]</td>
</tr>
<tr>
<td>Minimum water flux</td>
<td>11.8637</td>
<td>[kg/h \cdot m^2]</td>
</tr>
<tr>
<td>Number of implemented baffles</td>
<td>3</td>
<td>--</td>
</tr>
<tr>
<td>Minimum baffle-to-baffle distance</td>
<td>0.00775</td>
<td>[m]</td>
</tr>
<tr>
<td>Draw free-stream solute mass fraction</td>
<td>0.06</td>
<td>--</td>
</tr>
<tr>
<td>Feed free-stream solute mass fraction</td>
<td>0.00</td>
<td>--</td>
</tr>
<tr>
<td>Average free-stream velocity in all channels</td>
<td>0.1</td>
<td>[m/s]</td>
</tr>
</tbody>
</table>

Table 6.3: Final GA’s chromosome case parameters.

Important alterations include the increase in the GA’s mutation rate and population, as well the decrease in its crossover rate. The maximum draw channel pressure loss and minimum water flux constants required to evaluate \(f\) have also changed, as these values are dependent on the mesh the GA’s chromosome is assigned (see Section 4.4.3).

The final GA obtained a converged population within 12 generation evolutions (that is, 300 chromosome evaluations), and was therefore stopped before the prescribed 2000 run limit. The
graphical output for each chromosome's fitness value and baffle positions can be found below in Figure 6.7.

Figure 6.7: DAKOTA’s graphical output for the final GA.

The ‘Objective Fn’ graph on the left of Figure 6.7 shows a trend of increasing chromosome fitness, and the positions of second and third baffles (‘x1’ and x2’ graphs) have converged to a minimal solution range (disregarding variation due to mutation). Although DAKOTA still repetitively increases its first design variable (graph ‘x0’) to allow for a predictable solution search, the majority of this search pattern became confined around a mean value of 0.02m by the latter half of the run. The final GA’s population evolution shows all the same trends as that of the test GA (see Figure 6.2), and can be seen in Figure 6.8 below.

Figure 6.8: Fitness mapping of the final GA’s population evolution.
As is also demonstrated by Figure 6.7, the final GA’s run initially found a large number of null chromosomes due to the GA’s strict constraints (which did not change between the test and final runs). Again, by population six all null chromosomes had been eliminated, despite the fact that the population was 50% larger than that of the test GA. After two generations of complete chromosomes, constraint dominance was achieved, and the population converges desirably in the final four evolutions. These trends all show that the changes made to the GA have allowed it to become more efficient and robust. The Fitness values obtained by the final GA are found to be generally higher than that of the test GAs. This can be credited to the fact that the pressure field in the draw channel has been more accurately modeled due to the relatively refined mesh of Case 3: the physically-accurate pressure loss incurred by the baffles is, in basic terminology, lower than that generally found by Case 1. This decrease necessarily generates larger $f$ values.

Again, the best chromosome in the constraint-dominated populations is significantly larger than the others, although the difference in fitness values is not as drastic as that seen in Figure 6.3. To check if the reason for this relatively high fitness is due to the fitness function’s pressure loss sensitivity, the water flux and pressure loss values of the best 15 chromosomes of the final population were plotted. This comparison can be seen in Figure 6.9 below.

Figure 6.9: Pressure loss and water flux for the final GA’s fifteen best chromosomes.

Figure 6.9 shows that the top chromosome of the final Pop has a low pressure loss and average water flux relative to the other top chromosomes. To check if this pressure loss is due to the geometric ‘cut-off’ effect found in the test GAs best chromosome, the baffle positions of each chromosome were mapped in relation to the start and end of the draw channel membranes. This positional mapping can be seen in Figure 6.10 below.
Figure 6.10: Baffle positions for each chromosome of the test GA’s final population.

The best chromosome’s last baffle is no closer to the end of the membrane than the rest of the population, meaning that the relatively low pressure loss is not due to geometric cut-off. The decrease is pressure loss is therefore probably due to a sampling error created by the pLoss utility: there is no other reason that would cause it to have a pressure loss so much lower than, for example, chromosome 2, since their baffles positions are virtually the same. Thus, this top chromosome will not be treated as more important than the others. Since most of the pressure loss has been mitigated by the minimum baffle-to-baffle distance constraints, the water flux is now the most important factor to be considered. The top half of the final population has achieved nearly identical water flux results: their ordering is mostly dependent on their pressure loss values. Because of this flux similarity and the fact that most of the baffle positions are nearly identical, averaging each of them across the final population’s chromosomes will produce a reliable, ‘best’ chromosome that will dampen the effects of individual chromosome’s pressure loss variance. The averages of these baffle positions were (in order of appearance within the channel) 0.02037, 0.03342 and 0.04534 [m]. A simulation for this ‘best’ chromosome was generated with a ‘Case 5’ mesh to show all channel flow effects associated with the steady state pressure, velocity and solute mass fraction flow variable fields. These fields can be found on the in Figure 6.11. This case represents the final result of this
thesis. It proves that the DAKOTA GA is able to quickly and reliably converge a population of OpenFOAM FO case chromosomes to a set which can be averaged to achieve a ‘best’ chromosome with optimal baffle positions with respect to maximizing water flux and minimizing channel pressure loss. Although this process can theoretically be applied to any FO channel case geometry, it should be noted that the GA will take longer to converge if the channels become longer or a greater number of baffle positions are being optimized because the solution domain the GA must search within will become much larger.
Figure 6.11: ‘Best’ final GA chromosome’s mesh and relevant flow variable fields.
Chapter 7

Conclusion

The goal of this thesis was to form a genetic algorithm in such a way that future researchers could easily manipulate and alter it in order to explore the many implications of baffle placement optimization within Forward Osmosis asymmetric-membrane channels in crossflow.

Studies were performed which verified and validated the OpenFOAM-contained semi-compressible solver, asymmetric FO membrane BC and cross-flow channel meshes. The GA optimization was implemented through a DAKOTA/OpenFOAM coupling and was designed in such a way that a user could alter almost every relevant case parameter through minor manipulation of external python files. The OpenFOAM case parameters were optimized through pressure-loss-dependence, grid-convergence and pre-conditioner studies to ensure that the case was both accurate and computationally light. The GA was tested, improved, and run using settings which was known to produce an accurate solution. The run converged quickly, producing a final population which were then averaged to generate a best solution. The GA was thus proven to produce reliable, optimized results, and can be easily used to find the optimal positions of baffles within any FO asymmetric-membrane geometry such that the water flux through any present membranes is maximized and the pressure loss incurred by included baffles is minimized; realizing the goal of this thesis.
Chapter 8

Future Research

Having achieved the goal of constructing an easily-alterable GA which can model the effects of a cross-flow FO setup and optimize the positions of baffles within its channels, the author suggests that both it and its case chromosomes be used and altered to explore the following as topics for future research:

- Exploration of how the GA handles cases with different channel geometries or more than three implemented baffles.
- Exploration of the effects of various $d/b$ ratios on the water flux and channel pressure loss.
- Effects of baffle implementation in the feed channel (a channel-orientation inversion).
- Exploration of the effects of the baffles’ dynamic vortex shedding on the ECP for various $Re$, and a mapping of the minimum baffle-to-baffle distances those $Re$s require to maintain a low channel pressure loss.
- Exploration of the effects of vortex and boundary layer shear stress on the membrane’s porous support layer.
- Implementation of the zigzag baffle orientation (or allowing the baffles to vary vertically within the draw case mesh).
- Inclusion of non-circular baffle cross-sectional shapes into the snappyHexMesh meshing process.
- Alteration of the SOGA to a MOGA in order to mitigate pressure sensitivity.
- Semi-compressible turbulence model implementation.
Appendix A:

OpenFOAM Asymmetric Membrane Source Code

The solute and velocity boundary conditions for the asymmetric FO membrane are compiled as a separate library which is called within a case's file system (specifically the `system/topoSetDict` file). Each of the boundary conditions is implemented using a `.C` and `.H` (header) file. However, as the header file is simply used to instantiate variables and functions for the BC, only the relevant portions of the `.C` files will be shown. The page index for the boundary condition files is found below.

Solute Boundary Condition:
EFOMSFvPatchScalarField.C.................................................................................................................................114

Velocity Boundary Condition:
EFOMVFvPatchScalarField.C.................................................................................................................................116
114

EFOMSFvPatchScalarField.C

//OpenFOAM functionality and C++ constructor definitions here.

254 //Initialize autoMap from .IO:
255 void EFOMSFvPatchScalarField::autoMap
256 {
257     const fvPatchFieldMapper& m
258 )
259 {
260     fvPatchFields<scalar>::autoMap(m);
261 )
262 )
263 )
264 //Initialize reverse mapping from .IO:
265 void EFOMSFvPatchScalarField::rmap
266 {
267     const fvPatchField<scalar>::rmap<psf,
268     const labelList& addr
269 )
270 {
271     fvPatchFields<scalar>::rmap<psf, addr>
272 ;
273 )
274 //**************************************************************************/
275 )
276 )
277 void EFOMSFvPatchScalarField::updateCoeffs()
278 {
279     if (this->updated())
280 {
281         return;
282 )
283 )
284 // Execute the change to the openFraction only once per time-step:
285 if (curTimeIndex_ <= this->db().time().timeIndex())
286 {
287 // geometric help variables:
288     const vectorField& SF = patch().SF();
289 // creates a reciprocal matrix of each patch's scalar coeffs. this
290 // analogous to the j/A matrix specified within the thesis:
291     const scalarField deltax = 1.0/patch().deltaxCoeffs();
292     tmp = vectorField( tvnfn = patch().nF();
293     const vectorField& vfnf = tvnfn[];
294 )
295 )
296 // Velocity field along surface normal
297     const fvPatchField<volVectorField> upvf
298     = patch().lookupPatchField<volVectorField>, vector->(UName_);
299     scalarField magU = max( mag( cmpMultiply( upvf , vfnf ) ) )< SMALL );
300 )
301 )
302 // Used in test of BCs. Only strictly required for debugging */
303     const fvPatchScalarField& nA
304     = patch().lookupPatchField<volScalarField>, scalar("nA");
305     tmp = scalarField( temp = m_.patchInternalField());
306     const scalarField mInternal = temp[];
307 )
308 )
309 // Total salt flux and area:
310     scalar totalHeightFlux = 0;
311     scalar totalArea = 0;
312 )
313 )
314 // Variables used:
315     scalar A = 0;
316     scalar JS = 0;
317     scalar B = 0;
318     scalar rho = 0;
319 )
320 // Debug variables:
321     scalar nGrad;
322     scalar drawMassInbalance = 0;
323     scalar feedMassInbalance = 0;
324 )
325 forAll( patch() , facet )
326 {
327 /*
328     Determine side based on surface normals & flow direction
329     Afterwards implement following BC: D_A*dnA(dma/dn)+nA*MA = JS
330 */
331     rho = rho_.value();
332     * ( 1.0 + rho_eqCoeff_.value() ) operator[]( facet )
333     * (( 1.0 - DAB_eqCoeff_.value() ) operator[]( facet )
334     / ( 1.0 + DAB_eqCoeff_.value() ) operator[]( facet )
335     )().value();
336 )
337 )
338 if ( (upvf[facet] & nA[facet]) << 0 )
339 //This checks if the flow direction is from the draw to feed direction
340 //via the limit of the inner product of the velocity and surfaceVectorGeometric
341 //matrices being set to 0. It is NOT satisfactory for this portion of the code to
342 //become satisfied (WRT an RO setup), since this is in the negative direction into the
343 feed. [
115

```cpp
if ( (upvr.FacetP)^2 - MFacet^2) <= 0.0 )
{
    // Variables B and salt flux:
    B = rho*mag(J[facet]);
    JS = ( B + mag(J[facet]) ) / ( pLn Herc_value()/ 1000 + B );

    // Set coefficients:
    VJC[facet] = 1.0 / ( 1.0 + B * delta[s][facet] );
    VRC[facet] = JS / ( A / delta[s][facet] + B );
    GIC[facet] = -1.0 / ( A / delta[s][facet] );
    GRC[facet] = JS / ( A + B * delta[s][facet] );

    // Debugging:
    Following tests whether the equation
    A(DMA/dt)+b*MA = JS is satisfied
    if ( newbornGrad = GIC[facet]*MInternal[facet] + GRC[facet];
        DrawMassInbalance = A*newGrad - operator[2](Facet) - JB;
    )
    // For the case that the flow is from the feed to the draw (the normal,
    // positive direction):
    else
    {
        // Variables B and salt flux:
        B = -rho*mag(J[facet]);
        JS = ( B + mag(J[facet]) ) / ( pLn Herc_value() / 1000 + B );

        // Set coefficients:
        VJC[facet] = 1.0 / ( 1.0 + B * delta[s][facet] );
        VRC[facet] = JS / ( A / delta[s][facet] + B );
        GIC[facet] = -1.0 / ( A / delta[s][facet] );
        GRC[facet] = JS / ( A + B * delta[s][facet] );

        // Debugging:
        Following tests whether the equation
        A(DMA/dt)+b*MA = JS is satisfied
        if ( newbornGrad = GIC[facet]*MInternal[facet] + GRC[facet];
            DrawMassInbalance = A*newGrad + operator[2](Facet) - JB;
        )
        // Add to salt flux through membrane:
        totalWeightFlux += patch().magSF()[facet] * JS;
    }
}

} // Set time index:
curTimeIndex = time.db().time().timeIndex();

} // OpenFOAM patch field definitions here.

void Foam::Patch::calcFaceMapping()
{

    // Set up the face-index mapping based on cell centres
    const vectorField& CFVF = patch().CFVF();
    forAll(CFVF, facet)
    {
        // For label '1', i.e CFVF.size():
        if (label[facet] == 1)
        {
            // (mag(CFVF[facet] - CFVF[i]) <= 9)
            if (mag(CFVF[facet] - CFVF[i]) <= 9)
            {
                // Debug
                Info << "patch face " << facet << " : " << i << endl;
                Info << "DrawMassInbalance -- " << DrawMassInbalance << endl;
            }
        }
        break;
    }
}
```

115
```c
EFOMVFvPatchScalarField.C

116

```
forAll(fs_, facet)
{
    label fsi = fs_[facet];
    label dst = fn_[fsi];
    feedMem = NA[fsi];
    drawMem = NA[dst];

    /***************************************************************************/
    // User riddler's method to solve for the flux,
    scalar flux = riddlerSolve(
        feedMem,
        drawMem,
        minBound,
        maxBound,
        xec,
        t
    );
    /***************************************************************************/
    // Calculate the velocity for the asymmetric membrane
    vector v = vfnf[fsi] * flux;
    /***************************************************************************/
    // Set the feed-side velocity
    operator[](fsi) = v;
    /***************************************************************************/
    // Correct the velocity due to density change
    v *= (1.0 + rhoMCoeff_.value() * feedMem) / (1.0 + rhoMCoeff_.value() * drawMem);
    /***************************************************************************/
    // Total flux and area
    totalMassFlux += flux
    * rhoM_.value();
    * (1.0 + rhoMCoeff_.value() * feedMem)
    / magSF[dst];
    /***************************************************************************/
    Flux = totalMassFlux/(sum(magSF)/2)*3600;
    /***************************************************************************/
    // Slip boundary condition
    if( slipType() == "slip" )
    {
        slipUInternal = (internalU[dst]
                         - (internalU[dst] & vfnf[dst])
                         & vfnf[dst]);
        slipPrev = operator[](dst)
           - cmplMultiply
           ( cmplMultiply
              ( vfnf[dst], vfnf[dst] )
           , 
            operator[](dst)
           );
    /***************************************************************************/
    tf( mag(slipUInternal) > SMALL )
    {
        dudy = (mag(slipUInternal) + mag(slipPrev) ) / delaus[dst];
        slipUbounary = alpha*dxudy
        * (slipUInternal/mag(slipUInternal));
    /***************************************************************************/
    tf( mag(slipUbounary) > mag(maxSlip) )
    {
        maxSlip = slipUbounary;
    }
    /***************************************************************************/
    // Set the draw-side velocity
    operator[](dst) = v + slipUbounary;
    Info << "\n" << patch().name() << " : "
    << "\n\n(Ridders' Method - Total iterations = " << t
    << " \n\n(Water Flux, " << fluxType_ << ": "
    << Flux << " [kg/\(h\cdot m^2\)]")
    << "\n\n(\n\n(Draw)\n\nFeed mA estimate (across membrane): "
    << "\n\n(t)" << drawm << "  ||  " << feedm
    << "\n" << endl;
std::ofstream flux1("FluxOutput.txt", std::ios::app);
if (flux1.is_open())
    [ fux1 << flux
        << "\n"; }

std::ofstream flux2("CurrentFlux.txt");
if (flux2.is_open())
    [ fux2 << flux
        << "\n"; }

fixeValueFvPatchVectorField::updateCoeffs();
}

void EFOMVFpPatchVorticity::initAllUs()
{
calcFaceMapping();

// Fill out the fs_ list so it contains the indices of all the "feed-side" faces:
if (nagP_forwardDirection_ < VSMALL)
{
    info << patch().name()
        << " : forward direction specified by massFraction\n";
}

// The forward direction of the membrane has not been defined by the user so
// the mass fraction will be used to determine the feed side:
const fvPatchScalarField& mAsf
    = patch().lookupPatchField<volScalarField, scalar>(mName_);

tmp<vectorField> tfvnf;
    = mAsf.patchInternalField();

cost scalarField tvnf;
    = tfvnf();

forAll(fs_, facet)
{
    if (mAsf[facet] > mAsf[fn_[facet]])
        fn_[facet] = facet;
}

else
{
    info << patch().name()
        << " : forward direction specified by user\n";
}

forAll(fs_, facet)
{
    if (fn_[facet] ? fn_[facet]
        : facet;
}
}
std::ofstream flux1("FluxOutput.txt", std::ios::app);
if (flux1.is_open())
{
    flux1 << Flux
    << "\n";
}

std::ofstream flux2("CurrentFlux.txt");
tf(flux2.is_open())
{
    flux2 << Flux
    << "\n";
}
}
}
}


} // Patch Field definition here.

void EFGVFVPatchVectorField::initialise()
{
    calcFaceMapping();

    // Fill out the fs_ list so it contains the indices of all the 'feed-side' faces:
    if (nag(forwardDirection_) == VSMALL)
    {
        Info << patch().name() << " : forward direction specified by mass fraction\n"
        "end;
    }

    // The forward direction of the membrane has not been defined by the user so the
    // mass fraction will be used to determine the feed side:
    const TVPatchScalarField& MpsF
    = patch().lookupPatchField<volScalarField, scalar>(vName_);
    tmp<scalarField> tMpsF
    = MpsF.mixedInternalField();
    const scalarField& MpsF
    = tMpsF();

    forAll(fs_, facet)
    {
        fs_[facet] = (MpsF(facet) > MpsF[fn_[facet]]) ? fn_[facet] : facet;
    }
}

else
{
    Info << patch().name() << " : forward direction specified by user\n"
    << "end;
}

    tmp<vectorField> tvnfF
    = patch().nf();
    const vectorField& vnfF
    = tvnfF();

    forAll(fs_, facet)
    {
        fs_[facet] = (vnfF(facet) & forwardDirection_) < 0.0
            ? fn_[facet] : facet;
    }
}
540 void EFMUVfPatchVectorField::calcFaceMapping()
541 {
542   // Set up the face-index mapping based on cell centers
543   const vectorField& cvf = patch().Cf();
544   forAll(cvf, facei)
545   {
546     label label = 0;
547     for(label i=0; i<cvf.size(); ++i)
548     {
549       if (facei==i)
550         {
551           if (mag(cvf[facei]-cvf[i])<1e-9)
552             fn_[facei];
553         }
554         
555         Info << "patch face "
556             << facei << " -> " << i
557             << endl;
558         break;
559     }
560   }
561 }
562 }
563 
564 }/**
565 
566 */
567 
568 scalar EFMUVfPatchVectorField::fluxEquation(
569    const scalar& Jvalue,
570    const scalar& feedmA,
571    const scalar& drawmA
572 )
573 { 
574     if( FluxType_ == "simple" || B() < SMALL )
575     { 
576       /* Implicit flux equation,
577       Valid when B is low compared to other terms, i.e. high rejection.
578       See "Modelling Water Flux in Forward Osmosis: Implications for
579       Improved Membrane Design:
580      American Institute of Chemical Engineers, vol 53, No. 7, p. 1730-1744
581     */
582     
583     //eqn 2.14: the reduced assumption case:
584     return Jvalue
585       
586       - ( ( pl_nMacoff().value() )
587         + ( drawmA*exp( Jvalue*K() )
588           * feedmA
589         );
590     }
591 
592 }/**
593 
594 */
595 
596 */
597 else if( FluxType_ == "advanced" )
598 { 
599     /* Implicit flux equation
600     Valid at any B-value.
601     See "Coupled effects of internal concentration polarization and
602     Fouling on flux behaviors of forward osmosis membranes during
603     Humic Acid filtration" Journal of Membrane Science 354 (2010) 123-133
604     */
605     
606     //eqn 2.13: the non-reduced case:
607     return Jvalue
608       - ( ( pl_nMacoff().value()*drawmA
609         + ( Jvalue << B() )
610         * feedmA
611         * Jvalue << B() )
612         * Jvalue
613     }
614 
615 
616 }/**
617 
618 */
619 
620 
621 
622 }/**
623 */
// The only other option is that the user has not entered a correct statement/value:
else
{
    FatalError
    // In the file: explicitFOMembraneVelocity.C
    "No flux model was selected" << abort(FatalError);
}
return 0;
}

// This if statement asks if 0.0 is defined within the two bounds of the boundary functions:
if ( (fl > 0.0 && fh < 0.0) || (fl < 0.0 && fh > 0.0) )
{
    // Save bounds in new variables.
    scalar xl = minBound;
    scalar xh = maxBound;
    // An unlikely value, to simplify logic below
    scalar ans = -1.1e-30;
    // Variables used
    scalar x, f, s, xnew, fn;

    // Iteration counter
    while ( (int) j < 50 ; j++)
    {
        // Mean of the bound values:
        scalar x = 0.5*(xl+xhr);
        // Mean boundary function:
        f = fluxEquation( x, feedMen, drawMen );
        // The distance between the mean and the higher end lower bounds:
        s = sqrt( (f+fn- fl*fh );
        // The first evaluative function:
        if ( s < SMALL )
        {
            return ans;
        }
        // Update the formula and check answer
        if (fl >= fh)
        {
            xnew = x + (x-xl)*fx/s;
        }
        else
        {
            xnew = x - (x-xl)*fx/s;
        }
        if ( nag( xnew - ans ) <= xacc)
        {
            return ans;
        }
    }

}
/* ***********************************************/

// Redefines for the value evaluated with the logic limits:
ans = xnew;
// Definition of the new boundary condition via xNew:
fnew = fluxEquation( ans, feedmen, drawmen );

// Second evaluative function:
if ( mag(fnew) < SMALL )
{
  return ans;
}

/* ***********************************************/

// Bookkeeping to keep root bracketed on next iteration
if (checkSign(fn,fnew))
{
  xl = xm;
  sh = fn;
  th = fnew;
}
else if (checkSign(fl,fnew))
{
  sh = ans;
  Th = fnew;
}
else if (checkSign(fh,fnew))
{
  xl = ans;
  fl = fnew;
}
else
{
  fatalErrorIn
  /* In the file: explicitIFOMembraneVelocity.C */
  "Error in search logic" << abort(fatalError);
}

/* ***********************************************/

// Check the bounds
if ( mag( sh - xl ) <= xacc)
{
  return ans;
}

/* ***********************************************/

else
{
  if( mag(fl) < SMALL )
  [ return minBound;
  ]
  if( mag(fh) < SMALL )
  [ return maxBound;
  ]
  return 0;
}

// End file
Appendix B:
OpenFOAM Solver Source Code

This Appendix contains files pertaining to the two potential solvers and the SaltPimpleFoam solver developed in this thesis. These solvers are compiled as separate libraries, which are then referenced within a case through the system/controlDict file. The page index for the .C and related header files pertaining to each solver is found below.

Potential Solvers:
PotentialSolute1st.C............................................................124
PotentialSolute2nd.C...........................................................125

SaltPimpleFoam Solver:
SaltPimpleFoam.C (the main solver file).................................126
createFields.H (instantiates flow field)....................................127
mAContinuity.H (checks for mass continuity)............................129
mAEqn.H (solves the D-C equation)........................................129
mAInitialContinuity.H (finds the initial mass in the computational domain).........................................................129
pEqn.H (PISO algorithm correction steps).................................130
UEqn.H (assembles the N-S equation)......................................131
```cpp
#include "fvCFD.H"
int main(int argc, char *argv[])
{
    argList::validOptions.insert("write", "");
    #include "setHostCase.H"
    // set up case directory
    #include "createTime.H"
    // Instantiate time data base
    #include "createMesh.H"
    // create mesh object
    #include "readControls.H"
    // read included control file
    #include "createFields.H"
    // Instantiate flow fields
    Info<< nl << "Calculating potential flow" << endl;
    runtime.functionObjects().start();
    // Since solver contains no time loop it would never execute function objects: must do it ourselves.
    for (int nonOrth=0; nonOrth<=nNonOrthCorr; nonOrth++)
    {
        fvScalarMatrix pEqn;
        
        fvm::laplacian
        (       
            dimensionedScalar
            (       
                "1",
                dimTime/p.dimensions())dimensionSet(0,2,-1,0,0),
            P       
        )
        == fvc::div(phl);

        pEqn.setReference(pRefCell, pRefValue);
        pEqn.solve();
        if (nonOrth == nNonOrthCorr)
        {
            phl = pEqn.flux();
        }
    }

    Info<< "continuity error = "
    << mag(fvc::div(phl)).weightedAverage(mesh.V()).value()
    << endl;
    U = fvc::reconstruct(phl);
    U.correctBoundaryConditions();

    Info<< "Interpolated U error = "
    << (sqrt(
        sum(sqr((fvc::interpolate(U) & mesh.Sf()) - phl))
    )/sum(mesh.magSf())).value()
    << endl;
    // Force the write
    U.write();
    p.write();
    Info<< "ExecutionTime = " << runtime.elapsedCpuTime() << " s"
    << " time = " << runtime.elapsedClockTime() << " s"
    << endl;
    Info<< "End\n"
    << endl;
    return 0;
}
```
PotentialSolute2nd.C

/* ------------------------------ C++ ------------------------------ */

/#include "fvCFD.H"

/#include "setHostCase.H"
/#include "createMesh.H"
/#include "readControls.H"
/#include "createFields.in"

// -------- * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

int main(int argc, char *argv[])
{
  argList::validOptions.insert("write", ";");
  // #include "setHostCase.H"
  // #include "createMesh.H"
  // #include "readControls.H"
  // #include "createFields.in"

  Info<< nl << "Calculating potential flow" << endl;
  runTime.functionObjects().start();
  adjustPhi(p, U, p);

  for (int nonOrth=0; nonOrth<4; nonOrth++)
  {
    fvScalarMatrix pEqn;
    fv::laplacian
    dimensionedScalar
    "
    
    dimensionSet(0,2,-2,0,0),
    
    p);
    
    pEqn.solve();
    if (nonOrth == 4)
    {
      phi = pEqn.flux();
    }
  }

  Info<< "continuity error = "
  << mag(fvc::dlv(pphi)).weightedAverage(mesh.V()).value()    
  << endl;
  U = fvc::reconstruct(phi);
  U.correctBoundaryConditions();
  Info<< "Interpolated U error = "
  << sqrt((
    sumSqr(fvc::Interpolate(U) & mesh.Sf()) - pphi))
    
    /sum(nmesh.magSf())
  .value()    
  << endl;
  // force the write
  U.write();
  p.write();
  phi.write();
  rho.write();
  Info<< "Execution time = " << runTime.elapsedCpuTime() << " s"    
  << endl;
  Info<< "End()" << endl;
  return 0;
#}
SaltPimpleFoam.C

#include "bound.H"
#include "FvPatch.H"
#include "pimpleControl.H"
//Pimple functionality.
#include "fvOptions.H"

//Option to Include in-out finite volume source terms.
#include "IMPPZonesList.H"

//finite volume check functionalities used to correct U.
#include "FixedFluxPressureFvPatchScalarField.H"

//Boundary condition functionality which adjusts the pressure gradient such that the flux on the boundary is that specified by the velocity boundary condition.
int main(int argc, char *argv[]) {
  //C++ mats function: 'int argc' is number of parameters, 'char *argv[]' the number of actual parameters while running the application

  #include "setRootCase.H"
  #include "createMesh.H"
  #include "createTime.H"
  #include "readGravityionalAcceleration.H"
  #include "InitContinuityErrs.H"
  #include "createFields.H"
  #include "createOptions.H"
  #include "PresSitialContinuity.H"
  #include "readTimeControls.H"
  #include "CourantNo.H"
  #include "setInitialDeltaT.H"

  pimpleControl pimple(mesh);
  //PIMPLE control class to supply convergence information/checks for the PIMPLE loop.

  Info << "Starting time loop:\n" << endl;

  while (runTime.run()) {
    #include "readTimeControls.H"
    #include "compressibleCourantNo.H"
    #include "setInitialDeltaT.H"

    runTime++;

    Info << "time = "
        << runTime.timeName()
        << nl
        << endl;

    //******************************************************************************/
    // PIMPLE corrector loop
    while (pimple.loop()) {
      #include "RhoP.H"

      // PISO corrector loop
      while (pimple.correct()) {
        //FLUX CORRECTOR STEP
        #include "MAG.H"
        //PRESSURE CORRECTION STEP
        #include "pEqn.H"
      }

      //******************************************************************************/
      rho = rho == (1.0 + rho,maCoeff*PA);
      runTime.write();

      Info << "mContinuity.H"
        << "Checks for continuity of m across the membrane's opposite faces. simulation will stop running if this equation is not satisfied.

      //******************************************************************************/

      Info << "ExecutionTime = "
        << runTime.elapsedPTime() << " s"
        << "CreditTime - "
        << runTime.elapsedClockTime() << " s"
        << "m << nl
        << "m << endl;

    Info << "End\\n" << endl;
  return 0;
}
createFields.H

1 Info << "Reading field p" << endl;
2 volScalarField p
3 {
4   IOobject
5     (p,
6      runtime.timeName(),
7      mesh,
8      IOobject::MUST_READ,
9      IOobject::AUTO_WRITE
10    ),
11    mesh
12);
13
14 /***********************************************************************/
15 Info<< "Reading field mA" << endl;
16 volScalarField mA
17 {
18   IOobject
19     (mA,
20      runtime.timeName(),
21      mesh,
22      IOobject::MUST_READ,
23      IOobject::AUTO_WRITE
24    ),
25    mesh
26    );
27
28/***************************************************************************/
29 Info<< "Reading field U" << endl;
30 volVectorField U
31 {
32   IOobject
33     (U,
34      runtime.timeName(),
35      mesh,
36      IOobject::MUST_READ,
37      IOobject::AUTO_WRITE
38    ),
39    mesh
40    );
41
42/***************************************************************************/
43 Info<< "Reading transportProperties" << endl;
44 IOdictionary transportProperties
45 {
46   IOobject
47     (transportProperties,
48      runtime.constant(),
49      mesh,
50      IOobject::MUST_READ
51    );
52
53}  //This allows access to the transportProperties file within the constant directory of the case. Since the constant are fixed by definition, the 'IOobject::AUTO_WRITE' command is omitted.
54
dimensionedScalar pl_mCoeff(transportProperties.lookup("pL_mCoeff"));
55 dimensionedScalar nu(transportProperties.lookup("nu"));
56 dimensionedScalar nu_mCoeff(transportProperties.lookup("nu_mCoeff"));
57 dimensionedScalar babMin(transportProperties.lookup("babMin"));
58 dimensionedScalar babCoeff(transportProperties.lookup("babCoeff"));
59 dimensionedScalar bab_mCoeff(transportProperties.lookup("bab_mCoeff"));
60 dimensionedScalar rho_mCoeff(transportProperties.lookup("rho_mCoeff"));
61
do
62 volScalarField rho
63 {
64   IOobject
65     (rho,
66      runtime.timeName(),
67      mesh,
68      IOobject::NO_READ,
69      IOobject::AUTO_WRITE
70    ),
71    mesh
72    );
73
74}  //Will put a rho file within each time file, and saves it as a prior t step.
75
#include <compressibleCreatePhi.H>

//Will write an initial phi file which allows phi to depend on rho.

Info << "Flow is assumed to be laminar:\n"
<< "Turbulence is currently unsupported in this solver"
<< endl;

Info << "Calculating field (g,h)\n" << endl;

surfaceScalarField gh = surfaceScalarField(
  "gh",
  g & mesh.Cf()
);

label phiRefCell = 0;
scalar phiRefValue = 0.0;

setRefCell(
  p,
  mesh.solutionDict().subDict("PIMPLE"),
  phiRefCell,
  phiRefValue
);

//Formalizing of the reference information with respect to the PISO solver loop.
mACContinuity.H

1 2 scalar mA_influxOld = mA_influx;
3 scalar mA_massOld = mA_mass;
4
5 **************************************************************/
6
7 //Evaluate then reset the total solute mass contained within the geometry.
8 mA_mass = fvc::domainIntegrate(mA * rho * rhoMACoeff).value();
9 Info << "ln"
10 << mA.name()
11 << " mass: " << mA_massOld
12 << "new = " << mA_mass
13 << endl;
14 mA_influx = 0.0;
15
6 **************************************************************/
17
18 //Evaluate the salt through the membrane via patch. '-' is due to the stated membrane
direction, for which salt permeation is generally in the opposite direction of the
water flux (the pos flux direction).
19 forAll(mA.boundaryField(), patchi)
20 {
21 mA_influx += sum(
22       phi.boundaryField()[patchi] / rho.boundaryField()[patchi] * mA.boundaryField()[patchi];
23 )
24
25 mA_influx -= rho0.value() * rhoMACoeff.value();
26 //Alter the influx total according to each patch's permeate density.
27 Info << "ln"
28 << mA.name()
29 << " influx: " << mA_influxOld
30 << "new = " << mA_influx << "ln"
31 << endl;
32}
33

mAEqn.H

1 volScalarField rho0ab;
2 {
3   "rhoab",
4   rho + max(
5     DabCoeff * (1.0 - Dab_MACoeff * mA),
6     0
7   )
9);
10 //Explicitly [due to the fact that it is a volume scalar field and is calculated with
11 //constants] calculates the quantity rho0ab field from the mA field according to
12 //equation 3.31 in the thesis [altered due to an error within the thesis’ equation and
13 //the corresponding implemented code].
11
12 **************************************************************/
13
14 solve
15 {
16   fvm::ddt(rho, mA)
17   >> fvm::div(phi, mA)
18   >> fvm::laplacian(rho0ab, mA)
19         );
20 //This solves equation 3.36 using the newly calculated rho0ab value (from above), rho
21 //value (from the previous iteration's phi), and phi value (from the previous
22 //iteration's phi) to get a new mA field. This equation MAY NOT hold for nano or
23 //angstrom scale membrane pores.
21
22 Info << mA.name() << " Info:
23 << "max = " << max(mA).value()
24 << "max = " << mA.value()
25 << "ln" << endl;
26}
27
28 **************************************************************/
29
 mAInitialContinuity.H

1 scalar mA_influx = 0.0;
2 //The membrane salt-transmittance
3
4 scalar mA_mass = fvc::domainIntegrate(
5       rho0ab
6       * rho_MACoeff
7       * mA
8 ) .value();
9 //Calculates total mA mass over system geom for tub and stores as a scalar
13 Info << "mA: Initial mass = " << mA_mass
14 << endl << endl;
pEqn.H

1 rho = rho0 * (1.0 + rho_nhCoeff * NA);
2 // Explicitly calculate the density field with Equation 3.31.
3 
4 /******************************************************************************
5 */
6 // Calculate and store the 1/A term as both body and face values:
7 vo[ScalarField RAU("RAU", 1.0/UEqn.A)];
8 surfacScalarField RAUF("RAUF", fvc::interpolate(RAU));
9 
10 // Temporary U field for U alteration accounting in this iteration (replaces U):
11 vo[vectorField HByA("HByA", U)];
12 
13 // Equation 3.52:
14 HByA = RAU/UEqn.H();
15 
16 /******************************************************************************
17 */
18 // Calculates and temporarily stores the cell surface velocity flux field within the
19 // mesh via equation 3.54. This is the 'first flux divergence argument':
20 surfacScalarField phHByA
21 {
22 "phHByA",
23 fvc::interpolate(rho) * 
24 {
25 (fvc::interpolate(HByA) & mesh.Sf())
26 + fvc::interpolate(rho*RAU)*fvc::ddtCorr(rho, U, phi)
27 / fvc::interpolate(rho)
28 );
29 }
30 // 'ddtCorr' accounts for the divergence of the face velocity field by taking out the
31 // difference between the interpolated velocity, uH, rho, and flux.
32 fOptions.makenRelative(phHByA);
33 adjustPhi(phHByA, U, p);
34 
35 /******************************************************************************
36 */
37 // Update the fixedFluxPressure BCS to ensure flux consistency between U and
38 // temporary U;
39 setNGrad=fluxFixedPressureFVPatchScalarField
40 {
41 p.boundaryField(),
42 (pHByA.boundaryField())
43 - fOptions.relative(mesh.Sf().boundaryField() & U.boundaryField())
44 )*(mesh.nAbsf().boundaryField()+RAU.boundaryField())
45 );
46 /******************************************************************************
47 */
48 // The 'second flux divergence argument':
49 surfacScalarField phlg
50 {
51 -> fvc::interpolate(rho)
52 * ghf
53 -> fvc::snGrad(rho)
54 * RAUF
55 * mesh.nAbsf()
56 );
57 
58 phHByA *= phlg;
59 
60 /******************************************************************************
61 */
62 // Non-orthogonal pressure corrector loop
63 while (pimple.correctNonOrthogonal())
64 {
65 // Pressure corrector
66 fvScalMatrix pEqn
67 {
68     fvm::laplacian(fvc::interpolate(rho)*RAU, p)
69 ==
70     fvc::div(phHByA)
71 + fvc::ddt(rho);
72 }
73 // This equation assembles the pressure matrix equation found in eqn 3.53: the flux,
74 // rho and RAUF terms found via eqn updating earlier are used here.
75 /******************************************************************************
76 */
77 pEqn.setReference(pRefCell, pRefValue);
78 pEqn.solve(mesh.solver(p.select(pimple.finalInnerIter())));
79 }
88 // If the last correction step has been completed, then the new, conservative flux
field is rewritten to be that of the new pressure equation's flux matrix.
89   if (pimple.finalNonOrthogonalIter())
90     {
91       phl = phlHbyA - pEqn.flux();
92
93   // Calculates the new velocity field with the new pressure and flux so that it can be
94   // used in the nAeqn that precedes the pEqn within the following iteration within
95   // the .c solver's PISO loop file.
96     U =  HbyA
97     + rAU
98     / rho
99     * fvc::reconstruct((phlg - pEqn.flux())/rAU);
100     U.correctBoundaryConditions();
101   }
102
103   /**************************************************************************/
104
105   #include "continuityErrs.H"
106
107 }
108

UEqn.H

1 volScalarField mu
2 {
3   "mu",
4   mu8*(1.0 + mu_nACoeff*nA)
5 );
6   // Explicitity [due to the fact that it is a volume scalar field and is calculated with
7   // constants] calculate the viscosity field from the nA field according to equation 3.32.
8   /**************************************************************************/
9
10 fvVectorMatrix UEqn
11  // The following is equation 2.87:
12  {
13    fvm::ddt(rho,U)
14    + fvm::div(phi,U)
15    - fvm::laplacian(mu,U)
16    - fvc::div( mu*n*dev2f fvc::grad(U)((),T()) )
17    == fvOptions(rho,U);
18  );
19  // ()T()` creates a transpose version of the fvc::grad(U) matrix. dev2 is the
20  // deviatoric component of the matrix following it, and translates to [dev(A)=A-
21  // (1/3)*trace(A)]. This information can be found in OpenFOAM\primatives\Tensor
22  \TensorI.H.
23  UEqn.relax();
24  fvOptions.constrain(UEqn); |
25  /**************************************************************************/
26
27  // Eqn. 3.49: Obtains an initial approximination [intermediate field] of the velocity
28  // field that does not depend on the continuity equation. It takes the balance of the
29  // gravitational and pressure body forces and maps them onto the mesh via a 'surface
30  // scalar field' according to face area magnitude.
31  // i.e. generates a cell-based volumetric field from the face-flux field.
32  if (pimple.momentumPredictor())
33    {
34      solve( UEqn == fvc::reconstruct
35             {
36                - gH*fvc::snGrad(rho)
37                - fvc::snGrad(p)
38                + mesh.magSf()
39          };
40    }
41    fvOptions.correct(U);
Appendix C:

OpenFOAM Basic Case File System Source Code

This Appendix contains the code pertaining to the files in a basic FO OpenFOAM case which are discussed in the thesis. The page index for these files is found below. For clarity on the locations of the files found both in this appendix and Appendix D, a diagram mapping all files and folders within the basic DAKOTA/OpenFOAM case setup is included. In this diagram, the names of files that can be found in this appendix and Appendix D are bolded.

Total DAKOTA and OpenFOAM Case File Reference .......................................................... 133

Relevant OpenFOAM Case Files:
- 0/mA .......................................................................................................................... 134
- 0/p ............................................................................................................................. 135
- 0/U ............................................................................................................................. 136
- system/controlDict ................................................................................................. 137
- system/fvSchemes .................................................................................................... 138
- system/fvSolution ................................................................................................... 139
```
#include "./O/include/nonDaffies"
#include "./O/include/nonDaffies"
```
```cpp
1 /************************************************************** C++ */
2 | Field | OpenFOAM: The Open Source CFD Toolbox |
3 |        | Version: 2.1.1 |
4 \| / parallel |
5 |        | Web: www.OpenFOAM.org |
6 \| / Multiphase |
7 \*-----------------------------------------------------------*/
8 FoamFile
9 |
10 version 2.0;
11 format ascii;
12 class volScalarField;
13 location "p";
14 object p;
15
16 // ******************************************************************************
17 //Dimensions [0 2 -1 0 0 0 0]; //this is needed for the potential solver. It is
18 //later changed to [1 -1 2 0 0 0 0] for the SaltPimpleFoam solver.
19 boundaryField uniform 1000.0;
20 //this value tends to be arbitrary for FO due to the nature of the potential solver.
21 boundaryField
22 |
23 |
24 |
25 upperFeedEntry
26 |
27 |
28 |
29 |
30 |
31 |
32 |
33 |
34 |
35 |
36 |
37 |
38 |
39 |
40 |
41 |
42 |
43 |
44 |
45 |
46 |
47 |
48 |
49 |
50 |
51 |
52 |
53 |
54 |
55 |
56 |
57 |
58 |
59 |
60 |
61 |
62 |
63 |
64 |
65 |
66 |
67 |
68 |
69 |
70 |
71 |
72 |
73 #include "../include/nonMastication"
74 //its uncommented after All 'sed' commands:
75 //#include "../include/nonMasticated"
76 }
``
// Inlet averaged velocities range from .001 m/s to 1 m/s
// Held at 0.1 in this thesis.

dimensions [0 1 -1 0 0 0];

internalField uniform (0 0 0);

boundaryField
{
    upperfeedEntrance
    {
        type groovySC;
        valueExpression "vector( axyp(i1-(yp/h)), 0, 0 )";
        gradientExpression "1";
        fractionExpression "1";
        evaluateDuringConstruction 0;
        variables "um=0.1;yp=pos().y-0.85;h=0.002;a=(6*um)/(h);";
        value uniform (0 0 0);
    }

    upperfeedExit
    {
        type zeroGradient;
    }

    drawEntrance
    {
        type groovySC;
        valueExpression "vector( axyp(i1-(yp/h)), 0, 0 )";
        gradientExpression "vector(0,0,0)";
        fractionExpression "1";
        evaluateDuringConstruction 0;
        variables "um=0.1;yp=pos().y-0.85;h=0.002;a=(6*um)/(h);";
        value uniform (0 0 0);
    }

    drawExit
    {
        type zeroGradient;
    }

    lowerfeedEntrance
    {
        type groovySC;
        valueExpression "vector( axyp(i1-(yp/h)), 0, 0 )";
        gradientExpression "vector(0,0,0)";
        fractionExpression "1";
        evaluateDuringConstruction 0;
        variables "um=0.1;yp=pos().y-0.85;h=0.002;a=(6*um)/(h);";
        value uniform (0 0 0);
    }

    lowerfeedExit
    {
        type zeroGradient;
    }

top
    {
        type fixedValue;
        value uniform (0 0 0);
    }

    mid
    {
        type fixedValue;
        value uniform (0 0 0);
    }

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

    front
    {
        type empty;
    }

    back
    {
        type empty;
    }

    #include "./0/include/UMaffles"
    // Is uncommented after all `sed' commands.
    #include "./0/include/used"
system/controlDict

// Filled by OpenFOAM: The Open Source CFD Toolbox
// Version: 2.1.1
// Web: www.OpenFOAM.org

FoamFile

version 2.0;
format ascii;
class dictionary;
location "system";
object controlDict;

//**************************************************/

libs

"libOpenFOAM.so"
"libSimpleWakFunctionObjects.so"
"libWakFunctionObjects.so"
"libGroovyBC.so"
"AllUpperMembraneBoundaryConditions.so"
"AllLowerMembraneBoundaryConditions.so"

application SolfSimpleFoam;
startFrom latestTime;
startTime 0;
stopAt endTime;

//ENDTIME INDEX
endTime 10;
deltaT 0.00005;
writeControl adjustableRunTime;
writeInterval 10;
purgeWrite 0;
writeFormat ascii;
writePrecision 10;
writeCompression uncompressed;
timeFormat general;
timePrecision 10;
runTimeModifiable yes;
adjustTimeStep yes;
maxCo 0.7;
//could be between 0.5 and 1.0
maxDeltaT 0.5;
system/fvSchemes

1 /**
2 *----------------------------------------------------------------------------*/
3 \ \\
4 /// Field OpenFOAM: The Open Source CFD Toolbox
5 /// \\
6 /// Operation Version: 2.1.1
7 /// \\
8 /// Tecplot
9 |
10 | version 2.0;
11 |#
12 | class dictionary;
13 | location "system";
14 | object fvSchemes;
15 |
16 // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
17 |
18 ddtSchemes
19 |
20 //default Euler;
21 //for all dt. Euler is needed due to transience.
22 |
23 |
24 gradSchemes
25 |
26 //default Gauss linear;
27 | grad(p, rho) Gauss linear;
28 | grad(u) Gauss linear;
29 |
30 //for the UEqn (see divSchemes).
31 |
32 |
33 divSchemes
34 |
35 //default none;
36 | div(phi) bounded Gauss linear upwind;
37 |
38 //for the pEqn.
39 | div(p) bounded Gauss linear upwind grad(u);
40 | div(nu*dev2(grad(u).T())) Gauss linear;
41 | div(p, n) bounded Gauss linear upwind grad(n);
42 |
43 laplacianSchemes
44 |
45 //default none;
46 |
47 | laplacian(I, p) Gauss linear corrected;
48 | laplacian(rho*(I*A(U)), p) Gauss linear corrected;
49 |
50 //for the pEqn.
51 | laplacian(m, u) Gauss linear corrected;
52 |
53 interpolationSchemes
54 |
55 //default linear;
56 |
57 | interpolate(u) linear;
58 | interpolate(rho) linear;
59 |
60 //all for the pEqn.
61 |
62 enGradSchemes
63 |
64 //default corrected;
65 |
66 fluxRequired
67 |
68 //default no;
69 |
70 | p yes;
71 }
```
 foamFile

 version 2.0;
 format ascii;
 class dictionary;
 location "system";
 object fvSolution;

// ************************************************************************* //
solvers
{
    p
    {
        solver PCG;
        preconditioner DIC;
        tolerance 1e-05;
        relTol 0.075;
    }

    U
    {
        solver PBICG;
        preconditioner DIU;
        tolerance 1e-04;
        relTol 0.5;
    }

    rho
    {
        solver smoothSolver;
        smoother syncGaussSeidel;
        tolerance 1e-04;
        relTol 0;
    }

    pFinal
    {
        p;
    }

    UFinal
    {
        U;
    }

    rhoFinal
    {
        $rho;
    }

    potentialFlow
    {
        wenoOrthogonalCorrectors 10;
    }

    PIMPLE
    {
        momentumPredictor yes;
        nOuterCorrectors 1;
        nCorrectors 2;
        wenoOrthogonalCorrectors 10;
        pRefValue 0;
        pRefPoint (0 0 0);
    }
```
relaxationFactors
{
  fields
  {
    p 0.5;
    //50% of old solution must be used. Since p is used to calculate U and mA, this
    factor is more restricted than the others.
  }
  equations
  {
    U 0.7;
    mA 0.2;
  }
}
Appendix D:
DAKOTA Source Code

This Appendix contains the code pertaining to the files in a basic DAKOTA case which are discussed in the thesis. The page index for these files is found below. For clarity on the locations of the files found both in this appendix and Appendix C, a diagram mapping all files and folders within the basic DAKOTA/OpenFOAM case setup can be found in Appendix C. In this diagram, the names of files that can be found in this appendix and Appendix C are bolded.

Total DAKOTA and OpenFOAM Case File Reference.................................................................133

Relevant OpenFOAM Case Files:
baffle_ga.in.....................................................................................................................142
ConstraintsCheck.py ........................................................................................................143
ObjFncEval.py ..................................................................................................................144
pLoss.C..............................................................................................................................145
simulation............................................................................................................................147
snappyHexMeshDict.template.........................................................................................149
baffle_ga.in

0 environment
1 graphics
2 tabular_graphics_data
3 tabular_graphics_file = 'baffle.dat'
4
5 method
6 soga
7
8 output verbose
9 seed = 10001
10 final_solutions = 5
11 max_function_evaluations = 2000
12 population_size = 20
13 print_each_pop
14
15 initialization_type
16 unique_random
17
18 mutation_type
19 replace_uniform
20 mutation_rate = 0.1
21
22 crossover_type
23 multi_point_parameterized_binary = 2
24 crossover_rate = 0.8
25
26 replacement_type
27 elitist
28
29 fitness_type
30 merit_function
31
32 convergence_type
33 best_fitness_tracker
34 num_generations = 500
35 percent_change = 0.05
36
37 variables
38 continuous_design = 3
39
40 cdv_initial_point
41 .01875 .03000 .04925
42
43 cdv_lower_bounds
44 .01875 .01050 .02025
45
46 cdv_upper_bounds
47 .03750 .04150 .04925
48
49 cdv_descriptor
50 'x0' 'x1' 'x2'
51
52
53 interface
54 fork
55
56 analysis_driver = 'simulation'
57 parameters_file = 'params.in'
58 results_file = 'results.out'
59 work_directory directory_tag
60 template_directory = 'template'
61
62 # uncomment to leave params.in and results.out files in work_dir subdirectories:
63 named 'workDir' file_save directory_save
64 aperpro
65
66 responses
67 num_objective_functions = 1
68 #GAs do not need gradients:
69 no_gradients
70 no_hessians
71 sense 'max'
ConstraintsCheck.py

3 import os
4 import sys
5
6 n = 3
7 vals = []
8
9 f = open('draw/system/snappyHexMeshDict', 'r')
10 snappy = f.read()
11 geom_index = snappy.find('geometry')
12
13 for l in range(n):
14    baff_index = snappy.find('cylinder'+str(l)+'.geom_index')
15    point_index = snappy.find('points.'+baff_index)
16    end_index = snappy.find(' 0.0035'.point_index)
17    vals.append(float(snappy[point_index+9:end_index]))
18 f.close()
19
20 Dist = float(0.0075)
21 errorVal = '0.0'
22 boollist = [vals[i+1]-vals[i] > Dist for i in range(len(vals) - 1)]
23 f = open('2Ddraw/NOERRORscript','.w')
24 f.write('NOERROR')
25 f.close()
26
27 if not False in boollist:
28    f = open('2Ddraw/ErrorBool','.w')
29    f.write('ERROR')
30    f.close()
31 else:
32    f = open('2Ddraw/ERRORbool','.w')
33    f.write('ERROR')
34    f.close()
ObjFntcEval.py

3 import os
4
5 #case-specific variables
6 #---------------------------
7 #the total flux value from a geometrically-identical case with no implemented baffles
8 #will act as the 'original' value, and will be used to calculate the % increase of
9 #the values input into the objective function. This value will vary case-to-case.
10 J_wotOrig = float(11.047) # for the 60 x-cell case
11 J_2x_wotOrig = float(11.803) # for the 200 x-cell case
12 # the pressure loss value from a geometrically-identical case with the maximum number
13 # of implemented baffles will act as the 'max' value, and will be used to calculate the
14 # decrease of the values input into the objective function. This value will vary
15 # case-to-case.
16 PlossMax = float(132.536) # for the 60 x-cell case
17 PlossMax = float(130.118) # for the 200 x-cell case
18 #---------------------------
19 # Obtain the flux for the lower membrane:
20 #---------------------------
21 FL = open('lowerFluxOutput.txt')
22 lowVals = [float(line) for line in FL]
23 FL = len(lowVals)
24 J_Lower = float(lowVals[FL-1])
25 #---------------------------
26 # Obtain the flux of the upper membrane:
27 #---------------------------
28 FU = open('upperFluxOutput.txt')
29 upVals = [float(line) for line in FU]
30 FU = len(upVals)
31 J_Upper = float(upVals[FU-1])
32 #---------------------------
33 # Obtain the values required by the objective function:
34 #---------------------------
35 J_Lower = float(J_Lower+J_Upper)
36 JwotCmnc = 100*(J_wotOrig/J_Lower)
37 FP = open('FinalPloss.txt')
38 Ploss = [float(line) for line in FP]
39 Ploss = float(Ploss[0])
40 PlossfDec = (Ploss/PlossMax)
41 #---------------------------
42 # Calculate the objective function and export it to DAKOTA:
43 #---------------------------
44 ObjFntcVal = float(JwotCmnc * PlossfDec)
45 f = open('ObjFntcVal', 'w')
46 f.write(str(ObjFntcVal))
47 f.flush()
48 f.close()
pLoss.C

1/**************************************************************************
  
  
  OpenFOAM: The Open Source CFD Toolbox
  
  Copyright (C) 1991-2010 OpenCFD Ltd.
  
  License
  
  This file is part of OpenFOAM.
  
  OpenFOAM is free software: you can redistribute it and/or modify it
  under the terms of the GNU General Public License as published by
  the Free Software Foundation, either version 3 of the License, or
  (at your option) any later version.
  
  OpenFOAM is distributed in the hope that it will be useful, but WITHOUT
  ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
  FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public license
  for more details.
  
  You should have received a copy of the GNU General Public License
  along with OpenFOAM. If not, see <http://www.gnu.org/licenses/>.
  
  Application
  
  Description
  
  Calculates and reports the average pressure loss for the membrane channel.
  
  **************************************************************************/

#define "fvCFD.H"
#define "wetWallPatch.H"
#define "fstream"
#define "lststream"

/**
 *
 * @brief Main function
 *
 */

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, time)
    {
        runTime.setTime(timeDirs[time], time);
        Info<< "Time = " << runTime.timeName() << endl;
        fvMesh::readUpdateState state = mesh.readUpdate();

        volScalarField pEntrance
        {
            IOobject
            (
                "pEntrance",
                runtime.timeName(),
                mesh,
                IOobject::NO_READ,
                IOobject::NO_WRITE
            ),
            mesh,
            dimensionedScalar("pEntrance", dimless, 0.0)
        );

        volScalarField pexit
        {
            IOobject
            (
                "pexit",
                runtime.timeName(),
                mesh,
                IOobject::NO_READ,
                IOobject::NO_WRITE
            ),
            mesh,
            dimensionedScalar("pexit", dimless, 0.0)
        );
    }

    return 0;
}
Info<< "Reading field pIn" << endl;
volScalarField pIn
{
    IOobject
    {
        "pIn",
        runtime.timeName(),
        mesh,
        IOobject::READternal,
        IOobject::NO_WRITE
    },
    mesh
};

label entrancePatchID = mesh.boundaryIds().findPatchID("drawEntrance");
label exitPatchID = mesh.boundaryIds().findPatchID("drawExit");

pEntrance.boundaryField()[entrancePatchID] = p.boundaryField()[entrancePatchID];
pExit.boundaryField()[exitPatchID] = p.boundaryField()[exitPatchID];

const scalarField& pEnt = pEntrance.boundaryField()[entrancePatchID];
const scalarField& pExit = pExit.boundaryField()[exitPatchID];

Info<< "For drawEntrance:
" << setw(10) << min(pEnt)
<< setw(10) << max(pEnt)
<< setw(10) << average(pEnt)
<< endl;
Info<< "For drawExit:
" << setw(10) << min(pExit)
<< setw(10) << max(pExit)
<< setw(10) << average(pExit)
<< endl;
Info<< "Overall Channel Pressure Drop = "
<< (average(pEnt) - average(pExit))
<< " [kg/(m^2*s)]"
<< endl;

std::ofstream PL1("ProgressivePloss.txt", std::ios::app);
if(PL1.is_open())
{
    PL1 << (average(pEnt) - average(pExit))
    << "\n";
}
std::ofstream PL2("FinalPloss.txt");
if(PL2.is_open())
{
    PL2 << (average(pEnt) - average(pExit))
    << "\n";
}
Info<< "End\n" << endl;
return 0;
#!/bin/sh

# $1 is param.in FROM Dakota
# $2 is results.out returned to Dakota

# HAVE TO BE IN 'of2D' ALIAS:

# ----------------------------------------------V
# GENERATED VALUE IMPORT
# ----------------------------------------------V
dpro $1 snappyHexMeshDict.template snappyHexMeshDict.tm

# ----------------------------------------------V
# snappyHexMeshDict.in draw/system/snappyHexMeshDict
# ----------------------------------------------V

# ----------------------------------------------V
# CONDITIONAL-BASED ANALYSIS
# ----------------------------------------------V

# Check to see if the chosen baffle locations violate user-defined constraints, and
# choose the output patch accordingly:

cd 2Ddraw

A=$(tail -1 ErrorBool)
B=$(tail -1 NERRORScript)

if [[ "$A" = "A" ]]
then
cd ..

# User-chosen variables:
# ------------------------
# python VariablesSet.py
# ----------------------------------------------V

# Run the snappyHexMesh command sequence:
# ----------------------------------------
cd draw

blockMesh

snappyHexMesh -overwrite

cd ..;2Ddraw

rm -r constant/polyMesh

mkdir constant/polyMesh

executeMesh

createPatch -overwrite

# ----------------------------------------------V

# Replace the files within the 2D case's Omega folder with new files:
# ----------------------------------------

rm -r 0

mkdir 0

cp 0.org/a 0

cp 0.org/p 0

cp 0.org/U 0

# ----------------------------------------------V

# Uncomment the Omega folder's post-sed-command includes:
# ----------------------------------------

python sedUncomment.py

# ----------------------------------------------V

# Mesh, merge and stitch together the upper, middle and lower channels:
# ----------------------------------------
cd ..

blockMesh -case topfeed

mergeMeshes 2Ddraw topfeed

rm -r 2Ddraw/constant/polyMesh

cp -ar 2Ddraw/5e-05/polyMesh 2Ddraw/constant

cp -r 2Ddraw/5e-05

sed -i s/"top","/"j 2Ddraw/j"

sed -i s/"mid","/"j 2Ddraw/j"

stitchMesh -case 2Ddraw top3Bottom mids3Top

rm -r 2Ddraw/constant/polyMesh

cp -ar 2Ddraw/5e-05/polyMesh 2Ddraw/constant

rm -r 2Ddraw/5e-05

blockMesh -case bottomfeed

mergeMeshes 2Ddraw bottomFeed
rm -r 2Ddraw/constant/polyMesh
cp -ar 2Ddraw/Se-05/polyMesh 2Ddraw/constant
rm -r 2Ddraw/Se-05
sed -i.s/Bottom/"bottom"*/js 2Ddraw/0*
stichMesh -case 2Ddraw bottomTop midsBottom
rm -r 2Ddraw/constant/polyMesh
cp -ar 2Ddraw/Se-05/polyMesh 2Ddraw/constant
rm -r 2Ddraw/Se-05
# ---------------------------------------------

topoSet
crestBaffles
rm -r constant/polyMesh
cp -ar 5e-05/polyMesh constant
rm -r 0
mkdir 0
cp 5e-05/0 0
cp 5e-05/0 0
rm -r 5e-05
# ---------------------------------------------

# Set the solute fields and run the potential solvers:
# ---------------------------------------------
solveFields
potentialsolveTest
potentialsolve2nd
# ---------------------------------------------

# Alter the units of the O/p and O/phi files to account for incompressibility:
# ---------------------------------------------
python IncompUnitAlter.py
# ---------------------------------------------

# Run the case and find the draw channel's p loss:
# ---------------------------------------------
decomposePar
mpirun -np 4 saltPinlineFoam -parallel > decompLog
reconstructPar
# ---------------------------------------------

# Alter the system/controlDict file so that the case can be run again
# ---------------------------------------------
python VariablesPostSet.py
SaltPinlineFoam > log
# ---------------------------------------------

# POST-PROCESSING
# ---------------------------------------------
rm -r processor*
cd ..
# ---------------------------------------------

else
cd ..
# ---------------------------------------------

# OBJECTIVE VALUE EXPORT
# ---------------------------------------------
mv 2Ddraw/ObjectiveVal .
text . -n 1 ObjectiveVal > tmp.txt
text tmp.txt 2
# ---------------------------------------------

148
snappyHexMeshDict

1/-----------------------------------------------------------------------------\n 2 | Field | OpenFOAM: The Open Source CFD Toolbox | |
 3 \|/ 0peration | Version: 2.1.1 | |
 4 \|/ A nd | web: www.OpenFOAM.org | |
 6 \\| Manipulation | | |
 7 \------------------------------------------------------------------------------\n
0 FoamFile
9 |
10 | version 2.0;
11 | format ascii;
12 | class dictionary;
13 | object snappyHexMeshDict;
14 |
15 | // *******************************************************************************
16 |
17 | // Which of the steps to run:
18 | castellatedMesh true;
19 | snap true;
20 | addLayers true;
21 |
22 |
23 | // Geometry. Definition of all surfaces. All surfaces are of class searchableSurface.
24 | geometry
25 |
26 | //geomStartIterativeInput
27 | cylinder0
28 | {
29 | type searchableCylinder;
30 | point1 ( (x0) 0.0035 -0.001 ); //x0 is the x position of the first baffle.
31 | point2 ( (x0) 0.0035 0.002 );
32 | radius 0.00075;
33 |}
34 | cylinder1
35 | {
36 | type searchableCylinder;
37 | point1 ( (x1) 0.0035 -0.001 ); //x1 is the x position of the second baffle.
38 | point2 ( (x1) 0.0035 0.002 );
39 | radius 0.00075;
40 |}
41 | cylinder2
42 | {
43 | type searchableCylinder;
44 | point1 ( (x2) 0.0035 -0.001 ); //x2 is the x position of the third baffle.
45 | point2 ( (x2) 0.0035 0.002 );
46 | radius 0.00075;
47 |}
48 | //geomEndIterativeInput
49 | |
50 |
51 |
52 |
53 | // Settings for the castellatedMesh generation:
54 | castellatedMeshControls
55 |
56 | // Refinement parameters
57 | // --------------------------
58 | maxLocalCells 1000000;
59 | maxGlobalCells 2000000;
60 | minRefinementCells 10;
61 | maxloadimbalance 0.10;
62 | nCellsBetweenLevels 1;
63 |
64 |
65 |
66 | // Explicit feature edge refinement
67 | // -----------------------------
68 | features
69 | { 
70 | 
71 | };
// Surface based refinement
refinementSurfaces
{
  //refineStartIterativeInput
cylinder0
  {
    level (2 2);
  }
cylinder1
  {
    level (2 2);
  }
cylinder2
  {
    level (2 2);
  }
}
//refineEndIterativeInput
}
resolveFeatureAngle 30;

// Region-wise refinement
refinementRegions
{

  // Mesh selection
  locationInMesh (0.0024 0);
  allowFreeStandingZoneFaces true;
}

// Settings for the snapping:
snapControls
{
  nSmoothPatch 5;
  tolerance 4;
  nSolverItter 50;
  nRelaxIter 10;
}

// Settings for the layer addition:
addLayersControls
{
  relativeSizes true;

  layers
  {
    //layerStartIterativeInput
cylinder0^{+}=
    {
      nSurfaceLayers $;
    }
cylinder1^{+}=
    {
      nSurfaceLayers $;
    }
cylinder2^{+}=
    {
      nSurfaceLayers $;
    }
}
//layerEndIterativeInput
}

expansionRatio 1.15;
finalLayerThickness 0.8;

minThickness 0.1;
ngRow 0;

// Advanced settings:
featureAngle 75;

nRelaxIter 3;
nSmoothSurfaceNormals 3;
nSmoothNormals 3;
nSmoothThickness 10;
maxFaceThicknessRatio 0.5;
maxThicknessToMedianRatio 0.3;
minMedianAxisAngle 90;
bufferCellsNoExtrude 0;
layerIter 50;

// Generic mesh quality settings:
meshQualityControls

maxNonOrtho 60;
maxBoundarySkewness 20;
maxInternalSkewness 4;
maxConcave 0.05;
minVol 1e-17;
minTetQuality 1e-30;
minArea -1;

minTwist 0.95;
minDeterminant 0.0001;
minFaceWeight 0.05;
minVolRatio 0.01;
minTriangleTwist -1;

// Advanced:
smoothScale 4;
errorReduction 0.75;

// Advanced:

debug 0;
mergeTolerance 1e-6;

/*---------------------------------------------*/
References


