A NUMERICAL STUDY OF DROPLET FORMATION
AND BEHAVIOR USING INTERFACE TRACKING
METHODS

A Dissertation Presented
by
SANDEEP MENON

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

September 2011

Mechanical and Industrial Engineering
A NUMERICAL STUDY OF DROPLET FORMATION AND BEHAVIOR USING INTERFACE TRACKING METHODS

A Dissertation Presented
by
SANDEEP MENON

Approved as to style and content by:

________________________________________
David P. Schmidt, Chair

________________________________________
J. Blair Perot, Member

________________________________________
Sanjay R. Arwade, Member

________________________________________
Donald L. Fisher, Department Head
Mechanical and Industrial Engineering
To Amma and Papa
ACKNOWLEDGMENTS

First and foremost, I’d like to thank my advisor David Schmidt for his guidance, encouragement, and constant feedback during this endeavor. I’m especially grateful to him for allowing me to pursue my ideas, and providing support when I needed it most. I’m also greatly indebted towards Prof. Blair Perot, whose work and influence has altered my line of thinking. I’ve had the privilege of working with Blair during my Masters degree, and a lot of my knowledge in the area of CFD is thanks to him. I’d also like to acknowledge Prof. Sanjay Arwade and Prof. Jonathan Rothstein, who served on my committees.

I must also convey my gratitude to Hrvoje Jasak and Željko Tuković, whose vast expertise in OpenFOAM has impacted my research is so many ways. A lot of the work presented in this dissertation builds upon years of their efforts. I must thank Hrv in particular, who patiently answered all my embarrassing questions and provided helpful suggestions.

After years of learning, it’s easy to forget how little we knew in the first place. In this regard, I’d like to sincerely thank my colleagues Venkataramanan Subramanian and Shiva Gopalakrishnan for tons of support during my early days. I’ve also had the privilege of working with some really great colleagues at the CFD lab, who’ve made my stint both lively and memorable. In no particular order, I’d like to thank Kyle Mooney, Partha Valluri, Mike Martell, Kshitij Neroorkar, Tom Furlong, Nat Trask, Dnyanesh Digraskar, Raghuveer Vajapeyayajula, Mike Colarossi, Tim McGuinness, Saba Almalkie, Ali Khajeh Saeed, Bradley Shields, and Chris Zusi for such a wonderful time. A special thank you to several friends that I’ve made during
my stay at Amherst - that list would be a little too long to mention here, but you all know who you are.

I'd also like to thank Dorothy Adams, for handling so many of the administrative aspects with such ease, especially during the last few days.

I am hugely indebted to my parents and my sister for their unwavering support throughout my life - their understanding, patience and support cannot be expressed well enough through words. I hope I've made them proud, and will always give them my deepest gratitude. During my stay here at Amherst, I was also fortunate enough to meet Meredith Nelson, who is an amazing woman in so many different ways. She has been a constant source of moral support, and I hope that I can reciprocate some of all that affection she gives me.

Lastly, funding for this work was provided by the Army Research Office under Grant No. W911NF-08-1-0171, and the financial support is gratefully acknowledged.
ABSTRACT

A NUMERICAL STUDY OF DROPLET FORMATION AND BEHAVIOR USING INTERFACE TRACKING METHODS

SEPTEMBER 2011

SANDEEP MENON
B.E., PSG COLLEGE OF TECHNOLOGY
M.S.M.E, UNIVERSITY OF MASSACHUSETTS AMHERST
Ph.D, UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor David P. Schmidt

An adaptive remeshing algorithm has been developed for multiphase flow simulations using the moving-mesh interface tracking (MMIT) technique. The edge-swapping algorithm uses the Delaunay criterion (in 2D) and a dynamic programming technique (in 3D) to maximize the quality of mesh primitives surrounding edges in the mesh, and performs local remeshing to minimize interpolation errors. Edge bisection and contraction operations are also performed to adjust the mesh resolution around important features like fluid-interfaces, driven by a local length scale estimation algorithm that is efficient and easily parallelized. Flow-field interpolation after reconnection is achieved using a conservative, second-order accurate remapping scheme that can be extended to arbitrary mesh pairs. To minimize the number of mesh reconnection operations, vertices in the mesh are also moved in a manner that optimizes the quality of cells at every time step, using a spring-analogy.
based Laplacian smoother for surface meshes, and an optimization-based smoothing approach for interior points. To facilitate the simulation of large-scale problems, all smoothing and reconnection algorithms in this work have been parallelized for shared- and distributed-memory paradigms. This approach allows meshes to undergo very large deformations which are characteristic of multiphase flows, and the method is versatile enough to extend its applicability to a broad range of problems including error-driven mesh refinement, reciprocating machinery, fluid-structure interaction, and wing flapping simulations.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>v</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>vii</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>xiii</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>xiv</td>
</tr>
<tr>
<td><strong>CHAPTER</strong></td>
<td></td>
</tr>
<tr>
<td>1. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Eulerian Methods for Multi-phase Flow</td>
<td>2</td>
</tr>
<tr>
<td>1.1.1 Level Set Methods</td>
<td>3</td>
</tr>
<tr>
<td>1.1.2 Volume of Fluids</td>
<td>5</td>
</tr>
<tr>
<td>1.1.3 Marker Methods</td>
<td>11</td>
</tr>
<tr>
<td>1.2 Lagrangian Methods for Multi-phase Flow</td>
<td>12</td>
</tr>
<tr>
<td>2. GOVERNING EQUATIONS FOR MULTI-PHASE FLOW</td>
<td>16</td>
</tr>
<tr>
<td>2.1 Governing Equations for an Arbitrary Control Volume</td>
<td>16</td>
</tr>
<tr>
<td>2.2 Conditions at a Fluid Interface</td>
<td>19</td>
</tr>
<tr>
<td>2.3 Variation of Physical Properties</td>
<td>21</td>
</tr>
<tr>
<td>3. FINITE VOLUME DISCRETIZATION</td>
<td>22</td>
</tr>
<tr>
<td>3.1 Discretization of a General Transport Equation</td>
<td>25</td>
</tr>
<tr>
<td>3.1.1 Discretization of the Convection Term</td>
<td>28</td>
</tr>
<tr>
<td>3.1.2 Discretization of the Diffusion Term</td>
<td>31</td>
</tr>
<tr>
<td>3.1.3 Discretization of the Source Term</td>
<td>33</td>
</tr>
<tr>
<td>3.1.4 Boundary Conditions for Discretized Spatial Terms</td>
<td>33</td>
</tr>
<tr>
<td>3.1.5 Discretization of the Temporal Term</td>
<td>35</td>
</tr>
</tbody>
</table>
3.2 Solution to Linear Algebraic Equation Systems .................. 37
3.3 Discretization of the Navier-Stokes Equations .................... 38

4. ADAPTIVE MESH RECONNECTION ................................. 43

4.1 Mesh Reconnection for Improved Mesh Quality .................... 44
  4.1.1 Mesh Quality Measures for Tetrahedra ......................... 57

4.2 Mesh Reconnection for Length Scale Resolution .................... 60
  4.2.1 Estimation of Local Length Scale .......................... 61
  4.2.2 Estimation of Boundary Length Scale ...................... 65
  4.2.3 Edge Bisection ....................................... 67
  4.2.4 Edge Contraction ...................................... 68
  4.2.5 Face Trisection ....................................... 71

4.3 Sliver Detection and Removal ..................................... 71
  4.3.1 Elimination of Type I Cells ............................... 73
  4.3.2 Elimination of Type II Cells .............................. 75
  4.3.3 Elimination of Type III Cells .............................. 75
  4.3.4 Elimination of Type IV Cells .............................. 76

4.4 Variable Remapping after Mesh Reconnection ...................... 77
  4.4.1 Background ........................................... 78
  4.4.2 Definitions ........................................... 79
  4.4.3 Intersection algorithm .................................. 84
  4.4.4 Interpolation .......................................... 87
  4.4.5 Numerical Results ..................................... 89
  4.4.6 Linear function ........................................ 90
  4.4.7 Cosine hill function ................................... 90
  4.4.8 Sinusoidal function on simplical 3D meshes ................. 93
  4.4.9 Run-time efficiency on polyhedral meshes ................. 94
  4.4.10 Computational Fluid Dynamics test ......................... 96
  4.4.11 Divergence-free Face Flux Transfer ....................... 98

5. MESH SMOOTHING METHODS .................................... 100

5.1 Survey of Existing Mesh Smoothing Methods ...................... 100
5.2 Effective Smoothing of Curved Surfaces .......................... 105
  5.2.1 Constrained Optimization for the Node-based Measure .... 110
  5.2.2 Laplacian Smoothing with Untangling ....................... 113
6. PARALLEL MESH ADAPTATION .............................................. 117

6.1 Background ................................................................. 117
6.2 Shared Memory Parallelism ............................................. 119
   6.2.1 Threading Basics ..................................................... 120
   6.2.2 The Master/Worker Threading Model .......................... 123
   6.2.3 Thread-parallel Mesh Adaptation .............................. 126
      6.2.3.1 Fine-grained locking ........................................ 127
      6.2.3.2 Coarse-grained locking .................................... 128
6.3 Distributed Memory Parallelism .................................... 130
   6.3.1 Domain decomposition and Halo meshes ....................... 132
   6.3.2 Parallel Mesh Quality Improvement ........................... 135
   6.3.3 Parallel Length Scale Resolution ............................. 138
   6.3.4 Parallel Edge Refinement ....................................... 140
   6.3.5 Parallel Field Remapping ...................................... 143
6.4 Parallel Mesh Smoothing ............................................... 146
   6.4.1 Parallel Surface Smoothing .................................... 146
   6.4.2 Parallel Interior Mesh Smoothing ............................. 151

7. CASE STUDIES ................................................................. 152

7.1 Validation Cases for Mesh Adaptation ............................ 152
   7.1.1 Rotating and Translating Box in a Rectangular Domain ...... 152
   7.1.2 Rotating and Translating Ball in a Rectangular Domain ...... 153
   7.1.3 Internal Combustion Engine Case ............................. 154
   7.1.4 Field-based Mesh Adaptation Case ............................ 156
7.2 Validation Cases for Free-Surface Flows ........................... 157
   7.2.1 Droplet oscillations .............................................. 158
   7.2.2 Axi-symmetric Ink-jets ......................................... 159
      7.2.2.1 Pressure modulation ...................................... 160
      7.2.2.2 Temperature modulation .................................. 161
   7.2.3 Collapsing Ligament ............................................. 163
   7.2.4 Binary Droplet Collisions ..................................... 164
7.2.4.1 Head-on Droplet Collision ................. 164
7.2.4.2 Off-centre Droplet Collision ............... 166
7.2.5 Drop-on-demand Inkjet in 3D ................. 168
7.3 Validation Cases for Parallel Mesh Adaptation ......................... 169
7.3.1 Translating Circle in a Rectangular Domain ................. 169
7.3.2 Internal Combustion Engine Case ......................... 171
7.3.3 Parallel Speed-up Tests ......................... 173

8. CONCLUSIONS AND FUTURE WORK ......................... 177
  8.1 Future Work ...................................................... 179
    8.1.1 Two-phase Flows ......................................... 179
    8.1.2 Divergence-free Remapping .............................. 179
    8.1.3 Polyhedral Mesh Adaptation .............................. 180
  8.2 Closure ...................................................... 181

BIBLIOGRAPHY ................................................. 182
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 Global integral of $\phi$ using various methods</td>
<td>93</td>
</tr>
<tr>
<td>4.2 Details of meshes used for accuracy tests</td>
<td>94</td>
</tr>
<tr>
<td>4.3 Profiling test results</td>
<td>96</td>
</tr>
<tr>
<td>6.1 Equivalent pthreads function calls</td>
<td>123</td>
</tr>
<tr>
<td>7.1 Simulation parameters for the droplet oscillation case</td>
<td>158</td>
</tr>
<tr>
<td>7.2 Simulation parameters for Head-on Droplet Collision</td>
<td>165</td>
</tr>
<tr>
<td>7.3 Simulation parameters for Off-centre Droplet Collision</td>
<td>166</td>
</tr>
<tr>
<td>7.4 Itemization of components for the serial run</td>
<td>173</td>
</tr>
<tr>
<td>7.5 Itemization of components for the parallel run</td>
<td>173</td>
</tr>
</tbody>
</table>
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>SLIC and PLIC reconstruction</td>
</tr>
<tr>
<td>1.2</td>
<td>Mesh slicing for interface topology changes (See ref. [133])</td>
</tr>
<tr>
<td>3.1</td>
<td>Definition of various control-volume entities</td>
</tr>
<tr>
<td>4.1</td>
<td>Edge flipping operation using the Delaunay criterion.</td>
</tr>
<tr>
<td>4.2</td>
<td>Triangular prism mesh equivalent to Fig. 4.1</td>
</tr>
<tr>
<td>4.3</td>
<td>Complementary 2-3 and 3-2 swapping operations in 3D</td>
</tr>
<tr>
<td>4.4</td>
<td>Surface 2-2 swapping operation in 3D</td>
</tr>
<tr>
<td>4.5</td>
<td>Possible swapping sequences for 4 tetrahedra surrounding an edge</td>
</tr>
<tr>
<td>4.6</td>
<td>Possible triangulations for a 4- and 5-sided polygon</td>
</tr>
<tr>
<td>4.7</td>
<td>Klincsek’s recursive triangulation algorithm</td>
</tr>
<tr>
<td>4.8</td>
<td>Edge bisection in (a) 2D; and (b) 3D</td>
</tr>
<tr>
<td>4.9</td>
<td>Edge contraction in (a) 2D; and (b) 3D</td>
</tr>
<tr>
<td>4.10</td>
<td>Face trisection in 3D</td>
</tr>
<tr>
<td>4.11</td>
<td>Poor quality tetrahedra</td>
</tr>
<tr>
<td>4.12</td>
<td>Notations used in the sliver detection algorithm</td>
</tr>
<tr>
<td>4.13</td>
<td>Removing the Type I cell</td>
</tr>
<tr>
<td>4.14</td>
<td>Removing the Type II cell</td>
</tr>
<tr>
<td>4.15</td>
<td>Removing the Type III cell</td>
</tr>
</tbody>
</table>
6.8 Patch conversion in 2D ................................................. 144
6.9 Laplacian Smoothing ..................................................... 147
6.10 Point and edge markers for surface mesh smoothing ............... 150
6.11 Shared processor points ................................................. 151
7.1 Rotating and translating box in a rectangular domain .............. 153
7.2 Length scale contours for the 2D test case .......................... 153
7.3 Rotating and translating ball in a rectangular domain ................ 154
7.4 Geometry of the GDI engine ............................................ 155
7.5 Cut-section of the mesh ................................................ 156
7.6 Mesh adaptation using a refinement field .............................. 157
7.7 Amplitude decay history of the oscillating droplet .................... 159
7.8 Stages of inkjet evolution .............................................. 162
7.9 Thermal modulation of a continuous inkjet ............................ 163
7.10 Collapsing ligament case ($Oh = 0.1091$ and $t_c = 3.78 \times 10^{-9}s$) ................. 163
7.11 Case (h) from Qian and Law [131] .................................... 165
7.12 Head-on Droplet Collision .............................................. 166
7.13 Case (o) from Qian and Law [131] .................................... 167
7.14 Off-centre Droplet Collision ............................................. 167
7.15 Contours of length scale for off-centre droplet collision ............ 167
7.16 Drop-on-demand inkjet in 3D ......................................... 168
7.17 Translating circle in a rectangular domain ............................ 170
7.18 Length scale contours for the parallel 2D test case .................. 170
7.19 Parallel Decomposition of the GDI engine, showing halo meshes ... 171
7.20  Cell count history for four processors ......................... 172
7.21  Cell count history for the serial run ............................ 172
7.22  Parallel Reconnection: Stages of mesh evolution ............... 175
7.23  Solution time vs. processor count for 20 time-steps .......... 176
7.24  Speed-up for adaptation and flow-solver ....................... 176
7.25  Speed-up for adaptation ........................................... 176
CHAPTER 1
INTRODUCTION

In several areas of engineering and various industrial settings, multi-phase flows exhibit a strong presence. These flows occur over a range of length scales, from microscopic inkjets and engine fuel-injectors to large-scale bubble columns, coolant systems and transportation lines. The proper functioning of some these systems critically rely on multi-phase flow phenomena, whereas in other cases, they may be a hinderance to efficiency, or a side-effect that is simply too expensive to avoid. Either way, a thorough understanding of such flows and their behaviour is of paramount importance for the efficient design of such systems.

In the past, understanding of multi-phase flow regimes in industrial settings was limited to experimental observations, which required the use of expensive monitoring equipment that were either too dangerous, or caused disruptions to conventional efficient operation. Thus, the design of such systems required the use of small-scale experimental studies using convenient fluid substitutes. Scaling laws are often required to apply the results of these experiments to actual environments, which may not be well established (Bergles et al. [13]). Additionally, such studies are often time-consuming and expensive in terms of initial investment and operational costs, and paint an overall picture of the flow, while failing to capture local characteristics.

A numerical approach to the problem using Computational Fluid Dynamics is unique because it provides a detailed description of the flow-field in the entire domain, thereby allowing a multitude of statistical information to be extracted from it. This technique is limited only by the modeling assumptions (both numerical and physical),
and the computational resources devoted to the simulation, and is therefore a highly
efficient and cost-effective paradigm for the design process. Numerical simulations of
the Navier-Stokes equations for multi-phase flows is not new, but it does present a few
challenges. Most simulations solve the Navier-Stokes equations for the entire domain,
and treat the interface as an internal boundary across which the jump-conditions for
properties like density and viscosity are incorporated (Quan and Schmidt [133]). The
accuracy of the simulation is closely related to the manner in which the interface
capillary effects (including surface-tension and interface curvature) are resolved. An
under-resolved interface results in an imbalance in the surface-stress conditions (Jamet
et al. [83]), which in turn leads to spurious ‘parasitic’ currents, particularly in surface-
capturing methods like Volume of Fluids (VOF). In contrast, trying to resolve features
accurately quickly renders the problem intractable due to the high computational
expense. This leads to a situation where a compromise between accuracy and
computational cost is sought.

This first section aims to provide a detailed description of the various aspects
of the computational approach to multi-phase flows (spanning over four decades),
including their numerous benefits and caveats.

1.1 Eulerian Methods for Multi-phase Flow

Eulerian methods are characterized by the use of a fixed mesh (Cartesian or
unstructured) which does not move with the interface. A multi-phase simulation
using the Eulerian description must, therefore, define interfaces which cut across this
fixed mesh. This class can also be divided into capturing or tracking methods:

Capturing methods: This category of Eulerian methods requires the inference of
the interface locations using information on the fixed-mesh. Such information can
include volume-fractions, phase-fields and distance-functions.
Tracking methods: Eulerian tracking methods explicitly track the interface position using a variety of techniques including supplementary Lagrangian surface-grids and marker particles.

Eulerian methods are frequently preferred mainly because of their simplicity of definition, ease of incorporation into existing code frameworks, and simple extendability to three-dimensions. Although subject to accuracy requirements, these methods may also be cheaper in terms of computational cost, when compared to Lagrangian interface-tracking methods. Another attractive feature of interface-capturing Eulerian methods is the ability to automatically handle changes in interface topology, such as break-up and coalescence. A variety of approaches to multi-phase flows have been developed in the Eulerian framework, some of which are described in detail in the following sections.

1.1.1 Level Set Methods

Level set methods are a class of techniques which implicitly define an interface on an Eulerian mesh using the zero level-set of a signed distance function for a pair of fluids in the system. Formally, for an interface \( \Gamma(t) \) in \( \mathbb{R}^n \) bounding an open region \( \Omega \), a smooth function \( \phi(x,t) \) can be defined such that \( \phi(x,t) = 0 = \Gamma(t) \); while \( \phi(x,t) > 0 \) for \( x \in \Omega \), and \( \phi(x,t) < 0 \) for \( x \notin \Omega \).

The method was first devised by Osher and Sethian [117] to compute and analyze the motion of an interface under a velocity field. A particularly attractive feature of the Level Set method is the fact that changes in interface topology are well defined and can be performed automatically. Level Set methods also possess the quality of being very inexpensive in terms of computational cost, since it usually suffices to evaluate and solve for the evolution of \( \phi(x,t) \) at regions close to the interface (Peng et al. [123]), thereby making the technique very competitive with boundary-integral methods. The evolution of the level set function, \( \phi(x,t) \), given an underlying
flow-field, \( \mathbf{u} \), is governed by a transport equation which lacks a diffusion term, but numerical dissipation must be controlled carefully with an appropriate convection scheme. The unit normal, \( \mathbf{n} \), is defined by the normalized gradient of \( \varphi \) at the interface and the corresponding curvature, \( \kappa \), is given by the surface divergence of \( \mathbf{n} \).

With the evolution of \( \Gamma \) through time, a necessary requirement at every time-step is the reinitialization of \( \varphi \). This is usually done using the steady-state solution to the equation:

\[
\frac{\partial \psi}{\partial \tau} + \text{sgn}(\varphi) (|\psi| - 1) = 0 \quad (1.1a)
\]

\[
\psi(x, 0) = \varphi(x, t) \quad (1.1b)
\]

Properties such as density, \( \rho \), and viscosity, \( \mu \), are given by relations involving a smoothed Heaviside function \( (H_\alpha) \):

\[
\rho = \rho_l + (\rho_g - \rho_l) H_\alpha(\varphi) \quad (1.2a)
\]

\[
\mu = \mu_l + (\mu_g - \mu_l) H_\alpha(\varphi) \quad (1.2b)
\]

To track the evolution of the Level Set function, \( \varphi(x, t) \) (and therefore, the interface), Peng et al. [123] devised a fast evolution algorithm, with a complexity of \( \mathcal{O}(N) \), where \( N \) is the number of grid-points enclosed by the region defined by a specified cut-off width, \( \gamma \). This is achieved by masking grid-points that lie outside the cut-off region, and building a list of the points that lie within. The numerical value of \( \gamma \) depends on the advection scheme used for evolution, varying between \( 4\Delta x \) and \( 6\Delta x \). The masking step, however, is an \( \mathcal{O}(N^2) \) algorithm that is performed once per time-step.

Level set methods possess the attractiveness of simplicity, but mass-conservation remains quite challenging. Recently, hybridized methods (Sussman and Puckett [146],
Bourlioux [20], Sussman and Fatemi [145], van der Pijl et al. [157]) have been reported that take the advantage of both volume-of-fluids (VOF) and level-set methods in an effort to tackle the mass conservation issues. On similar lines, Enright et al. [51] have combined level-set and particle methods for improved accuracy of simulations. However, these methods employ local corrections to the level set field in order to satisfy mass conservation and as a result, introduce artificial fluctuations to interface curvature. Herrmann et al. [96] showed that high-order advection methods, when combined with multi-level meshes (called the Refined Level Set Grid method), have improved mass conservation properties. Parallelization efforts using Level Set methods have also been successful (Croce et al. [35]).

1.1.2 Volume of Fluids

Perhaps the most popular method for the simulation of free-surface/multi-phase flows in the Eulerian framework (and also in general) is the Volume of Fluids (VOF) approach. Current VOF methods are built on nearly three decades of research in the area - dating back to work by Noh et al. [115], Nichols et al. [113] and DeBar [42]. The basic principle behind the VOF method is the advection of a volume-fraction, \( f_k \), on an Eulerian mesh (The \( k \) subscript refers to the component index in a multi-component configuration). Any cell on this mesh can consist of a volume fraction equal to zero or one (and therefore designated as a pure cell) or, a fractional value if the cell contains an interface (called a mixed cell). The sum of all volume fractions for any given cell must be equal to one. Typically, in a multi-fluid configuration, volume fractions are specified with respect to a reference phase. This information is then used to reconstruct an interface between the fluid phases in the mixed cell.

The evolution of volume fractions, \( f_k \), given an underlying flow-field, \( \mathbf{u} \), is governed by the transport equation:

\[
\frac{\partial f_k}{\partial t} + \mathbf{u} \cdot \nabla f_k = 0 \tag{1.3}
\]
As in Level Set methods, numerical dissipation must be controlled carefully. Properties such as density, $\rho$, and viscosity, $\mu$, are given by relations involving the volume fraction $f_k$:

$$\rho = f_k \rho_l + (1 - f_k) \rho_g \quad (1.4a)$$

$$\mu = f_k \mu_l + (1 - f_k) \mu_g \quad (1.4b)$$

The basic reconstruction approach is the method by Nichols et al. [113] (also called the Simple Line Interface Construction (SLIC) technique by Noh et al. [115]), where the interface in a mixed cell is aligned with a particular Cartesian direction, and positioned to satisfy the volume fraction (shown in Fig. 1.1(c)). Although the method conserves volume and keeps the transitional area between phases over one cell, the SLIC approach is first-order accurate, and shows severe limitations in its
ability to accurately represent an interface. It does not preserve local boundedness; i.e., a volume fraction value for a cell that lies in between the values for its neighbours does not necessarily maintain the condition in plain advection. Thus, the advection of these fractions quickly deteriorates the quality of the interface, thereby generating what is commonly referred to as ‘flotsam’ - random fluid volumes which are ejected off the surface (Noh et al. [115]). Since the method extends one dimensional schemes to higher dimensions by operator splitting, it is limited to Cartesian meshes.

A simple and widely used VOF approach is the Compressive Interface Capturing Scheme for Arbitrary Meshes (CICSAM) method by Ubbink and Issa [154]. The authors of CICSAM noticed that the approach taken by Nichols et al. included a compressive downwind differencing scheme to achieve a sharp interface. Since this leads to wrinkling when the interface is tangential to the flow direction, Nichols et al. locally switched to upwind differencing when the condition occurred, which in turn leads to interface smearing. The CICSAM method locally switches to a higher order QUICK scheme, weighted by a factor based on the angle between the interface and the flow direction. This preserves boundedness while maintaining a relatively sharp, wrinkle-free interface.

An alternative approach is to approximate the interface in a mixed cell to a line (2D) or plane (3D) segment using a piecewise-linear function - the PLIC method (shown in Fig. 1.1(d)). It is usually convenient to define this segment in the Hessian-normal form \( \mathbf{n} \cdot \mathbf{r} + d = 0 \), where \( \mathbf{r} \) is a point on the segment, \( \mathbf{n} \) is its normal, and \( d \) is the signed distance from the origin to the segment. Once the segment normal is computed, \( d \) is obtained by positioning the plane such that the volume fraction is satisfied with respect to the reference cell.

A reconstruction using this technique shows better agreement with the true interface, although with minor discontinuities between the linear segments, leading to errors of \( O(\kappa h^2) \), where \( \kappa \) is the true interface curvature and \( h \) is the grid spacing.
The primary challenge in a PLIC method is defining the interface-segment normal ($n$) for such a cell. Although the intent of the PLIC method is to achieve a second-order accurate representation of the true interface, naïve normal estimation methods will often exhibit overall first-order tendencies, i.e., an inability to reproduce linear surfaces (Rider and Kothe [135]). With the basic constraint of volume conservation, almost all current VOF techniques are variants of the PLIC approach, and vary by the method in which the segment normal is estimated.

A popular approach is to define the interface normal in a mixed cell based on a local estimate of the volume-fraction gradient. Youngs’ method [164] used a simple finite-difference estimate of the volume-fraction gradient on orthogonal grids, and was shown to be first-order accurate by Kothe et al. [135]. An alternative approach, devised by Pilliod and Puckett [128], is to perform a least-squares gradient (LSG) based on error minimization principles. This method, known as the Least-square Volume Interface Reconstruction Algorithm (LVIRA, and its derivative - ELVIRA) is iterative in nature and known to be second-order accurate.

Another iterative method by Swartz [147] (although originally devised for computer-vision algorithms) uses a multi-step procedure. The algorithm is initialized by calculating a normal for the reference mixed cell using Youngs’ method, and an interface line (or plane in 3D) segment is positioned in this cell based on the estimate. Interface segments for all neighbouring mixed cells are estimated using the normal from the reference cell. A new segment is then created by connecting the averaged centroid of all surrounding mixed cell segments with the reference cell segment-centroid, and a new normal is estimated from it. This procedure is then repeated until convergence.

This method (along with Pilliod’s approach), although second-order accurate, has two drawbacks. Firstly, these techniques can sometimes be prohibitively expensive,
especially in 3D. Furthermore, both methods use an evaluation stencil that extends to the neighbouring cells, which is particularly undesirable while trying to reconstruct features that are smaller than the cluster of cells used in the estimation.

Dyadechko and Shashkov [49] overcome this situation by enriching the volume-fraction data-set \((f_k)\) with corresponding reference centroids \((x_{c}^{ref})\). This information is sufficient to define volume-conserving PLIC information without the need for neighbouring cell data and so, the method is able to resolve features as small as the cell size. The basic principle behind this approach, known as the moment-of-fluid method, is to minimize the following non-linear functional:

\[
E_{c}^{MoF}(n) = |x_{c}^{ref} - x_{c}(n)|^2
\]  

where \(x_{c}^{ref}\) is the reference material centroid and \(x_{c}(n)\) is the actual (reconstructed) material centroid with a given interface normal, \(n\). The first step in the moment-of-fluid approach is to make an initial guess for the interface normal by defining the vector between \(x_{c}^{ref}\) and the actual cell-centroid. The value of the signed distance \(d\) (for the Hessian-normal form of the plane segment), is then calculated to match the volume fraction, \(f_k\). Following this step, the centroid of the resulting polyhedron \((x_c(n))\) is computed, and the distance between the two centroids is measured.

The method is more efficient than LVIRA owing to the fact that no neighbour information is taken into account, and the determination of centroid information is relatively cheap. Ahn and Shashkov [2] also extended this method to multi-material interfaces on polyhedral meshes.

Marker methods have also been used in conjunction with VOF techniques with a fair amount of success by Popinet and Zaleski [129] and Aulisa et al. [7]. The markers are used to construct piecewise-polynomials which give a parametric representation of the interface. This approach shows advantages over conventional PLIC approaches,
since sub-grid features are resolved with better accuracy, and the method also reduces spurious currents substantially.

Once the interface is reconstructed using the described methods and volume fluxes are computed from updated interface locations, the volume fraction field is evolved in time using Eq.(1.3). A problematic area in VOF (and other surface capturing) methods is the treatment of surface tension. Inaccurate implementations of these effects show up in the form of numerical instabilities and/or noise at the interface, in addition to the development of spurious ‘parasitic’ currents.

For VOF methods which do not explicitly track the interface, a technique which is frequently used to model surface tension effects is the Continuum Surface Force (CSF) approach by Brackbill et al. [21]. The basic principle behind the CSF approach is the use of an indicator function, \( \gamma \), that varies continuously between fluids across the interface. A desirable property of \( \gamma \) would be to have it as thin as possible, i.e., reminiscent of a true interface. However, attempting to calculate the unit interface normal (required to estimate capillary forces) from the gradient of this function is extremely difficult and so, a smoothing convolution kernel, \( H \), is applied to \( \gamma \). This is given by:

\[
\gamma^*(\mathbf{x}) = \int_V \gamma(\mathbf{x}') H(\mathbf{x} - \mathbf{x}'; \epsilon) d\mathbf{x}' 
\]  

(1.6)

where \( H(\mathbf{x}; \epsilon) \) has the property \( H \to \delta_S \) as \( \epsilon \to 0 \). The interface curvature is computed based on the divergence of the surface normal calculated from the smoothed indicator function.

For PLIC or marker-based VOF methods, where the normal is measured explicitly, capillary forces can be evaluated with better accuracy, thereby leading to a reduction in spurious currents.
1.1.3 Marker Methods

Marker methods are characterized by the use of massless marker particles to locate the presence of an interface. The method was first introduced by Harlow and Welch [76] in 1965. Such particles can be either surface-based (markers only on interfaces), or volume-based (markers in the entire domain). Information is interchanged from Eulerian (bulk-fluid) to Lagrangian (marker) grids by conservative interpolation. Marker-based methods vary in the method by which this interpolation is performed, and a smoothing technique is often necessary for numerical stability. A typical set up using this approach would involve the steps:

- Set initial conditions for the system and solve for fluid momentum on the Eulerian mesh.

- Conservatively transfer momentum to the Lagrangian grid (composed of either particles or a supplementary mesh), and advect it.

- Update surface-tension forces on the Lagrangian grid and transfer them back to the Eulerian grid.

- Apply transferred forces to the momentum equations and solve for the next time step. Repeat.

Surface-based markers explicitly track the interface, but require numerical surgery when handling cases involving topological changes and triple-points (i.e., the gas-solid-liquid interface) in multi-phase flow. Volume-based markers can alleviate such issues, but in general, a larger number of volume-markers are necessary to accurately represent an interface, making it an expensive approach in three-dimensions.

Marker particles show obvious benefits in their ability to capture sub-grid features like thin liquid filament structures, which may otherwise have been overlooked by conventional interface capturing methods. But this gain is often short-lived, as
pointed out by Scardovelli et al. [138], since small-scale physics like velocity and pressure fields are not captured by coarse meshes (where marker particles are most effective) and so, grid refinement is required anyway. However, these methods can accurately measure interface curvature (and therefore, surface tension), and conserve volume very well.

Univerdi et al. [155], Tryggvason et al. [150, 151] and de Sousa et al. [41] used marker particles in conjunction with a supplemental triangular surface-mesh in an Arbitrary Lagrangian Eulerian (ALE) setting to track interfaces with greater accuracy. Surface-tension forces were calculated on the supplementary mesh and then conservatively transferred to the Eulerian grid by interpolation. This approach was extended to study the direct numerical simulations of bubbly flows by Esmaeeli et al. [52, 53, 151] and boiling applications by Juric et al. [92]. Glimm et al. have also used marker-based front tracking methods to simulate Rayleigh-Taylor [69] and Richtmyer-Meshkov [68] instabilities.

1.2 Lagrangian Methods for Multi-phase Flow

In contrast to the Eulerian fixed mesh description, Lagrangian methods employ a moving mesh technique where the interface is tracked explicitly. In this approach, the interface actually corresponds to a (tessellated) surface of zero thickness, as opposed to interface-capturing Eulerian techniques, where a interface of finite thickness with smoothly varying properties is usually the norm. A particularly attractive feature of this approach is the precise description of interface curvature, which allows surface tension calculations to be performed with a high degree of precision and ensures mass conservation up to machine accuracy. Jump conditions are often formulated to be equivalent to the physical system, but owing to the sharp nature of the interface, some form of local averaging may sometimes be necessary for numerical stability and accuracy (Quan and Schmidt [133]).
From the mesh perspective, the interface corresponds to faces that are shared by two fluid cells having dissimilar properties on either side. A common problem associated with moving mesh methods is cell tangling and distortion, particularly with cases involving large domain deformations. Distorted cells are usually detrimental to numerical accuracy and convergence, and care must be taken to ensure that they are eliminated during the course of the simulation process (Shewchuk [142]). In general, Lagrangian methods involve a combination of mesh motion and topology modifications to circumvent the distortion problem.

An explicit Euler update of the interface introduces a time-step restriction that is generally a factor smaller than the smallest timescale in the bulk fluid. Slikkerveer et al. [143] quantified this time-step using dimensional analysis:

\[ \tau_{surf} \approx \frac{\mu R}{\sigma} = \frac{\mu}{\sigma} \left( \frac{\lambda^2}{4\pi^2a} \right) \quad (1.7) \]

where \( R \) is the radius of curvature, \( \mu \) is the fluid viscosity, \( \sigma \) is the surface-tension, and \( \lambda \) is wavelength of a sinusoidal disturbance on the interface, with amplitude \( a \). In the discrete sense, \( \lambda \) is equivalent to the mesh size on the interface and consequently, any attempt at mesh-refinement contributes toward numerical stiffness. The natural solution is to treat the surface-tension effects in an implicit manner, and there have been a number of initiatives in this direction (Slikkerveer et al. [143], Hysing [81]).

In a Lagrangian simulation of multi-phase flow, motion of the interface is defined by velocity attributes interpolated from mesh variables (located at cells, faces, edges or points) at the interface, subject to physical conditions like surface-tension, normal and shear-stresses. Motion of points in the interior of the mesh (away from the interface) is specified in a manner that maximizes the quality of cells in the mesh, in an effort to minimize distortion (Perot and Nallapati [111]). However, there are instances when mesh motion alone cannot alleviate extreme distortions, and re-meshing of the
domain is necessary. The subject of adaptive mesh reconnection is discussed in detail in Chapter 4, followed by a discussion of mesh smoothing methods in Chapter 5.

Another challenge with the Lagrangian approach involves changes to interface topology, such as coalescence and break up. Since moving meshes use a discretized surface to represent the interface, situations involving topology changes require explicit (and sometimes, manual) intervention. These situations are also demanding due to the disparity of length and time scales involved. For instance, a thin liquid bridge connecting two large droplets on the verge of break up involves time scales that are much smaller than the bulk flow. This is clearly a numerical stiffness problem, but it is unclear how such a situation is to be handled from a continuum point of view, and indeed, a continuum approach simply may not be adequate. Perhaps an approach which incorporates molecular effects would provide a more accurate description.

![Figure 1.2. Mesh slicing for interface topology changes (See ref. [133])](image)

Frequently, from a macroscopic perspective, the actual mechanics of break up or coalescence is considered insignificant, and it suffices to establish that a change in topology has occurred, without detailing the intricate physics involved. In these situations, a combination of interface mesh-slicing and phase-conversion (from liquid to gas, for instance) is performed. This is the approach taken by several authors, including Cristini et al. [33, 34], Quan et al. [132], Zheng et al. [166] and Anderson et al. [6].
These kind of topology modification techniques come with additional problems as well. For instance, cases involving mesh slicing or phase-change rarely provide a smooth interface, thereby complicating calculations for interface curvature. Thus, some form of surface smoothing if often necessary to attain a well-behaved mesh, which in turn requires some sacrifices to be made for mass conservation (Dai [37]). Recently, Bargteil et al. [10] used an explicit interface combined with a semi-Lagrangian contouring method to track surface topology characteristics automatically. This method combines the versatility of the level-set method with the accuracy of interface tracking. The algorithm starts with a known explicit triangular mesh interface, and a signed scalar distance field ($\varphi$) which evolves according to the standard level-set transport equation. Rather than updating the explicit interface, the next step is to advect $\varphi$ using the explicit Euler method, followed by an extraction of its zero level-set to represent the new (implicit) interface representation. A new explicit mesh interface is then generated on the implicit representation. The scalar field is then reinitialized (or redistanced) for the next iteration.

To reinitialize $\varphi$, the authors use an distance-tree structure in order to quickly detect cells close to the interface. The structure also contains signed distance values which are updated continuously, thereby providing a rough estimate for redistancing at the following time-step. A similar approach was taken by Walker [160] using a variational level set approach. Walker chose not to regenerate the mesh interface, but to locally project vertices of the existing mesh toward the zero level-set contour using an optimization step.

In Chapter 2, the equations governing the dynamics of multi-phase flows for moving and deforming control volumes (on which Lagrangian interface-tracking methods are based) is described, including details of boundary conditions at the interface between two fluids.
CHAPTER 2

GOVERNING EQUATIONS FOR MULTI-PHASE FLOW

This chapter will describe the integral conservation laws governing a moving, deforming control volume, which is necessary for simulations involving a moving-mesh interface-tracking approach. Since multi-phase flows are also considered in this paradigm, boundary conditions at the interface between two fluids (both dynamic and kinematic) are also described.

2.1 Governing Equations for an Arbitrary Control Volume

The conservation law for an intensive physical property \( \phi \), given a material volume \( V_M \), bounded by a closed surface \( S_M \), is given by:

\[
\frac{d}{dt} \int_{V_M} \rho \phi \, dV = - \int_{S_M} n \cdot q_\phi \, dS + \int_{V_M} s_\phi \, dV
\]  

(2.1)

where \( \rho \) is the density, \( n \) is the outward pointing unit normal on \( S_M \), \( q_\phi \) and \( s_\phi \) are surface and volume sources of \( \phi \).

The Reynolds Transport theorem governing a general property \( \phi \), for a deforming control volume (which occupies volume \( V \), bounded by a surface \( S \) moving at a velocity \( v_s \), at time \( t \)) is given by (Bird et al. [17]):

\[
\frac{d}{dt} \int_V \rho \phi \, dV = \int_V \frac{\partial \rho \phi}{\partial t} \, dV + \int_S n \cdot \rho v_s \phi \, dS
\]  

(2.2)
Assuming that at time $t$, the material volume $V_M$ also occupies volume $V$, the Reynolds Transport theorem for the material volume (with material velocity $\mathbf{v}$), is given by:

$$
\frac{d}{dt} \int_{V_M} \rho \phi \, dV = \int_{V_M} \frac{\partial \rho \phi}{\partial t} \, dV + \int_{S_M} \mathbf{n} \cdot \rho \mathbf{v} \phi \, dS \quad (2.3)
$$

At a new time $t+dt$, the control volume ($V$) and the material volume ($V_M$) do not coincide, but by combining Eq.(2.2) and Eq.(2.3), the following relation is obtained:

$$
\frac{d}{dt} \int_{V_M} \rho \phi \, dV = \frac{d}{dt} \int_{V} \rho \phi \, dV + \int_{S} \mathbf{n} \cdot \rho (\mathbf{v} - \mathbf{v}_s) \phi \, dS \quad (2.4)
$$

Applying Eq.(2.4) to Eq.(2.1), the integral conservation law for an arbitrary volume $V$, bounded by a surface $S$, is obtained:

$$
\frac{d}{dt} \int_{V} \rho \phi \, dV + \int_{S} \mathbf{n} \cdot \rho (\mathbf{v} - \mathbf{v}_s) \phi \, dS = -\int_{S} \mathbf{n} \cdot \mathbf{q}_\phi \, dS + \int_{V} \mathbf{s}_\phi \, dV \quad (2.5)
$$

The surface source, $\mathbf{q}_\phi$, usually represents a diffusion flux which is expressed by the relation:

$$
\mathbf{q}_\phi = -\Gamma_\phi \nabla \phi \quad (2.6)
$$

where $\Gamma_\phi$ is the diffusion coefficient.

The basic laws of continuum mechanics can be obtained from Eq.(2.5), by choosing various properties for $\phi$:

- **Conservation of Mass:**

  $$
  \frac{d}{dt} \int_{V} \rho \, dV + \int_{S} \mathbf{n} \cdot \rho (\mathbf{v} - \mathbf{v}_s) \, dS = 0 \quad (2.7)
  $$

- **Conservation of Linear Momentum:**

  $$
  \frac{d}{dt} \int_{V} \rho \mathbf{v} \, dV + \int_{S} \mathbf{n} \cdot \rho (\mathbf{v} - \mathbf{v}_s) \mathbf{v} \, dS = \int_{V} \rho \mathbf{g} \, dV + \int_{S} \mathbf{n} \cdot \mathbf{\sigma} \, dS \quad (2.8)
  $$
• Conservation of Angular Momentum:

\[
\frac{d}{dt} \int_V \rho (\mathbf{r} \times \mathbf{v}) \, dV + \int_S \mathbf{n} \cdot \rho (\mathbf{v} - \mathbf{v}_s) (\mathbf{r} \times \mathbf{v}) \, dS
\]

\[
= \int_V \rho (\mathbf{r} \times \mathbf{g}) \, dV + \int_V (\mathbf{r} \times \nabla \cdot \mathbf{\sigma}) \, dV
\]

where \( \mathbf{g} \) is the acceleration due to gravity, \( \mathbf{r} \) is a position vector, and \( \mathbf{\sigma} \) is the stress tensor.

In the case of constant density, another consequence of choosing \( \phi = 1 \) with the Reynolds Transport equation, Eq.(2.2), is the space-conservation law for a deforming control volume. This law relates the rate of change of the control volume to the velocity of its boundary:

\[
\frac{d}{dt} \int_V dV - \int_S \mathbf{n} \cdot \mathbf{v}_s \, dS = 0
\]

Also, for constant density Newtonian fluids, the stress tensor \( \mathbf{\sigma} \) can be expressed in the normal and deviatoric components:

\[
\mathbf{\sigma} = -p I + \mathbf{\tau}
\]

\[
\mathbf{\tau} = \mu [\nabla \mathbf{v} + (\nabla \mathbf{v})^T]
\]

where \( p \) is the pressure, \( \mu \) is the dynamic viscosity, and \( I \) is the second-rank unit tensor. For situations involving constant density and constant viscosity, equations Eq.(2.8) and Eq.(2.9) can be combined to obtain the following general form:

\[
\int_S \mathbf{n} \cdot \mathbf{v} \, dS = 0
\]

\[
\frac{d}{dt} \int_V \rho \mathbf{v} \, dV + \int_S \mathbf{n} \cdot \rho (\mathbf{v} - \mathbf{v}_s) \mathbf{v} \, dS = \int_V \rho \mathbf{g} \, dV - \int_V \nabla p \, dV + \int_S \mathbf{n} \cdot (\mu \nabla \mathbf{v}) \, dS
\]
2.2 Conditions at a Fluid Interface

While equations Eq.(2.12) and Eq.(2.13) are valid under the continuum hypothesis, conditions at an interface between two immiscible fluids require special treatment. Some conditions like velocity and viscous shear-stress (at constant surface-tension) remain continuous, while others like density, viscosity, viscous normal-stress and pressure do not. These jump-conditions are most apparent when the equation Eq.(2.13) is taken to the singular limit at the interface, yielding a surface $S$ bounded by a curve $\partial S$. In this situation, the volume integrals on the left-hand side of Eq.(2.13) become negligible, and the following stress balance is obtained (Deen [43]), known as the \textit{dynamic condition}:

\begin{equation}
\int_{S} \mathbf{n} \cdot \sigma_{2} \, dS - \int_{S} \mathbf{n} \cdot \sigma_{1} \, dS + \int_{\partial S} \mathbf{m} \sigma \, dL = 0 \tag{2.14}
\end{equation}

where $\sigma_{1}$ and $\sigma_{2}$ are the stress tensors for either fluid, $\sigma$ is the surface tension, $\mathbf{m}$ is the unit bi-normal vector to $\partial S$ (with an arc-length $L$). The line integral in Eq.(2.14) can be converted to a surface integral using the surface Gauss theorem:

\begin{equation}
\int_{S} \mathbf{n} \cdot \sigma_{2} \, dS - \int_{S} \mathbf{n} \cdot \sigma_{1} \, dS + \int_{S} \nabla_{s} \sigma \, dS + \int_{S} \kappa \mathbf{n} \, dS = 0 \tag{2.15}
\end{equation}

where $\mathbf{n}$ is the unit normal vector at the interface, $\kappa = -\nabla_{s} \cdot \mathbf{n}$ is twice the mean surface curvature, and $\nabla_{s}$ is the surface gradient operator. Expressing the stress tensor as a sum of the normal and deviatoric components ($\sigma = -p\mathbf{I} + \tau$), and letting equation Eq.(2.15) converge to a point on the interface, the following differential form can be obtained:

\begin{equation}
(p_{2} - p_{1})\mathbf{n} - \mathbf{n} \cdot (\tau_{2} - \tau_{1}) = \nabla_{s} \sigma + \kappa \mathbf{n} \tag{2.16}
\end{equation}

The normal component of the surface stress balance is obtained by taking the dot-product of the interface unit normal with equation Eq.(2.16):
\[(p_2 - p_1) - \mathbf{n}\mathbf{n} : (\mathbf{\tau}_2 - \mathbf{\tau}_1) = \kappa\sigma \quad (2.17)\]

Equation Eq.(2.17) represents a pressure-jump condition across an interface, owing to the capillary effects of surface tension and curvature.

The tangential component is obtained by multiplying Eq.(2.17) with \( \mathbf{n} \) and subtracting it from Eq.(2.16):

\[
\mathbf{n} \cdot (\mathbf{\tau}_2 - \mathbf{\tau}_1) - \mathbf{n}[\mathbf{n}\mathbf{n} : (\mathbf{\tau}_2 - \mathbf{\tau}_1)] = -\nabla_s \sigma \quad (2.18)
\]

For a rigid, no-slip surface and incompressible flow, the term \( \mathbf{n} \cdot \mathbf{\tau} \) is identically zero (Batchelor [11]). It is evident from Eq.(2.18), that the normal component of the deviatoric stress at an interface is directly related to the surface gradient of surface-tension. In the absence of such gradients, using the relation between the rate-of-strain tensor and the deviatoric stress tensor Eq.(2.11b), the following continuity holds:

\[
\mu_2 (\mathbf{n} \cdot \nabla \mathbf{v}_2) = \mu_1 (\mathbf{n} \cdot \nabla \mathbf{v}_1) \quad (2.19)
\]

The relationship between fluid velocities (\( \mathbf{v}_1 \) and \( \mathbf{v}_2 \)) at either side of the interface is dictated by the *kinematic condition* (Batchelor [11]), which specifies that the normal velocity is continuous:

\[
\mathbf{n} \cdot \mathbf{v}_2 - \mathbf{n} \cdot \mathbf{v}_1 = 0 \quad (2.20)
\]

Assuming momentum transfer across the interface, the tangential component of fluid velocity should also be continuous:

\[
(I - \mathbf{n}\mathbf{n}) \cdot \mathbf{v}_2 - (I - \mathbf{n}\mathbf{n}) \cdot \mathbf{v}_1 = 0 \quad (2.21)
\]

Combining equations Eq.(2.20) and Eq.(2.21) provides the condition specifying continuity of velocity:

\[
\mathbf{v}_2 = \mathbf{v}_1 \quad (2.22)
\]
2.3 Variation of Physical Properties

In situations where variations in operating conditions like temperature are present, physical properties like fluid viscosity and surface tension must also adapt to accommodate these fluctuations. Gradients of surface-tension (either due surfactant concentration or temperature related Marangoni effects) contribute towards the shear-stress at an interface.

Variations in viscosity due to fluid temperature are modeled using the Vogel-Fulcher-Tammann (VFT) relation [65]:

\[ \mu = \mu_o \exp \left[ a \left( \frac{T_o}{T - b} - c \right) \right] \]  
\[ a = \left( \frac{T_A}{T_o} \right) \]  
\[ b = T_V \]  
\[ c = \frac{T_o}{T_o - T_V} \]

where \( \mu_o, T_o, T_V \) and \( T_A \) are the reference viscosity, reference temperature, Vogel and activation temperatures, respectively. For water, these values are given as: \( \mu_o = 10^{-3} \), \( T_o = 298.15 K \), \( T_V = 156.8 K \) and \( T_A = 937.38 K \).

In accordance with the Eötvos rule, surface tension varies linearly with temperature, and is described by the following relation:

\[ \sigma = \sigma_o - \beta (T - T_o) \]

For water, these values are: \( \sigma_o = 72.7 \) dyne/cm, \( \beta = 0.18 \) dyne/cm/K at \( T_o = 298 K \).
CHAPTER 3

FINITE VOLUME DISCRETIZATION

The primary intent of any discretization method is to transform the continuous form of a governing partial differential equation into a discrete set of algebraic equations. This procedure usually requires the sub-division of the solution space (or domain) into a finite number of points at which the unknowns are defined. The division is performed spatially, by defining a finite set of control volumes (or cells), and in cases where the solution is transient in nature, temporally as well. Temporal sub-division requires time to be split into discrete time-steps, with each step describing a discrete spatial solution at that instance.

Before proceeding to the process of discretizing equations, some important definitions about spatial discretization are necessary. A collection of non-overlapping cells constitute a mesh, such that cells are connected to each other by faces, which in turn, are described by points, and edges that connect these points. It follows naturally that any interior face in the domain is connected to exactly two cells, while faces on the boundary of the domain are connected to only one. In this work, it is assumed that all faces are flat, with a centroid for each face $x_f$ defined such that:

$$\int_{S_f} (x - x_f) \, dS = 0$$  \hspace{1cm} (3.1)

The face area vector $S_f$, is defined such that it points out of a cell of interest $P$ into a neighbouring cell $N$, with a magnitude equal to the area of face. Such a face is said to be owned by the cell $P$, with its neighbour $N$. The points describing each face
Figure 3.1. Definition of various control-volume entities

are ordered in a counter-clockwise manner around the face normal vector, according to the right-hand rule, as shown in Fig. 3.1. The ordering of internal faces is also rigorously maintained such that every face has an ‘owner’ cell label which is less than the ‘neighbour’ cell label. For boundary faces, the face area vector always points out of the domain, and a neighbour cell label is not needed. The unit normal vector \( \mathbf{n} \), is defined as the normalized face area vector, \( \mathbf{n} = \mathbf{S}_f / |\mathbf{S}_f| \).

Centroids for cells \( \mathbf{x}_P \), with volume \( V_P \), are defined such that:

\[
\int_{V_P} (\mathbf{x} - \mathbf{x}_P) \, dV = 0 \quad (3.2)
\]

The vector \( \mathbf{d} \), is defined as the vector connecting the cell position vectors \( \mathbf{x}_P \) and \( \mathbf{x}_N \), on either side of an internal face.

\[
\mathbf{d} = (\mathbf{x}_N - \mathbf{x}_P) \quad (3.3)
\]

while for boundary faces, this vector is defined as:
\[ d = (x_f - x_P) \]  

Any given mesh is said to be orthogonal if the vector \( d \) is parallel to the face normal vector \( n \), for every face in the mesh. The orthogonal case, however, is more of an exception rather than the rule, and usually applies only to the simplest of domains. Spatial discretization of an arbitrary domain can often become very complicated, thereby requiring a more general description of mesh cells. In this framework, any given cell in the mesh can be bounded by an arbitrary number of faces. Such meshes are described as unstructured, and the possess several advantages over regularly structured meshes. These meshes can conform to complicated domain boundaries very well, especially in cases where the boundaries change with time, and considerably ease the process of mesh generation and manipulation.

The space-conservation law, described by Eq.(2.10), can be discretized to provide the mesh velocity \( v_s \), for any triangular face in mesh (Perot and Nallapati [125]). Considering a triangular mesh face with points \( x_o^0, x_1^0, x_2^0 \) at time \( t \), which moves to a new position \( x_n^0, x_1^n, x_2^n \) at time \( t + \Delta t \), then the mesh velocity is given by the expression:

\[ v_{s,f} = \frac{1}{\Delta t} (x_n^f - x_o^f) \cdot \left[ \frac{1}{2} (S_n^f + S_o^f) - \frac{1}{12} (v_0 \times v_1 + v_1 \times v_2 + v_2 \times v_0) \right] \tag{3.5} \]

where \( v_i = x_n^i - x_o^i \) for a given point \( i \). For general polyhedral meshes, the swept volume for a polygonal face is defined by first decomposing it into triangles and applying Eq.(3.5) (Tuković [153]).

Another important decision that has to be made during the finite volume discretization process is the choice of variable arrangement on the mesh. Dependent variables (for which the discretized governing equations are solved) have been stored at points, edges, faces and cells in previous work by many researchers in the past, but the two most popular arrangements seem to be collocated and staggered mesh methods.
The staggered mesh approach, as the name suggests, places the dependent variables at multiple locations on the mesh (like faces and cells, for example), and generally possess different control volumes. A typical staggered mesh arrangement involves pressure variables stored at cell centroids and face fluxes are stored at face centroids. The primary advantage of the staggered mesh approach is that it fits very well in the paradigm of pressure-velocity coupling, presents several advantages pertaining to conservation properties, and avoids the problems of spurious pressure modes (Zhang et al. [165]). Discontinuous boundary conditions and face-normal gradient evaluation on unstructured staggered meshes are sometimes complicated, since the cell-centroid variable (such as fluid velocity), must often be reconstructed from face-fluxes, but these issues have been successfully handled in the past (Perot et al. [127, 125]).

The collocated mesh method places all dependent variables in a manner such that they share the same control volume. An common choice is the centroid of the cell, as adopted in this work, but other locations are clearly possible. Collocated arrangements were largely unpopular due to an unwanted ‘checker-boarding’ effect which introduces spurious oscillations in the pressure field (Patankar [121], Ferziger and Perić [58]). A convenient work-around, suggested by Rhie and Chow [134], involves the interpolation of the cell-centred pressure variable to faces while estimating the cell-centred pressure gradient in the momentum equation. This approach has since been adopted in almost all collocated CFD codes, and has turned out to be quite popular. This study adopts the collocated variable arrangement, and the details of its application in the discretization process are discussed in forthcoming sections.

3.1 Discretization of a General Transport Equation

The standard form of a partial differential equation governing a tensorial variable $\phi$ is given as:
\[
\frac{\partial \rho \phi}{\partial t} + \nabla \cdot (\rho \phi \mathbf{v}) = \nabla \cdot (\rho \Gamma \phi \nabla \phi) + S_\phi \tag{3.6}
\]

where \(\rho\) is the density, \(\mathbf{v}\) is the material or fluid velocity, and \(\Gamma\) is the diffusivity. The temporal term represents the rate of change in the variable \(\phi\), the convection term represents the convective flux of \(\phi\) due to the fluid velocity \(\mathbf{v}\), the diffusion term represents the rate of transport due to diffusion, and finally, the source term represents the rate of production/destruction of \(\phi\). All terms in equation Eq.(3.6) are defined per unit volume.

The finite volume discretization approach is formulated by integrating equation Eq.(3.6) over the control volume \(V\) and time interval \(\Delta t\):

\[
\int_{t}^{t+\Delta t} \left[ \int_{V} \frac{\partial \rho \phi}{\partial t} \, dV + \int_{V} \nabla \cdot (\rho \phi \mathbf{v}) \, dV \right] \, dt
= \int_{t}^{t+\Delta t} \left[ \int_{V} \nabla \cdot (\rho \Gamma \phi \nabla \phi) \, dV + \int_{V} S_\phi \, dV \right] \, dt \tag{3.7}
\]

All the spatial terms can be converted to surface integrals by the Gauss’ theorem (for scalar \(\phi\) or tensor \(\phi\)), which is given by the following identities:

\[
\int_{V} \nabla \cdot \phi \, dV = \int_{S} dS \cdot \phi \tag{3.8}
\]

\[
\int_{V} \nabla \phi \, dV = \int_{S} dS \phi \tag{3.9}
\]

\[
\int_{V} \nabla \phi \, dV = \int_{S} dS \phi \tag{3.10}
\]

Using the assumption of flat faces for cells in the mesh, Eq.(3.8) can be transformed into a sum of integrals over cell faces:
\[
\int \nabla \cdot \phi \, dV = \int_S dS \cdot \phi = \sum_f \left( \int_f dS \cdot \phi \right)
\] (3.11)

In order to ensure that the discretization method is second order accurate in space, a linear variation of the variable \( \phi(x) \) is assumed, using a truncated Taylor series expansion around a point \( P \), such that \( \phi_P = \phi(x_P) \):

\[
\phi(x) = \phi_P + (x - x_P) \cdot (\nabla \phi)_P
\] (3.12)

Integrating Eq.(3.12) over a control volume \( V_P \) around point \( P \), it follows that:

\[
\int_{V_P} \phi(x) \, dV = \phi_P \int_{V_P} dV + \left[ \int_{V_P} (x - x_P) \, dV \right] \cdot (\nabla \phi)_P
\]
\[
= \phi_P V_P
\] (3.13)

The second integral in equation Eq.(3.13) is identically zero due to the assumption in equation Eq.(3.2). This useful property can also be applied to the surface integral in Eq.(3.11):

\[
\int_S dS \cdot \phi = \left( \int_S dS \right) \cdot \phi_f + \left[ \int_S (x - x_f) \, dS \right] : (\nabla \phi)_f
\]
\[
= \phi_f \cdot S
\] (3.14)

Here again, the second integral in equation Eq.(3.14) is identically zero due to the assumption in equation Eq.(3.1). The term \( \phi_f \) denotes the value of the variable at the face centroid, which must be interpolated from cells on either side of the face, and \( S \) is the face normal vector pointing out of the control volume. In the current
paradigm, the face normal \( S_f \) points out of the cell in question only if it is 'owned' by that cell. Taking this into account while summing over all cell faces, and combining equations Eq.(3.11), Eq.(3.13) and Eq.(3.14), the following split is obtained:

\[
(\nabla \cdot \phi)V_p = \sum_f S \cdot \phi_f
\]

\[
= \sum_{owner} S_f \cdot \phi_f - \sum_{neighbour} S_f \cdot \phi_f
\]

(3.15)

This expression represents the second-order accurate discretization of the Gauss’ theorem.

### 3.1.1 Discretization of the Convection Term

The convection term is discretized by the application of the discrete Gauss divergence theorem according to the following relations:

\[
\int_{V_p} \nabla \cdot (\rho \phi \mathbf{v}) = \sum_f S \cdot (\rho \phi \mathbf{v})_f
\]

\[
= \sum_f S \cdot (\rho \mathbf{v})_f \phi_f
\]

\[
= \sum_f F \phi_f
\]

(3.16)

where \( F \) represents the mass flux through the face \( f \):

\[
F = S \cdot (\rho \mathbf{v})_f
\]

(3.17)

Notice that the density \((\rho)\), velocity \((\mathbf{v})\) and dependent variable \((\phi)\) fields are interpolated to faces for this operation. The choice of interpolation scheme for obtaining the interpolant of \( \phi \) at faces is critical to the finite volume method. An important requirement for any conservative convection scheme is boundedness, which is defined by the ability of the scheme to preserve the value of \( \phi \) given by its initial
distribution. In plain advection, the value of $\phi_f$ for a face must lie in between the values for its neighbouring cells to maintain the condition. For schemes that only use information from cells on either side of the face, a few choices of differencing are available:

The **upwind differencing** scheme determines the face interpolant value $\phi_f$ based on the direction of the flow through the face. More formally, it is defined according to the relation:

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

The upwind differencing scheme is guaranteed to provide a bounded solution, but it is known to be only first-order accurate (Ferziger and Perić [58]). The leading term in the truncation error resembles a diffusive flux and hence, this scheme tends to be excessively diffusive, thereby causing a reduced order of accuracy in the solution.

The **central differencing** scheme, on the other hand, uses a weighted average of cell values:

$$
\phi_f = f_x\phi_P + (1 - f_x)\phi_N
$$

$$
f_x = \frac{|x_f - x_N|}{|d|}
$$

Central differencing is known to be second-order accurate even on non-uniform meshes (Ferziger and Perić [58]), but introduces produces unphysical oscillations into the solution and does not preserve boundedness, especially on convection dominated simulations.

An attempt to combine the stability of upwind differencing and the accuracy of central differencing is the **blended differencing** approach, first suggested by Perić [124]. The face interpolant is calculated using a linear combination of upwinding and central differencing, according to the following expression:
\[ \phi_f = (1 - \gamma)(\phi_f)_{UD} + \gamma(\phi_f)_{CD} \]  

(3.20)

where the blending factor \( \gamma \), is a fraction which determines the amount of numerical diffusion that will be introduced due to upwind differencing. It naturally follows that \( \gamma = 0 \) reduces to upwind differencing.

Several approaches have been taken in the past to modify blended schemes in ways that can guarantee both accuracy and solution boundedness, including higher-order upwind schemes (QUICK by Leonard [104], for example) and flux-limiting schemes. Flux limiting is an approach that results in a scheme that is higher than first-order accurate, but without as much spurious oscillation as a second-order accurate central differencing approach. The concept of flux-limiting is used extensively in Total Variation Diminishing (TVD) schemes, first introduced by Harten [77]. The general form of a TVD scheme is given by the following expression (Sweby [148]):

\[ \phi_f = (\phi_f)_{UD} + \Psi \left[ (\phi_f)_{HO} - (\phi_f)_{UD} \right] \]  

(3.21)

where \((\phi_f)_{HO}\) is a selected higher-order scheme, and \(\Psi\) is a flux-limiter which is a function of the downwind, upwind, and ‘virtual’ upwind cells around the face \(f\). Details of various TVD schemes can be found in work by Darwish [39], Harten [77], Sweby [148], Roe [136] and van Leer [159]. This work uses a blended differencing scheme introduced by Jasak et al. [87], based on the Normalized Variable Approach introduced by Leonard [104] and Gaskell and Lau [67]. The scheme uses blending to switch between upwind and central differencing without an explicit reference to a far upwind node, thereby resulting in very low numerical diffusion. However, NVD-like schemes do tend to be very sensitive to mesh non-orthogonality, which often results in convergence problems for simplical meshes. For situations where severe mesh non-orthogonality affects solution convergence, an upwind scheme with explicit corrections
is used. This approach is more diffusive due to upwinding, but slightly better in terms of accuracy and stability.

3.1.2 Discretization of the Diffusion Term

The diffusion term is also discretized in a manner similar to the convection term:

$$\int_{V_P} \nabla \cdot (\rho \Gamma_\phi \nabla \phi) = \sum_f S \cdot (\rho \Gamma_\phi \nabla \phi)_f$$

$$= \sum_f (\rho \Gamma_\phi)_f S \cdot (\nabla \phi)_f$$

(3.22)

In the case of orthogonal meshes, the following expression can be used in Eq.(3.22):

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

(3.23)

One other possibility for $(\nabla \phi)_f$ is to compute it using Gauss’ theorem, where the cell-centred gradient on each side of the face is calculated thus:

$$(\nabla \phi)_P = \frac{1}{V_P} \sum_f S\phi_f$$

(3.24)

and the result is interpolated to the face:

$$(\nabla \phi)_f = f_x (\nabla \phi)_P + (1 - f_x)(\nabla \phi)_N$$

(3.25)

This method, however, uses a large computational molecule, involves a larger truncation error than the first method, and is highly error-prone on non-orthogonal meshes, providing only first-order accuracy. An alternative is to evaluate the face-normal gradient using a Least-Squares fit. The method first assumes a linear variation
in \( \phi \) (which is consistent with the second-order accuracy requirements), and evaluates the gradient error at each neighbouring cell (N) using the expression:

\[
\epsilon_N = \phi_N - (\phi_P + \mathbf{d} \cdot (\nabla \phi)_P)
\] (3.26)

The objective is to minimize the least-square error at (P), given by the expression:

\[
\epsilon^2_P = \sum_N w^2_N \epsilon^2_N
\] (3.27)

where the weighting function is given by \( w_N = 1/|\mathbf{d}| \). The following expression is used to evaluate the cell-centred gradient at P, which can then be interpolated to faces:

\[
(\nabla \phi)_P = \sum_N w^2_N \mathbf{G}^{-1} \cdot \mathbf{d} (\phi_N - \phi_P)
\] (3.28)

\[
\mathbf{G} = \sum_N w^2_N \mathbf{d} \cdot \mathbf{d}
\] (3.29)

Note that \( \mathbf{G} \) is a symmetric \( n \times n \) matrix (\( n \) being the number of spatial dimensions) which can easily be inverted. This leads to a second-order accurate gradient which is independent of mesh geometry.

On non-orthogonal meshes, in order to preserve second-order accuracy in the diffusion term, Eq.(3.23) is split into an orthogonal contribution and a non-orthogonal correction term:

\[
\mathbf{S} \cdot (\nabla \phi)_f = \Delta \cdot (\nabla \phi)_f + k \cdot (\nabla \phi)_f
\] (3.30)

where \( \mathbf{S} = \Delta + k \).

As described by Jasak [85], several combinations for \( \Delta \) are possible (with \( k \) being obtained by \( \mathbf{S} - \Delta \)).
The **minimum correction** approach attempts to minimize the non-orthogonal contribution by making $\Delta$ and $k$ orthogonal:

$$\Delta = \frac{d \cdot S}{d \cdot d}$$  \hspace{1cm} (3.31)

The **orthogonal correction** approach attempts to maintain the condition of orthogonality, irrespective of whether non-orthogonality exists:

$$\Delta = \frac{d}{|d|} |S|$$  \hspace{1cm} (3.32)

Unlike the minimum correction approach, the **over-relaxed** approach increases the contribution from $\phi_P$ and $\phi_N$ as non-orthogonality increases:

$$\Delta = \frac{d}{d \cdot S} |S|^2$$  \hspace{1cm} (3.33)

### 3.1.3 Discretization of the Source Term

The source term is first linearized to obtain the following expression:

$$S_\phi = S_u + S_p \phi$$  \hspace{1cm} (3.34)

where $S_u$ and $S_p$ can depend on $\phi$. The volume integral is then calculated using Eq.(3.13):

$$\int_{V_P} S_\phi \, dV = S_u V_P + S_p V_P \phi_P$$  \hspace{1cm} (3.35)

### 3.1.4 Boundary Conditions for Discretized Spatial Terms

Any system governed by partial differential equations must be provided with boundary and/or initial conditions for closure. Boundary conditions belong to two
broad categories, namely, Dirichlet and Neumann. Both types of boundary conditions can be applied to either the convection or diffusion terms in the following manner:

**Dirichlet**, or ‘fixed-value’ boundary conditions, prescribe the value of the dependent variable $\phi_B$, at the boundary.

- **Convection term.** The discretized form of the convection term is given as:

$$\int_{V_P} \nabla \cdot (\rho \phi \mathbf{v}) = \sum_f F \phi_f$$

Hence, the sum over cell faces include the term $F_B \phi_B$ at the boundary, where $F_B$ is the boundary face-flux.

- **Diffusion term.** The discretized form of the diffusion term is given as:

$$\int_{V_P} \nabla \cdot (\rho \Gamma_\phi \nabla \phi) = \sum_f (\rho \Gamma_\phi) f S \cdot (\nabla \phi)_f$$

The face gradient at the boundary $\nabla \phi_B$ is calculated from the prescribed value $\phi_B$ using the relation:

$$\nabla \phi_B = \left| S \right| \frac{\phi_B - \phi_P}{|d_n|}$$

where $d_n$ is the orthogonal part of $d$ (parallel to the boundary unit-normal vector $n$), given by the expression:

$$d_n = (d \cdot n)n$$

**Neumann**, or ‘fixed-gradient’ boundary conditions, prescribe the gradient of the dependent variable $\nabla \phi_B$, at the boundary.

- **Convection term.** The face value of $\phi_B$ is calculated from the cell-centred value and the boundary gradient using the following expression:

$$\phi_B = \phi_P + d_n \cdot (\nabla \phi_B)$$
- **Diffusion term.** Since the gradient value $\nabla \phi_B$ is specified, it can be substituted directly at the boundary:

$$S \cdot (\nabla \phi_B)_f = |S|(\nabla \phi_B \cdot n) \quad (3.41)$$

### 3.1.5 Discretization of the Temporal Term

The temporal term is discretized using the Euler scheme, where:

$$\int_{V_P} \rho \frac{\partial \phi}{\partial t} \, dV \approx \frac{\rho^n \phi^n V^n - \rho^o \phi^o V^o}{\Delta t} \quad (3.42)$$

where the subscript ‘$n$’ represents values at time $t+\Delta t$, and the subscript ‘$o$’ represents values at the previous time step, i.e., at time $t$. The Euler scheme can be designated as either *explicit* or *implicit*, based on the instance in time at which the spatial terms are evaluated. Consider the discretized equivalent of the general transport equation:

$$\frac{\rho^n \phi^n V^n - \rho^o \phi^o V^o}{\Delta t} + \sum_f F \phi_f = \sum_f (\rho \Gamma_f) S \cdot (\nabla \phi)_f + S_u V_P + S_p V_P \phi_P \quad (3.43)$$

In this relation, the explicit Euler scheme is obtained by the following choices for the various terms:

$$V_P = V_P^o \quad \phi_P = \phi_P^o \quad \phi_N = \phi_N^o \quad \phi_f = \phi_f^o \quad S \cdot (\nabla \phi)_f = \Delta \cdot (\nabla \phi)^o_f + k \cdot (\nabla \phi)^o_f \quad (3.44)$$
The explicit Euler scheme is first-order accurate, and is restricted by the Courant number $C_0$ (which must be less than unity to be stable):

$$C_0 = \frac{(v \cdot n) \Delta t}{|d|} \quad (3.45)$$

The implicit Euler scheme, which has been used extensively in this work, is obtained by making the following choices:

$$V_P = V_P^n$$
$$\phi_P = \phi_P^n$$
$$\phi_N = \phi_N^n$$
$$\phi_f = \phi_f^n$$

$$S \cdot (\nabla \phi)_f = \Delta \cdot (\nabla \phi)_f^n + k \cdot (\nabla \phi)_f^o \quad (3.46)$$

This scheme is unconditionally stable (and hence, does not depend on the Courant number), but is still only first-order accurate. Note that the non-orthogonal diffusion term given in Eq.(3.44), must be treated explicitly to guarantee boundedness (Jasak [85]). This yields an algebraic relation for every control volume in the mesh, taking the general form:

$$a_P \phi_P + \sum_N a_N \phi_N = R_P \quad (3.47)$$

where $a_P$ and $a_N$ are coefficients associated face interpolants for the control volume $P$, and its neighbours $N$, respectively. The $R_P$ term denotes any explicit contributions to the system, such as the non-orthogonal correction, boundary conditions and explicit source-terms. Due to the implicit nature of this scheme, all dependent variables must be solved simultaneously, thereby yielding a system of linear algebraic equations of the form:

$$[A] \phi = [R] \quad (3.48)$$
where \([A]\) is a square \(N \times N\) matrix (where \(N\) is the number of cells in the mesh). The manner in which this system is solved for a discrete solution is the subject of the next section.

### 3.2 Solution to Linear Algebraic Equation Systems

The previous section outlined the manner in which the finite volume discretization method yields a system of linear algebraic equations of the form:

\[
[A] \phi = [R] \tag{3.49}
\]

It is interesting to note that the matrix \([A]\) is generally quite sparse, with coefficients \(a_P\) on the main diagonal, and \(a_N\) on the off-diagonals. For unstructured meshes, due to the irregular connectivity between cells, the matrix inherently lacks a coherent structure. The off-diagonal coefficients \(a_N\) at any given row (and therefore, a particular cell) corresponds to the face coefficients for that cell.

In general, linear systems taking the form of Eq.\((3.49)\) can be solved either by direct inversion of \([A]\), or by an iterative approach which tries to minimize the condition \(r = [R] - [A] \phi\), until some specified tolerance is met. While the direct approach is suitable for small problems, the cost of matrix storage and inversion becomes prohibitively expensive with larger meshes, thereby making the iterative method a much better option. The convergence rate of iterative methods largely depend on the magnitude of matrix coefficients in \([A]\). A diagonally dominant matrix, which is necessary for accelerated convergence, is given by the following condition:

\[
|a_P| \geq \sum_N |a_N| \tag{3.50}
\]

In this work, the Conjugate Gradient algorithm (by Hestenes and Stiefel [79], preconditioned using Incomplete Cholesky decomposition), is used for systems
involving symmetric matrices, while the Bi-CGStab algorithm by van der Vorst [158] is usually used for asymmetric systems.

### 3.3 Discretization of the Navier-Stokes Equations

This section will describe the discrete solution procedure of the Navier-Stokes equations for fluid-flow. In particular, it will focus on issues that pertain to pressure-velocity coupling for unsteady flows, and the methods used to deal with them.

For clarity, the incompressible Navier-Stokes equations (at constant density) for a Newtonian fluid are outlined below:

\[
\nabla \cdot \mathbf{v} = 0 \tag{3.51}
\]

\[
\frac{\partial \rho \mathbf{v}}{\partial t} + \nabla \cdot (\rho \mathbf{v} (\mathbf{v} - \mathbf{v}_s)) = -\nabla p + \nabla \cdot (\mu \nabla \mathbf{v}) \tag{3.52}
\]

where \( \rho \) is the fluid density, \( \mathbf{v} \) is the fluid velocity, \( \mathbf{v}_s \) is the velocity of the control volume boundaries (i.e., the mesh velocity), \( p \) is the pressure, and \( \mu \) is the dynamic viscosity of the fluid.

The convection term in the Navier-Stokes system defines the transport of momentum, where the fluid is advecting itself. This term slightly complicates the solution procedure because it is non-linear in nature. While it is possible to perform a non-linear discretization of the term and solve the system using non-linear equation solvers, the computational effort involved with the approach is seldom justified. A simpler alternative is to linearize the term using the procedure described in Section 3.1.1:
\[
\int_{V_P} \nabla \cdot (\rho \mathbf{v} (\mathbf{v} - \mathbf{v}_s)) = \sum_f S \cdot (\rho \mathbf{v} (\mathbf{v} - \mathbf{v}_s))_f = \sum_f S \cdot (\mathbf{v}_s)_f \mathbf{v}_f = \sum_f F \mathbf{v}_f
\] (3.53)

where \( F \) represents the mass flux through the face \( f \), relative to the mesh:

\[
F = S \cdot (\rho (\mathbf{v} - \mathbf{v}_s))_f
\] (3.54)

This procedure essentially ‘lags’ part of the non-linear convection term, by using the value of the previous iteration. In convection-dominated transient flows, this term can become significant and so, several iterations may be necessary per time-step to fully resolve the non-linear effects.

To understand the concept of pressure-velocity coupling, it is convenient to discretize the Navier-Stokes equations into a block LU decomposition (Perot [126]) of the form:

\[
\begin{bmatrix}
A & G \\
D & 0
\end{bmatrix}
\begin{bmatrix}
\mathbf{v}^n \\
p^n
\end{bmatrix} = \begin{bmatrix}
r^o \\
0
\end{bmatrix}
\] (3.55)

where \( G \) and \( D \) are matrices that represent the discrete gradient and divergence operators, and \( A \) is a sub-matrix (which lacks the pressure-gradient term) whose structure depends on the form of temporal and spatial discretization. The pressure \( p \) must always be solved implicitly when the equations are incompressible to enforce the incompressibility constraint (which must be true at the next time level \( n \)). This is defined by the bottom row of the matrix in Eq.(3.55). The vector \( r^o \) is the explicit right-hand side of the momentum equations (including boundary conditions) for the momentum and pressure equations.

The matrix system in Eq.(3.55) has both positive and negative eigenvalues and is therefore, not easy to invert. If an iterative method is used to solve this system
then it must be converged nearly to machine precision, as any errors in the iterative solution mean that the incompressibility constraint is not exactly satisfied. These iteration errors show up effectively as local mass creation and destruction, and are highly detrimental to the overall solution accuracy.

Most methods used to solve Eq.(3.55) involve some form of pressure-projection in order to project the velocity field to a divergence-free space. Two popular methods are the Fractional Step method by Chorin [30] and Temam [149], and the PISO algorithm by Issa [82]. The present work chooses the latter approach.

Projection methods are relatively popular because it allows the momentum equations to be split in to a conventional advection-diffusion equation, and a separate pressure Poisson equation. This is achieved by pre-multiplying the first row of Eq.(3.55) by the inverse of $A$:

$$
\begin{bmatrix}
I & A^{-1}G \\
D & 0
\end{bmatrix}
\begin{bmatrix}
v^n \\
p^n
\end{bmatrix}
= 
\begin{bmatrix}
A^{-1}r^o \\
0
\end{bmatrix}
$$

(3.56)

Writing out the first row explicitly, this becomes:

$$
v^n = A^{-1}r^o - A^{-1}Gp
$$

(3.57)

Substituting Eq.(3.57) into the second row of Eq.(3.55) yields:

$$
DA^{-1}Gp^n = DA^{-1}r^o
$$

$$
Lp^n = DA^{-1}r^o
$$

(3.58)

The term $L = DA^{-1}G$ is the discrete equivalent of the Laplacian operator, resulting in Poisson equation for pressure, which has to be solved at each time-step to ensure incompressibility. It is at this step that most incompressible fluid-flow solvers
spend the most time, and therefore justification for the need to perform this step efficiently. Since it is impractical to compute the inverse of the sparse matrix $A$, it is usually approximated (as the inverse diagonal of $A$, for example).

Writing out the momentum equation for a given control volume in the mesh using the notation given in Eq.(3.47) yields:

$$a_P v_P + \sum_N a_N v_N = R_P - \nabla p$$

$$v_P = (a_P)^{-1}(R_P - \sum_N a_N v_N - \nabla p) \quad (3.59)$$

$$v_P = (a_P)^{-1}(H(v) - \nabla p) \quad (3.60)$$

where $H(v) = R_P - \sum_N a_N v_N$ represents the ‘transport’ part of the momentum equation, such as the advection-diffusion terms and source part of the transient term, but not the pressure gradient term.

Taking the divergence of velocity in Eq.(3.60) to satisfy the incompressibility criterion, it follows:

$$\nabla \cdot ((a_P)^{-1}\nabla p) = \nabla \cdot ((a_P)^{-1}H(v)) \quad (3.61)$$

Using the spatial discretization of the gradient and divergence operators yields the final form of the discrete Poisson equation for pressure:

$$\sum_f (a_P)^{-1}\frac{|S_f|}{|d|}(p_N - p_F)_f = \sum_f |S_f|(a_P)^{-1}_f H(v)_f \quad (3.62)$$

The following sequence elaborates on the steps in the PISO algorithm, assuming that divergence-free fluxes are available:

- Assemble the momentum equation using the available conservative fluxes.
- Solve the momentum equation in Eq.(3.63), known as a momentum-predictor, either with or without an existing pressure field $p^*$, to obtain a non-conservative
face-flux, \( \phi^* = v_f \cdot S_f \). The use of an existing pressure field helps to accelerate convergence.

\[
v_p = (a_P)^{-1}(H(v) - \nabla p^*)
\]  

(3.63)

- Calculate the pressure based on the updated velocity field using the discrete Poisson equation from Eq.(3.62). This is called a pressure-correction step.

- Correct the non-conservative flux \( \phi^* \), using the correction flux \( F \) obtained using the pressure-correction step:

\[
F = |S_f|(a_P)^{-1}H_f(v) - (a_P)^{-1}|S_f||d|(p_N - p_P)_f
\]  

(3.64)

Note that the terms \( a_P, H(v) \) and \( \nabla p \) have all been interpolated to faces prior to the dot-product with the face-normal vector \( S_f \), to obtain the correction flux \( F \). The divergence-free fluxes \( (\phi) \) are obtained by Eq.(3.65).

\[
\phi = \phi^* - F
\]  

(3.65)

- Update the cell-centred velocity using the gradient of the updated pressure:

\[
v_p = (a_P)^{-1}(H(v) - \nabla p)
\]  

(3.66)

- Repeat from the pressure-correction step until conservative face fluxes are obtained for a specified tolerance.

It is interesting to note that the face-flux \( \phi \) is responsible for maintaining the incompressibility criterion along with pressure, while the cell-centred velocity is merely a secondary variable that is used during the discretization process.
Adaptive mesh reconnection is a fairly broad term that is used to describe the process of re-meshing (or re-gridding) an existing mesh, subject to certain requirements. In the framework of Lagrangian interface-tracking, such methods are often necessary in situations where cells have become excessively distorted, and mesh smoothing can do very little to mitigate the issue. One approach is to re-mesh the domain entirely, using an appropriate mesh-generation algorithm. There are two major drawbacks to this approach:

- Mesh-generation can be a particularly time-consuming process, and currently, automatic mesh generation (i.e., without any user intervention) is not well established.

- Re-meshing requires the interpolation of flow variables to the new mesh, which can frequently induce errors and fluctuations to the flow-field. By re-meshing the entire domain, these errors can often be difficult to contain, and might even magnify as the simulation proceeds over time.

Therefore, a more logical approach is local re-meshing, which works well in minimizing interpolation errors and, given the right algorithms, can be quite efficient [38]. The mesh reconnection algorithms in this work are limited to simplical meshes (triangles in 2D and tetrahedra in 3D), since generalization of these concepts to arbitrary polyhedra is complicated.

The topic of mesh reconnection can also be extended to include refinement and derefinement of cells. Physical phenomena can often develop near-singular solutions
with large gradients in very localized regions of the mesh (like the neck-region during droplet break-up, for instance), and in most cases, the only solution is to resolve these variations using an increased number of cells in the area. The option of uniformly refining the entire mesh is immediately rejected, because the exponential increase in computational effort (particularly in three dimensions) is not really justified, and areas away from the singularity don’t have to be resolved that well anyway. Local refinement allows an increase in mesh density around areas that need it most, thus providing improved solution accuracy at an acceptable computational cost.

This chapter will discuss both topics - mesh reconnection for improved mesh quality and re-meshing methods for adaptive refinement, in the context of two- and three-dimensional simplical meshes.

4.1 Mesh Reconnection for Improved Mesh Quality

The quality of a simplical mesh can be locally improved by an operation known as edge-swapping (sometimes also known as edge-flipping), which is applicable in both two- and three-dimensions.

![Figure 4.1. Edge flipping operation using the Delaunay criterion.](image)
In two-dimensions, the condition for edge-swapping is defined by the Delaunay criterion, which specifies that no mesh points are to be contained in the circumcircle of any cell of the mesh. This concept was first introduced by Lawson [103], who also extended edge-swapping to three and higher dimensions. Fig. 4.1 shows the Delaunay criterion in 2D. The point marked $d$ is contained within the circumcircle of the triangle $(abc)$ and so, edge $bc$ must be flipped. The flipped configuration (and new circumcircle) is shown in the figure on the right.

![Figure 4.2. Triangular prism mesh equivalent to Fig. 4.1](image)

This approach is mathematically guaranteed to provide a mesh of better quality; as it maximizes the minimum angle of a triangulation and is also irreversible, thereby preventing infinite loops. In the current code framework, 2D simulations are performed by extruding a two-dimensional surface-mesh by one cell in the direction normal to the mesh-plane. Thus, two-dimensional simulations are actually performed in 3D. In this context, a 2D simplical mesh is now no longer composed of triangles, but triangular prisms, as shown in Fig. 4.2.

From the description of the 2D edge-swapping algorithm, it becomes clear the operation can only be performed on edges on the interior of the mesh (i.e., with two cells on either side). Referring again to Fig. 4.2, the face $(bcgf)$ is the equivalent...
of an interior edge in a 2D mesh. The algorithm for edge-swapping is given in the
\texttt{Swap2DEdges} function.

\begin{verbatim}
\textbf{Swap2DEdges()}
\textbf{Input} : Stack of interior faces (M)
\textbf{while} M is not empty \textbf{do}
  \textbf{f} = \text{pop}(M)
  \textbf{failed} = \text{TestDelaunay}(f)
  \textbf{if} failed \textbf{then}
    \text{Swap}(f)
\end{verbatim}

\textbf{Function Swap2DEdges}

The \texttt{TestDelaunay} function takes an internal face \( f \) (which must be a
quadrilateral), and looks for two triangular faces that share one of its edges. The
circumcenter for the first triangular face, \( \mathbf{x}_{CC} \), given its vertex positions, \( \mathbf{x}_0, \mathbf{x}_1 \) and
\( \mathbf{x}_2 \), is calculated using the relation:

\[ \mathbf{x}_{CC} = \frac{(c_1 + c_2)\mathbf{x}_0 + (c_0 + c_2)\mathbf{x}_1 + (c_0 + c_1)\mathbf{x}_2}{2(c_0 + c_1 + c_2)} \]  
\[ c_0 = (\mathbf{x}_2 - \mathbf{x}_0) \cdot (\mathbf{x}_1 - \mathbf{x}_0) \]
\[ c_1 = -(\mathbf{x}_2 - \mathbf{x}_1) \cdot (\mathbf{x}_1 - \mathbf{x}_0) \]
\[ c_2 = (\mathbf{x}_2 - \mathbf{x}_0) \cdot (\mathbf{x}_2 - \mathbf{x}_1) \] (4.1)

The function then looks for a point on the second triangular face \( \mathbf{x}_d \), which does
not belong to \( f \), and tests for the following criterion:

\[ (\mathbf{x}_d - \mathbf{x}_{CC}) \cdot (\mathbf{x}_d - \mathbf{x}_{CC}) < (\mathbf{x}_0 - \mathbf{x}_{CC}) \cdot (\mathbf{x}_0 - \mathbf{x}_{CC}) \]  
\[ (\mathbf{x}_d - \mathbf{x}_{CC}) \cdot (\mathbf{x}_d - \mathbf{x}_{CC}) < (\mathbf{x}_0 - \mathbf{x}_{CC}) \cdot (\mathbf{x}_0 - \mathbf{x}_{CC}) \] (4.2)

If this condition given by Eq.(4.2) is true, then the face \( f \) fails the Delaunay test, and
must therefore be flipped.

Edge swapping in three-dimensions is much more complicated when compared
to the 2D case. In a tetrahedral mesh, an edge can be connected to an arbitrary
number of cells and so, it becomes unclear how an edge-swapping operation is to be implemented in a general manner. One of the first proposed solutions to this problem was defined by Briere de L’isle and George [23], called the edge-removal operation.

Edge-removal is a topological operation that removes a single edge from the mesh (and therefore, all the tetrahedra connected to it), and replaces the local void with a new configuration of tetrahedra with an improved value of a pre-defined mesh quality metric $q$. The quality metric is typically a scalar value defined for an individual tetrahedral cell, normalized to vary in the range between 0 and 1, where a quality of 0 denotes a degenerate element with zero volume, and 1 denotes an element that is close to ideal (usually an equilateral tetrahedron). It naturally follows that an inverted tetrahedron (one with negative volume) possesses a negative value for the quality metric. The topic of mesh quality will be revisited in Section 4.1.1. In general, if an edge is surrounded by $m$ tetrahedra, it is replaced by $(2m - 4)$ cells during the edge-removal operation. It must also be noted that the operation can (and usually does) introduce more edges to the mesh. Before delving into the specifics of the edge-removal operation, it is important to enumerate the three basic swap operations in 3D, shown in Fig. 4.3 and Fig. 4.4.

![Figure 4.3. Complementary 2-3 and 3-2 swapping operations in 3D](image)
The 2-3 swap is an operation which removes a common face shared by two tetrahedra and replaces it with an edge, thus creating three new cells instead. The 3-2 swap is complementary to this operation, which works by removing an edge connected to three tetrahedra and replacing it with a common face.

![Figure 4.4. Surface 2-2 swapping operation in 3D](image)

The 2-2 swap shown in Fig. 4.4 is applicable to surface edges, where the edge $ab$ is swapped to edge $cd$. Later in the section, it will also be shown that any edge in the mesh, surrounded by an arbitrary number of tetrahedra connected to it, can be removed by using a sequence of these three elementary operations.

Take a minimal example involving an interior edge $ab$ connected to four tetrahedra, as shown in Fig. 4.5. There are two possible ways in which this edge can be removed using only 2-3 and 3-2 swaps.

The first sequence of operations involves the creation of an edge $p_1p_3$, thereby replacing the face $(abp_0)$ using a 2-3 swap, as shown in Fig. 4.5(b). The next step in the first sequence is to remove the edge $ab$ by creating a face $(p_1p_2p_3)$ using a 3-2 swap, as shown in Fig. 4.5(c).

The second sequence of operations involves the creation of an edge $p_0p_2$, thereby replacing the face $(abp_1)$ using a 2-3 swap, as shown in Fig. 4.5(d). The next step in
the second sequence is to remove the edge $ab$ by creating a face $(p_0p_2p_3)$ using a 3-2 swap, as shown in Fig. 4.5(e).

It now becomes apparent that these two possible sequences are a result of the two possible ways in which the polygon $(p_0p_1p_2p_3)$ (which Dai [37] calls the ‘equitorial polygon’) has been triangulated. Although the equitorial polygon is shown to be almost planar in Fig. 4.5 for clarity, it is rarely so in typical tetrahedral meshes. Taking Fig. 4.5(a), for instance, when looking down on edge $ab$ from the side of point $a$, Fig. 4.6(a,b) shows the two possible triangulations obtained. A more general triangulation involving an equitorial polygon with five sides is shown in Fig. 4.6(c).

After having calculated the minimum quality of the cells ($q_{\text{orig}}$) surrounding edge $ab$ in Fig. 4.5(a), the natural choice of a particular triangulation (and the associated sequence) would be one which maximizes the minimum quality metric ($q_{\text{min}}$) of the cells after the edge-removal process (Fig. 4.5(c) or (e)), provided $q_{\text{min}} > q_{\text{orig}}$. While...
considering equitorial polygons with an increasing number of sides, the number of possible triangulations quickly becomes quite daunting. For instance, a 6-sided polygon can be triangulated in 14 ways, a 7-sided polygon in 42 ways, and for an 8-sided polygon - 132 ways. Thus, checking against every possible triangulation for each edge in the mesh is clearly impractical.

Freitag and Ollivier-Gooch [63] attempted to simplify this problem by classifying any triangulation as a member of a particular topological case for a given equitorial polygon. Taking Fig. 4.6(c) as an example, all five cases can be regarded as a rotation of the first triangulation. Freitag also limited the number of combinations to 7-sided polygons, since empirically, mesh quality rarely improves for cases involving a larger number of sides.

The algorithm used in this work, devised by Shewchuk [140], is an optimization-based alternative to the one by Freitag and Ollivier-Gooch, which is simple to implement and performs the optimal triangulation in polynomial time. The algorithm uses a dynamic programming approach to the triangulation problem, which was
originally introduced by Klincsek [97], and does not impose an upper bound on the number of sides of the equitorial polygon. Shewchuk also noticed that once an optimal triangulation is obtained, the edge removal process consists of a sequence of \((m - 3)\) 2-3 swaps followed by a final 3-2 swap, where \(m\) is the number of vertices in the ring surrounding the edge (See Fig.(4.7)).

Given an edge \(ab\) which is to be checked for removal, define a closed ring of points \(R\) around \(ab\) defined by the equitorial polygon. The points \((p_0, p_1, \ldots, p_n)\) in \(R\) are ordered in a counter-clockwise manner around \(ab\), when viewed from \(a\). If the edge \(ab\) lies on the boundary of the mesh, then the ring \(R\) roughly subtends an angle of 180° around \(ab\). If this angle is less than 180°, then the edge probably lies on a bounding-curve, which is a curve that distinctly defines the geometry of the mesh, and must not be removed to preserve the shape of the domain.

Assume \(T\) to be a (non-planar) triangulation of \(R\) (as shown in Fig. 4.7). Each triangle in \(T\), when connected to points \(a\) and \(b\) located on either side of the polygon, form two new tetrahedra. For a given triangulation, its quality is defined by the minimum quality of the tetrahedra induced by connecting the triangles to points \(a\) and \(b\).

![Figure 4.7. Klincsek’s recursive triangulation algorithm](image)

Figure 4.7. Klincsek’s recursive triangulation algorithm
From the original ring of points \( R \), consider a sub-ring \( R_{ij} \) (with \( i < j \)), composed of points \( (p_i, p_{i+1}, \ldots, p_j) \), and an additional edge \( ij \). Assume that \( T_{ij} \) is a triangulation of this sub-ring. It naturally follows (from Fig. 4.7) that a triangulation in \( T_{ij} \) is possible only if \( j > i + 1 \). Now, if \( j \geq i + 2 \), then the edge \( pi pj \) is considered to be the side of a triangle in \( T_{ij} \). Choosing any point \( p_k \) in \( R_{ij} \), the triangulation is sub-divided into three parts - \( T_{ik} \), the triangle \( (ikj) \), and \( T_{kj} \). From this point, \( T_{ik} \) and \( T_{kj} \) are triangulated recursively until no more sub-triangulations are possible. The goal here is to maximize the quality of each sub-triangulation. While considering every possible sub-triangulation, the likelihood of a particular case being tested more than once is very high. To avoid this situation, Klincsek’s dynamic programming algorithm is used, such that no sub-problem is solved more than once.

\[
\text{SWAP3DEdges}()
\]

\[
\text{Input : Stack of edges (M)}
\]

\[
\text{INITTABLES(Q, K)}
\]

\[
\text{while M is not empty do}
\]

\[
\begin{align*}
\text{\hspace{2em} e} & = \text{POP(M)} \\
\text{\hspace{2em} if} \text{ CHECKBOUNDINGCURVE}(e) \text{ then} \\
\text{\hspace{4em} continue} \\
\text{\hspace{2em} minQuality} & = \text{COMPUTE.MINQUALITY}(e) \\
\text{\hspace{2em} FILLTABLES}(e, Q, K) \\
\text{\hspace{2em} if} \text{ CHECKQUALITY}(e, Q, \text{minQuality}) \text{ then} \\
\text{\hspace{4em} REMOVE.EDGE.FLIPS}(e, K)
\end{align*}
\]

\[
\text{Function SWAP3DEdges}
\]

The dynamic programming algorithm uses two tables \( Q \) and \( K \), both sized to \( (m - 1) \times m \). For convenience, these tables are indexed from 0. The table \( Q \) stores the scalar qualities of triangulations, such that \( Q[i][j] \) specifies the quality of the optimal triangulation of \( T_{ij} \), and is filled in using the recurrence relation given in Eq.(4.3):

\[
Q[i][j] = \max_{k \in [i+1, j-1]} \min\{Q[i][k], q(x_i, x_k, x_j, x_a), q(x_j, x_k, x_i, x_a), Q[k][j]\} 
\]  (4.3)
where \( q(x_0, x_1, x_2, x_3) \) is a cell quality metric computed using one of the measures given in Section 4.1.1, using points corresponding to indices \( p_0, p_1, p_2 \) and \( p_3 \). The entries in \( Q \) are initialized to large values, and are filled in the order of decreasing \( i \) and increasing \( j \). The table \( K \) stores the indices of the ring that are used to reconstruct the optimal triangulation, such that \( K[i][j] \) stores the index \( k \) that maximizes Eq.(4.3). Once the tables have been filled in, if the quality value specified in \( Q[0][m-1] \) is better than the initial configuration, then the edge is to be removed.

**ComputeMinQuality**

**Function ComputeMinQuality**

The top level routine **Swap3DEdges** picks edges off a stack and computes the minimum quality of cells surrounding each one using the **ComputeMinQuality** function. This function uses a supplied edge-ring connectivity list (\( \text{ring} \)), which is a list of point indices ordered in a counter-clockwise manner around the first point of edge \( e \). If edge \( e \) is on a boundary, then \( \text{ring} \) must start and end with boundary points. The function \( \text{TetQuality}(x_a, x_b, x_c, x_d) \) computes the quality of the tetrahedron with the specified points, while the \( \text{rcIndex}(\text{list}, i) \) function for a container \( \text{list} \) provides the reverse-circular index for a particular index \( i \), where
The \textsc{FillTables} function then fills the programming tables for the edge under consideration. The running time of this function is $O(m^3)$, where $m = \text{size}(\text{ring})$. Because $m$ is typically a small value (usually around 5 to 7 cells for each edge), this is not a large limitation, but it is the most time consuming aspect of the entire algorithm, particularly for large mesh sizes.

\begin{verbatim}
FILLTABLES(e, Q, K)

Inputs : Edge-ring connectivity (ring), mesh points list (points), mesh edges list (edges)

checkEdge = edges[e]
m = size(ring)
x_a = points[checkEdge[0]]
x_b = points[checkEdge[1]]
for i = m-3, down to 0 do
  for j = i+2, m do
    for k = i+1, j do
      x_i = points[ring[i]]
x_j = points[ring[j]]
x_k = points[ring[k]]
      qa = TETQUALITY(x_i,x_k,x_j,x_a)
x_b = TETQUALITY(x_j,x_k,x_i,x_b)
      q = MIN(qa, qb)
      if k < j - 1 then
        q = MIN(q, Q[k][j])
      if k > i + 1 then
        q = MIN(q, Q[i][k])
      if k = i + 1 or q > Q[i][j] then
        Q[i][j] = q
        K[i][j] = k

Function FILLTABLES

The \textsc{CheckBoundingCurve} function, as the name suggests, checks whether the edge lies on a bounding curve and if it does, prevents it from being removed. If the \textsc{CheckQuality} function detects that an improved triangulation is available, the edge removal operation is carried out by the \textsc{RemoveEdgeFlips} function.

\end{verbatim}
**CheckQuality**$(e, Q, minQuality)$

**Inputs** : Edge-ring connectivity $(ring)$

$m = size(ring)$

if $Q[0][m - 1] > minQuality$ then
  return true

return false

**Function CheckQuality**

The approach taken here for the swap sequence is identical to the one by Dai [37], where triangulations are first extracted from the table $K$ returned by **FillTables**, followed by a sequence of 2-3 swaps and a final 3-2 or 2-2 swap. This is because the original algorithm by Shewchuk permits the intermediate creation of inverted tetrahedra during the sequence, although it is guaranteed that the final configuration will not contain any invalid cells. Dai’s approach involved the interpolation of flow variables directly after each 2-3 / 3-2 swap operation, which required valid tetrahedra at each step. This work chooses to perform the interpolation step only after all mesh reconnection operations have been made to the mesh and so, the mesh validity restriction does not strictly apply. However, using a pre-determined sequence does simplify algorithm development, particularly with regard to maintaining coherent connectivity structures during swap operations.

**RemoveEdgeFlips**$(e, K)$

**Inputs** : Edge-ring connectivity $(ring)$, Empty $3 \times m_{\text{max}}$ list $(\text{triangulations})$

$m = size(ring)$

$numT = 0$

**ExtractTriangulation**(0, $m - 1$, $K$, $numT$, $\text{triangulations}$)

$t_f = \text{Identify32Swap}(e, \text{ring}, \text{triangulations})$

$nSwaps = 0$

while $nSwaps < (m - 3)$ do

  for $i = 0, (m - 2)$ do

    if $i \neq t_f$ and **BoundaryTriangulation**$(i, \text{triangulations})$ then

    **Swap23**(triangulations)$[:][i])$

    $nSwaps = nSwaps + 1$

**Swap32**(triangulations)$[:][t_f])$

**Function RemoveEdgeFlips**
The \texttt{ExtractTriangulation} function recursively calls itself to obtain all sub-triangulations from the table $K$, given a start and end vertex on the ring.

\begin{verbatim}
ExtractTriangulation(i, j, K, numT, triangulations)
  if j > (i + 2) then
    triangulations[0][numT] = i
    triangulations[1][numT] = k
    triangulations[2][numT] = j
    numT = numT + 1
    ExtractTriangulation(i, K[i][j], K, numT, triangulations)
    ExtractTriangulation(K[i][j], j, K, numT, triangulations)
\end{verbatim}

\textbf{Function ExtractTriangulation}

The \texttt{BoundaryTriangulation} function determines whether the triangulation stored at a particular index in the \texttt{triangulations} list corresponds to one on the boundary. The function counts the number of times that the vertices in the input triangulation are repeated in all triangulations. If any vertex on the triangulation is counted only once, then it must be on the boundary.

\begin{verbatim}
BoundaryTriangulation(i, triangulations)
  f = 0, s = 0, t = 0
  for row = 1, size(triangulations) do
    for col = 1, size(triangulations[row]) do
      if triangulations[row][col] = triangulations[0][i] then
        f = f + 1
      if triangulations[row][col] = triangulations[1][i] then
        s = s + 1
      if triangulations[row][col] = triangulations[2][i] then
        t = t + 1
  if f = 1 or s = 1 or t = 1 then
    return true
  else
    return false
\end{verbatim}

\textbf{Function BoundaryTriangulation}

The \texttt{Identify32Swap} routine is used to locate the triangulation in the sequence which corresponds to a 3-2 or 2-2 swap. This is done by identifying a triangulation which intersects the edge provided to the function, using a simple segment-triangle intersection algorithm that is both robust and efficient.
Identify32Swap(e, ring, triangulations)

Inputs: mesh points list (points), mesh edges list (edges)

\[ m = \text{size(ring)} \]
\[ e = \text{edges}[e] \]
\[ e_0 = \text{points}[e[0]] \]
\[ e_1 = \text{points}[e[1]] \]

for \( i = 0, (m - 2) \) do

\[ t_a = \text{points}[\text{triangulation}[i][0]] \]
\[ t_b = \text{points}[\text{triangulation}[i][1]] \]
\[ t_c = \text{points}[\text{triangulation}[i][2]] \]

\[ \text{intersects} = \text{SegmentTriFaceIntersection}(t_a, t_b, t_c, e_0, e_1) \]

if \( \text{intersects} \) then

\[ \text{return } i \]

Function Identify32Swap

The case of identifying a 2-2 swap depends on the convexity of the surface on which the input edge lies. If the surface is too concave at the edge, an intersection may not even exist, and a 2-2 swap would result in a small change in the local shape of the surface and therefore, a proportional change in mesh volume. A mildly concave surface swap may be desirable, especially if it can improve the local mesh quality, at the expense of a negligible change to overall mesh volume. The SegmentTriFaceIntersection function expects the points \( x_a, x_b \) and \( x_c \) to be ordered in a counter-clockwise manner when viewed from \( e_0 \). Since this order is inherently maintained by the FillTables function, this limitation is not a cause for concern.

When put together, the RemoveEdgeFlips function performs \((m - 3)\) 2-3 swaps followed by a final 3-2 swap. The final swap32 function is also designed to handle a 2-2 swap, if the edge \( e \) lies on the boundary of the mesh.

4.1.1 Mesh Quality Measures for Tetrahedra

The quality of a particular mesh is of great importance, particularly with regard to solution accuracy and convergence rate. However, it is often difficult to quantify how ‘good’ a mesh actually is. Past experience seems to indicate that meshes which possess nearly equilateral cells often produce solutions with sufficient accuracy.
\text{SegmentTriFaceIntersection}(x_a, x_b, x_c, e_0, e_1)

\begin{align*}
n_f &= \frac{1}{2} ((x_b - x_a) \times (x_c - x_a)) \\
n_f &= n_f / |n_f| \\
n &= n_f \cdot (x_a - e_0) \\
d &= n_f \cdot (e_1 - e_0) \\
\text{if } |d| < \epsilon \text{ then} & \\
\quad \text{return false} \\
\text{if } u \geq 0.0 \text{ and } u \leq 1.0 \text{ then} & \\
\quad c_p &= e_0 + u(e_1 - e_0) \\
\quad s_0 &= ((x_b - x_a) \times (c_p - x_a)) \cdot n_f \\
\quad \text{if } s_0 < 0.0 \text{ then} & \\
\quad \quad \text{return false} \\
\quad s_1 &= ((x_c - x_b) \times (c_p - x_b)) \cdot n_f \\
\quad \text{if } s_1 < 0.0 \text{ then} & \\
\quad \quad \text{return false} \\
\quad s_2 &= ((x_a - x_c) \times (c_p - x_c)) \cdot n_f \\
\quad \text{if } s_2 < 0.0 \text{ then} & \\
\quad \quad \text{return false} \\
\quad \text{return true} \\
\end{align*}

\text{return false}

\textbf{Function SegmentTriFaceIntersection}

(Shewchuk [142]), but it is also true that anisotropy is sometimes preferred in certain situations like boundary layers, where slender cells aligned in the flow direction can provide an optimal trade-off between cost and accuracy. Aspects such as interpolation accuracy and gradient evaluation are strongly influenced by how well-shaped the underlying cell is, and often, the rule of thumb is that extreme angles are to be avoided. Poorly shaped elements often contribute toward increasing the condition number of the solution matrix, which has a negative effect on convergence characteristics of the iterative solver.

The most popular approach to evaluating mesh quality is the use of an easily computed algebraic \textit{quality metric}, which is a function of the point positions of the cell. These metrics can be size dependent, which is sometimes useful when dealing with cases involving spatial variations in mesh density, but they are taken to be size-invariant in this work, and are used only for shape optimization. As stated earlier, the quality metric is a scalar value which is normalized to vary in the range between
0 and 1, where a quality of 0 denotes a *degenerate* cell with zero volume, and 1 denotes a cell that is close to ideal. The measures are usually multiplied by the sign of the cell volume, so that inverted tetrahedra possess a negative value for the quality metric. These metrics also inherently assume that the points of the cell are ordered in a topologically consistent manner to produce a positive value for cell volume if it is valid. The equation for the volume of a tetrahedral cell with points \((x_0, x_1, x_2, x_3)\) is given in Eq.(4.5):

\[
V_c = \frac{1}{6} \left[ \left( (x_1 - x_0) \times (x_2 - x_0) \right) \cdot (x_3 - x_0) \right]
\] (4.5)

which assumes that \(x_0, x_1\) and \(x_2\) are oriented in a counter-clockwise manner when viewed from point \(x_3\).

Some of the metrics used in this work for the optimization of mesh quality are given here. For all measures, \(V_c\) denotes the cell volume, \(L_e\) denotes the edge-length, while \(L_{rms}\) and \(A_{rms}\) denote the root-mean-square values of edge-lengths and face-areas of a tetrahedral cell.

The first measure, suggested by Knupp [100], uses a normalized ratio between the volume of a cell to the sum of its squared edge lengths, as shown in Eq.(4.6):

\[
q = \frac{6^{5/3} \sqrt{3} V_c^{2/3}}{6 \sum_{i=1}^{6} L_e^2}
\] (4.6)

It becomes apparent that this metric would tend to penalize tetrahedra with small volumes compared to an optimal tetrahedron with equivalent edge lengths, since \(q \rightarrow 0\) as \(V_c \rightarrow 0\).

Knupp [98] also suggests an alternative metric that is based on the Frobenius norm of the element condition number, which simplifies to the ratio specified in Eq.(4.7):

\[
q = \frac{3 \sqrt{6} V_c}{L_{rms} A_{rms}}
\] (4.7)
Another metric suggested by Liu and Joe [106] (which is numerically equivalent to Eq.(4.6)) is the Mean Ratio metric, given in Eq.(4.8):

\[ q = \frac{12 (3 V_c)^{2/3}}{\sum_{i=1}^{6} L_i^2} \]  

(4.8)

### 4.2 Mesh Reconnection for Length Scale Resolution

Adaptive refinement and derefinement of cells in a moving-mesh paradigm is a particularly attractive feature, mainly because it allows an increase in mesh density at locations involving interesting physical phenomena. For example, the accuracy of the discrete surface curvature calculation largely depends on whether the mesh is adequately refined is at a given point on the surface, particularly at regions involving rapid spatial variations. Alternatively, at regions where solution variations are more gradual, such as interior locations of the mesh away from the surface, increased mesh density does very little to improve spatial accuracy and only contributes towards unnecessary computational effort.

Mesh refinement can be performed in a variety of ways. The most common approach, also taken in this work, is the **h-refinement** method, which involves the addition of points at regions which require higher mesh density to resolve high solution gradients. Derefinement (or the removal of points) is also possible in regions where the solution error is low. Due to the continuous addition and deletion of points during the solution process, this approach to adaptive mesh refinement has the drawback of yielding a highly unpredictable computational solution time per time-step. Examples of h-refinement can be found in work by Jasak [85], Muzaferija [110], and Coelho et al. [31], among others.

The **r-refinement** method keeps the number of points in the domain constant and redistributes them across the domain during the solution process based on error requirements, thereby leading to a constant solution time at every time-step.
Laplacian spring analogy based smoothing, as discussed in Chapter 5, can also be used to perform r-refinement by using a variable spring constant that is proportional to the gradient of the solution. However, it is often difficult to judge \textit{a-priori} whether the number of available points at the start of the simulation is sufficient to resolve high spatial gradients (which are often unpredictable themselves), and this method is also known to degrade mesh-quality in regions where refinement is not needed (Hawken et al. [78]).

The \textbf{p-refinement} approach is usually applicable to Finite-Element methods which use a polynomial shape function for spatial discretization. Refinement is achieved by increasing the polynomial degree for elements in areas where the discretization error is large, and is usually accompanied by a proportionate increase in computational effort comparable to h-refinement. Increase in polynomial degree in the vicinity of high solution gradients is typically associated by overshoots in interpolation and other spurious oscillations which must be taken into account.

\subsection{4.2.1 Estimation of Local Length Scale}

Prior to an adaptive mesh refinement procedure, it is important to define a criterion for which points are to be added to (or removed from) the mesh. While there are several choices (such as solution error, for example), the criterion used in this work is based on an estimate of the \textit{local length scale} field \( L(x) \), which is a scalar value defined for every cell in the mesh. Such a field needs to satisfy two requirements:

\begin{itemize}
  \item The value of \( L(x) \) at \( x = \partial \Omega \) (where \( \partial \Omega \) is the boundary of the solution domain \( \Omega \), like a fluid interface), must be sufficiently small so that the curvature of the surface is captured with sufficient accuracy. It must not be too small, so as to increase numerical stiffness induced by reducing the time-step, dictated by Eq.(1.7).
\end{itemize}
• Also, \( L(\mathbf{x}) \) for \( \mathbf{x} \in \Omega \) must gradually increase as \( \mathbf{x} \) tends away from \( \partial \Omega \) towards the interior, so that computational effort for the solution is reduced.

Dai [37] used a spatially varying vector indicator function to achieve an estimate for \( L(\mathbf{x}) \). This approach requires the solution to a discrete vector Laplace equation at each time-step (albeit with a relaxed tolerance for the iterative solver), which is comparable to the cost of the pressure Poisson equation.

The algorithm devised in this work uses a greedy approach in conjunction with mesh connectivity information, and requires a length scale value to be specified for boundary faces of the domain. Optionally, an adaptation-field (\( \alpha \), representing a volume phase-fraction between 0.0 and 1.0, for example) can also be provided as input. The fixed-value length scale is not required to be specified for all boundary faces, but must be specified for at least one of them (unless an \( \alpha \) field is specified). If an \( \alpha \) field is specified, a specified internal length-scale (\( L_\alpha \)) and a bounding threshold (\( \alpha_l \) and \( \alpha_h \)) must also be specified. In the \texttt{CALCULATELENGTHSCALE} function, the cell-levels list (\texttt{cLevel}) is an integer array for every cell in the domain, which is initialized to zero. The cell-cell connectivity list (\texttt{cellCells}) is an array of size \( n_c \) (where \( n_c \) is the number of cells in the mesh), such that \texttt{cellCells}[i] points to a list of indices indicating the cells that share faces with cell \( i \). The growth-factor (\( \gamma \)), is a scalar value greater than (or equal to) 1.0, which specifies how quickly the cell-size is expected to grow toward the interior of the mesh. The container (\texttt{cLc}) is an empty stack which is updated at each iteration, and contains cell indices of the current ‘level’. The variable \texttt{level} is initialized to 1. The ‘owner’ list, as defined in Chapter 3, specifies the cell index which ‘owns’ a given face, while the ‘neighbour’ list specifies the cell index on the other side of the given face. The objective is to obtain a discrete equivalent to \( L(\mathbf{x}) \), given in this algorithm by the container (\texttt{L}), which is an array of size \( n_c \) such that \texttt{L}[i] defines the length scale for cell \( i \). To achieve a concentration of mesh density...
in specific regions, the number of levels can be limited \((\text{maxLevel})\), and growth can be capped off to a mean length-scale \((L_{mean})\).

The algorithm begins by tagging all cells adjacent to fixed-value boundaries with a cell level of 1, and simultaneously adds tagged cells to the stack \((cLc)\). The main loop of the algorithm starts by copying cell indices in \(cLc\) to a new list, clears the existing stack, and queries for the list of cells \((cList)\) neighbouring each cell in \(\text{currentLevelCells}\). The next inner loop checks if any cells in \(cList\) have to be tagged \(\text{(i.e., have a 'level' of 0)}\). If un-tagged, the cell is marked with an incremented level, and the length scale for the cell is computed as the scaled average of all tagged cells connected to it. This cell is then added to \(cLc\) (which tracks cell indices of the next level). This procedure essentially tags each layer of cells towards the interior with a progressively higher ‘level’, where each level scales the length scale of the previous level by \(\gamma\). The algorithm is terminated when all cells in the mesh have been visited.

When adaptive refinement operations are applied using the length scale field as a criterion, adequate surface refinement is guaranteed based on choices for the boundary length scale \((bL, \text{which is the discrete equivalent of } bL, \text{described in Section 4.2.2})\), with progressively coarser length scales toward the interior of the mesh. Coarse scales trigger derefinement operations which reduce the mesh density at the interior, resulting in substantial computational cost savings.

The length scale field is augmented by the specification of a minimum \((\rho_{min} < 1.0)\) and maximum \((\rho_{max} > 1.0)\) ratio, which is used to control the length for interior and boundary edges. Thus, for an edge \(e_m\) with length \(l\), refinement is applied when \(l > \rho_{max}L[e_m]\), and derefinement is applied when \(l < \rho_{min}L[e_m]\); where \(L[e_m]\) is the local length scale value at the center of the edge. Since the field \(L\) is specified only for cells, \(L[e_m]\) is obtained by averaging the length scale for all cells surrounding \(e_m\) for interior edges, and specified by \(bL\) for boundary edges.
**CalculateLengthScale()**

**Inputs**
- Specified length scale values on \( \partial \Omega \) (\( bL \)), cell levels (\( cLevel \)), cell-cell connectivity (\( cellCells \)), growth-factor (\( \gamma \)), current-level stack (\( cLc \)), field parameters (\( \alpha, \alpha_l, \alpha_h, L_\alpha, L_{mean} \))

**Output**
- Length scale field (\( L \))

\[ visitedCells = 0, \ level = 1 \]

**forAll** Fixed-value boundary faces (\( fI \)) on \( \partial \Omega \) do

\[ ownerCell = owner[fI] \]

if \( cLevel[ownerCell] = 0 \) then

\[ cLevel[ownerCell] = level \]

\[ L[ownerCell] = \gamma \cdot bL[fI] \]

\[ visitedCells = visitedCells + 1 \]

\[ push(cLc, ownerCell) \]

**forAll** Fixed-value internal faces (\( fI \)) in \( \partial \Omega \) do

\[ ownerCell = owner[fI], neighbourCell = neighbour[fI] \]

\[ \alpha_{avg} = 0.5 \cdot (\alpha[ownerCell] + \alpha[neighbour[fI]]) \]

if \( \alpha_{avg} > \alpha_l \) and \( \alpha_{avg} < \alpha_h \) then

if \( cLevel[ownerCell] = 0 \) then

\[ cLevel[ownerCell] = level \]

\[ L[ownerCell] = L_\alpha \]

\[ visitedCells = visitedCells + 1 \]

\[ push(cLc, ownerCell) \]

if \( cLevel[neighbourCell] = 0 \) then

\[ cLevel[neighbourCell] = level \]

\[ L[neighbourCell] = L_\alpha \]

\[ visitedCells = visitedCells + 1 \]

\[ push(cLc, neighbourCell) \]

while \( visitedCells \leq n_c \) do

\[ currentLevelCells = cLc, clear(cLc) \]

for \( cI = 0, size(currentLevelCells) \) do

\[ cList = cellCells[currentLevelCells[cI]] \]

for \( cJ = 0, size(cList) \) do

\[ ngbLevel = cLevel[cList[cJ]] \]

if \( ngbLevel = 0 \) then

\[ cLevel[ncList[cJ]] = cLevel[ncList[cJ]] + 1 \]

\[ sumL = 0.0, nNgb = 0, ncList = cellCells[cList[cJ]] \]

for \( cK = 0, size(ncList) \) do

\[ nLevel = cLevel[ncList[cK]] \]

if \( sLevel < ngbLevel \) and \( sLevel > 0 \) then

\[ sumL = sumL + L[ncList[cK]] \]

\[ nNgb = nNgb + 1 \]

if \( level < maxLevel \) then

\[ sumL = (sumL/nNgb), sLength = \gamma \cdot (sumL) \]

else if \( L_{mean} > 0.0 \) then

\[ sLength = L_{mean} \]

\[ L[cList[cJ]] = sLength, push(cLc, cList[cJ]) \]

\[ visitedCells = visitedCells + 1 \]

\[ level = level + 1 \]

**Function** **CalculateLengthScale**
4.2.2 Estimation of Boundary Length Scale

The algorithm specified in Section 4.2.1 requires a fixed-value length scale to be defined at a certain number of boundary faces in the domain, based on which length scales at the interior of the mesh are established. This length scale can be specified in various ways, and the following criteria have been used in this work:

Constant surface length scale, \( (bL_f) \): The choice of a constant length scale allows a fixed edge length to be maintained at the surface during the course of the simulation. This approach will typically fail to refine the surface adequately at interesting regions (involving high curvature, for example), and is almost always combined with other boundary length scale specifications.

Length scale dictated by local curvature, \( (bL_\kappa) \): Local curvature at a specified surface edge is roughly estimated by comparing the normals of the two faces connected to it. If the projection of one normal on the other falls below a specified fraction, the edge is marked for refinement.

Length scale dictated by proximity to a surface \( \mathcal{S} \), \( (bL_p) \): In certain situations like droplet coalescence, where one section of the interface approaches another, it is often desirable to refine the approaching surfaces so that the transition for interface topology changes is relatively smooth. This would require the length scale at these sections to be defined by the proximity to the approaching surface. Clearly, checking for the distance to every other edge on the interface is impractical and therefore, a reduced search algorithm needs to be implemented. At each time-step in the simulation, an axis-aligned bounding box (\( B \)) for the centroids (points) of all faces of the surface (\( \mathcal{S} \)) is defined. This bounding box is then divided into a finite number of smaller boxes called bins. The size of these boxes is dictated by a specified spatial resolution \( (s_r) \). The container bins is a single-dimensional array of some arbitrary size (preferably prime), such that
bins[i] points to a list of faces in that particular bin. The faces are then hashed into these bins using the spatial hash algorithm given in the SpatialHash function.

\[\text{SpatialHash}(\text{points, } B, s_r)\]

**Output**: Spatially hashed points (bins)

\[dx = s_r / (\max(B) - \min(B))\]

for \(pI = 0, \text{ points} \) do

\[x_i = \text{points}[pI] \]

\[p = x_i - \min(B) \]

\[i = \text{floor}(p_x, dx_x) \]

\[j = \text{floor}(p_y, dx_y) \]

\[k = \text{floor}(p_z, dx_z) \]

\[\text{pos} = (k \cdot s_r, s_r) + (j \cdot s_r) + i \mod \text{size}(bins) \]

push(bins[pos], pI)

**Function** SpatialHash

Proximity of the specified surface edge to (\(S\)) is then calculated by the following steps:

1. The edge-normal (\(n_e\)) is calculated for each edge by adding the normals of two boundary faces adjacent to it.

2. With a step-size of the edge length, take multiple steps in the edge-normal direction and perform a spatial hash of each step to obtain the corresponding bin. This bin is added to the list of bins to be checked (binList).

3. Now loop through all bins in binList, and calculate the distance to faces in each bin, and select the minimum distance (\(d_{min}\)) to opposing face candidates. Opposing face candidates (with a face normal, \(n_{of}\)), is defined such that \(n_{of} \cdot n_e < 0.0\).

4. The length scale at the edge location is now defined as \(\xi d_{min}\), where \(\xi\) is a multiplicative constant (chosen to be 0.2 in this work).
Length scale dictated by time-step restrictions, $(bL_d)$: To avoid problems associated with excessive numerical stiffness and stability, the overall minimum mesh length scale may have to be limited to a certain value, such as the one defined in Eq.(1.7). The relation is repeated here for convenience:

$$\tau_{surf} \approx \frac{\mu R}{\sigma} = \frac{\mu}{\sigma} \left( \frac{\lambda^2}{4\pi^2 a} \right)$$  \hspace{1cm} (4.9)

If none of the criteria described have been imposed, the existing surface mesh length-scale is used. When these criteria are used in combinations, the surface length scale can be allowed to vary spatially, resulting in an adaptive scheme that focuses the mesh density in areas that require it most.

Sections 4.2.3 and 4.2.4 will describe two important operations required for h-refinement in simplical 2D and 3D simplical meshes - Edge Bisection and Edge Contraction. Section 4.2.5 will describe an additional refinement feature specific to 3D tetrahedral meshes - Face Trisection.

### 4.2.3 Edge Bisection

Edge bisection is a basic topology modifying operation which splits an edge in the mesh at its center, resulting in the addition of a single point. Consequently, all cells connected to the original edge must also be split, resulting in additional faces, edges and cells. The bisection operation applied to 2D (applicable to triangular prism meshes as well) is shown in Fig. 4.8(a), and the 3D equivalent applied to tetrahedra is shown in Fig. 4.8(b). In both cases, the edge $ab$ is split at the center to introduce a new point $c$.

Edge-bisection is particularly useful for the refinement of interface edges, which is based on any of the criteria specified in Section 4.2.2. This ensures that the mesh is adequately refined in regions involving either high surface curvature or interface proximity. Naïvely attempting to achieve increased mesh refinement by successively
Figure 4.8. Edge bisection in (a) 2D; and (b) 3D

bisecting edges in a certain region is usually ineffective, since it causes the mesh quality to rapidly deteriorate (Baker and Cavallo [9]). To avoid this situation, an additional check is performed to ensure that the cells resulting from the bisection operation possesses a quality metric value above a certain threshold. If an edge fails this check and additional refinement is necessary, the bisection is delayed till the next time-step, where smoothing and swapping operations would have been applied to improve the current configuration.

4.2.4 Edge Contraction

Edge contraction (or edge collapse) is complementary to the bisection operation, where one of the two end points of an edge (and the edge itself) is deleted from the mesh. This operation would involve the deletion of all faces and cells connected to the edge, and several edges connected to the deleted point have to be either renumbered
or replaced. The edge contraction operation for 2D and 3D is depicted in Fig. 4.9. In both cases, point $d$ is deleted and edge $cd$ is collapsed to point $c$.

![Figure 4.9. Edge contraction in (a) 2D; and (b) 3D](image)

In the 3D case, cells $(abcd)$ and $(bcde)$ are removed from the mesh, edges $ad$ and $de$ are replaced by edges $ac$ and $ce$, and edge $fd$ must now be renumbered to $fc$. Edge contraction is slightly complicated because the operation might yield invalid cells under certain configurations. Dey et al. [45] present the various conditions under which edge contraction can be applied to the mesh, based on various principles of algebraic topology. However, the data-structures required to represent (and therefore check) these conditions cannot be incorporated easily into the current paradigm and so, a simpler alternative is used. The alternative check involves looping through all edges connected to the point slated for removal, and checks the volume of all cells connected to these edges when the replacement point is substituted. In Fig. 4.9(b) for instance, cells $(abdf)$ and $(bdef)$ are checked for the case when point $d$ is replaced.
by point $c$ (note that the cells to be removed, cells $(abcd)$ and $(bcde)$, are not included in the check). If these cells are inverted or degenerate after the substitution, then the operation cannot be performed.

The choice of a particular point of the edge for removal is based on the constraints imposed on it by the mesh topology. If a particular point lies on a boundary surface, that point should be preferentially retained. However, when both points of an edge (and the edge itself) lie on a boundary surface, the contraction operation is actually valid and must therefore be considered. In situations where no constraints are present, a mid-point collapse is performed, i.e., point $d$ is deleted and point $c$ is moved to the mid-point of edge $cd$.

Having defined the elementary refinement operations in 2D and 3D, the top-level routine `EdgeBisectCollapse` is introduced. The functions `BisectEdge` and `CollapseEdge` take an input edge index and perform their respective refinement actions, provided the resulting cell quality does not fall below a user-specified threshold.

```
EdgeBisectCollapse()
Input : Stack of edges (M)
while M is not empty do
    e = pop(M)
    if CheckEdgeBisection(e) then
        BisectEdge(e)
    else if CheckEdgeCollapse(e) then
        CollapseEdge(e)
Function EdgeBisectCollapse
```

The `CheckEdgeBisection` and `CheckEdgeCollapse` functions merely average the length scale values of all cells touching the edge. If a particular edge lies on the boundary, the boundary length scale specified in Section 4.2.2 is used instead.
4.2.5 Face Trisection

Another option for mesh refinement in three dimensional meshes is the face trisection operation, which introduces a new point at the centroid of a triangular face. This splits the face into three smaller faces, and each cell on either side of the original face is split into three smaller ones. The face trisection operation is depicted in Fig. 4.10, where the face \((abc)\) is split at the centroid point \(d\).

![Figure 4.10. Face trisection in 3D](image)

Recursive trisection can also degrade the quality of a mesh, and checks are performed to ensure that the quality does not fall below a specified threshold.

4.3 Sliver Detection and Removal

In certain situations, the swapping process can sometimes fail to remove poor quality tetrahedra because a quality-improving triangulation could not be found. This case often occurs when boundaries (such as a free-surface, for instance) are deforming too quickly for the mesh-motion algorithm to adapt and improve surface cells, or when all four points of a tetrahedral cell lie on the boundary. These cells are usually small in number (less than 5 to 10 cells in a large mesh), and occur very infrequently. Degenerate, inverted or tangled tetrahedra, no matter how few, are highly detrimental to solution accuracy and can almost certainly halt a simulation;
and so they must often be eliminated by some special means. This section will describe a fairly general sliver detection and removal algorithm, adapted from work by Li, Shephard and Beall [105].

Figure 4.11. Poor quality tetrahedra

Four general cases of poor quality tetrahedra, as described by Freitag and Knupp [62], are shown in Fig. 4.11. All four cases involve cells with edges of acceptable length, but their height (and therefore, volume) is very small. Most metrics (like those described in Section 4.1.1) can identify these cells as bad quality elements, but effective removal of these cells from the mesh requires them to be classified under one the four types shown in Fig. 4.11. The algorithm for sliver type detection, given by the routine IDENTIFYTETSLIVERTYPE in Algorithm 14, requires the list of four points that describe the tetrahedron (c{4}).

The first part of the algorithm identifies the apex point (point ‘a’ in Fig. 4.12) that possesses the minimum perpendicular distance to the base triangular face (‘b’, given by face (p_0 p_1 p_2) in Fig. 4.12). The function PERMUTE(c, pI) returns an ordered set of three points from p (describing a face) which does not contain the point pI. Given this information, the position vectors r_1,...,6 are defined as shown in Fig. 4.12(b), which is then used to obtain the signed triangle areas (t_1, t_2, and t_3) shown in Fig. 4.12(a).

From Fig. 4.12, it becomes apparent that a wedge (Type IV) is a special case of the cap cell (Type II), i.e., when the projection of the apex point on to the base is too close to any of the points which describe the face. Likewise, the spade (Type III)
is a special type of sliver cell (Type I), which occurs when the projection of the apex point is too close to one the base edges. The scalar $\epsilon$ is a parameter which defines a threshold for a cap cell to be classified as a wedge, or a sliver cell to be classified as a spade. Based on the values (and magnitudes) of these signed areas, the type of sliver is determined by the \texttt{TriAreaSliverType} function. This algorithm can correctly classify any bad quality cell, regardless of the spatial orientation of the tetrahedron.

After having identified the sliver type from Algorithm 14, the four general types of poor quality tetrahedra can be eliminated from the mesh using a sequence of the three refinement operators defined in Section 4.2. These elimination methods are described in the following subsections.

### 4.3.1 Elimination of Type I Cells

The characteristic features of the Type I (sliver) tetrahedron are two opposite edges which are located very close to each other, making the cell nearly flat in appearance. When this type of cell is detected by Algorithm 14, the two closely spaced edges are first identified (edges $ab$ and $cd$ in Fig. 4.13(a)). To eliminate this cell from the mesh, edge $ab$ is first bisected to introduce point $e$ (Fig. 4.13(b)), and a temporary interior face ($cde$). Following this, edge $cd$ is bisected to introduce point $f$.
IdentifyTetSliverType(c)

Output : Sliver type: (I) Sliver; (II) Cap; (III) Spade; or (IV) Wedge

\[ \text{minDistance} = \infty \]

\[
\text{forAll Points (pI) in c do}
\]

\[
\text{testFace} = \text{Permute(c, pI)}
\]

\[
\text{n}_t = \text{NORM}(\text{testFace})
\]

\[
p = pI - \text{testFace}[0]
\]

\[
q = p - (p \cdot \text{n}_t)\text{n}_t
\]

\[
distance = |p - q|
\]

\[
\text{if distance} < \text{minDistance} \text{ then}
\]

\[
a = pI
\]

\[
b = \text{testFace}
\]

\[
\text{minDistance} = \text{distance}
\]

\[
n = \text{NORM}(b)
\]

\[
r_1 = b[1] - b[0]
\]

\[
\]

\[
r_3 = b[0] - b[1]
\]

\[
r_4 = a - b[0]
\]

\[
r_5 = r_4 - r_1
\]

\[
r_6 = r_5 - r_2
\]

\[
t_1 = n \cdot (0.5(r_1 \times r_4))
\]

\[
t_2 = n \cdot (0.5(r_2 \times r_5))
\]

\[
t_3 = n \cdot (0.5(r_3 \times r_6))
\]

\[
\epsilon = 0.1(\text{AREA}(b))
\]

\[
\text{return TriAreaSliverType(t_1, t_2, t_3, \epsilon)}
\]

Function IdentifyTetSliverType

(Fig. 4.13(c)), and two temporary faces \((aef)\) and \((bef)\). These two operations result in the creation of the new edge \(ef\), and splits the sliver cell \((abcd)\) into four temporary cells. These bisection operations also split adjacent cells in the mesh, but they are not shown in Fig. 4.13 for clarity. Finally, edge \(ef\) is collapsed and the sliver cell is removed.

![Figure 4.13. Removing the Type I cell](image)
TriAreaSliverType\((t_1, t_2, t_3, \epsilon)\)

\[\text{Output} \quad \text{type: (I) Sliver; (II) Cap; (III) Spade; or (IV) Wedge}\]

if \(t_1 > 0.0 \text{ and } t_2 > 0.0 \text{ and } t_3 > 0.0\) then \(\text{type} = \text{II}\)
if \(t_1 < 0.0 \text{ and } t_2 > 0.0 \text{ and } t_3 > 0.0\) then \(\text{type} = \text{I}\)
if \(t_1 > 0.0 \text{ and } t_2 < 0.0 \text{ and } t_3 > 0.0\) then \(\text{type} = \text{I}\)
if \(t_1 > 0.0 \text{ and } t_2 > 0.0 \text{ and } t_3 < 0.0\) then \(\text{type} = \text{I}\)
if \(t_1 < 0.0 \text{ and } t_2 > 0.0 \text{ and } t_3 < 0.0\) then \(\text{type} = \text{II}\)
if \(t_1 < 0.0 \text{ and } t_2 < 0.0 \text{ and } t_3 > 0.0\) then \(\text{type} = \text{II}\)
if \(t_1 > 0.0 \text{ and } t_2 < 0.0 \text{ and } t_3 < 0.0\) then \(\text{type} = \text{II}\)
if \(t_1 < 0.0 \text{ and } t_2 < 0.0 \text{ and } t_3 < 0.0\) then \(\text{type} = \text{II}\)
if \(|t_1| < \epsilon\) then
  if \(|t_3| < \epsilon\) then
    \(\text{type} = \text{IV}\)
  else if \(|t_2| < \epsilon\) then
    \(\text{type} = \text{IV}\)
  else if \(|t_2| > \epsilon \text{ and } |t_3| > \epsilon\) then
    \(\text{type} = \text{III}\)
if \(|t_2| < \epsilon\) then
  if \(|t_3| < \epsilon\) then
    \(\text{type} = \text{IV}\)
  else if \(|t_1| > \epsilon \text{ and } |t_3| > \epsilon\) then
    \(\text{type} = \text{III}\)
if \(|t_3| < \epsilon\) then
  if \(|t_1| > \epsilon \text{ and } |t_2| > \epsilon\) then
    \(\text{type} = \text{III}\)
return \(\text{type}\)

**Function** TriAreaSliverType

### 4.3.2 Elimination of Type II Cells

Cap cells can be identified by the situation where a particular point of the tetrahedron is located too close to its opposite face, and unlike the Type I cell, the edges are evenly spaced. This type of cell is eliminated by first trisecting the opposite face (face \((bcd)\) in Fig. 4.14(a)), thus creating an intermediate edge \(ae\), and three temporary cells. Other cells and edges that are created during the trisection process in adjoining cells are not shown for clarity. The cap cell is finally removed by collapsing the edge \(ae\), and removing the three temporary cells in the process.

### 4.3.3 Elimination of Type III Cells

The spade cell is a special case of the Type I sliver cell, where the projection of the apex point \((a)\) on to the opposite face \((bcd)\) is too close to one of the edges. Once
the apex point and the corresponding face is identified, this cell is removed by a two step process, involving one bisection and one collapse operation of the intermediate edge $ae$, as shown in Fig. 4.15.

4.3.4 Elimination of Type IV Cells

Wedge cells are eliminated by simply collapsing the short edge of the tetrahedron. In Fig. 4.16, this would be edge $ad$. 

---

Figure 4.14. Removing the Type II cell

Figure 4.15. Removing the Type III cell

Figure 4.16. Removing the Type IV cell
4.4 Variable Remapping after Mesh Reconnection

Any mesh reconnection process requires flow variables to be transferred from the original mesh to the modified one. A method is presented here for conservatively transferring, or remapping, cell-centered variable fields from one mesh to another with second-order accuracy. The method is generally applicable to any polyhedral source or target mesh. Like the work of Farrell et al. [57], which was designed for finite-element computations, the proposed methodology uses a logical supermesh consisting of the intersections of polyhedra from both meshes. The resulting transfer process is well-suited for finite-volume methods that rely on cell-centered variables. The accuracy and efficacy of the new remapping process is demonstrated with numerical experiments and a computational fluid dynamics test.

The most general case for the applicability of this method is a situation where the computational mesh that deforms gradually, by simply moving node locations, until the degradation in mesh quality necessitates transferring the computation to a newly created mesh which shares the same exterior boundary, but bears no resemblance in terms of element connectivity. As an example, this is common practice in simulations of internal-combustion engines, where it is referred to as a key-grid or target-mesh approach [24], and requires a global transfer of solution variables prior to a solution restart.

Alternatively, a mesh can undergo local changes in topology, requiring transfer to a new mesh that is very similar to the old one, which is the approach taken in this work. In contrast to the key-grid methodology, the local approach changes mesh connectivity frequently, but in incremental stages, so that only a few new cells are created or destroyed per time-step. Similar transfer processes can occur in steady-state scenarios as well [119]. If a converged solution is available on a given mesh, field-transfers can also be used to accelerate initial convergence on a new mesh which shares the same domain, but possesses different resolution or grid-connectivity.
4.4.1 Background

Field-transfers are commonplace in Arbitrary Lagrangian-Eulerian (ALE) settings. The ALE method usually involves a Lagrangian step, where vertices are repositioned continuously until certain elements become extremely distorted, followed by a remeshing (or rezoning) step, at which point a field remap becomes necessary. A choice for the frequency of the rezoning step is usually available, but the approach preferred by most is continuous rezoning, mainly for reasons of simplicity and efficiency of the algorithms involved. Examples of work in this area include those of Margolin and Shashkov [109], Loubère et al. [108], Berndt et al. [14] and Kucharik et al. [101]. The remapping scheme by Berndt et al. uses a hybrid swept-volume/intersection approach for multi-material flows in two-dimensions. Reconnection-based remapping for two-dimensional Voronoi meshes was considered by Loubère et al. [107]. Global remapping involving hexahedral grids was also considered by Dukowicz and Padial [48], and for two-dimensional simplical grids by Alauzet et al. [4]. Extensions involving three-dimensional polyhedra are given in work by Grandy [70], and Garimella et al. [66]. The work of Garimella et al. provides an accurate and fast remapping algorithm for situations where the change in the mesh occurs in small increments, while the work of Grandy is only first-order accurate.

A related problem occurs in lower dimensions with non-conforming surface meshes, particularly in geophysical fluid flows. This problem was investigated by Dukowicz [46, 47] and Jones [91] for quadrilateral and unstructured meshes described in spherical coordinates, respectively. Prior work such as Azarenok [8] looked specifically at how to calculate intersections for hexahedral meshes. An early application of the supermesh concept is considered for non-conformal surface meshes in work by Jiao [90].

Ideally, the transfer of fields from one mesh to another mesh will be both accurate and conservative. Simple interpolation methods can be accurate, but
usually do not maintain conservation, causing undesirable additional perturbations to the solution beyond what truncation error would indicate. Farrell et al. [57] recognized this challenge and presented a vertex-based solution using a bounded Galerkin projection approach, that is well suited for finite-element methods. Their second-order methodology was shown to be conservative, and its efficacy was proven in numerical tests. The work presented here is an extension of the supermesh concept by Farrell et al., accompanied by tests for cell-centered variables.

Using different methods, a procedure is presented that is second-order accurate, conservative, and suited for collocated variable, finite volume computations. Unlike rezoning approaches [66], there is no assumed similarity in the original mesh or the target mesh. The present work is independent of mesh topology and can be applied to transfer of fields on any general polyhedral mesh. This greater generality is achieved at the cost of calculating mesh volume intersections.

4.4.2 Definitions

For clarity, most definitions of the supermesh are kept identical to those introduced by Farrell et al. [57], and repeated here for convenience. We first consider two arbitrarily unstructured polyhedral meshes $\mathcal{T}_A$ and $\mathcal{T}_B$ of the same domain $\Omega \subset \mathbb{R}^d$, with nodes $N_A, N_B$, and edges $\mathcal{D}_A, \mathcal{D}_B$. $K \in \mathcal{T}$ is used to denote an element (or cell) in a mesh.

In the finite-volume framework, cells of each mesh are described as a collection of $(d - 1)$-dimensional facets (edges in 2 dimensions, and faces in 3 dimensions). So, an additional set for faces $\mathcal{F}_A, \mathcal{F}_B$ is introduced in 3D, where $F \in \mathcal{T}$ is used to denote a polygonal face in a mesh. It follows naturally that any interior face in a mesh is connected to exactly two cells, while faces on the boundary of a mesh are connected to only one. In 3D, the situation is more complicated because faces of a polyhedral cell can also be non-planar. This warrants the decomposition of
the cell into tetrahedra using Steiner points, as shown in Fig. 4.17. This approach inherently assumes that such a decomposition exists, which may not necessarily be true in situations involving highly non-convex polyhedra. The choice of Steiner points is also somewhat arbitrary, and in this work, they are chosen to coincide with cell and face centroids ($x_{Ka}$ and $x_f$, respectively). The process of polyhedral decomposition is fairly simple and efficient, only requiring a counter-clockwise walk through points of the polyhedral face, connecting the $i^{th}$ and $i+1^{th}$ point with $x_{Ka}$ and $x_f$ to form a tetrahedron, with the cost being linear in the number of faces visited.

![Figure 4.17. Face decomposition with Steiner points](image)

One such example is shown in Fig. 4.18. In this case, Fig. 4.18(a) is an arbitrary polyhedral cell with 14 non-planar faces, while Fig. 4.18(b) shows the decomposition of this cell into 72 tetrahedra. Finally, Fig. 4.18(c) shows the intersections of the cell against a background tetrahedral mesh, colored by source-cell index. A possible alternative involves the triangulation of faces (without face Steiner points), followed by subtending these triangles to cell-centroids to obtain tetrahedra. This type of decomposition yields fewer tetrahedra, which alleviates the computational cost for intersection calculations, but may not represent non-planar boundaries with sufficient accuracy.
The computational point used in this work is located at the centroid \( \mathbf{x}_K \) of each cell (with volume \( V_K \)) given by Eq. 3.2, and repeated here for convenience:

\[
\int_{V_K} \left( \mathbf{x} - \mathbf{x}_K \right) \, dV = 0 \tag{4.10}
\]

For arbitrary polyhedral cells, the centroid \( \mathbf{x}_K \) can be calculated from the decomposition itself, using Eq.(4.11).

\[
\mathbf{x}_K = \frac{1}{V_K} \sum_i V_i \mathbf{x}_i \tag{4.11}
\]

where \( V_i, \mathbf{x}_i \) represent the volume and centroid of the \( i^{th} \) tetrahedron (respectively), and \( V_K = \sum_i V_i \).

The accuracy of the discretization method depends on the assumed variation of the underlying tensorial function \( \phi(\mathbf{x}) \) in space. For a method to be second-order accurate in space, it must be capable of preserving a linearly-varying function. Using a Taylor series expansion around a point \( \mathbf{x}_K \), such that \( \phi_K = \phi(\mathbf{x}_K) \) (from Eq. 3.12, repeated here):

\[
\phi(\mathbf{x}) = \phi_K + (\mathbf{x} - \mathbf{x}_K) \cdot (\nabla \phi)_K + \mathcal{O}(\mathbf{x} - \mathbf{x}_K)^2 \tag{4.12}
\]
Truncating the second-order terms in the series and integrating Eq.(4.12) over a control volume $V_K$ around point $K$ to obtain a cell-averaged quantity $\bar{\phi}$, it follows that:

\[
\bar{\phi} = \frac{1}{V_K} \int_{V_K} \phi \; dV \tag{4.13a}
\]

\[
= \frac{1}{V_K} \int_{V_K} [\phi_K + (x - x_K) \cdot (\nabla \phi)_K] \; dV \tag{4.13b}
\]

\[
= \frac{\phi_K}{V_K} \int_{V_K} dV + \frac{1}{V_K} \left[ \left( \int_{V_K} (x - x_P) \; dV \right) \cdot (\nabla \phi)_K \right] \tag{4.13c}
\]

\[
= \phi_K \tag{4.13d}
\]

The second integral in equation Eq.(4.13c) is identically zero due to the assumption in equation Eq.(4.10). Thus, $\phi_K$ represents the average value of the linearly varying variable $\phi$ in cell $K$.

The supermesh $\mathcal{T}_C$ of $\{\mathcal{T}_A, \mathcal{T}_B\}$ of $\Omega$ can now be defined, such that:

- $\mathcal{N}_C \supseteq \mathcal{N}_A \cup \mathcal{N}_B$

- $V(K_c \cap K) \in \{0, V(K_c)\} \forall K_c \in \mathcal{T}_C, K \in \mathcal{T}, \mathcal{T} \in \{\mathcal{T}_A, \mathcal{T}_B\}$

where $V(.)$ is the volume function, such that $V_K = V(K)$.

The first item above states that nodes in the supermesh must contain nodes from both meshes $\mathcal{T}_A$ and $\mathcal{T}_B$. The second item states that the intersection of every cell in $\mathcal{T}_C$ with cells in $\mathcal{T}_A$ and $\mathcal{T}_B$ must be either zero or the volume of $K_c$.

An example of this concept is depicted in Fig. 4.19, where the supermesh in Fig. 4.19(c) is obtained by the intersection of cells in meshes Fig. 4.19(a) and Fig. 4.19(b). It is clear that the mesh in Fig. 4.19(b) was obtained by removing the edge $ab$ (and therefore, vertex $b$) and by adding vertex $e$ at the mid-point of edge $cd$. 82
While the supermesh concept applies to arbitrary mesh-pairs which bear no resemblance to each other, one can also apply the technique in a more localized sense. This approach is particularly attractive for a few reasons. The primary advantage is that it allows the use of localized re-meshing algorithms in transient simulations, where a mesh is typically inspected for bad elements at each time-step, and a local set of cells around bad elements are agglomerated and remeshed to improve mesh-quality. Because the number of bad quality cells at each time-step are very few (typically less than 5% of the total), this results in an efficient algorithm, and also restricts interpolation errors to areas undergoing local re-meshing operations. This also has the added benefit of having a substantially reduced number of intersection calculations required to define the superset cells, when compared to a global re-meshing approach.

Intersection calculations are also particularly vulnerable to floating-point round-off errors, especially for degenerate cases where vertices are nearly coincident, or lie exactly on edges / faces in the mesh. The local nature of mesh-topology modifications...
allows algorithms to re-use existing vertices and therefore, circumvent a vast majority of such degenerate cases, thereby leading to very robust run-time performance.

4.4.3 Intersection algorithm

The first step in the intersection process is the identification of a candidate parent cell from the source mesh. In the local reconnection case, each operation involves only a local set of cells, which makes the identification process trivial. For the global remapping case, however, this process is more involved. In this work, the candidate parent cell is identified by using an octree algorithm [137], although other efficient methods are clearly possible. Once a candidate parent cell has been identified, it is tested for intersection using the advancing-front algorithm suggested by Farrell and Maddison [56].

An example of this approach is shown in Fig. 4.20, where the target cell is depicted (in red) against a background source mesh (in black), and the candidate parent cell is shaded (in blue). The candidate cell is first tested for intersection, and if it does not intersect, a rescue mechanism is set up by agglomerating several layers of cells around the candidate and testing each one in-turn until a suitable intersection is found. Once the initial intersection is found, the neighbours of the intersecting cell are tested using a face-cell walk, as shown in Fig. 4.20(b). This procedure is repeated for all untested neighbours, until no new intersections are found. The final list of source cells is shown in Fig. 4.20(e).

Since polyhedral meshes are decomposed prior to interpolation, the task of computing the volume and centroid of intersections simplifies to algorithms specific to triangles and tetrahedra. The process of finding intersections is clearly not unique, and a variety of algorithms can deal with this aspect in a very efficient manner. Since the algorithm only involves triangles and tetrahedra, any intersection is guaranteed to be a convex hull. Prior efforts in this area include work by Ahn and Shashkov [3],
Eberly [50] and Chazelle [29]. Owing to simplicity in implementation, run-time efficiency and numerical robustness, the half-space intersection algorithm described by Eberly [50] is used in this work. The algorithm is schematically represented for 2D meshes in Fig. 4.21, where edges of the source cell (in black) are repeatedly clipped by edges of the target cell (in red), with the clipping edge (or half-space) depicted in green. At each step, the clipping process sub-divides each simplex of the existing set into a list of simplices depending on the location of vertices with respect to the half-space. This procedure is fairly simple because the sub-division can be classified into a specific set of cases (3 in 2D and 6 in 3D). The final set of simplices is shown in Fig. 4.21(e).
In short, the overall algorithm for polyhedral source/target meshes can be summarized as follows:

- If source mesh $\mathcal{T}_A$ is polyhedral, decompose into tetrahedral mesh $\mathcal{T}_{At}$
- If target mesh $\mathcal{T}_B$ is polyhedral, decompose into tetrahedral mesh $\mathcal{T}_{Bt}$
- Create source map $\mathcal{X}_{At} : \mathcal{T}_{At} \to \mathcal{T}_A$, taking a tetrahedral cell in $\mathcal{T}_{At}$ to its parent polyhedral cell in $\mathcal{T}_A$
- Create target map $\mathcal{X}_{Bt} : \mathcal{T}_B \to \mathcal{T}_{Bt}$, taking a parent polyhedral cell in $\mathcal{T}_B$ to its decomposition set in $\mathcal{T}_{Bt}$
- Agglomeration:
- Fetch the set of tetrahedral cells $K_{bl} = \mathcal{I}_{Bl}(\mathcal{I}_{Bl}(K_B))$ for each $K_B \in \mathcal{T}_B$

- For each tetrahedral cell in $K_{bl}$, if $V(K_{bl} \cap K_{At}^j) > 0$, add $\mathcal{I}_{At}(K_{At}^j)$ to the list of parent source cells, and accumulate intersection volume / centroid. $K_{At}^j$ is initially identified by octree, and subsequently by face-cell walk.

If the $\mathcal{T}_A$ or $\mathcal{T}_B$ is tetrahedral, the corresponding source / target $\mathcal{X}$ simplifies to an identity map.

### 4.4.4 Interpolation

In this section, the problem is viewed as an interpolation approach between a source mesh $\mathcal{T}_A$ and a target mesh $\mathcal{T}_B$. The logical supermesh $\mathcal{T}_C$ serves as an intermediate during the process. Depending on the application, the source mesh and target mesh may or may not have regions that are identical.

Each cell $K_c$ in the supermesh has a distinct centroid $x_{K_c}$, which is given by Eq.(4.14):

$$\int_{V_{K_c}} (x - x_{K_c}) \, dV = 0$$  \hspace{1cm} \text{(4.14)}$$

Let $K_a$ be an element on the source mesh $\mathcal{T}_A$ containing $K_c$. Using the definition of the second-order Taylor series given in Eq.(3.12), the value of any variable $\phi$ at the location $x_{K_c}$ is given in terms of the value of the variable at a location $K_a \in \mathcal{T}_A$. This is identical to the approach taken by Alauzet et al. [4], given by Eq.(4.15):

$$\phi(x_{K_c}) = \phi_{K_a} + (x_{K_c} - x_{K_a}) \cdot (\nabla \phi)_{K_a}$$  \hspace{1cm} \text{(4.15)}$$

Certain centroids on the supermesh may occasionally coincide with cell centroids on $\mathcal{T}_A$, due to the localized nature of the re-meshing algorithm. In such cases, the second term in Eq.(4.15) drops out and $\phi(x_{K_c})$ naturally reduces to the original value $\phi(x_{K_c})$, thereby denoting a direct map. Dropping the gradient term altogether reduces the remapping to first-order, as pointed out by Jones [91].

87
The process of estimating the gradient term in Eq.(4.15) is not uniquely defined. Typical estimates of the gradient (like a weighted least-squares approach, for example) on unstructured meshes are known to be second-order accurate and are thus sufficient to preserve the linearity of the underlying field. A more critical requirement is the boundedness of the gradient estimate, and schemes which are strongly monotone can guarantee that $\phi(x_{K_c})$ lies in between values of $\phi$ for cell $K_a \in \mathcal{T}_A$ and its immediate neighbours. A discussion on gradient boundedness is given in work by Alauzet et al. [4] and a treatment for discontinuous fields is given by Farrell et al. [56] and Garimella et al. [66].

Assume that any given cell $K_a \in \mathcal{T}_A$ consists of $m$ supermesh cells from $\mathcal{T}_C$, such that:

$$V_{K_a} = \sum_{i=1}^{m} V_{K_c}^i$$ (4.16)

$$x_{K_a} = \frac{1}{V_{K_a}} \left[ \sum_{i=1}^{m} x_{K_c}^i V_{K_c}^i \right]$$ (4.17)

Strong conservation requires that the volume-weighted sum of $\phi$ for $m$ cells $K_c \in \mathcal{T}_C$ be equal to $\phi$ for $K_a \in \mathcal{T}_A$. This is proven in Eq.(4.18d) by using the fact that the volume-weighted sum of supermesh cell-centroids is identical to cell-centroids in the source mesh $\mathcal{T}_A$ (given by Eq.(4.17)), and that the product of the average cell-value with cell-volume is identical to the integral over the cell (given by Eq.(4.13d)).

$$\bar{\phi}_{K_a} \ V_{K_a} = \sum_{i=1}^{m} \phi(x_{K_c}^i) V_{K_c}^i$$ (4.18a)

$$= \sum_{i=1}^{m} \left[ \phi_{K_a} + (x_{K_c}^i - x_{K_a}) \cdot (\nabla \phi)_{K_a} \right] V_{K_c}^i$$ (4.18b)

$$= \phi_{K_a} \sum_{i=1}^{m} V_{K_c}^i + \left[ \sum_{i=1}^{m} x_{K_c}^i V_{K_c}^i - x_{K_a} \sum_{i=1}^{m} V_{K_c}^i \right] \cdot (\nabla \phi)_{K_a}$$ (4.18c)

$$= \phi \ V_{K_a}$$ (4.18d)
Conservation is therefore independent of the gradient estimation method, and although Eq. (4.15) is only an approximation of the Taylor series, the conservation given by Eq. (4.18) is exact.

The final step in the remapping process is to agglomerate the interpolants from the supermesh to cells in the target mesh $\mathcal{T}_B$. This is a simple volume-weighted sum, which does not affect conservation. Thus, assuming that any given cell $K_b \in \mathcal{T}_B$ consists of $n$ supermesh cells from $\mathcal{T}_C$, such that:

$$V_{K_b} = \sum_{i=1}^{n} V_{K_c}^i$$

(4.19)

Then,

$$\phi(x_{K_b}) = \frac{1}{V_{K_b}} \sum_{i=1}^{n} \phi(x_{K_c}) \frac{V_{K_c}^i}{V_{K_b}}$$

(4.20)

The agglomeration also does not affect the linearity-preserving nature of the scheme. If the underlying field $\phi$ is linearly varying, then the values at $\phi(x_{K_c})$ are exact, and consequently, so will the values at $K_b \in \mathcal{T}_B$. This can be easily verified by revisiting Eq. (4.18b), and relating supermesh cells to cells in the target mesh, $K_b \in \mathcal{T}_B$. For polyhedra, the agglomeration process involves the superset cells arising from the intersections of all decomposed tetrahedra.

### 4.4.5 Numerical Results

A logical first step is to test the remapping in the context of field-interpolation between two arbitrary meshes that share the same domain. To demonstrate this, an underlying function is defined on an initial mesh, and then remapped to a target mesh of similar mesh density, but bearing no resemblance in terms of vertex-positions and element connectivity.
For comparison, a more conventional remapping based on inverse-distance weighting that is given by Eq.(4.21) is also considered:

\[
\phi(x_{K_b}) = \frac{1}{w_t} \sum_{i=1}^{n} w_i^2 \phi(x_{K_a}^i) \tag{4.21}
\]

\[
w_i = \frac{1}{|x_{K_a}^i - x_{K_b}|}
\]

\[
w_t = \sum_{i=1}^{n} w_i^2
\]

where \(\phi(x_{K_b})\) is the field value at the target cell, and \(\phi(x_{K_a}^i)\) is the field value associated with the \(i^{th}\) cell in the source-mesh. For consistency, the set of cells used for weighting factors \(w_i\) are identical to those used in the conservative method. While the inverse-distance method is not conservative, it serves as a useful reference to gauge the accuracy of remapping.

### 4.4.6 Linear function

The first test function is a linear scalar field, given by Eq.(4.22):

\[
\phi(x, y, z) = 2x + 3y + z \tag{4.22}
\]

Numerical tests confirm that the remapping is exact (to machine round-off) for the linear field.

### 4.4.7 Cosine hill function

The second field is a cosine hill function, identical to the one used by Jones [91], given by Eq.(4.23), where \(r\) is the distance from the centre of the hill and \(L\) is a length.
**Figure 4.22.** Test mesh colored by function value: \( \phi(x, y, z) = 2x + 3y + z \)

scale. This function is very convenient while demonstrating the effects of repeated remapping between mesh pairs.

\[
\phi(r) = 2 + \cos(\pi r / L) \tag{4.23}
\]

**Figure 4.23.** Meshes used for repeated remapping tests

The test function was repeatedly remapped 250 times between two arbitrary planar meshes, with \( n_{Ca} = 3592 \) and \( n_{Cb} = 1893 \), shown in Fig. 4.23, where the
height and color represents the magnitude of $\phi$. The first-order conservative method is obtained by dropping the gradient term in Eq. (4.15), which reduces to a volume-weighted remap. The second-order weights are therefore obtained by including a volume-weighted distance from the source-cell centroid. The results in Fig. 4.24 show the diffusive nature of the inverse-distance and first-order conservative schemes, while the second-order conservative scheme preserves the shape and magnitude of the function. In practice, however, this form of cyclic mapping is almost never performed, and the test only serves the purpose of demonstrating the accuracy of the scheme.

![Figure 4.24. Results of repeated remapping](image)

Results for the global integral of $\phi$ after repeated remapping between two meshes ($n_{Ca} = 14360$ and $n_{Cb} = 7405$) for 250 cycles using the cosine hill field is shown in Table. 4.1. The value of the global integral on the source mesh is
0.05327452406622132. As expected, the inverse-distance approach lacks the property of conservation, while both first- and second-order methods reproduce the integral to machine accuracy.

<table>
<thead>
<tr>
<th>Method</th>
<th>Global integral of $\phi$</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inverse-distance</td>
<td>0.05427098440184993</td>
<td>0.0009964603356286128</td>
</tr>
<tr>
<td>First-order conservative</td>
<td>0.05327452406622409</td>
<td>2.7686186667658984e-15</td>
</tr>
<tr>
<td>Second-order conservative</td>
<td>0.05327452406622302</td>
<td>1.700029006457271e-15</td>
</tr>
</tbody>
</table>

Table 4.1. Global integral of $\phi$ using various methods

### 4.4.8 Sinusoidal function on simplical 3D meshes

The third function is a smooth sinusoidal field, identical to the one used by Garimella et al. [66], given by Eq. (4.24).

$$\phi(x, y, z) = 1 + \sin(2\pi x) \sin(2\pi y) \sin(2\pi z)$$  \hspace{1cm} (4.24)

A typical mesh used in the test is given in Fig. 4.25.

![Figure 4.25. Typical mesh used for the sinusoidal function](image.png)
The mesh spacing \((dx)\) is given by Eq.\((4.25)\)

\[
dx = \left[ \frac{V_\Omega}{n_{Cb}} \right]^{1/n_d}
\]

(4.25)

where, \(V_\Omega\) is the domain volume (selected to be a unit cube), \(n_{Cb}\) is the number of cells in the target-mesh, and \(n_d\) is the dimensionality of the mesh. Table 4.2 provides details of the source and target meshes.

<table>
<thead>
<tr>
<th>(n_{Ca})</th>
<th>(n_{Cb})</th>
<th>(dx)</th>
</tr>
</thead>
<tbody>
<tr>
<td>6972</td>
<td>7442</td>
<td>0.0512</td>
</tr>
<tr>
<td>36230</td>
<td>35693</td>
<td>0.0304</td>
</tr>
<tr>
<td>62732</td>
<td>61669</td>
<td>0.0253</td>
</tr>
<tr>
<td>97646</td>
<td>101226</td>
<td>0.0215</td>
</tr>
<tr>
<td>809371</td>
<td>801841</td>
<td>0.0108</td>
</tr>
</tbody>
</table>

Table 4.2. Details of meshes used for accuracy tests

The results of the remapping process are presented in Fig. 4.26. The inverse-distance approach is slightly better than first-order accurate. For the test, the gradient term in Eq.\((4.15)\) was evaluated numerically using a weighted least-squares approach.

4.4.9 Run-time efficiency on polyhedral meshes

An important consideration for most algorithms, particularly those involving global remapping in three dimensions, is run-time efficiency. Since this work considers the general case involving polyhedra, the efficiency of the method is clearly dictated by the choice of intersection algorithm. Intersection algorithms also have the benefit of being trivially parallel in a shared-memory paradigm using threads on a multi-core computer. The timings reported in the section were obtained using an Intel Core2 Quad computer running at 2.83GHz with 4Gb of RAM. The algorithms were compiled using gcc-4.4.2 at -O3 optimization, while multi-threading capabilities were
implemented using pthreads and the Master/Worker threading model described by Nichols et al. [112].

Intersections calculations were performed between several sets of tetrahedral / polyhedral mesh pairs, as shown in Fig. 4.27a and Fig. 4.27b. The target mesh was chosen to be polyhedral, while the source mesh was chosen to be tetrahedral, with an additional halo layer of cells to account for boundary elements on the target mesh that may protrude out of the source domain. Details of the profiling results are given in Table. 4.3. The performance of the algorithm relies on several factors, such as the density of both source and target meshes and the number of tets per polyhedron. The symbol $n_{Ct}$ is the number of cells in the tetrahedral mesh, $n_{Cp}$ is the number of cells in the polyhedral mesh, $n_{Cpt}$ is the number of tetrahedra resulting from decomposition of polyhedra, and $n_t$ is the number of threads.

Since both conservative approaches share the step of calculating intersections, it is worthwhile to include the gradient contribution for second-order accuracy at a minor
computational cost. When compared to the conservative remapping approaches, the cost of inverse-distance remapping is trivial, with the highest cost being the initial octree-based source cell identification step.

<table>
<thead>
<tr>
<th>Mesh details</th>
<th>Calculation time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$n_{C_p}$</td>
</tr>
<tr>
<td>3384</td>
<td>17008</td>
</tr>
</tbody>
</table>

Table 4.3. Profiling test results

4.4.10 Computational Fluid Dynamics test

The final test is the driven-cavity problem for an incompressible fluid, coupled with a transport equation for a passively-advected scalar field, $\phi$, with no diffusion term. The domain is identical to the cosine hill case shown in Fig. 4.23, with lid-
velocity $U = 20 \text{ms}^{-1}$, kinematic viscosity $\nu = 0.01 \text{m}^2\text{s}^{-1}$, and box dimension $d = 1 \text{m}$. The initial field for $\phi = 0.5 + (0.5 \cdot \cos(\pi r/d))$, where $r$ is the distance from the centre of the box. A snapshot of the flow-field is shown in Fig. 4.28. The solution is initially obtained on a triangular mesh ($n_{Ca} = 14360$) for the time-range $t = [0 - 1] \text{s}$, using the SuperBee scheme [136] for advection, at a Courant number of 0.1. The solution process is then repeated, and all fields are mapped to a new polygonal mesh ($n_{Cb} = 7405$) at $t = 0.5 \text{s}$, and then remapped to the original mesh at $t = 0.75 \text{s}$. For comparison, both the conservative 2$^{nd}$ order and inverse-distance methods are used for the remapping steps.

Figure 4.28. Snapshots of the driven-cavity problem

The global integral of $\phi$ is computed every 50 time-steps, and plotted for the three cases, shown in Fig. 4.29.
4.4.11 Divergence-free Face Flux Transfer

The transfer of face fluxes ($\phi$, where $\phi = \mathbf{v}_f \cdot \mathbf{S}_f$) requires some care because of the additional constraint due to incompressibility, which requires it to be divergence free. The failure to maintain this condition is equivalent to the local creation or destruction of mass, which results in solution unboundedness. Operations such as 2D face-flipping, 2-3 / 3-2 swaps in 3D, and edge-bisection (both 2D and 3D) result in the creation of faces which do not have an equivalent on the original mesh and so, a general procedure to obtain divergence-free flux transfer needs to be incorporated.

The first step in the transfer process is to obtain a non-conservative flux estimate from the cell-centred velocity, which is interpolated from the original mesh using the methods described in Section 4.4.

$$\phi^* = \mathbf{v}_f \cdot \mathbf{S}_f$$  \hspace{1cm} (4.26)

Next, a pressure Poisson equation is solved using $\phi^*$, given in Eq.(4.27)
\[(a_P)^{-1}_f \frac{|S_f|}{|d|} (p_N - p_P)_f = \phi^* \tag{4.27}\]

Again, the terms \(a_P\) and \(p\) are interpolated cell-centred variables from the original mesh. The divergence-free flux \(\phi\), is obtained by modifying \(\phi^*\) using the correction-fluxes obtained from Eq.(4.27). This relation is the same as Eq.(3.64) in Chapter 3, repeated here for convenience.

\[F = \phi^* - (a_P)^{-1}_f \frac{|S_f|}{|d|} (p_N - p_P)_f\]

This procedure ensures that divergence-free fluxes are always available after changes to mesh topology, and is general enough to be applicable to situations where there is little similarity between original and modified meshes. The trade-off is the computational expense associated with the solution of an extra Poisson equation, which may or may not be significant.
CHAPTER 5
MESH SMOOTHING METHODS

Mesh smoothing, sometimes referred to in literature as mesh motion, or mesh re-zoning, is critical to the success of Lagrangian surface tracking methods. An effective smoothing technique continuously maintains the quality of the mesh during the course of the simulation, and can often delay the requirement for local re-meshing, which in turn minimizes interpolation errors. Pure mesh motion (i.e., without any re-meshing) can easily be incorporated into the governing momentum equations without any loss of accuracy.

5.1 Survey of Existing Mesh Smoothing Methods

Most attempts at effective mesh smoothing take either of two approaches - (a) global or (b) local optimization of mesh vertex (or node) positions, each of them having their fair share of benefits and flaws. Perhaps the most popular approach is spring analogy based Laplacian mesh smoothing (Blom [18], Huang and Russell [80]) - a simple algorithm that is quick, both in implementation and run-time efficiency. In its basic form, this method moves every vertex in the mesh to the average position of its nearest neighbours (as shown in Fig. 5.1), and is governed by the minimizing the following functional:

\[ \sum_j (x_{ij} - x_i) = 0 \]  \hspace{1cm} (5.1)

where, \( x_{ij} \) denotes the position of every neighbouring vertex immediately connected to vertex \( x_i \).
This is analogous to the case where edges connecting two neighbouring vertices are regarded as springs in tension, and are allowed to relax to their equilibrium state. This is a well-posed energy minimization problem that can be solved effectively using the Conjugate Gradient method [79]. It is also given in the weighted form:

\[ \sum_j k_{ij} (x_{ij} - x_i) = 0 \]  \hspace{1cm} (5.2)

where \( k_{ij} \) represents a spring constant. Several variants of Laplacian smoothing can be derived by choosing appropriate values for weighting factors. For instance, weighting each vertex by the sum of cell-volumes touching it results in moving mesh vertices to the centroid of volumes surrounding the vertex, rather than averaged positions. Spring constants can also be allowed to vary spatially according to solution gradients, in an effort to coax mesh smoothing to perform mesh adaptation for better solution accuracy (Habashi et al. [74]). The approach can also be used to perform mesh smoothing on surfaces (which may or may not be planar) by removing the surface-normal component of the spring forces at each vertex:

\[ (I - \mathbf{n}\mathbf{n}^T) \sum_j k_{ij} (x_{ij} - x_i) = 0 \]  \hspace{1cm} (5.3)
Since surface-normals are closely coupled to vertex positions at any given time, the relation becomes non-linear, and must therefore be linearized by lagging normal information in several steps until convergence is obtained. This linearization forces surface points to move in a plane defined by the normal at that time (which is a first-order approximation) and so, care must be taken to ensure that displacements are not too large as to violate volume conservation constraints.

In two dimensions, Laplacian smoothing has proven to be highly effective. In three dimensions, however, several complications such as degenerate or inverted cells start to appear. These cells possess either zero or negative volumes, which affect interpolation accuracy, matrix conditioning, and stability as well (Shewchuk [142]). One possible method of avoiding this condition is to implement a smart Laplacian smoothing technique, where a node in the mesh is moved only if it does not cause an inversion of the cells connected to it (Field [59], Canann et al. [26]). A particularly notorious type of degenerate cell that Laplacian smoothing produces is a sliver tetrahedron (See Fig. 5.2), first reported by Cavendish et al. [28]. Slivers possess edges of comparable size, but one dimension (and therefore its volume) is small. These cells are sometimes hard to detect, and typically require explicit removal to prevent harmful effects on the simulation (Freitag and Olliver-Gooch [63]).

![Figure 5.2. Types of degenerate cells](image)

The reason behind the inefficiency of Laplacian smoothing in three dimensions is the fact that it does not directly relate to the quality of the mesh in any way. There have been attempts to improve it by adding additional constraints.
Farhat et al. [55, 54] used torsional springs at each vertex location to prevent the formation of degenerate cells. An interesting approach is the ball-vertex method by Ackigoz [1], where virtual springs connecting vertices to opposing tetrahedral faces were introduced in an attempt to prevent motion towards them. Jasak and Tuković [86] and Compere et al. [32] use the finite element method based on strain-displacement relations for linear elasticity to achieve mesh motion. However, such modifications often come with additional computational cost and/or non-linearity, which is an important factor to consider.

Optimization can also be performed based on a measure of cell quality. This approach was first introduced by de Cougny et al. [40] to optimize tetrahedral meshes. de Cougny defines a distortion metric of cell quality (the scaled ratio of the element’s volume to it face areas), and for each vertex in the mesh, performs a line search in the direction which minimizes the maximum cell distortion metric.

Since cell quality measures are difficult to define for arbitrary polyhedra, they are often described only for simplices (triangles or tetrahedra), and rely on the fact that polyhedral meshes can often be decomposed into simplical ones. In the most severe cases, a single degenerate cell is sufficient to halt a simulation, and it is therefore the $L_\infty$ norm of quality that is more important than the $L_1$ (mean quality), or $L_2$ (root-mean-square) norms. For a mesh motion solver that globally optimizes for a given quality metric, it is important to severely penalize sliver-like tetrahedra so that the $L_\infty$ norm is given priority. Global optimization based on mesh quality metrics has been investigated by Parthasarathy et al. [120] and Canaan et al. [25], with good results.

Existing methods for local mesh optimization attempt to maximize the minimum quality of cells in the mesh. Examples of this approach include work by Amenta et al. [5], Freitag et al. [61, 62, 63] and Zavattieri et al. [118]. This is done by locally constructing a hull of cells surrounding a vertex and moving it in a manner that
maximizes the quality of all cells in that local group. The procedure is then repeated for the next vertex. This sequence is essentially a Gauss-Seidel iteration over all vertices in the mesh, and is similar to local Laplacian smoothing in that respect, the difference being that each vertex is now subjected to a complex (and often expensive) optimization problem rather than plain weighted arithmetic averages. It is unclear whether this approach actually converges to an optimal global quality (even if it does converge, it is also uncertain what it converges to). Empirical results published by several authors using this approach seems to show that good results are often obtained.

Freitag et al. [63] describe the optimization based mesh smoothing algorithm as follows:

- Assume that \( x \) is the position of a given mesh vertex, \( f_i(x) \) is the quality of cells attached to the vertex, and \( g_i(x) \) is the gradient of \( f_i(x) \) at that position.

- Define a composite function:

\[
\phi(x) = \min f_i(x). \tag{5.4}
\]

Although \( f_i(x) \) is continuous, \( \phi(x) \) is not, and usually contains discontinuous partial derivatives. Thus, Eq.(5.4) is a non-smooth problem which is to be solved.

- Analogously, Eq.(5.4) can be solved by the following relation for \( \beta \) to obtain an effective gradient search direction, \( \bar{g} \):

\[
\min g^T \bar{g}; \quad \text{where } g = \sum_i \beta_i g_i(x) \text{ and } \sum_i \beta_i = 1 \tag{5.5}
\]
• Once \( g \) is found using Eq.(5.5), the distance along the search direction is obtained using a bisection method, constrained to quality improvement for a specified tolerance.

Freitag at al. used the steepest descent approach to solving the problem and noticed that, empirically, the optimum point placement is achieved in an average of 2.5 steps. Dai [37] used a Quadratic Programming (QP) approach. Amenta et al. [5] showed for certain choices of cell quality measures, the problem becomes quasiconvex, and can therefore be solved in linear time using the Generalized Linear Programming (GLP) paradigm. Owing to the computational expense involved with optimization methods, there have been efforts to combine it with Laplacian smoothing, which have proved to be quite successful (Canann et al. [26]). Moreover, since the algorithm performs only local modifications to vertex positions, this approach also parallelizes well (Freitag et al. [61]). Zavattieri et al. [118] restrict the optimization effort to a relatively smaller number of vertices surrounding poor quality cells in the mesh, thereby achieving a good trade-off between cost and mesh quality.

5.2 Effective Smoothing of Curved Surfaces

The rapid evolution of fluid interfaces in the moving-mesh interface tracking approach requires a robust surface smoothing technique combined with effective topology modifications to maintain a valid mesh. When subjected to specified surface length scales, bisection and/or collapse operations are used to account for expanding or diminishing interface area. It is well known that successive bisection and contraction operations often degrade quality and so, continuous swapping and smoothing is necessary to maintain an acceptable mesh that is free from slivers. The smoothing of three-dimensional mesh surfaces can sometimes become fairly complicated because the vertices on the surface must be displaced in a manner that maximizes cell quality, in addition to maintaining the local shape of the surface.
In this work, the triangular surfaces of tetrahedral meshes are smoothed using the spring-analogy Laplacian solver (by solving Eq.(5.3) in Section 5.1). While this approach is cost-effective (and in most cases, often sufficient), it does have a fairly obvious drawback - it does not account for the quality of tetrahedra connected to surface vertices in any way. There may be occasional instances where the displacement of surface vertices are too large, thereby invalidating a few cells at the immediate interior.

One possible approach to surface smoothing, given by Yin and Teodosiu [163], is to take interior tetrahedral mesh quality into account. This involves the use of a quadric surface constraint to be imposed for each point on the surface, given by the relation:

$$z_l(x, y) = a_0 + a_1 x_l + a_2 y_l + a_3 x_l y_l + a_4 x_l^2 + a_5 y_l^2$$  \hspace{1cm} (5.6)

This surface is defined for a local coordinate system \((x_l, y_l, z_l)\) with the origin defined at the point \(x_i\) to be optimized, and the z-axis corresponding the the point-normal at \(x_i\). The point-normal is obtained by taking the resultant of all face-normals connected to \(x_i\). The orientation of the x- and y-axes are arbitrary, and any choice of mutually orthogonal vectors is sufficient. It is also convenient to compute a transformation tensor from global to local coordinates \(T\), given by the relation in Eq.(5.7)

$$T = \begin{bmatrix} x_g \cdot x_l & x_g \cdot y_l & x_g \cdot z_l \\ y_g \cdot x_l & y_g \cdot y_l & y_g \cdot z_l \\ z_g \cdot x_l & z_g \cdot y_l & z_g \cdot z_l \end{bmatrix}$$  \hspace{1cm} (5.7)

where the global cartesian axes are given: \(x_g^T = [1\ 0\ 0]\), \(y_g^T = [0\ 1\ 0]\) and \(z_g^T = [0\ 0\ 1]\). Also, \(x_l\), \(y_l\) and \(z_l\) are unit vectors (in global coordinates) denoting the axes for the local coordinate system. The tensor \(T\) is particularly attractive because it is orthogonal, \(i.e.,\) its inverse is also its transpose. To calculate the coefficients \(a_0 \ldots a_5\) of the quadric surface, it is necessary to obtain at least 6 points in the neighbourhood.
of \(x_i\). If exactly six points are available, a \(6 \times 6\) system can be constructed and solved by direct inversion:

\[
\begin{bmatrix}
1 & x_0 & y_0 & x_0y_0 & x_0^2 & y_0^2 \\
1 & x_1 & y_1 & x_1y_1 & x_1^2 & y_1^2 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
1 & x_5 & y_5 & x_5y_5 & x_5^2 & y_5^2 \\
\end{bmatrix}
\begin{bmatrix}
a_0 \\
a_1 \\
\vdots \\
a_5 \\
\end{bmatrix}
=
\begin{bmatrix}
z_0 \\
z_1 \\
\vdots \\
z_5 \\
\end{bmatrix}
\]

(5.8)

If more than six points are available, a weighted least-squares approximation can be obtained by solving the system:

\[
X^T W X \{a\} = X^T W \{z\}
\]

(5.9)

where \(X\) is similar to the first matrix in Eq.(5.8) with \(n\) points instead of 6, \(W\) is an \(n \times n\) diagonal weight matrix, \(\{a\}\) is the vector of coefficients, and \(\{z\}\) is the vector of \(z\)-coordinates for each point. This approach uses a larger stencil to obtain a better approximation to the local surface curvature.

Having defined the quadric surface for \(x_i\), the optimization algorithm must now achieve an improvement in the mesh quality of tetrahedra attached to this point in addition to maintaining the constraint during the process. Before delving into the details of the optimization algorithm, a quality measure must first be defined. A large variety of cell-based quality metrics have been defined in literature, such as those given in Section 4.1.1. Since the intent in this case is the optimization of all cells connected to a surface point, a node-based definition is now necessary. Consider \(M = 1 \ldots m\) to be the set of elements connected to \(x_i\), then the node-based metric \(q_n\) for the point is given by Eq.(5.10):

\[
q_n = \sum_{m=1}^{M} q_m
\]

(5.10)
where $q_m$ is the quality metric for the $m$th cell connected to $x_i$. Three edge-vectors can be defined from $x_i$ to the $m$th tetrahedral cell, such that:

$$
\begin{align*}
\mathbf{e}_{m,k} &= \mathbf{x}_{m,k} - \mathbf{x}_i \\
\quad k &= 1, 2, 3
\end{align*}
$$

The discrete Jacobian matrix for the $m$th cell can now be defined:

$$
\mathbf{J}_m = [\mathbf{e}_{m,1} \mathbf{e}_{m,2} \mathbf{e}_{m,3}] = \\
\begin{bmatrix}
x_{m,1} - x_i & x_{m,2} - x_i & x_{m,3} - x_i \\
y_{m,1} - y_i & y_{m,2} - y_i & y_{m,3} - y_i \\
z_{m,1} - z_i & z_{m,2} - z_i & z_{m,3} - z_i
\end{bmatrix}
$$

The Jacobian matrix assumes a clockwise ordering for $\mathbf{x}_{m,k}$ when viewed from $\mathbf{x}_i$. If this convention is adhered to, then the determinant of $\mathbf{J}_m$ is given by Eq.(5.13).

$$
\alpha = \det(\mathbf{J}_m) = \mathbf{e}_{m,1} \cdot (\mathbf{e}_{m,2} \times \mathbf{e}_{m,3}) \tag{5.13}
$$

Thus, $\alpha$ is positive if the cell is valid, and negative if it is inverted. It is also convenient to define the Frobenius norm of $\mathbf{J}_m$, given in Eq.(5.14).

$$
|\mathbf{J}_m|_F = \sqrt{\text{trace}(\mathbf{J}_m^T \mathbf{J}_m)} \tag{5.14}
$$

Having defined the building blocks for the node-based metric, the Inverse Smoothness measure outlined by Knupp [99] is now given in Eq.(5.15).

$$
q_n = \sum_{m=1}^{M} \frac{|\mathbf{J}_m|_F^2}{\alpha^{2/3}} \tag{5.15}
$$

This metric is non-dimensional, but it is not normalized and does not possess an ideal quality of 1 (unlike the metrics defined in Section 4.1.1). Instead, it behaves like an
inverse, where better cell-quality is obtained by minimizing $q_n$. This fits well with the constrained minimization objective of any optimization algorithm, but the case of a maximizing metric requires only a trivial change in the objective, where $q_n$ is replaced by $-q_n$. Clearly, any choice of a node-quality metric with properties similar to the Inverse Smoothness would suffice for $q_n$.

Typically, an optimization algorithm requires the gradient of the $q_n$ at a given location during the iteration process. This can be evaluated using finite-differences in space, but based on the differencing scheme, this usually requires several function calls for $q_n$ which attributes towards computational cost, and may not be sufficiently accurate. Another approach is to analytically differentiate $q_n$ to obtain a gradient, which is beneficial because $\nabla q_n$ can be easily computed at a given point at the same time as $q_n$, with very little additional cost. As explained by Knupp [99], the differentiation process can quickly become quite daunting because $|J_m|^2$ contains 9 terms which must be differentiated, while $\alpha$ contains 6. When differentiated, this results in a possibly unwieldy set of terms. This process is simplified by acknowledging that the quality measure is basically a scalar function of several matrices. Thus, by applying the chain rule of differentiation,

$$q_n = \sum_{m=1}^{M} f(J_m)$$

$$\nabla q_n = \sum_{m=1}^{M} \text{trace}((\partial q_n/\partial J_m)^T(\partial J_m/\partial x))$$

$$= \sum_{m=1}^{M} (\partial q_n/\partial J_m)u$$

$$= \sum_{m=1}^{M} (\partial q_n/\partial J_m)u = u^T = [-1, -1, -1] \quad (5.16)$$

Thus, differentiating the Inverse Smoothness metric for the $m^{th}$ cell yields the expression given in Eq.(5.17).
\[
\frac{\partial q_m}{\partial J_m} = \frac{\partial \left( \frac{|J_m|^2_F}{\alpha^{2/3}} \right)}{\partial J_m} = \frac{2}{3\alpha^{2/3}} \left[ 3J_m - |J_m|^2 F J_m^{-T} \right] \tag{5.17}
\]

where the following identities were used to obtain the result.

\[
\frac{|J_m|^2_F}{\partial J_m} = 2J_m \tag{5.18}
\]

\[
\frac{\alpha^n}{\partial J_m} = n\alpha^2 J_m^{-T} \tag{5.19}
\]

Similarly, the constraint gradient is obtained by differentiating Eq.(5.6) to obtain:

\[
C_{eq} = a_0 + a_1 x_l + a_2 y_l + a_3 x_l y_l + a_4 x_l^2 + a_5 y_l^2 - z_l \tag{5.20}
\]

\[
\nabla C_{eq} = \begin{bmatrix}
a_1 + a_3 y_l + 2a_4 x_l \\
a_2 + a_3 x_l + 2a_5 y_l \\
-1
\end{bmatrix} \tag{5.21}
\]

5.2.1 Constrained Optimization for the Node-based Measure

The optimization procedure for a surface point \(x_i\) is rather complicated because it falls under the category of a general Non-linear Problem (NLP), given by:

\[
\min_x f(x) \begin{cases}
c(x) \leq 0 \\
c_{eq}(x) = 0
\end{cases} \tag{5.22}
\]

where both \(f(x)\) and the constraint relations \(c(x), c_{eq}(x)\) are non-linear. Earlier attempts to tackle such problems (including the work by Yin and Teodosiu [163]) involved the use of a large multiplicative penalty parameter \(\mu\) for the constraint function, so that it is redefined as an unconstrained problem:

\[
\min_x f(x) + \mu c(x) + \mu c_{eq}(x) \tag{5.23}
\]
While this approach might be feasible for certain cases, it is often difficult to decide on the magnitude of $\mu$ that is appropriate for satisfactory convergence. Choosing extremely large penalty values tends to make the system ill-conditioned, and the solution may even diverge. A constrained optimization algorithm limits the feasible area in the search-space, and can therefore make informed decisions on optimal search directions and step-lengths based on continuously updated information during the iteration process. This advantage is not available for an unconstrained method, thereby resulting in a significantly larger number of iterations to obtain an equivalent result. Owing to these factors, the unconstrained penalty approach has been largely replaced by methods that seek solutions to the Kuhn-Tucker (KT) equations in Eq.(5.24a), which augment Eq.(5.22).

$$\nabla f(x) + \sum_{i=1}^{n} \lambda_i \cdot \nabla c_i(x) = 0 \quad (5.24a)$$

$$\lambda_i \cdot c_i(x) = 0 \quad (5.24b)$$

$$\lambda_i \geq 0 \quad (5.24c)$$

where $\lambda$ is the vector of Lagrange multipliers required to balance the deviation in the gradients of the objective function $f(x)$ and the constraints $c(x)$. If only certain constraints are considered active during the optimization process, the values of $\lambda$ corresponding to the inactive constraints are given a value of 0, which is stated by Eq.(5.24b) and Eq.(5.24c). These optimization methods aim to linearize Eq.(5.22) using an iterative process which solves for the Lagrange multipliers at each step. Each iteration performs a quasi-Newton update based on accumulated information from previous steps, thereby guaranteeing superlinear convergence properties. In literature, this class of methods are commonly referred to as Sequential Quadratic Programming (SQP) algorithms, since a quadratic subproblem (QP) is solved at each outer iteration. The method was first introduced by Wilson [162] as an approach to
convex optimization, and subsequently made popular by Biggs [15], Han [75] and Powell [130].

Referring to Eq.(5.22), the associated Lagrange function is given by Eq.(5.25):

$$L(x, \lambda) = f(x) + \sum_{i=1}^{n} \lambda_i \cdot c_i(x) \quad (5.25)$$

The quadratic sub-problem (QP) at the $k^{th}$ iteration is obtained by a quadratic approximation to Eq.(5.25), using linearized Taylor series approximations to the constraints, $c(x)$:

$$\min_{d_k} \frac{1}{2} d_k^T H_k d_k + [\nabla f(x_k)]^T d_k$$

$$[\nabla c_i(x_k)]^T d_k + c_i(x_k) = 0 \quad (5.26)$$

where $H_k$ is an approximation at the $k^{th}$ iteration to the Hessian of the Lagrange function, and $d_k$ is the search direction. Eq.(5.26) can be solved using any one of the several approaches to the general Quadratic Programming problem, and the active-set method is used in this work to obtain a solution for $d_k$ and its associated vector of Lagrange multipliers $\lambda$. The search direction $d_k$ is then used in a line-search to obtain a new guess $x_{k+1}$:

$$x_{k+1} = x_k + \alpha_k d_k \quad (5.27)$$

The line-search uses a merit-function $\psi_k(\alpha)$ (with $0 < \alpha \leq 1$), such that the term $[\psi_k(\alpha_k) - \psi_k(0)]$ is sufficiently reduced to a specified tolerance. The Hessian approximation is initialized as the identity matrix, and then subsequently updated at each outer iteration using the BFGS formula (named in recognition of its inventors - Broyden, Fletcher, Goldfarb and Shanno), given in Eq.(5.28).
\begin{align}
H_{k+1} &= H_k + \frac{q_k^T q_k}{q_k^T s_k} - \frac{H_k^T H_k}{s_k^T H_k s_k} \\
 s_k &= x_{k+1} - x_k \\
 q_k &= \left[ \nabla f(x_{k+1}) + \sum_{i=1}^n \lambda_i \cdot \nabla c_i(x_{k+1}) \right] - \left[ \nabla f(x_k) + \sum_{i=1}^n \lambda_i \cdot \nabla c_i(x_k) \right]
\end{align}

(5.28)

The SQP algorithm is usually the preferred approach for systems involving relatively smaller number of unknowns. In this case, the unknowns correspond to the three cartesian coordinates of the optimized point and so, this method is quite efficient. Owing to the local nature of the optimization process, this algorithm also parallelizes well.

The optimization approach, however, can quickly become expensive for cases involving a large number of surface vertices. This is mainly attributed to the fact that multiple Gauss-Seidel style sweeps are required to achieve a mesh of good quality. If computational cost is a factor, a Laplacian mesh smoothing approach combined with an untangling algorithm may be preferable.

### 5.2.2 Laplacian Smoothing with Untangling

An alternate approach for robust surface mesh smoothing is to utilize the spring-based Laplacian technique, combined with an untangling algorithm to account for situations where excessive vertex displacement causes invalid cells. The first step is to loop through all surface vertices and compute the Jacobian of tetrahedra connected to each one, using Eq. (5.12). If the determinant of the Jacobian is negative, then the cell is considered to be inverted, and the untangling algorithm is then invoked.

Mesh untangling on boundary vertices is considerably more complicated than interior ones, owing to the surface constraint. Rather than opting for a constrained optimization algorithm at the boundary, an easier approach would be to detect all interior vertices connected to the invalid cell, and perform an unconstrained optimization on that set alone.
The algorithm devised by Freitag et al. [64] solves the non-smooth problem given by Eq.(5.29).

$$\max \left( \min (\alpha_m) \right)_{m=1,...,M} \quad (5.29)$$

Since $\alpha_m = \det(J_m)$ is a linear function of vertex positions, this problem can be conveniently optimized using linear programming. The approach by Vachal et al. [156] is particularly attractive because it converts the non-smooth problem into one that is smooth and convex, given by Eq.(5.30).

$$q_u = \sum_{m=1}^{M} [|\alpha_m - \beta| - (\alpha_m - \beta)]^2 \quad (5.30)$$

where $\beta$ is a slack parameter. The choice of $\beta$ is somewhat ad-hoc, since an extremely small value results in nearly degenerate (but still positive volume) cells, while large values tend to cause divergent solutions. Yin and Teodosiu [163] suggest a value of $|\sum \alpha_m|/10M$, based on experience.

To optimize for $q_u$, a Quasi-Newton approach based on the limited-memory BFGS algorithm by Nocedal [114] is used in this work. Since this approach requires gradients of the function to be evaluated at a particular location, an analytical gradient is provided here.

Consider the following form for $q_u$:

$$q_u = \sum_{m=1}^{M} [\sqrt{u^2} - u]^2 \quad (5.31)$$

$$u = [\alpha_m - \beta]$$

Using the chain-rule of differentiation,
\[
\frac{\partial q_u}{\partial J_m} = \sum_{m=1}^{M} \frac{\partial q_u}{\partial u} \frac{\partial u}{\partial J_m} \tag{5.32}
\]
\[
\frac{\partial q_u}{\partial u} = 2 \left[ \sqrt{u^2} - u \right] \left( \frac{u}{|u|} - 1 \right) \tag{5.33}
\]
\[
\frac{\partial u}{\partial J_m} = \frac{\partial [\alpha_m - \beta]}{\partial J_m} = \alpha_m J_m^{-T} \tag{5.34}
\]
\[
\frac{\partial q_u}{\partial J_m} = -2 \left[ |\alpha_m - \beta| - (\alpha_m - \beta) \right]^2 \frac{\alpha_m J_m^{-T}}{|\alpha_m - \beta|} \tag{5.35}
\]

where the identity from Eq.(5.19) has been used to obtain the relation in Eq.(5.34).

It is important to realize that every tangled mesh does not necessarily possess an untangled configuration, and that the algorithm described above can fail, simply because a solution does not exist. In such situations, assuming that the mesh is in an untangled configuration the previous time step, the mesh vertex positions are relaxed to an intermediate configuration that doesn’t invert cells, by using the following relation:

\[
x^o_{nb} = \lambda x^o_{nb} + (1 - \lambda) x^s_{nb} \tag{5.36}
\]

where \(x^o_{nb}\) are boundary vertex positions at the previous time step, \(x^s_{nb}\) are the positions after Laplacian smoothing, and \(x^o_{nb}\) is the final configuration. The parameter \(\lambda\) is a relaxation parameter in the range \([0 \ldots 1]\), determined using a bisection method.

### 5.3 Smoothing of Interior Mesh Vertices

To maintain the quality of cells away from mesh boundaries towards the interior, this work uses the Mesquite Mesh Improvement library from Sandia National Labs [22] for three dimensional meshes. The library is independent by design and can be linked via functional interfaces, thereby making it versatile. It also provides a variety of options, including quality metrics, assessors, objective functions and optimization algorithms that include user-driven termination criteria for efficiency. Mesquite cannot, however, optimize vertex positions on complicated boundaries like
the situations involved in this work, and assumes that these points on the input mesh are fixed. Thus, surface smoothing is first performed using the method described in Section 5.2 prior to the optimization of interior points. The library, at the time of writing, only provides experimental support for smoothing in distributed-memory parallel configurations. This issue is addressed in Chapter 6.

For two-dimensional meshes, a conventional spring-analogy Laplacian solver using the Conjugate Gradient method was used predominantly for reasons of efficiency, in addition to work carried out by Jasak and Tuković [86], which uses the finite-element method to optimize vertex positions by decomposing polyhedral cells into tetrahedra. Parallelization of the spring-analogy Laplacian solver is also discussed in Chapter 6.
CHAPTER 6
PARALLEL MESH ADAPTATION

With an efficient implementation of the basic mesh reconnection operations in place, test cases involving a large number of cells are usually the next step in the simulation process. While it is not unusual to see cases involving 250,000 cells or more, simulations which include multiple physical phenomena (like fluid-flow coupled with interfaces and heat-transfer effects, for instance) typically require running times ranging from a few hours to a few days, even on recent machine configurations. There is always a point at which the computational expense of a simulation is so immense that users typically run out of patience while waiting for an answer. Frequently, these large cases are used solely for the purpose of gathering statistics about the physics involved and so, it is in the user’s best interest to obtain results in a timely manner.

6.1 Background

Perhaps the most popular approach to achieving this goal is through parallelization, a technique which involves the use of multiple computational resources which share the effort of solving the problem at hand. The basic idea is to split the simulation work into smaller (and therefore more manageable) sizes, each of which can be tackled simultaneously by an individual computational resource. The term ‘computational resource’ (sometimes referred to as a node) is used here to broadly categorize the different types of computer hardware available for computational purposes. These include examples such as a single processor, or multi-core processors - which contain several CPUs on a single processor die. Processors can also be used
in cluster-like configurations - which involve several nodes connected to each other through communication hardware. In all these cases, the efficiency of communication between individual processing components is an important consideration. The best performance is usually obtained from configurations with several processors on a single mother-board, but they also possess a limitation on the amount of physical memory available on-board, thereby restricting the size of problems that can be solved. Configurations with network hardware are widely regarded to be the least effective in terms of communication, but problem sizes are only restricted by the availability of resources (which can theoretically be infinite). The primary challenge in any parallelization effort is to minimize the amount of communication necessary to achieve the task at hand, because this usually implies that the implementation will be highly efficient. A task which requires no communication at all is called a trivially parallel problem, but such instances are usually rare in practice.

From the perspective of software implementation, the parallelization effort involves a choice between two alternatives - using (a) shared memory, and (b) distributed memory. A shared memory approach involves a situation where the entire memory space is accessible to all nodes. This paradigm has the advantage of efficient memory access characteristics, but compromises on safety because read/write access to memory is not regulated in any way. The shared-memory approach is usually limited by the hardware resources available on a given node, and can thus impose a restriction on problem size. When an algorithm is memory-bound (which is typical of most practical CFD problems), mother-board memory bandwidth is also a limiting factor with shared-memory parallelism.

A distributed approach, on the other hand, provides a dedicated section of memory to each node for purposes of access safety, but this benefit usually corresponds to increased communication costs between nodes, particularly in non-trivial parallel algorithms. This effect can be mitigated to a certain extent by asynchronous
communication, which allows individual nodes to perform useful work while data transfers occur in the background. A related problem with the distributed memory approach is task synchronization. Since data communication is to be minimized in this approach, a typical section of the problem being tackled is usually unaware of the state of sections other than its nearest neighbours, and must therefore find ways to periodically synchronize changes across nodes. An innovative approach is the combination of both methods, known as hybrid parallelization, which can be highly efficient when implemented correctly.

6.2 Shared Memory Parallelism

In this work, sections of the mesh adaptation algorithm are parallelized using POSIX threads (pthreads) [112] on the shared memory paradigm. Threads are technically defined as a set of instructions that can be executed independently within a certain process. An example of a process would be a computer program, which can instantiate (or fork) any number of threads that it deems necessary during its period of execution. Multiple threads within a process can share resources such as memory space, but it is very rare to see threads being shared across processes.

The IEEE POSIX 1003.1c specification was defined as a means of providing a standard for various thread library implementations, thereby easing the concerns of portability across platforms. For UNIX-like systems, threads are defined as light-weight processes which require far less resources than a traditional system process. Threads can be instantiated and managed with relatively less effort, while communication between threads is vastly simplified and highly efficient since they all see the same region in system memory. However, it is largely left to the programmer to avoid complications related to unsafe read/write accesses by multiple threads within a process. There are several mechanisms detailed by the standard to deal with such situations, such as mutexes, signals and condition variables, to name a few. These
mechanisms provide large degrees of freedom while implementing an algorithm, but the complexity involved in coding effort is usually unwieldy, and this has traditionally been a major barrier for the popularity of parallel programming.

One particular effort to simplify algorithm development is the OpenMP programming interface [36], which allows the use of compiler directives to parallelize sections of code (like for-loops), to achieve both task and data level parallelism. OpenMP requires a compiler that supports it, and most modern compilers usually do. However, fine tuning for performance in OpenMP is somewhat limited, but the paradigm itself is in no way inferior to the pthreads approach. Most compilers usually translate OpenMP directives to pthreads function calls anyway, so compile time is sometimes reduced by opting for the pthreads approach.

6.2.1 Threading Basics

This section will deal with a few basics of thread management, including the creation and destruction of threads. Every process starts execution with a single thread, and all subsequent threads are explicitly instantiated (or destroyed) during the program’s execution lifetime. A thread can be created using the pthread_create function call, which is given by the following C prototype:

```c
int pthread_create
(    
    pthread_t *thread,
    const pthread_attr_t *attr,
    void *(*start_routine)(void*),
    void *arg
);
```

where the `pthread_t` and `pthread_attr_t` are data-types defined in the `pthread.h` header file. Thread attributes are specified using options included through the `attr` pointer. The third argument is a function-pointer to a procedure called `start_routine`, which is to be executed in the newly created thread, and any arguments to the function are provided through `arg`. Since this a C-style function
call, programs written in C++ must pass only static class-member functions for the third argument to avoid complications due to name-mangling issues. Also, since static member-functions cannot access class member data, a pointer to the object must be supplied in the argument list to allow this behaviour. Typically, \texttt{arg} is a pointer to a C structure that contains appropriate data, and type-casting to/from \texttt{(void *)} is necessary. This function returns immediately, and \texttt{start\_routine} will be allowed to run independently in the new thread.

An important attribute is to specify whether the thread is joinable or detached. A joinable thread requires a \texttt{pthread\_join} function call, which is given by the following C prototype:

\begin{verbatim}
int pthread_join(pthread_t thread, void **status);
\end{verbatim}

This mechanism allows the calling function to wait for a thread to complete before resuming execution. A detached thread, on the other hand, does not require such a call, and will terminate normally when \texttt{start\_routine} runs to completion. When created, each thread is allotted its own stack space in memory (within the resources of the parent process), with the bare minimum required to execute. This allocation cost may sometimes be significant, so it is unwise to repeatedly create and destroy threads. While the stack resources allocated to each thread is unique, any thread is allowed to freely access any resource of its parent process. Thus, in the shared memory paradigm, a disciplined approach is necessary to ensure that data shared by several threads is accessed in a safe and predictable manner. This is achieved using \textit{mutexes} and \textit{condition variables}.

A mutex is an abbreviation for ‘mutual exclusion’, which allows threads to \textit{lock} sections of data (or code) while it is being accessed. This prevents all other threads from accessing the region while the lock is held. When all access operations have succeeded, the thread can \textit{unlock} the section and proceed normally. Only one thread can hold a mutex lock at any particular time, while all other threads are forced
to wait on the mutex until it becomes unlocked. If several threads are waiting on a mutex when it is unlocked, the system uses an internal scheduling policy to determine the next thread which will be allowed to lock the mutex. Mutexes can therefore be regarded as a mechanism to serialize critical sections of the process. While this provides security, excessive locking can lead to heavy performance degradation, particularly in situations where several threads are contending for the same region of shared data. Mutexes are created, destroyed, locked and unlocked (respectively) using the following calls:

```c
int pthread_mutex_init
(
    pthread_mutex_t *mutex,
    const pthread_mutexattr_t *attr
);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Condition variables are another mechanism by which threads can synchronize with each other. Consider a situation where several threads are waiting on a particular thread to access a critical section of data. In the absence of a condition variable, all waiting threads would have to continuously poll the critical section to check whether it is available. This is clearly wasteful because these threads could perhaps be doing other useful work while they wait. When a mutex is used in conjunction with a condition variable, the thread holding the mutex can signal other waiting threads and inform them that the section is available after the work is done. This signal can be either directly to another thread contending for the mutex, or broadcast globally to all threads waiting on the mutex. The pthreads function calls for the creation, destruction, signalling, broadcasting and waiting (respectively) of condition variables are given as follows:

```c
int pthread_cond_init(pthread_cond_t *cond);
int pthread_cond_destroy(pthread_cond_t *cond);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```
Of particular importance is the `pthread_cond_wait` function call, which is used in conjunction with a mutex. Any thread calling this function must first lock the mutex. When the function is called, the mutex is automatically unlocked (which allows others threads to acquire it), and the calling thread waits on the condition. The thread can be later awakened either by a specific signal from another thread, or a broadcast which wakes all threads waiting on the condition. In a broadcast situation, the pthreads library uses an internal scheduling policy to determine the order of awakened threads.

For the sake of algorithm simplicity, the conventions used in the next section to denote equivalent pthreads entities are given in Table 6.1.

<table>
<thead>
<tr>
<th>pthreads entity</th>
<th>Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>pthread_mutex_t</td>
<td>Mutex</td>
</tr>
<tr>
<td>pthread_mutex_lock</td>
<td>LOCK(Mutex)</td>
</tr>
<tr>
<td>pthread_mutex_unlock</td>
<td>UNLOCK(Mutex)</td>
</tr>
<tr>
<td>pthread_cond_t</td>
<td>Conditional</td>
</tr>
<tr>
<td>pthread_cond_signal</td>
<td>SIGNAL(Conditional)</td>
</tr>
<tr>
<td>pthread_cond_broadcast</td>
<td>BROADCAST(Conditional)</td>
</tr>
<tr>
<td>pthread_cond_wait</td>
<td>WAIT(Conditional, Mutex)</td>
</tr>
</tbody>
</table>

Table 6.1. Equivalent pthreads function calls

### 6.2.2 The Master/Worker Threading Model

An easy solution to avoiding thread instantiation costs is to employ a master/worker model [112], where a pool of worker threads are initially created by the master thread, and each worker is made to wait for a job (submitted by the master thread) in a work-queue. When a job is available, a worker thread picks it off the queue and works on it. When it completes, the worker returns to the queue and waits for the next job to arrive.
To adequately describe the model, it is first necessary to introduce the infoPool structure that will be used in this section. The contents of the infoPool structure is given in Fig. 6.1.

It is fairly evident that the infoPool structure is a linked-list of function pointers, and their corresponding arguments. It also consists of a mutex and several condition variables for work-queue synchronization. At the start of program execution, the requested number of threads are instantiated using the INITIALIZE_THREAD_POOL function.

**INITIALIZE_THREAD_POOL**

**Inputs**: Number of threads \( n \), Queue structure (infoPool), Array of pthread_t (tID[\( n \)])

\[
\begin{align*}
\text{infoPool} & \Rightarrow \text{numThreads} = n \\
\text{infoPool} & \Rightarrow \text{queueSize} = 0 \\
\text{infoPool} & \Rightarrow \text{head} = \text{NULL} \\
\text{infoPool} & \Rightarrow \text{tail} = \text{NULL} \\
\text{infoPool} & \Rightarrow \text{queueClosed} = \text{false} \\
\text{infoPool} & \Rightarrow \text{shutDown} = \text{false} \\
\text{for} \ i = 0, \ n \ \text{do} \\
\text{thread_create}(\text{tID}[i], \text{NULL}, \text{POOL_THREAD}, \text{infoPool})
\end{align*}
\]

**Function** INITIALIZE_THREAD_POOL

Each thread runs one instance of the POOL_THREAD function, and upon entry, immediately locks the queue and waits for a condition which specifies that the queue is not empty. If and when this condition occurs, the function first picks the first item off the queue, executes the function provided in the workQueueItem, decrements the queue counter and returns to the queue to resume waiting for the next item.
**PoolThread**

```plaintext
while true do
  Lock(infoPool⇒queueLock)
  while infoPool⇒queueSize = 0 and infoPool⇒shutDown = false do
    Wait(infoPool⇒queueNotEmpty, infoPool⇒queueLock)
  if infoPool⇒shutDown = true then
    Unlock(infoPool⇒queueLock)
    return
  infoPool⇒queueSize = infoPool⇒queueSize - 1
  newItem = infoPool⇒head
  if infoPool⇒queueSize = 0 then
    infoPool⇒head = infoPool⇒tail = NULL
    Signal(infoPool⇒queueEmpty)
  else
    infoPool⇒head = newItem⇒next
  Unlock(infoPool⇒queueLock)
  newItem⇒function(newItem⇒arg)
  DELETE(newItem)
```

**Function PoolThread**

This condition can also be triggered by a shut-down signal, in which case the thread unlocks the work queue and exits the function normally.

Jobs are added to the work queue by invoking the `AddToWorkQueue` function, which merely takes a function pointer along with its argument, creates a new `workQueueItem` and adds it to the `infoPool` linked-list. If the queue is either closed or in the process of shutting down, no further jobs can be added, and the function returns normally. If a new job is added, the function broadcasts a signal to all threads waiting on the queue. The first worker thread that can access the queue picks the new work item, while other threads resume waiting.

Finally, the `DestroyThreadPool` function is used to shut-down the work queue. The function first waits for the queue to become empty, and then sets a flag prompting all waiting threads to shut down. All threads are then joined to ensure that the process was completed in a clean manner.
ADDWORKQUEUE(nFunction, nArg)

Inputs: Queue structure (infoPool)

LOCK(infoPool⇒queueLock)

while infoPool⇒queueClosed = false or infoPool⇒shutDown = false do
  WAIT(infoPool⇒queueNotFull, infoPool⇒queueLock)

if infoPool⇒shutDown = true or infoPool⇒queueClosed = true then
  UNLOCK(infoPool⇒queueLock)
  RETURN

newItem = NEW workQueueItem
newItem⇒function = nFunction
newItem⇒arg = nArg
newItem⇒next = NULL

if infoPool⇒queueSize = 0 then
  infoPool⇒head = infoPool⇒tail = newItem
  BROADCAST(infoPool⇒queueNotEmpty)
else
  infoPool⇒tail⇒next = newItem
  infoPool⇒tail = newItem

infoPool⇒queueSize = infoPool⇒queueSize + 1
UNLOCK(infoPool⇒queueLock)

Function ADDWORKQUEUE

6.2.3 Thread-parallel Mesh Adaptation

Having introduced the threading model in Section 6.2.2, its application to thread-parallel mesh adaptation is now discussed. Multi-threaded algorithms are intrinsically quite complicated because of the need to frequently lock / unlock (and therefore serialize) sections of the code to avoid data corruption. Any implementation that can effectively handle thread-parallel execution without corruption issues is described as thread-safe. Thread-safety is a primary concern, but it also tends to be a detrimental factor for efficiency, since excessive locking and unlocking comes with a fixed-cost that can accumulate quite rapidly.

The most expensive components of mesh adaptation algorithms in Chapter 4 are the EDGEBISECTCOLLAPSE and SWAP2DEDGES/SWAP3DEDGES routines. On closer inspection, two possible thread parallelization approaches become apparent, differing primarily in the granularity of locks. The first step (common to both
Function DestroyThreadPool

approaches) is to split the stack of faces / edges among the number of running threads, so that they are roughly equal in number.

6.2.3.1 Fine-grained locking

The first approach is to have each thread pick an entity (face or edge) off its stack, have it decide whether the entity is a good candidate for a refinement or swapping operation and if it is, proceed with the task. While this sounds simple at first, the numerous possibilities of data corruption quickly become daunting when other threads (possibly working on adjacent entities) are taken into account.

Consider the scenario where an edge \( e_{ta} \) is being considered for a swapping operation by a thread \( ta \). Assume that \( c_{ta} \) is the set of cells connected to \( e_{ta} \). The ComputeMinQuality function is firstly required to ensure that none of the cells in \( c_{ta} \) are being worked on by other threads. Since the ComputeMinQuality function merely checks the quality of these cells, but doesn’t actually manipulate them, only read-access to \( c_{ta} \) is required. If another thread (say, \( tb \)) is in the process of modifying any of the cells in \( c_{ta} \), thread \( tb \) would possess write-access to the cell, in which case the thread \( ta \) should not be allowed to perform a read operation. This would require
a special read-write mutex to be defined for each cell in the mesh. Therefore, each cell and subsequently, all lower topology entities like faces, edges and points must be locked prior to any form of read/write operations.

In practice, such conflicts are usually rare, particularly for large-scale meshes, but they can and do occur. Eventually, the decision to opt for this approach comes down to whether the extensive book-keeping effort is justified by substantial gains in performance. Since the only way to observe the real-time performance of this method is to implement it, and considering the fact that extensive mutex locking and unlocking would be necessary, this approach is not considered in this work.

6.2.3.2 Coarse-grained locking

The second approach is to have a very high-level locking mechanism, where each of the worker threads possess a stack of their own, in addition to an empty master stack. The worker threads are now expected to check for refinement and/or swapping candidacy (with only read-access at all times), and if a candidate entity is found, add it to the master stack. This vastly simplifies the locking conundrum, since only a single mutex is now required - for the master stack. Once all threads have completed checking their allotted stacks, the main thread loops through the master stack, re-checks all candidate entities, and sequentially performs each operation, thereby making this a trivially parallel algorithm.

At first, it would appear that this approach places an excessive burden on the master thread, but in practice, the number of swapping/bisection/collapse operations required per time-step are far fewer than the number of entities itself, and the bulk of the computational effort is spent checking all entities for candidacy. This is particularly true for the COMPUTE_MIN_QUALITY routine, which evaluates the TET_QUALITY function several times for each edge, and is clearly the most expensive component of the adaptation algorithm. The overall mesh-adaptation algorithm is
now provided by the top-level ThreadedTopoModifier function, in addition to the modified EdgeBisectCollapse and SwapEdges (shown here for 3D) routines.

**ThreadedTopoModifier()**

*Input*: Number of worker threads \((n)\), Array of Stacks \((M[n+1])\)

**InitStacks(M)**

if \(n > 0\) then

\[
\text{for } i = 1, n \text{ do } \text{EdgeBisectCollapse}(M[n])
\]

EdgeBisectCollapse(M[0])

**InitStacks(M)**

if \(n > 0\) then

\[
\text{for } i = 1, n \text{ do } \text{SwapEdges}(M[n])
\]

SwapEdges(M[0])

**Function ThreadedTopoModifier**

**SwapEdges()**

*Input*: Stack of edges \((M)\), Thread index \((tIndex)\)

**InitTables(Q, K)**

while \(M\) is not empty do

\[
e = \text{pop}(M)
\]

if CheckBoundingCurve(e) then

continue

\[
\text{minQuality} = \text{ComputeMinQuality}(e)
\]

FillTables(e, Q, K)

if CheckQuality(e, Q, minQuality) then

if \(tIndex = 0\) then

RemoveEdgeFlips(e, K)

else

\[
\text{push}(M[0], e)
\]

**Function SwapEdges**

When \(n > 0\), the **InitStacks** function distributes all entities among \(n\) stacks, but leaves \(M[0]\) empty. If \(n = 0\), **InitStacks** pushes all entities on to \(M[0]\). A practical caveat to this approach is the first adaptation step, since the input mesh that is typically generated by a grid-generation algorithm, and may possess either excessive or insufficient mesh-refinement. Owing to this, an inordinate number of topology changes may be initially required. However, by limiting the number of changes per
EdgeBisectCollapse()

Input: Stack of edges (M), Thread index (tIndex)

while M is not empty do
    e = pop(M)
    if CheckEdgeBisection(e) then
        if tIndex = 0 then
            BisectEdge(e)
        else
            push(M[0], e)
    else if CheckEdgeCollapse(e) then
        if tIndex = 0 then
            CollapseEdge(e)
        else
            push(M[0], e)

Function EdgeBisectCollapse

sweep and allowing the mesh to relax to an equilibrium state over several time-steps using a mesh-smoothing algorithm, this can be ameliorated to a certain extent.

6.3 Distributed Memory Parallelism

The distributed approach to parallel mesh adaptation considered in this work is achieved using the Message Passing Interface (MPI) paradigm [71]. This involves the partitioning of the computational mesh into several sub-domains which represent a section of problem being solved. The topic of MPI programming is fairly broad, and several resources which describe various APIs, their implementation and usage are commonly available [72, 144].

The topic of distributed-memory parallel mesh adaptation has been explored by several researchers in the past. Adaptive techniques for octree-based refinement with dynamic load-balancing was considered by De Keyser et al. [95] for structured meshes, and for unstructured grids by Flaherty et al. [60]. Waltz [161] explored recursive refinement and derefinement techniques for tetrahedral grids. Oliker et al. [116] discusses parallel load-balancing strategies that focus on the recursive refinement of tetrahedral meshes, while hybrid three-dimensional meshes is considered by Kavouklis
et al. [94]. Octree-based techniques are particularly attractive because of local adaptation capabilities that are inherent to the approach. While the method is limited to structured quad (2D) and hexahedral (3D) meshes, it has proven to be very popular in situations involving shock-capturing and volume-of-fluid (VOF) techniques. Adaptation algorithms of this nature also parallelize very well because the task of synchronization across processor boundaries can be pre-determined before the (de)refinement process and therefore, only minimal communication effort is necessary. The dominant problem with the approach then becomes the issue of load-balancing - a task that requires the dynamic equi-distribution of cells across processors so that maximal efficiency is obtained. While the octree-based approach is convenient for error-driven mesh refinement, it does not readily extend to situations involving moving/deforming domains, because of the additional mesh quality constraint. Cavallo et al. [27] tackle this problem on unstructured hybrid 3D grids by a cell-migration paradigm that shifts processor boundaries during the quality-driven adaptation procedure. This approach has the benefit of circumventing tricky situations involving inter-processor synchronization, but requires a dynamic re-partitioning step that can potentially be quite expensive, depending on the frequency of mesh adaptation. On the other hand, Guoy et al. [73] choose to re-generate the sub-domain meshes in parallel when the element quality falls below a certain threshold. The following sections describe the approach taken to implement distributed memory parallelism for the mesh adaptation methods used in this work.

Before delving into details of the implementation, a few definitions are in order. For the sake of simplicity, all algorithms assume that a single processor node is responsible for each sub-domain (i.e, single-threaded). The mesh boundary is split into a set of distinct patches, where a patch is defined as a coherent set of boundary faces. In addition to physical patches, each sub-domain now has additional processor patches representing inter-processor boundary patches which act as communication
channels to neighbouring sub-domains. The algorithms described in this section inherently assume that faces on processor patches match both geometrically and topologically. For example, if patch #4 on processor 3 communicates with patch #1 on processor 5, face # 2 for patch #4 on processor 3 must match face # 2 for patch #1 on processor 5, and so on. Additionally, the face-centres and zeroth point of each processor patch face-pair must match geometrically. All processors are ranked numerically in ascending order, starting from 0 to \( nProcs - 1 \), where \( nProcs \) is the number of processors involved in the simulation. In general, for all algorithms described in this section, if an entity (such as a point or edge) is simultaneously shared by several processors, the processor with the lowest rank is responsible for its manipulation.

### 6.3.1 Domain decomposition and Halo meshes

The first step in the distributed memory approach is to decompose the solution domain into several parts, such that each sub-domain is allocated to a processor. An example of this process is shown in Fig. 6.2, where the triangular mesh of a rectangular domain is decomposed into 5 sub-domains. A desirable trait of this procedure is to yield a decomposition that equally divides the cells across all processors, while minimizing the size of inter-processor boundaries so that the communication effort is minimized. The simplest approach is geometric sub-division, where processor boundaries roughly coincide with global cartesian directions, but this approach does not guarantee an equal cell-distribution, and is usually only applicable to the simplest of domains. An approach that is particularly attractive for unstructured meshes is graph-partitioning, where grid-connectivity is taken into account during the decomposition process. Several popular examples of work in this area include the METIS library by Karypis and Kumar [93], the Zoltan library by Devine et al. [44], and the Scotch partitioning library by Pellegrini and Roman [122]. Although the
partitioning process is often considered to be a pre-processing step for most static parallel runs, simulations that dynamically change the mesh structure during run-time must also occasionally re-partition the domain due to the gradual imbalance in sub-domain cell sizes over time. This process is frequently referred to in literature as *dynamic load-balancing*.

Prior to any mesh adaption procedure, each sub-domain must have a minimal description of the mesh structure connected to processor-boundary points on neighbouring sub-domains, defined as a *halo mesh*. This is depicted in Fig. 6.2(c), where halo meshes for each of the 5 sub-domains are colored by original processor ID. An equivalent decomposition is shown for 3D in Fig. 6.3. Note that a ‘neighbour’ sub-domain in this context can share points, edges or faces. In addition to basic connectivity structures, each halo mesh must also carry a description of its boundary patches, since this information is often necessary during the adaptation procedure. Quite often, decomposed domains contain points that are simultaneously shared by several processors at the same time, known as *global* points. These points must first
be identified in order to assist in the halo mesh construction process. In this work, these points are identified by an efficient algorithm developed by Jasak [84] that uses only nearest-neighbour information.

Once halo meshes have been constructed, each sub-domain must send and receive them to neighbouring sub-domains using MPI. This transfer can be done asynchronously, so each sub-domain is free to do other useful work while transfers occur in the background. After all transfers have completed, each sub-domain must prepare entity maps that relate points, edges and faces on processor boundaries with equivalent entities on the received halo meshes, and vice-versa. Thus, for example, a point map $\mathcal{X}_p^j : \mathcal{N} \to \mathcal{N}^j$ takes a point from the sub-domain point-set ($\mathcal{N}$) to its equivalent point on the halo mesh point-set ($\mathcal{N}^j$) from processor $j$, while a reverse point map $\mathcal{R}_p^j : \mathcal{N}^j \to \mathcal{N}$ does the opposite. Similar maps are constructed for edges ($\mathcal{X}_e^j , \mathcal{R}_e^j$) and faces ($\mathcal{X}_f^j , \mathcal{R}_f^j$). Naturally, each entity map is defined for only a subset of all entities on each sub-domain.
6.3.2 Parallel Mesh Quality Improvement

The mesh adaptation procedure implemented in this work is a natural extension of the algorithms defined in Chapter 4, but with special consideration for edges and faces located on processor boundaries. For the parallel mesh adaptation procedure, all halo mesh entities are dealt with first, before the interior ones. If entities of the sub-domain were sent to a lower-ranked processor, then those entities must be left untouched, since the other processor may choose to modify the mesh topology locally.

![Figure 6.4. Cell migration for swapping in 2D](image)

The concept of local cell-migration is now introduced to explain localized topology changes in this context. Consider the two-dimensional case in Fig. 6.4(a), where triangle $abc$ is located on processor 0, while triangle $b'a'd'$ is located on processor 3. At this point, since halo meshes have been transferred across neighbouring processors, triangle $b'a'd'$ is easily accessed by a look-up from the edge map (in this case, $X^3_e$) for $ab$ to obtain $a'b'$ on the local halo mesh. The isolated point on triangle $b'a'd'$ (namely, $d'$) is now obtained, and the Delaunay test (refer Eq. 4.2 from Chapter 4) for edge $ab$ can be performed. If edge $ab$ fails the test, then triangle $b'a'd'$ is migrated from the local halo mesh for processor 3 to processor 0, as shown in Fig. 6.4(b). Since both triangles are located on processor 0, $ab$ now becomes an interior edge, which is swapped conventionally. After this is done, the corresponding edge maps ($X^3_e$...
and \( R_3 \) are updated to include edges \( ad \) and \( bd \), which now represent the processor boundary. Since triangle \( b'a'd' \) is no longer present on processor 3, an entry for cell-removal is made into a separate operations list. When multiple cells are migrated, the entries are made sequentially and accumulated until all operations on shared entities have been performed. This list is then transferred to the neighbouring sub-domain (using MPI) for sequential removal. The operation-transfer process can be performed asynchronously to hide communication latency. Thus, while transfers occur in the background, topology changes for interior entities can be performed locally by each sub-domain.

This approach to cell-migration makes local changes to inter-processor boundaries, and only involves cells that are present on the halo mesh. In contrast, the migration approach by Cavallo et al. [27] requires two global shifts to inter-processor boundaries, and a potentially large transfer of halo mesh connectivity and/or geometry information.

In 3D, the cell quality optimization step is much more involved, since an edge can be shared by an arbitrary number of processors. An example is shown for edge \( ab \) in Fig. 6.5(a), which is surrounded by 4 processors (the square-brackets denote processor rank). The ComputeMinQuality and FillTables functions from Chapter 4 must now be modified to handle edges of this type. When the Swap3DEdges function encounters an edge on a processor boundary, the edge maps \( X_e^j \) are checked to obtain equivalent edges for each halo mesh \( (j) \). All faces connected to edge \( ab \) are collected, and for each triangular face \( (f_i^j) \), the isolated vertex \( v_i^j \) on the face (i.e., the point other than \( a \) or \( b \)) is added to a list \( (v_p) \). Since faces are duplicated on processor boundaries, only the isolated vertex corresponding to the lower sub-domain processor rank is added, thereby avoiding point duplicates (and therefore, justifying the requirement for halo meshes to possess boundary information). A local coordinate system is now defined, with the origin passing through the edge centre \( (x_e) \), and the
z-axis along the vector $t_e$ which is tangential to edge $ab$, as shown in Fig. 6.5(d). The choice of the x-axis $d_x$ depends on whether edge $ab$ lies on (i) a physical boundary (like a wall, for instance), or (ii) purely on a processor boundary (a *pure* processor edge, in this context, is one which has all attached faces either on processor boundaries or the interior of the mesh).

For the former case, the starting vertex $v_s = v_i$ is chosen such that it lies on a physical boundary patch. A *pure* processor edge is logically considered to be an interior edge and so, any choice from $v_p$ would suffice for $v_s$. The axis $d_x$ is then calculated by projecting the $(v_p[i] - x_e)$ vector on to the plane passing through $x_e$, with $t_e$ as its normal. The next step is to project all points in $v_p$ on to the $d_x - d_y$ plane, and then compute the angles subtended by vertices with respect to $d_x$ (given

![Figure 6.5. Parallel edge in 3D](image-url)
by $\theta$ in Fig. 6.5(d)). The angles can be conveniently computed using arc-tangents (namely, the C/C++ \texttt{atan2} function). Now that all angles are obtained, the list is sorted by $\theta$ to obtain the counter-clockwise vertex ring $R$ around edge $ab$, which is then evaluated for minimum quality (using the \texttt{COMPUTE_MIN_QUALITY} function) and optimized quality (using the \texttt{FILL_TABLES} function). If the optimized value is indeed better than the original hull-quality, then all cells surrounding $ab$ are migrated to the lowest-rank processor involved in the set (namely, processor 0 in Fig. 6.5(a)), and the corresponding maps are updated to account for the modified processor boundary. As with the 2D case, edge $ab$ now becomes internal to the mesh and can be removed using the \texttt{REMOVE_EDGE_FLIPS} function. In situations where edge $ab$ lies on a physical boundary (considered to be an \textit{impure} processor edge), the constructed vertex ring $R$ naturally starts and ends with boundary points, because all points are sorted by angles with respect to $d_x$, which now aligns with $v_s$.

\textbf{6.3.3 Parallel Length Scale Resolution}

This section describes the approach taken to incorporate the length scale resolution operators (like edge bisection and contraction, described in Chapter 4) in a distributed-memory parallel paradigm. Since these operators require a local length scale field to be defined throughout the mesh, the \texttt{CALCULATE_LENGTH_SCALE} algorithm defined in Section 4.2.1 must also be parallelized.

Conceptually, the algorithm is a face-cell wave that propagates the length scale value by a layer of cells at each iteration, using previously initialized values as input. Only the $L$, $cLc$, and $cLevel$ arrays in the algorithm need to be parallel-aware (and therefore updated) at each iteration. This is achieved using the \texttt{WRITE_LENGTH_SCALE_INFO} and \texttt{READ_LENGTH_SCALE_INFO} functions.

The \texttt{Send} and \texttt{Recv} functions in \texttt{WRITE_LENGTH_SCALE_INFO} represent an abstraction to the MPI send and receive function calls, which in this case is performed
**WriteLengthScaleInfo**($cLevel$, $L$)

**Output**: Face buffer ($RecvFaces$), Level buffer ($RecvLevel$), Scale buffer ($RecvScale$)

**forAll** Boundary patches ($patchI$) on $\partial \Omega$ do

if $patchI$ is a processor patch then

   procID = Neighbouring processor-rank for $patchI$

   $nSf = 0$

   $fCells = \text{Cells adjacent to faces on } patchI$

   **for** $fI = 0$; $\text{size}(fCells)$ do

      $SendFaces[[patchI][nSf]] = fI$

      $SendLevel[[patchI][nSf]] = cLevel[fI]$

      $SendScale[[patchI][nSf]] = L[fI]$

      $nSf = nSf + 1$

   **SEND**($procID, SendFaces[patchI]$)

   **SEND**($procID, SendLevel[patchI]$)

   **SEND**($procID, SendScale[patchI]$)

   **RECV**($procID, RecvFaces[patchI]$)

   **RECV**($procID, RecvLevel[patchI]$)

   **RECV**($procID, RecvScale[patchI]$)

**Function** WriteLengthScaleInfo

asynchronously. Thus, the call to **SEND** and **RECV** returns immediately, thereby allowing normal program execution to continue. However, each process must wait for all transfers to complete before the data in containers **RecvFaces**, **RecvLevel** and **RecvScale** can be considered safe for use.

This synchronization step is performed only later by the **ReadLengthScaleInfo** function through the **WaitForBuffers** call. Once all buffers have been received, the **ReadLengthScaleInfo** function loops through cells adjacent to processor patches and re-calculate cell length scale values based on inputs from the buffer (using the **ReInitializeCell** function), in addition to updating the $cLevel$ and $cLc$ lists to account for changes made on other processors. The asynchronous nature of MPI transfers also allow the main body of the loop to execute normally while transfers occur in the background, and can be effective in hiding latencies, particularly for well-proportioned domain decompositions. Both the **ReadLengthScaleInfo** and **WriteLengthScaleInfo** functions can now be added to the **CalculateLengthScale** function at the locations shown to obtain

139
the\, \texttt{CalculateParallelLengthScale} \, function, which has been abbreviated to
display the relevant sections.

\subsection{Parallel Edge Refinement}

The next step is to incorporate coupled bisection and collapse operators that
drive across processor boundaries. The parallel bisection operation is relatively
straightforward in 2D - for each parallel edge $ab$, the equivalent edge $a'b'$ is obtained
using the edge map ($\mathcal{D}_e^j$) on processor $j$. Since only one such coupling is possible both
edges are bisected to introduce points $c$ and $c'$, respectively (shown in Fig. 6.6(a)).

This procedure can be extended to 3D (shown in Fig. 6.6(b)), but there are now
several entries for $ab$ in edge maps ($\mathcal{D}_e^j \ldots \mathcal{D}_e^n$), since an arbitrary number of
processors can share an edge. The first step is to accumulate and check all map edges
$(a'b')^j \ldots (a'b')^n$ for bisection feasibility, followed by an actual bisection operation for
all $(a'b')^j \ldots (a'b')^n$. Entries are then made into the \texttt{operations} list for each processor.
ReInitializeCell($cI$, $cLevel$, $L$)

**Inputs**
- Face buffer (RecvFaces), Level buffer (RecvLevel), Scale buffer (RecvScale), cells list (cells), face owner list (owner), face neighbour list (neighbour)

**Output**
- Re-initialized cell length scale ($sumL$)

```
sumL = 0.0, nNgb = 0, checkCell = cells[cI]
```

```c
for faceI = 0; size(checkCell) do
    facePatch is the boundary patch index for checkCell[faceI]
    if facePatch is interior then
        if owner[checkCell[faceI]] = cI then
            sLCell = neighbour[checkCell[faceI]]
        else
            sLCell = owner[checkCell[faceI]]
        sLevel = cLevel[sLCell]
        if sLevel < cLevel[cI] and sLevel > 0 then
            sumL = sumL + L[sLCell]
            nNgb = nNgb + 1
    else if facePatch is a processor patch then
        localFaceI is the local position of checkCell[faceI] in facePatch
        if localFaceI is found in RecvFaces[facePatch] then
            j is the index of localFaceI in RecvFaces[facePatch]
            rLevel = RecvLevel[facePatch][j]
            if rLevel < cLevel[cI] and rLevel > 0 then
                sumL = sumL + RecvLevel[facePatch][j]
                nNgb = nNgb + 1
        else if facePatch is a physical patch then
            sumL = bL[checkCell[faceI]]
            nNgb = nNgb + 1
    avgL = (sumL/nNgb)
    if level < maxLevel then
        avgL = $\gamma \cdot$ (avgL)
    else if $L_{mean} > 0.0$ then
        avgL = $L_{mean}$
return avgL
```

**Function** ReInitializeCell

edge. Finally, the point maps ($\mathcal{X}_p^j, \mathcal{R}_p^j$) are updated to include the new points introduced by bisection.

Parallel edge collapse, on the other hand, tends to introduce a few additional complications. In two dimensions, the simplest case is one where an edge (and both its points) is shared by exactly two processors. In this situation, the operation is similar to the edge-bisection case - an initial check for feasibility is performed, followed by the actual collapse operation and updates to the operations list and point / edge
maps. Occasionally, more complicated collapse situations can occur, such as the one depicted in Fig. 6.7. In this case, while edge $cd$ is shared by processors 0 and 3, point $c$ is additionally shared by processor 2. If a decision has been made by processor 0 to collapse edge $cd$, then an additional entry (to move point $\hat{c}$) has to be made in the operations list for processor 2.

An additional scenario can occur in certain situations, where an operation can require the reassignment of a processor patch-face to another patch. In Fig. 6.8, for example, collapsing edge $cd$ requires edge $\hat{d}\hat{e}$ to be converted from a patch communicating with processor 3, to a patch communicating with processor 2 (namely, edge $\hat{c}\hat{e}$). Also, if such a patch did not previously exist, then it must be created prior this step. If a patch creation / conversion operation is required, a corresponding entry is made in the operations list for respective processors. While updating point maps, the entries for $\hat{d}$ is reassigned to $c$, since point $d$ was deleted during the collapse process. All these cases also carry over to the 3D situation, but with the additional possibility of multiply-connected processor edges.
Figure 6.6. Parallel edge bisection in (a) 2D; and (b) 3D

6.3.5 Parallel Field Remapping

After parallel mesh reconnection, conservative field remapping using the methods described in Section 4.4 (in Chapter 4) tends to be complicated due to the fact that cells often migrate from one sub-domain to another. The task of finding intersections for elements adjacent to processor boundaries can sometimes become quite difficult, simply because all the required source-cells may not currently exist. This can be remedied, however, by recognizing that the halo meshes contain all the geometric information necessary for intersection calculations. In the advancing-front algorithm described in Section 4.4.3, if any of the source cells are found to be adjacent to processor boundaries, the halo meshes are also checked for intersections. The actual field variables needn’t be present for this process and so,
sub-sets of fields (corresponding to cells in halo meshes) can be transferred in the background using MPI while intersection calculations are being performed, thereby hiding communication latencies to a certain extent. Once all transfers have completed, the sub-domain fields are augmented with sub-fields corresponding to halo-meshes, and with some manipulation of the source-cell addressing, all fields can be remapped conventionally. Note that second-order conservative remapping requires cell-centered gradients, which must be transferred as well.

To conclude this section, the steps involved in the parallel reconnection algorithm are summarized. These steps are described from the stand-point of a particular sub-domain, since all sub-domains execute them in an identical sequence.
• Calculate a length scale estimate for the domain, using the modified methods described in Section 6.3.3.

• Identify neighbouring processors, including those that are connected only by points / edges.

• Construct halo meshes for each neighbouring sub-domain. Send and receive halo meshes.

• Once halo meshes have been transferred, build maps for entities (point maps $\mathcal{X}_p$ and $\mathcal{R}_p$, edge maps $\mathcal{X}_e$ and $\mathcal{R}_e$, etc).

• Send and receive length scale values for halo cells.

• Coupled modifications:
  
  – Initialize stack with processor-coupled edges on halo meshes sent to other sub-domains, avoiding those sent to lower-ranked processors.
  
  – Perform coupled-modifications using operations described in Section 6.3.2 and Section 6.3.4.
  
  – Schedule transfer of the operations list to higher-ranked processors.

• Re-initialize stack with non-halo edges.

• Perform modifications using operations described in Chapter 4.

• Synchronize coupled modifications:
  
  – Wait for transfer of the operations list from lower-ranked processors to complete.
  
  – Sequentially execute operations from list.

• Prepare and schedule transfer for sub-fields corresponding to halo meshes.
• Perform coupled intersection calculations using approach described in Section 6.3.5.

• Wait for sub-field transfers to complete, augment existing fields, and remap conventionally.

• Re-order boundaries so that processor-patches are matched both geometrically and topologically.

6.4 Parallel Mesh Smoothing

This section will discuss the various details of the mesh smoothing algorithms presented in Chapter 5, when extended to a distributed-memory paradigm. The topic of parallel mesh smoothing has been investigated by several researchers in the past. Freitag et al. [61] extended their local-optimization algorithm to a Parallel Random Access Machine (PRAM) computational model, which allows sub-domains to access other sub-domains in a shared-memory paradigm. Tsai et al. [152] used a transfinite interpolation (TFI) technique to smooth multi-block meshes in parallel, with a minimal amount of inter-processor communication. Jiao et al. [89] extended a feature-preserving surface-mesh smoother to a distributed memory model, and observed near-linear scaling for up to 128 processors using a Myrinet interconnect. Guoy et al. [73] used a halo mesh approach to extend the Mesquite Mesh Improvement Library to perform smoothing in parallel.

6.4.1 Parallel Surface Smoothing

Unlike the work by Jiao et al. [89], where surface-mesh vertices are moved in a localized manner, the approach taken in this work is to perform a global repositioning of surface vertices by extending the spring-analogy Laplacian smoothing method described in Chapter 5 to a distributed-memory paradigm. This is depicted by Fig. 6.9, which is repeated from Chapter 5 for convenience.
The relation governing the position of surface vertices is given by Eq. 6.1.

\[(I - nn^T) \sum_j k_{ij} \cdot (x_{ij} - x_i) = 0\]  

(6.1)

where \(k_{ij}\) represents a spring constant for an edge connecting vertices \(x_i\) and \(x_j\), and \(n\) represents the surface-normal vector defined at point \(x_i\). When the point-normals are treated explicitly, the relation in Eq. 6.1 yields a symmetric positive-definite matrix of coefficients which can be solved efficiently using the Conjugate Gradient method [79]. Since the algorithm itself is integral to the parallelization process, it is discussed here in further detail.

The Conjugate Gradient (CG) algorithm is a popular iterative method for solving systems of the form \(Ax = b\), where the matrix \(A\) is symmetric and positive-definite. Direct solvers like Gaussian elimination and LU decomposition techniques have the advantage of reusability, since the matrix \(A\) has to be factored only once in the solution process and is then applicable for multiple cases of \(b\). They are also less prone to round-off issues, as opposed to iterative techniques which gradually accumulate errors with increasing iterations. However, direct methods usually require the entire matrix to be stored in memory, and this becomes impossible for even moderately sized problems.
When \( A \) is sparse, factoring of such matrices generally tends to yield triangular factors that contain many more non-zero elements than the matrix \( A \) itself (Shewchuk [141]) and therefore, direct methods are no longer advantageous. Iterative techniques are generally more memory- and cost-efficient in these cases. Such systems frequently arise in the solution of discretized linear and non-linear partial differential equations.

In theory, the Conjugate Gradient algorithm is guaranteed to converge in \( N \) iterations, where \( N \) is the number of unknowns in the system. However, in practice, convergence is usually achieved at a much faster rate. The algorithm primarily consists of three operations that must be highly efficient for the solution to be competitive in terms of computational cost - reduction operations like a vector dot-product \( (\rho_{i+1} = r_{i+1} \cdot z_{i+1}) \), the axpy operation \( (x_{i+1} = x_i + \alpha p_i) \), and the sparse-matrix multiply \( (w_i = A p_i) \).

\[
\begin{align*}
\text{ConjugateGradient}(A, b, x_0) \\
r_0 &= b - Ax_0 \\
p_0 &= r_0 \\
\rho_0 &= r_0 \cdot p_0 \\
\text{for } i = 0, 1, 2, \ldots \text{ do} \\
&\quad w_i = Ap_i \\
&\quad \alpha = \rho_i / (p_i \cdot w_i) \\
&\quad x_{i+1} = x_i + \alpha p_i \\
&\quad \text{Exit if convergence criteria is satisfied} \\
&\quad r_{i+1} = r_i - \alpha w_i \\
&\quad \rho_{i+1} = r_{i+1} \cdot r_{i+1} \\
&\quad \beta_{i+1} = \rho_{i+1} / \rho_i \\
&\quad p_{i+1} = r_{i+1} + \beta_{i+1} p_i
\end{align*}
\]

\textbf{Procedure ConjugateGradient}

From the perspective of surface mesh-smoothing, the vector of unknowns \( (x) \) in the CG algorithm corresponds to the positions of mesh vertices \( (x_i) \), and the
number of unknowns \((N = 3n)\), where \(n\) is the number of surface-vertices. The effect of sub-domain distribution is apparent at only two locations in the algorithm - in the sparse-matrix multiply and the vector dot-product. The idea is to exchange and/or accumulate vector information (stored at vertices) across sub-domains without double-counting. An example of this situation is depicted in Fig. 6.10, where an arbitrary surface-mesh is shared by 3 processors. While a surface-edge can be shared by only two sub-domains, a surface-vertex can be shared by an arbitrary number of them, and the concept of point- and edge-markers is now introduced to deal with this situation. If \(r_v\) is the processor-rank for a particular sub-domain, and \(R_v\) is the set of processor-ranks for all processors sharing vertex \(i\), then the point-marker for the vertex \(\delta^i_p\) is defined:

\[
\delta^i_p = \begin{cases} 
1 & \text{if } r_v \text{ is less than all ranks in } R_v \\
0 & \text{if } r_v \text{ is greater than any rank in } R_v
\end{cases}
\] (6.2)

On similar lines, if \(r_a\) is the rank of sub-domain ‘a’ that shares an edge with another sub-domain ‘b’ with rank \(r_b\), then the edge-marker \(\delta^ij_e\), for an edge connecting points \(i\) and \(j\) is defined for sub-domain ‘a’ as:

\[
\delta^ij_e = \begin{cases} 
1 & \text{if } r_a < r_b \\
0 & \text{if } r_a > r_b
\end{cases}
\] (6.3)

This is also apparent from Fig. 6.10, where the point- and edge-markers for a particular sub-domain are marked 1 if the processor owns the entity, and 0 otherwise. Points and edges which are not shared by more than one processor automatically get marked 1, thereby defining the marker field for every surface vertex and edge.

The parallel version of the dot-product operation now becomes a slightly modified version of the original, now including an additional marker field \(\delta_p\). This is given
by the `ParallelDotProduct` function. The `SumReduce` function combines the value of its argument across all processors.

```plaintext
ParallelDotProduct(x, y, δ_p)
    dot = 0.0
    for i = 0, N do
        dot = dot + (δ_p[i] \cdot x[i] \cdot y[i])
    SumReduce(dot)
    return dot
```

**Function ParallelDotProduct**

The parallel version of the sparse-matrix multiplication is realized by slightly modifying the relation given in Eq. 6.1 to include the edge-marker field defined earlier. Note that averaging the normals for points on processor boundaries requires some communication, so that the correct estimate of \( \mathbf{n} \) is obtained.

\[
(I - \mathbf{n} \mathbf{n}^T) \sum_j \delta_{ij}^e \cdot k_{ij} \cdot (\mathbf{x}_{ij} - \mathbf{x}_i) = 0
\] (6.4)
Experience shows that this approach to parallel surface-mesh smoothing is both robust and computationally efficient for practical use.

### 6.4.2 Parallel Interior Mesh Smoothing

The approach taken by Gouy et al. [73] is used in this work for the parallel smoothing of interior vertices using Mesquite. The preparation of halo meshes for parallel smoothing is identical to the description given in Section 6.3.1, which is repeated in Fig. 6.11 for convenience. As shown in the figure, all points on the boundary of the sub-domain are held fixed (shown in red), while any shared processor points (shown in white) are allowed to be optimized by Mesquite. After the optimization process, all shared processor points are averaged to obtain the final position. Experience shows that this averaging procedure is sufficient to maintain mesh quality during the course of the simulation process, while being computationally efficient.

**Figure 6.11.** Shared processor points
CHAPTER 7

CASE STUDIES

This chapter discusses several test cases to demonstrate the efficiency and robustness of the mesh adaptation algorithms, and their general applicability to a variety of flows.

7.1 Validation Cases for Mesh Adaptation

To test the ability of the described mesh adaptation algorithms to handle large domain deformations, a variety of validation cases were used. All cases in this section do not involve fluid flow, and only serve the purpose of demonstrating the versatility of the method.

7.1.1 Rotating and Translating Box in a Rectangular Domain

The first test case shown in Fig. 7.1, involves a two-dimensional mesh with approximately 1000 cells, similar to the demonstrations by Baker [9]. The outer rectangle is 5 units wide, 2 units tall and 0.1 units deep. The inner box has a side of 0.5 units. The case was run for a total time of 5 units, with a time-step of 0.01. The inner box translates by a distance of 0.007 units and an angle of 0.25° per time-step. Mesh smoothing was performed at every time-step using the spring-based Laplacian method. Edge swapping ensures that the mesh is strictly Delaunay at every time-step, while adequate refinement is dictated by edge bisection and collapse operations based on the length scale field. Contour levels of the length scale field are shown in Fig. 7.2.
7.1.2 Rotating and Translating Ball in a Rectangular Domain

The next test case involves a three-dimensional tetrahedral mesh with approximately 50,000 cells, shown in Fig. 7.3. The outer rectangle is 5 units wide, 2 units tall and 2 units deep. The inner sphere has a diameter of 0.5 units. The case was run for a total time of 42 units, with a time-step of 0.1. The rotation and translation parameters were same as the 2D case. The minimum cell quality, as defined by the quality metric given by Eq.(4.6), was 0.48.

Interior mesh vertices were smoothed using the Mesquite Mesh Improvement library, while surface vertices were smoothed using the spring-based Laplacian
Figure 7.3. Rotating and translating ball in a rectangular domain

method. Mesh smoothing (both interior and surface vertices) accounted for 1.25 seconds at every time-step. Swapping and refinement accounted for about 0.25 seconds per time-step, using four threads on an Intel Core2 Quad CPU running at 2.83 GHz per core.

7.1.3 Internal Combustion Engine Case

To demonstrate the general applicability of the adaptation algorithm, the domain of a Mitsubishi internal combustion engine designed for Gasoline Direct Injection (GDI) was selected. The domain consists of highly complicated piston and manifold
geometries, with a pent-roof configuration and two intake valves, as shown in Fig. 7.4. The geometry is initially split into three parts, consisting of the two intake manifolds and the combustion chamber. Each part is then meshed separately with tetrahedra, amounting to a total of approximately 80,000 cells when the piston is at top-dead-centre (TDC). The manifolds are connected to the combustion chamber domain by means of a Generalized Grid Interface (GGI), developed by Beaudoin and Jasak [12].

![Figure 7.4. Geometry of the GDI engine](image)

The engine bore is approximately 80mm, while the piston stroke is 45mm, and the valve lift is 4mm. The case was run for 180 crank-angle degrees, with a time-step of 0.0001, so that the boundary motion at each time-step is sufficiently small to avoid invalid cells. At the bottom-dead-centre (BDC), the total number of cells increases to approximately 200,000. Despite the large number of cells, thread-parallel mesh adaptation (on average) accounted for roughly 1.25 seconds per time-step, while mesh
smoothing was restricted to 1.25 seconds at each step. The minimum cell quality, as defined by the quality metric given by Eq.(4.6), was 0.35. Cut-sections of the mesh at TDC and BDC during the simulation are shown in Fig. 7.5.

![Cut-sections of the mesh at TDC and BDC](image)

**Figure 7.5.** Cut-section of the mesh

### 7.1.4 Field-based Mesh Adaptation Case

The next test demonstrates the ability of the length-scale estimation algorithm to assist mesh refinement and smoothing in situations which involve a field-based criterion. Such situations commonly occur in Volume-of-Fluid (VOF) methods, error-based refinement and shock-capturing schemes, where a trade-off between computational cost and solution accuracy if sought. The test involves an initial uniform triangular mesh consisting of 14360 cells in a unit-square domain. To re-create a situation similar to a VOF method, a ‘bubble’ with radius 0.25 units is defined within the domain, and the mesh is allowed to adapt using a combination of smoothing, refinement, and length-scale algorithms defined in Chapter 4. The mean
length-scale \( (L_{\text{mean}}) \) was defined to be 0.03 units, with \( \alpha_l = 0.001 \), \( \alpha_h = 0.999 \) and \( L_\alpha = 0.005 \). After 10 adaptation cycles, the cell count in the mesh reduces to 5148, with a concentration of mesh density in the region representing the interface of the bubble, as shown in Fig. 7.6.

![Initial Mesh and Adapted Mesh](image)

**Figure 7.6.** Mesh adaptation using a refinement field

The cases demonstrated in this section show that the described adaptive mesh algorithms, in conjunction with smoothing methods, can efficiently handle simulations involving extreme deformation with acceptable cell quality.

### 7.2 Validation Cases for Free-Surface Flows

To validate the implementation of interface boundary conditions, several test cases are considered in this section. In situations where only the one fluid phase is considered (i.e., free-surface flows), the zero stress boundary condition for velocity and a fixed value condition for pressure is used. The effects of gravity are also neglected in all cases.
7.2.1 Droplet oscillations

A useful test to determine the accuracy of numerical surface-tension calculations is an oscillating droplet test. A viscous droplet behaves like a damped oscillator, with surface tension as a restoring force. Analytical solutions to the viscous decay constant \((t_c)\) and oscillation period \((T)\) are available from work by Lamb [102], and given by Eq. 7.1 for a single-phase.

\[
t_c = \frac{\rho r_0^2}{(n-1)(2n+1)\mu} \tag{7.1a}
\]

\[
T = \frac{2\pi}{\sqrt{n(n-1)(n+2)(\sigma/\rho r_0^3)}} \tag{7.1b}
\]

where \(n\) is the oscillation mode (taken to be the lowest mode: 2 in this case), \(r_0\) is the initial droplet radius, \(\mu\) is the dynamic viscosity, \(\rho\) is the liquid density, and \(\sigma\) is the surface tension. The amplitude should therefore decay according to the relation: \(A_n(t) = A_0 e^{-t/t_c}\). The simulation details for this case are given by Table. 7.1.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>(r_0)</td>
<td>(1.01 \times 10^{-3} \text{ m})</td>
</tr>
<tr>
<td>(r_a)</td>
<td>(1.03 \times 10^{-3} \text{ m})</td>
</tr>
<tr>
<td>(r_b)</td>
<td>(1.00 \times 10^{-3} \text{ m})</td>
</tr>
<tr>
<td>(\mu)</td>
<td>(1 \times 10^{-2} \text{ kg/m/s})</td>
</tr>
<tr>
<td>(\rho)</td>
<td>(1 \times 10^{3} \text{ kg/m}^3)</td>
</tr>
<tr>
<td>(\sigma)</td>
<td>(0.17 \text{ N/m})</td>
</tr>
</tbody>
</table>

Table 7.1. Simulation parameters for the droplet oscillation case

The initial droplet is a prolate spheroid with its major axis 2\% longer than a sphere of equivalent volume. With these parameters, the theoretical values for \(t_c\) and \(T\) are \(2.0402 \times 10^{-2}\) and \(5.46 \times 10^{-3}\), respectively. The obtained numerical values from the simulation are \(1.9477 \times 10^{-2}\) and \(5.06 \times 10^{-3}\), corresponding to relative errors of 4.5\% and 2.4\%, respectively. The amplitude decay history is given in Fig. 7.7, where
the x- and y-axes are scaled by time-constant \( (t_c) \) and spherical droplet radius \( (r_0) \), respectively.

![Figure 7.7. Amplitude decay history of the oscillating droplet](image)

For comparison, the theoretical time-period \( (T) \) is also plotted in Fig. 7.7, which should ideally coincide with the intersection of the curves corresponding to the oblate and prolate axis displacements.

### 7.2.2 Axi-symmetric Ink-jets

Ink-jet printing has matured to a level where it finds widespread use in several areas of modern technology, including printed electronics, microarray fabrication and large-scale paper-based printing. In the past, the development of systems employed for ink-jet printing relied largely on prototyping, which quickly proved to be expensive. The numerical study of droplet formation is therefore particularly attractive to
researchers who develop such technologies, since it contributes significantly towards shorter development cycles, in addition to the obvious cost benefits.

This section shows the results obtained from simulations subjected to several parametric variations. These cases involve only the liquid phase, and effects of the gas-phase, though significant, are neglected here. Unless stated otherwise, the fixed system conditions are as follows: Inlet jet velocity of 20 m/s; an orifice diameter of 8.8 \( \mu m \). This corresponds to a Weber number, \( (\text{We} = \rho v^2 L/\sigma) \), of 50.28, and a Capillary number, \( (\text{Ca} = \mu v/\sigma) \), of 0.285. Here, \( v \) is the characteristic velocity magnitude of the flow, \( L \) is the characteristic length scale, \( \mu \) is the dynamic viscosity, \( \rho \) is the fluid density, and \( \sigma \) is the surface tension.

Non-newtonian fluids exhibit viscosities that depend on the strain-rate of the fluid and so, corrections must be made to the viscosity as the simulation progresses. This kind of behaviour is reproduced fairly well by the Bird-Carreau model \([16]\), which is given by the constitutive relation in Eq.(7.2):

\[
\left( \frac{\eta - \eta_\infty}{\eta_0 - \eta_\infty} \right) = \left[ \frac{1}{1 + (\lambda \dot{\gamma})^2} \right]^{(1-n)}
\]  

(7.2)

The model contains four parameters: the zero strain-rate viscosity \( \eta_0 \), infinite strain-rate viscosity \( \eta_\infty \), viscoelastic time constant \( \lambda \), and a dimensionless constant \( n \) - which describes the slope of the power-law region of \( \log \eta \) vs. \( \log \dot{\gamma} \), where \( \dot{\gamma} \) is the strain-rate and is given by \( |\nabla \mathbf{v}| \). Here, the kinematic viscosity \( \eta \) is defined as \( (\mu/\rho) \) for convenience.

7.2.2.1 Pressure modulation

Drop-on-demand methods for inkjet printing always employ some form of pressure modulation to achieve droplets, including bubble-jets, piezoelectric, or diaphragm-based actuation. The quantity of ink dispensed from the orifice is dependent on the duration of the pressure pulse. Depending on the waveform of the actuating signal and
various fluid properties, satellite droplet formation can also occur. Satellite droplets are known to be a major cause of inferior print quality, and careful modulation of properties if often required to avoid them. This is depicted quite clearly in Fig. 7.8u, where the tail end of the drop eventually pinches off into a smaller droplet. The inlet is subjected to a fixed velocity of 20 m/s, which is subsequently switched off after 1.4 μs. At this point, surface tension causes necking to occur, and the drop eventually breaks off. Higher values of surface-tension accelerate the time to pinch-off. Higher shear-rates occur in the region around the orifice, causing minor shear-thinning effects with the Bird-Carreau model. This effect modifies the initial flow profile slightly, but does not show any significant changes in drop formation behavior. Non-Newtonian models which capture extensional behavior may influence satellite droplet formation, but their implementation is much more complicated.

### 7.2.2.2 Temperature modulation

High speed printing applications frequently require a continuous inkjet approach. In such cases, the formation of instabilities along the length of the jet cause it to break up into drops (and smaller satellite droplets). Most continuous inkjet methods employ thermal modulation to create spatial gradients of surface-tension. The resultant shear-stress at the interface results in Marangoni flow along the jet, eventually leading to instabilities and droplet formation. Temperature was modulated using a square wave profile. The percentage fraction of the heating pulse period relative to the modulation period is referred to as the duty cycle. In this work, with a period of 2μs, heat was applied at a 50% duty cycle (i.e., for 1μs). Fig. 7.8v shows the evolution of a continuous ink-jet subjected to thermal modulation in the range [300K-350K], set at a frequency of 500 kHz. The sequence depicts stages at approx. 0.2μs intervals, starting at 0.7μs after start of injection. Temperature contours depicting the thermal modulation is shown in Fig. 7.9, shown at 5.5μs after start of injection. Choosing
Figure 7.8. Stages of inkjet evolution
larger values for $\beta$ (in Eq.(2.24)) leads to larger spatial gradients of surface-tension along the jet, leading to a shorter break-off length (BOL). Shorter BOL can also be achieved by increasing the frequency of modulation.

![Temperature profile](image)

**Figure 7.9.** Thermal modulation of a continuous inkjet

### 7.2.3 Collapsing Ligament

The case of a collapsing ligament is of considerable significance in spray modeling. In most situations, the edges of liquid sheets frequently break up into ligaments due to instabilities, followed by break up into droplets. Depending on the length and fluid viscosity, the ligament can either coalesce into a single drop, or pinch off into several drops. This test considers the former case, where a stationary ligament collapses into an oscillating droplet under the influence of surface tension. The Ohnesorge number, $Oh$, given by the relation $\frac{\mu}{\sqrt{\rho L\sigma}}$, for this case is 0.1091, while the characteristic time, $t_c$, with initial radius $r_0$, given by the relation $\sqrt{\frac{\rho r_0^3}{\sigma}}$, is $3.78 \times 10^{-9} s$. The sequence of images at various dimensionless times during the simulation is shown in Fig. 7.10, where vectors denote surface velocity.

![Collapsing ligament images](image)

**Figure 7.10.** Collapsing ligament case ($Oh = 0.1091$ and $t_c = 3.78 \times 10^{-9} s$)
7.2.4 Binary Droplet Collisions

The phenomenon of binary droplet collisions are of significant interest for studies involving spray characteristics. A relevant example would be the evolution of a fuel spray in the combustion chamber of reciprocating engines, where the dense nature of droplet concentrations downstream from the injector lends to frequent droplet collisions that can significantly impact spray development and combustion. Droplet collision cases are also a fairly rigorous test of the re-meshing scheme’s ability to handle free-surface cases with significant magnitudes of mesh deformation. Jiang et al. [88] defines the collision Weber ($We$) and Reynolds ($Re$) number by the relations given in Eq. 7.3.

\[ We = 4 \rho_l d U_0^2 / \sigma \]  
\[ Re = 2 \rho_l U_0 d / \mu \]

where $\rho_l$ is the liquid density, $U_0$ is the initial relative droplet velocity, $\mu$ is the dynamic viscosity of the liquid, and $\sigma$ is the surface tension. Also of interest is the impact parameter $B = \chi / d$, where $\chi$ is the projection of the separation distance between droplet centres in the direction normal to $U_0$. Thus, values of $B = 0$ and $B = 1$ correspond to head-on and grazing collisions, respectively. Qian and Law [131] investigated the effects of Weber number ($We$) and impact parameter ($B$) on the binary collision of equal-sized hydrocarbon droplets, and classified outcomes into separate regimes which indicate either coalescence, separation, or even bouncing. More importantly, their work also serves as a practical benchmark for CFD validations.

7.2.4.1 Head-on Droplet Collision

The first case considered here is that of head-on droplet collision for two equal-sized droplets. The simulations parameters are given in Table. 7.2.
Table 7.2. Simulation parameters for Head-on Droplet Collision

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$d$</td>
<td>$3.8 \times 10^{-4} \text{ m}$</td>
</tr>
<tr>
<td>$\mu$</td>
<td>$3.88 \times 10^{-3} \text{ kg/ms}$</td>
</tr>
<tr>
<td>$\rho$</td>
<td>$758 \text{ kg/m}^3$</td>
</tr>
<tr>
<td>$\sigma$</td>
<td>$0.075 \text{ N/m}$</td>
</tr>
<tr>
<td>$U_0$</td>
<td>$2.0 \text{ m/s}$</td>
</tr>
<tr>
<td>$B$</td>
<td>$0.0$</td>
</tr>
</tbody>
</table>

This corresponds to a Weber number of 61.4, and a Reynolds number of 296.5, as given by Eq. 7.3. This would roughly correspond to the collision case (h) from Qian and Law [131], as shown in Fig. 7.11.

To avoid complications with the change in interface topology during droplet coalescence, the initial configuration is set such that the droplets touch with a small liquid bridge between them. The sequence of images shown in Fig. 7.12 shows results of the various stages in the collision process, up to the point where the fluid pinches off into three separate droplets. Depending on the Weber / Reynolds number, the colliding drops may form either a thin membrane (as shown in Fig. 7.12(d)), or break-up at the centre, thus forming a donut shape that eventually merges back to the configuration in Fig. 7.12(f).
Figure 7.12. Head-on Droplet Collision

7.2.4.2 Off-centre Droplet Collision

The next simulation depicts an off-centre droplet collision between two equal-sized droplets. The simulations parameters are given in Table 7.3.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>3.8 × 10^{-4} m</td>
</tr>
<tr>
<td>μ</td>
<td>2.16 × 10^{-3} kg/ms</td>
</tr>
<tr>
<td>ρ</td>
<td>758 kg/m^3</td>
</tr>
<tr>
<td>σ</td>
<td>0.026 N/m</td>
</tr>
<tr>
<td>U_0</td>
<td>1.17 m/s</td>
</tr>
<tr>
<td>B</td>
<td>0.68</td>
</tr>
</tbody>
</table>

Table 7.3. Simulation parameters for Off-centre Droplet Collision

This corresponds to a Weber number of 60.8, and a Reynolds number of 313.7, as given by Eq. 7.3. This would correspond to the collision case (o) from Qian and Law [131], as shown in Fig. 7.13.

Due to the larger impact parameter of this case, the coalesced drops attain sufficient angular momentum to break-off into smaller droplets. The actual separation of droplets is not considered in these cases, so the simulation is halted when cells in the neck region have become too small to continue.
Figure 7.13. Case (o) from Qian and Law [131]

Figure 7.14. Off-centre Droplet Collision

Figure 7.15. Contours of length scale for off-centre droplet collision
Contours of the length scale field for the off-centre droplet collision case is shown in Fig. 7.15, which clearly demonstrates the ability of the length estimation algorithm (from Section 4.2.1 in Chapter 4) to account for detailed features of the free-surface while coarsening cells towards the mesh interior. By comparison, the numerical results show good qualitative agreement with the image sets from Qian and Law.

7.2.5 Drop-on-demand Inkjet in 3D

Similar to the axi-symmetric case, a drop-on-demand inkjet is simulated in 3D. The fixed system conditions are as follows: Inlet jet velocity of 20 m/s, which at 0.2 µs is ramped down to 0 m/s at 0.8 µs; an orifice diameter of 8.8 µm. This corresponds to a Weber number, \( We = \rho v^2 L / \sigma \) of 50.28, and a Capillary number, \( Ca = \mu v / \sigma \) of 0.285. Here, \( v \) is the characteristic velocity magnitude of the flow, \( L \) is the characteristic length scale, \( \mu \) is the dynamic viscosity, \( \rho \) is the fluid density, and \( \sigma \) is the surface tension. The sequence of images in Fig. 7.16 show the evolution of the jet at 0.5 µs intervals, and coloured by velocity magnitude.

![Figure 7.16. Drop-on-demand inkjet in 3D](image)
7.3 Validation Cases for Parallel Mesh Adaptation

To test the ability of the described mesh adaptation algorithms to handle large domain deformations in distributed-parallel configurations, a variety of validation cases were used. All cases in this section do not involve fluid flow, and only serve the purpose of demonstration.

7.3.1 Translating Circle in a Rectangular Domain

The first test case shown in Fig. 7.17, involves a two-dimensional mesh with 1362 cells, similar to the demonstrations by Baker [9], decomposed into 4 sub-domains. The outer rectangle is 5 units wide, 2 units tall and 0.1 units deep. The inner circle has a diameter of 0.5 units. The case was run for a total time of 3.5 units, with a time-step of 0.01. The inner box translates by a distance of 0.01 units per time-step. Mesh smoothing was performed at every time-step using the parallel spring-based Laplacian method. Edge swapping ensures that the mesh is strictly Delaunay at every time-step, while adequate refinement is dictated by edge bisection and collapse operations based on the length scale field. The sequence of images in the left column clearly shows the sub-domain boundaries becoming excessively distorted over time, due to the lack of a dynamic load-balancing scheme.

The growth of inter-processor boundary sizes tend to increase communication costs, while imbalanced sub-domain sizes increases processor idle-time. This particular case has a fairly large ratio of processor boundary-faces to sub-domain cells (called a surface-to-volume ratio, $R_{SV}$), which tends to make it fairly inefficient in parallel. The sequence of images on the right column show results of the simulation when dynamic load-balancing is invoked at every 50th time-step, using the ParMetis parallel graph-partitioning library [139]. The re-partitioning step clearly improves the $R_{SV}$ ratio. Contour levels of the length scale field are shown in Fig. 7.18, where individual sub-domains are separated for clarity.
Figure 7.17. Translating circle in a rectangular domain

Figure 7.18. Length scale contours for the parallel 2D test case
7.3.2 Internal Combustion Engine Case

The next test case is slightly more demanding from a computational perspective. The initial domain is identical to the GDI engine from Section 7.1.3, and is partitioned into four sub-domains, as shown in Fig. 7.19. At the start of the simulation, individual sub-domains have approximately 17,000 cells per processor, which increases to about 45,000 per processor at the mid-point, when the piston is at bottom-dead-centre. The mesh is dynamically repartitioned at an arbitrarily chosen interval of 200 time-steps.

Figure 7.19. Parallel Decomposition of the GDI engine, showing halo meshes

Fig. 7.20 shows the cell count history for each processor during the course of the simulation. The re-distribution steps can be clearly distinguished by the jumps in cell count as the simulation progresses. It is also interesting to note is that the histograms are not symmetric about the mid-point of the simulation, but stays roughly constant during the second half, with a rapid decrease in cell count towards the end. This is probably because the mesh smoother continuously maintains mesh quality to the point where edge contractions are deemed necessary to account for the reduced domain size at top-dead-centre.

The total running time of the mesh adaptation on four processors was 5.7 hours, using a four core Intel Core2 Quad CPU running at 2.83 GHz per core. Due to the
absence of network hardware in this simulation, this is an unfair comparison, because it does not realistically exhibit communication latencies. An equivalent serial run, using identical simulation parameters, on one processor core took 11.45 hours. A cell-count history of the serial run is given in Fig. 7.21.

The itemization of individual costs is for the serial run is tabulated in Table. 7.4. A substantial portion of run-time is spent smoothing the mesh, but this fraction of the total cost is expected to reduce when a flow-solver is introduced, since a bulk of the solution cost is typically attributed to the pressure solver.
<table>
<thead>
<tr>
<th>Operation</th>
<th>Time (s)</th>
<th>Cost (% of total)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mesh smoothing</td>
<td>34237</td>
<td>83.03%</td>
</tr>
<tr>
<td>Mesh reconnection</td>
<td>5912.66</td>
<td>14.33%</td>
</tr>
<tr>
<td>Remapping</td>
<td>30.55</td>
<td>0.07%</td>
</tr>
<tr>
<td>Mesh re-ordering</td>
<td>1052.51</td>
<td>2.55%</td>
</tr>
<tr>
<td>Total</td>
<td>41232.99</td>
<td>100.00%</td>
</tr>
</tbody>
</table>

**Table 7.4.** Itemization of components for the serial run

A similar trend can be observed in the parallel run. The remapping step is slightly more time-consuming due to additional intersection checks against halo meshes. The re-ordering step also consumes a higher percentage of the total cost, since processor patches have to be matched both geometrically and topologically after mesh reconnection, thereby requiring communication.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time (s)</th>
<th>Cost (% of total)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mesh smoothing</td>
<td>16898.22</td>
<td>81.89%</td>
</tr>
<tr>
<td>Mesh reconnection</td>
<td>2704.87</td>
<td>13.11%</td>
</tr>
<tr>
<td>Remapping</td>
<td>205.34</td>
<td>0.99%</td>
</tr>
<tr>
<td>Mesh re-ordering</td>
<td>824.58</td>
<td>4.01%</td>
</tr>
<tr>
<td>Total</td>
<td>20633.01</td>
<td>100.00%</td>
</tr>
</tbody>
</table>

**Table 7.5.** Itemization of components for the parallel run

The sequence of images in Fig. 7.22 show the evolution of the mesh at various stages during the simulation for a total of 360 crank-angle degrees, where the colours represent sub-domain processor IDs. The inlet valves follow a sinusoidal lift-profile from $0^\circ$ to $180^\circ$, with a maximum lift of 4mm, and valve opening / closure events are not currently handled in this simulation.

### 7.3.3 Parallel Speed-up Tests

An important consideration for the mesh-adaptation algorithm is to assess how well it scales with an increase in processor count. To demonstrate this, the test case
of a solid sphere (with a radius of 0.1 units) translating through a cubical domain (of side 1 unit) is used. The initial domain is meshed with 1,562,629 tetrahedral cells, and subsequently split into sub-domains ranging between 4, 8, 16 and 32 processors. The timings reported in this section were obtained using the QueenBee supercomputer from the Louisiana Optical Network Initiative (LONI), which is a part of the Teragrid project. The cluster consists of 668 compute nodes running RedHat Enterprise Linux, with each node containing dual Quad Core Xeon 64-bit processors operating at 2.33 GHz with 8 Gb of RAM. All nodes are connected using a 10 Gb/sec Infiniband network interface, using the MVAPICH-2 MPI implementation.

Statistics for the simulations were collected over a period of 20 time-steps, where mesh-adaptation was used both with and without an incompressible flow-solver. Both timings are plotted against processor-count in Fig. 7.23. The mesh adaptation cost is observed to be approximately 60% of the total simulation time, of which about 80% is spent in the mesh smoothing process. Using the 4-processor case as a baseline, the speed-up for both adaptation and flow-solver is plotted in Fig. 7.24, and for adaptation alone in Fig. 7.25. Note that the ideal slope using 4-processors is now 0.25, as opposed to the conventional slope of 1.0. Interestingly, the 8- and 16-processor cases show a speed-up that is better than ideal, possibly owing to the large L2 cache sizes on the Xeon processor. The 32-processor case, however, shows the effect of increased communication costs, since each sub-domain now consists of approx. 48,000 cells, with a comparable size of processor boundaries.
Figure 7.22. Parallel Reconnection: Stages of mesh evolution
Figure 7.23. Solution time vs. processor count for 20 time-steps

Figure 7.24. Speed-up for adaptation and flow-solver

Figure 7.25. Speed-up for adaptation
CHAPTER 8
CONCLUSIONS AND FUTURE WORK

The research presented in the preceding chapters of this thesis demonstrate several notable advances in the area of mesh adaptivity. While the intent of this effort is for the numerical simulation of free-surface flows, a fairly comprehensive framework has been implemented in this work, which combines local adaptation with mesh smoothing methods to simulate a wide variety of cases.

The local nature of the adaptation process allows interpolation errors to be contained in regions undergoing connectivity changes, and is computationally efficient when compared to global mesh regeneration. Delaunay flipping is used in two-dimensional meshes to maintain cell-quality in situations involving higher degrees of mesh deformation, while a local optimization-based approach (proposed by Shewchuk [140] and Dai [38]) is used in 3D.

While there is no guarantee that the local approach will yield a globally optimum solution to mesh quality, it has been found to be sufficient for all the cases simulated in this work. The adaptation procedure also allows for local refinement and de-refinement of cells, so that mesh density is preferentially directed towards regions that require higher solution accuracy, while being competitive in terms of computational cost. In particular, the unique aspects of this dissertation can be summarized as follows:

- To direct the local refinement algorithms towards appropriate locations in the mesh, a length scale estimation algorithm has been devised in this work, which is both efficient and well-suited for parallelization.
• In rare situations where the adaptation algorithms are unable to deal with nearly degenerate cells in 3D, a sliver detection / removal algorithm (adapted from work by Li, Shephard and Beall [105]) is used.

• A generalized, second-order conservative remapping scheme has been introduced to handle the interpolation of solution fields after mesh reconnection, using an extension of the supermesh approach by Farrell et al. [57] for cell-centered finite volume variables. While the application of the scheme in this work is more localized in nature, the procedure has been demonstrated to successfully remap fields between arbitrary polyhedral meshes with second-order accuracy.

• To minimize the frequency of mesh reconnection, mesh smoothing algorithms have also been incorporated into this work. In two dimensional meshes, a smoother based on the spring-analogy Laplacian is used to effectively maintain mesh-quality. In 3D, mesh vertices are continually optimized using the Mesquite Mesh Improvement library from Sandia National Labs [22]. Vertices on curved surfaces are optimized using a Laplacian smoother in combination with an untangling algorithm. An optional optimization-based approach is also proposed to effectively smooth surface point positions, but at a much higher computational cost.

• Despite the significant complexity involved, all the mesh reconnection and smoothing algorithms used in this work have also been successfully extended to work in both shared- and distributed-memory parallel configurations, so that simulations are no longer limited by solution time or domain size. Unlike previous efforts in the field by Cavallo et al. [27], the approach in this work involves a local cell-migration approach that aligns well with the nature of the reconnection algorithms used.
• When sub-domain cell distributions are insufficiently balanced, a global redistribution is performed to minimize idle-time and communication costs. Since the redistribution procedure has the potential of being computationally expensive, it is performed only at occasional intervals, while relying on the smoothing and local reconnection algorithms to maintain cell quality.

8.1 Future Work

A few possible directions can be proposed for future work in the area:

8.1.1 Two-phase Flows

While demonstrations in this work have been primarily in the area of free-surface simulations, the mesh adaptation techniques described in Chapter 4 can clearly be extended to involve two-phase flows. Multi-phase flows are notoriously difficult to simulate numerically, mainly owing to the large variations in physical properties at the fluid interface. While the framework for this currently exists, the stability of the approach (particularly with mesh-reconnection), is an important consideration. The successful implementation of two-phase simulations permits separation and coalescence regimes to be modeled using the interface tracking approach, as shown by Quan et al. [132]. The introduction of a second phase automatically doubles the cost of the numerical simulation, which immediately brings forth the topic of parallelization. With that aspect already considered in this work, the task of extending the current scheme to two-phase flows is expected to be less daunting.

8.1.2 Divergence-free Remapping

While the issue of field remapping for cell-centred finite volume variables has been addressed in Chapter 4, divergence-free face flux transfer is achieved by using a Poisson pressure-correction step. While it is often sufficient, this approach can unnecessarily slow down solution times, thereby forcing a reduction in the frequency
of mesh reconnection operations during the course of the simulation. Since all reconnection operations in this work are local in nature, an attractive alternative would be to locally correct fluxes on newly introduced faces, while leaving the fluxes on existing faces untouched. A divergence-free remapping approach essentially imposes a Lagrangian constraint on the transfer process. Thus, one possible approach, suggested by Farrell [56], would be to consider the Helmholtz-Hodge decomposition to transfer a divergence-free source field $v_s$ to a target field $v_t$ via:

$$M_t v_t = M_s v_s + G \varphi$$ (8.1)

where $M_t$ and $M_s$ are diagonal mass-matrices, $G$ is the discrete gradient operator, and $\varphi$ is a scalar potential. Some relevant work in the area has also been attempted by Bochev and Shashkov [19], but their work is limited to two-dimensional cartesian grids.

### 8.1.3 Polyhedral Mesh Adaptation

A vast majority of research in the past (including this work) have primarily focussed on the use of adaptivity techniques that are tailored towards simplical two- and three-dimensional grids. While this has proven to be sufficient in a large variety of cases, simplical grids usually tend to be numerically diffusive and so, increased mesh-resolution is often necessary to counter these effects. In this regard, polyhedral meshes are particularly attractive because they provide higher accuracy (when compared to simplical grids) for similar mesh resolutions, and align very well with the finite volume paradigm, which places no restrictions on cell topology. This generality, however, comes with the cost of higher algorithmic complexity.
8.2 Closure

The generality of the algorithms used in this work is demonstrated using several test cases involving meshes with arbitrary boundary motions and free surface flows. Deforming domain problems are vastly simplified using this framework, where only an initial mesh and a description of the moving boundaries are required, and no further user intervention is necessary. The extension of these algorithms to parallel paradigms considerably expands the ability of the described CFD methods to provide rapid solutions to problems with increased accuracy. While most of the cases considered in this work are transient in nature, these algorithms can be applied to facilitate other aspects of CFD like grid-refinement studies on steady-state calculations, and error-based mesh adaptation. All the work described in this thesis has been included in the OpenFOAM continuum mechanics library, which is free and open-source. It is hoped that the algorithms devised in this work will be applied to a variety of situations involving dynamic meshes, far beyond the scope of original intent.
BIBLIOGRAPHY


182


