A Discrete Exterior Calculus Finite Element Method for Solving Two Phase Flow Problems

by

Peter Klimas

A Thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfilment of
the requirements for the degree of
Master of Applied Science

Ottawa-Carleton Institute for
Mechanical and Aerospace Engineering

Department of Mechanical and Aerospace Engineering
Carleton University
Ottawa, Ontario, Canada

May 2009

Copyright ©

2009 - Peter Klimas

Library and Archives
Canada

Published Heritage
Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et
Archives Canada

Direction du
Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file *Votre référence*
ISBN: 978-0-494-60260-7
Our file *Notre référence*
ISBN: 978-0-494-60260-7

NOTICE:

The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protègent cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

■ ■ ■
Canada

Abstract

The understanding of bubble nucleation, growth, and collapse has many practical applications ranging from nuclear to naval and even space. For example, high efficiency nuclear stations, which include cooling systems with high pressure two-phase flows, could be improved through the modeling of vapour-gas interactions. Because of this widespread applicability, ongoing research to develop efficient, high-accuracy algorithms and software to solve complex simulation scenarios of three-dimensional vapour bubble interactions with their surrounding fluid has significant implications.

This thesis proposes an efficient approach to solving flow problems accurately (with an emphasis on two-phase flow) by applying discrete exterior calculus to a vorticity based method. To solve flow problems computationally, they must be resolved at a discrete level with minimal loss in accuracy. As discrete exterior calculus applies differential and integral calculus of vector functions to a discrete model, it is ideally suited to building discrete mathematical operators such as Grad, Curl, Div and Laplace directly, resulting in sparse matrix operators that are computationally efficient. The proposed approach bridges the gap between the engineering discipline and discrete exterior calculus, a commonly overlooked mathematical field that is ideally suited for solving complex problems in a discrete domain. Furthermore, through the use of the vorticity formulation of the Navier-Stokes equations, this discrete model intrinsically preserves angular momentum.

Acknowledgments

I would like to acknowledge my supervisors Dr. Kaya and Dr. Goldak for their patience and direction while working on my thesis. I would also like to thank my parents Barbara and Stan, my sister Maria, and of course my wife Katerina for their unconditional support throughout my university career.

Table of Contents

Abstract	iii
Acknowledgments	iv
Table of Contents	v
List of Figures	x
Acronyms	xii
Nomenclature	xiv
1 Introduction	1
1.1 Background	1
1.2 Research Problem	3
1.3 Purpose and Objectives	4
1.4 Thesis Organization	5
2 A Review of Current Approaches to Solving Multi-Phase Flow Problems	6
2.1 Introduction	6
2.2 Brief Review of Fluid Dynamics	6
2.2.1 Approximate Analytical Solutions	8

2.2.2	Multi-phase problems	9
2.3	Computational Fluid Dynamics	10
2.3.1	Form of the Governing Equations	10
2.3.2	Discretization Schemes	11
2.3.3	Reference Frame	13
2.3.4	Interface Tracking	14
2.3.5	Numerical Solutions	14
2.4	Geometry-based Discrete Exterior Calculus	15
3	Analytical Framework	17
3.1	Mathematic Definitions and Derivations	17
3.2	Geometric Algebra	23
3.3	Differential Geometry	24
3.3.1	The Differential Structure	24
3.3.2	Introduction to the Underlying Structure of DEC	25
4	Governing Equations of Fluid Dynamics	27
4.1	Conservation of Mass	27
4.2	Conservation of Momentum	28
4.3	Conservation of Energy	30
4.4	Direct Modelling of Boundary and Interface Conditions	31
4.5	Simplified Spherical Bubble Model	33
4.5.1	Simplifying Assumptions	33
4.6	Vorticity Formulation	35
5	Numerical Approach	36
5.1	Finite Element Analysis with Discrete Exterior Calculus	37
5.1.1	Discrete Domain	38

5.1.2	Primal Complex	41
5.1.3	Dual Complex	42
5.1.4	Discrete Exterior Derivative	44
5.1.5	Hodge Star	46
5.1.6	Discrete Curl	47
5.1.7	Discrete Laplacian	47
5.1.8	Summary of Operators	48
5.2	Decomposition of the Flow Field	49
5.2.1	Helmholtz-Hodge Decomposition	49
5.3	Time Step Advection	51
5.4	Solution Methodology	51
5.4.1	Time-step Advection Loop	53
5.5	Boundary Conditions	54
5.6	Flow Discontinuities On Boundaries	55
5.6.1	Possible Solutions To Avoid Flow Discontinuities	55
5.7	Interface Tracking	56
5.7.1	The Level-Set Method	57
5.7.2	Signed Distance Function	58
6	Results	61
6.1	Stefan Problem	61
6.1.1	Conservation of Energy	62
6.1.2	Conservation of Mass	63
6.1.3	Stefan Problem Results	65
6.2	Sucking Problem	67
6.2.1	Conservation of Energy	68
6.2.2	Conservation of Mass	69

6.2.3	Interface Conditions	69
6.2.4	Sucking Problem Results	70
6.3	Prototype DEC Solver Results	71
6.3.1	Couette Flow	73
6.3.2	Poiseuille Flow	77
6.4	Vr Fluid Flow DEC Solver	77
7	Conclusions	87
	List of References	90
	Appendix A Examples of the DEC Machinery	94
A.1	Exterior Derivative	94
A.2	Primal and Dual Volumes	98
A.3	Hodge Star	100
A.4	Gradient, Curl and Divergence	103
A.5	Laplacian	106
	Appendix B Transcendental Equation Solution	108
	Appendix C 1D Bubble Analysis	110
C.1	Summary of simplifying assumptions	110
C.2	Formulation of governing equations	110
C.2.1	Equilibrium condition	113
C.2.2	The Rayleigh solution	114
C.2.3	Cooling effect	115
C.2.4	Solution to the equation of motion	117
	Appendix D Prototype DEC Solver Code	119
D.1	Main Function	119

D.2	DEC Operator Construction	135
D.2.1	Exterior Derivative	135
D.2.2	Hodge Star	138
D.3	DEC Specific Functions	142
D.4	Mesh Construction and Operation Functions	177
D.5	FEM Interpolation Functions	195
D.6	Output Functions	202

List of Figures

1.1	Sketch by Leonardo da Vinci of a spiralling bubble [1] and [2].	2
4.1	Problem domain and boundary interface	31
5.1	Primal simplices	41
5.2	Primal and corresponding dual simplices	43
5.3	Primal and dual mesh	43
5.4	2D mesh example	45
5.5	Summary of operators	48
5.6	Tangential velocity components	55
5.7	Flow discontinuities at vertices arising from using a linear flow profile on each face	56
5.8	Possible methods of avoiding flow discontinuities at vertices	57
6.1	Definition of the “Stefan” problem	62
6.2	Temperature profile for the analytical solution of the Stefan problem	65
6.3	Stefan problem interface location at 1600s	66
6.4	Analytical and numerical solutions for the interface position of the Stefan problem	66
6.5	Definition of the Sucking problem	67
6.6	Initial conditions of the Sucking problem	68
6.7	Numerical solution for the interface position of the Sucking problem .	72
6.8	Numerical solution for the interface velocity of the Sucking problem .	72

6.9	Prototype Scilab DEC solver configuration	74
6.10	Couette flow schematic	74
6.11	Couette velocity results from prototype DEC solver	76
6.12	Poiseuille vorticity results from prototype DEC solver	78
6.13	Poiseuille velocity results from prototype DEC solver	79
6.14	Poiseuille particle advection results from prototype DEC solver	80
6.15	Vr Fluid Flow solution domain with mesh	82
6.16	Couette flow vorticity	83
6.17	Couette flow velocity	84
6.18	Poiseuille flow vorticity	85
6.19	Poiseuille flow velocity	86
B.1	Graphical solution of transcendental equation	109

Acronyms

<i>ALE</i>	Arbitrary Lagrangian-Eulerian
<i>CFD</i>	Computational Fluid Dynamics
<i>DEC</i>	Discrete Exterior Calculus
<i>XFEM</i>	Extended Finite Element Method
<i>FDM</i>	Finite Difference Method
<i>FEM</i>	Finite Element Method
<i>FLIC</i>	Fluid-In-Cell
<i>FVM</i>	Finite Volume Method
<i>GFEM</i>	Generalized Finite Element Method
<i>GFM</i>	Ghost Fluid Method
<i>LSM</i>	Level-Set Method
<i>MAC</i>	Marker-and-Cell
<i>PDE</i>	Partial Differential Equation
<i>PIC</i>	Particle-In-Cell

PUM   Partition of Unity Method

Nomenclature

μ	dynamic viscosity $\left[\frac{kg}{m \cdot s}\right]$
ρ	density of fluid $\left[\frac{kg}{m^3}\right]$
Γ	fluid boundary or interface
Ω_A	fluid A domain
Ω_B	fluid B domain
C_p	specific heat at constant pressure $\left[\frac{kJ}{kg \cdot K}\right]$
$\mathbf{I}$	identity matrix
$\hat{n}$	normal vector to fluid interface
A	area $[m^2]$
T	temperature $[K]$
V	volume $[m^3]$
J	Jacobian matrix
d	discrete derivative
h	harmonic component of the flow field

t	time [s]
x	location in domain [m]

Discrete Exterior Calculus

$\flat$	discrete flat operator
$\sharp$	discrete sharp operator
$\star$	Hodge star
$\wedge$	wedge product
β	coefficient of expansion
ω/Ω	vorticity vector/discrete vorticity
ϕ/Φ	vector potential/discrete vector potential
v/U	velocity vector $[\frac{m}{s}]$ /Discrete flow $[\frac{kg}{s}]$
ψ/Ψ	scalar/scalar
σ	primal k-simplex or element
$\tilde{\sigma}$	dual k-simplex
k	0/1/2/3-form

Chapter 1

Introduction

1.1 Background

The study of two-phase flow dates back to Leonardo da Vinci (1452 - 1519) in his studies on “Movement in Water and Air”. He observed: [1]:

The air which is submerged together with the water ... returns to the air, penetrating the water in sinuous movement ... And this occurs because the light thing cannot remain under the heavy ... ; and because the water that stands there perpendicular is more powerful than the other in its descent, this water is always driven away by the part of the water that forms its coverings, and so moves continually sideways where it is less heavy and in consequence offers less resistance ... And because this has to make its movement by the shortest way it never spreads itself out from its path except to the extent to which it avoids that water which covers it above.

This sinuous motion, now known as Leonardo’s Paradox, was sketched by Leonardo and is shown in Figure 1.1. Leonardo’s Paradox is still not fully understood and is a continuing area of research [2].

Figure 1.1: Sketch by Leonardo da Vinci of a spiralling bubble [1] and [2].

As demonstrated by the sinuous motion of bubbles, many intricate phenomena play a role in the movement of vapour bubbles. As many engineering applications rely on the benefits of two-phase flow, mathematic modelling is of paramount importance. Engineering disciplines such as heat transfer, which deals with transferring thermal energy from one point to another, commonly employ phase transitions to reap the benefits of the latent heat of vaporization. One does not have to look far for examples of using latent heat to aid in heat transfer; even nature readily uses this phenomenon during perspiration.

The proper study of two-phase flow requires consideration of the motion of each fluid. A fluid is characterized by the relative motion of a substance's molecules. Similarly to how the stress in a solid body is proportional to strain (deformation per unit length), the viscous stress in a fluid is proportional to the time rate of strain (rate of deformation), and is commonly characterized by the viscosity.

Flow models are usually based on Euler or Navier-Stokes equations which arise from Newton's second law of physics

$$F = ma \tag{1.1}$$

Unfortunately, aside from some special cases, analytical solutions to the Euler and Navier-Stokes equations have resisted closed-form solutions and are still a study of interest in the mathematics community. For example, the lack of solutions to these equations are listed as one of seven classical problems that have resisted solutions by The Clay Mathematics Institute of Cambridge, Massachusetts. The Clay Mathematics Institute considers these problems of such high interest that it designated a \$7 million prize, \$1 million for each problem, to entice continuous research in this area [3].

Since the advent of computers, solutions to the Euler and Navier-Stokes equations have been commonly approximated through numerical solutions and are considered in the realm of computational fluid dynamics (CFD). Many engineering disciplines, ranging from aerospace to electronics, commonly apply CFD when striving for better and more efficient designs. Because of this widespread use, ongoing research to develop algorithms and software that improves the accuracy and speed of complex simulation scenarios has significant implications.

The addition of two phases further complicates the analysis, and therefore requires great care when modelling mathematically. A two-phase liquid/vapour bubble flow can be modelled as two immiscible fluids with property discontinuities across the fluid-bubble interface. Boundary effects, such as surface tension acting on the interface, play a large role in governing the bubble deformation, while mass transfer across the interface and bubble merging during fluid interactions will also change their topology significantly.

1.2 Research Problem

Due to the widespread application of CFD and heat transfer throughout engineering disciplines, fast and accurate algorithms are necessary. The search for faster and

more accurate algorithms is important as the discipline is continuously attempting to tackle increasingly complex design problems.

Furthermore, bridging the gap between science and engineering is an ongoing issue as new tools for the engineering community are developed. Fields such as discrete exterior calculus (DEC) (a branch of mathematics that applies calculus onto various geometrical features, referred to as manifolds, directly to a discrete setting) are ideally suited, but rarely used in CFD. By developing tools that can be directly applied to engineering problems, one can close the gap between science and engineering and thus benefit from faster and more accurate CFD algorithms.

1.3 Purpose and Objectives

This thesis proposes a unique and efficient approach to solving flow problems accurately (with an emphasis on two-phase flow) by applying discrete exterior calculus to a vorticity based method, which intrinsically preserves angular momentum. This approach bridges a gap between the engineering discipline and discrete exterior calculus, a commonly overlooked mathematical field that is ideally suited for solving complex problems in a discrete domain.

The objectives of this thesis are to:

- Review the literature in order to clarify current methods used to solve two-phase flow problems and their shortcomings
- Develop an analytical framework necessary to implement a unique and efficient approach to solving two-phase flow problems
- Develop a numerical framework that can be readily applied in a discrete setting
- Apply the numerical framework to evaluate the validity and accuracy of the proposed approach

- develop test cases and compare to known solutions
- Propose conclusions and recommendations for future work

It should be noted that the original goal of this thesis was to develop a high fidelity 3-D model of two-phase flow. As it became apparent that the motion of a bubble results in shedding of von Karman street like vortices, which in turn grossly affect subsequent bubbles, a vorticity preserving solution method would be very advantageous. For this reason, a discrete exterior calculus vorticity based formulation of the Navier-Stokes equations was selected. As commercial solvers were not available, the majority of this thesis focuses on the development of the DEC solver with a focus on two-phase flow applications.

1.4 Thesis Organization

Chapter 2 briefly reviews current approaches to solving multi-phase flow problems and their limitations, as well as new numerical approaches not commonly implemented in the engineering community. Chapter 3 reviews the mathematical theory required to implement a numerical solution based on discrete exterior calculus, while chapter 4 develops the necessary theory that governs fluid dynamics. Chapter 5 develops the suggested finite element solver based on discrete exterior calculus. Chapter 6 evaluates the validity of the developed numerical solver by implementing simple test problems and comparing the results to previous solutions. Chapter 7 summarizes the results and provides recommendations for future work.

Chapter 2

A Review of Current Approaches to Solving Multi-Phase Flow Problems

2.1 Introduction

To aid in the fulfilment of the thesis purpose of developing fast and accurate CFD solutions to tackle increasingly complex design problems, an understanding of current methods in computational fluid mechanics is required. This chapter will review widespread solution techniques used throughout engineering in order to clarify current methods used to solve two-phase flow problems and their shortcomings.

2.2 Brief Review of Fluid Dynamics

Modern day fluid dynamics can be attributed to distinguished mathematicians such as Daniel Bernoulli, who published *Hydrodynamica* in 1738. Bernoulli introduced what today is called Bernoulli's equation, a form of the conservation of energy that is applied to fluid dynamics. Bernoulli's equation state that the total energy in a steady-flowing, non-viscous, incompressible fluid (measured by pressure, velocity, and height), must remain constant [4]. In 1759, the equations of motion were introduced

by Leonhard Euler, and described the conservation of mass and momentum for an inviscid fluid [5]. In 1815 Augustin Cauchy, a French mathematician known for his contributions to partial differential equations and complex variables, contributed to the theory of fluid flow by introducing the concept of the average rotation at a point in the flow. This idea was later extended to the concept of instantaneous rotation of a fluid element by George Stokes [6]. In 1822, Claude Navier introduced Newtonian viscosity into the Euler equations, giving them broad applicability in the engineering community. George Stokes addressed another major shortcoming of the Euler equations in 1845 by introducing compressibility effects. Since incompressible flow assumed that changes in pressure do not affect density and temperature significantly, the addition of compressibility allows the equations to be used in cases such as high speed flows, where the incompressibility assumptions are no longer valid. These now famous equations are commonly referred to as the Navier-Stokes equations [5].

At this point it should be noted that to simplify the implementation of a solver based on discrete exterior calculus, compressibility effects were omitted, and are therefore no longer considered. For general incompressible flow, the Navier-Stokes equation has the following terms:

$$\rho \left(\underbrace{\frac{\partial \mathbf{u}}{\partial t}}_{\text{Unsteady acceleration}} + \underbrace{\mathbf{u} \cdot \nabla \mathbf{u}}_{\text{Convective acceleration}} \right) = \underbrace{-\nabla p}_{\text{Pressure gradient}} + \underbrace{\mu \nabla^2 \mathbf{u}}_{\text{Viscosity}} + \underbrace{\mathbf{f}}_{\text{Other body forces}} \quad (2.1)$$

$\frac{\partial \mathbf{u}}{\partial t}$ - **The acceleration term.** The derivative of velocity with respect to time.

$\mathbf{u} \cdot \nabla \mathbf{u}$ - **The convection term.** This term preserves momentum in the flow. The momentum of the fluid is moved or “convected” with the fluid flow.

$-\nabla p$ - **The pressure term.** Captures forces generated by pressure differences within the fluid. ρ is the fluid’s density.

$\mu \nabla^2 \mathbf{u}$ - **The viscosity term.** In viscous fluids, friction forces cause the velocity of the fluid to move toward the neighbourhood average. The viscous properties of the fluid are diffused through the ∇^2 operator scaled by the fluids kinematic viscosity (μ).

f - External force. Any external body forces, such as gravity, are included in this term.

The addition of the mass continuity equation (a mathematical statement that says that mass must be conserved), conservation of energy (mathematical statement that says that energy must be conserved), and boundary conditions, allows the flow to be fully described.

Conservation of mass:

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

Conservation of energy:

$$\rho C_p \left(\frac{\partial T}{\partial t} + \mathbf{u} \cdot \nabla T \right) - \nabla \cdot (\mathbf{k} \nabla T) - Q - \Phi = 0 \quad (2.3)$$

Unfortunately, due to the complexity of the Navier-Stokes equations, general closed-form solutions have yet to be developed. They are therefore either simplified and solved analytically or approximated through numerical solutions.

2.2.1 Approximate Analytical Solutions

By being the first to decompose the motion of a fluid element into three components - pure translation, pure rotation, and pure strain - George Stokes paved for future analysis methods by allowing a component of the flow to be solved. A few years after Stokes, Hermann von Helmholtz also applied the concept of rotation of a fluid element to inviscid flows. In 1858, von Helmholtz published a paper entitled “On

the Integrals of the Hydrodynamical Equations Corresponding to Vortex Motions,” in which he observed that the velocity components along all three axes in a flow could be expressed as a derivative of a single function. He called this function “potential of velocity” [6]. The solution of the velocity potential describes the advection of an irrotational and inviscid flow, and is expressed through the gradient of a scalar function, which has the form of Laplace’s equation below.

$$\nabla^2 \phi = 0 \tag{2.4}$$

Due to the inviscid characteristic of the flow, it advects any pre-existing vorticities or, if starting from rest, results in a purely irrotational flow (no vorticity generation). Therefore, the potential flow approximation is not useful when attempting to model any flow with strong vorticity effects.

Only in some cases, such as those described by Poiseuille’ Law and Bernoulli’s equation - special cases of 1-D Navier-Stokes equations - do solutions exist. These partial differential equations (PDEs) describe many complex flows and are therefore commonly solved with the aid of numerical methods.

2.2.2 Multi-phase problems

The origins of multi-phase problems were first studied by John William Strutt (Lord Rayleigh) in the late 19th century when he considered the collapse of a spherical void within a liquid. Strutt was approached by the Royal Navy during World War I because of their concern with the problem of damaged propellers of boats and submarines [7]. Further work on multi-phase problems includes the phase-change free-boundary heat transfer problem named the “Stefan problem”. The classic Stefan problem was first introduced by Lamé and Clapeyron in 1831. The solution to the

general Stefan problem was developed by Franz Neumann in 1860. The Neumann solution is an analytical solution based on error functions applied to special cases of the heat equation. However, the modern Stefan problem was officially recognized in 1890 by Jožef Stefan, a Slovene physicist, who first discussed the problem in publications with relation to ice formation.

S.C. Gupta [8] has constructed the authoritative source on the Stefan problem by piecing together many different mathematical methodologies that are widely scattered throughout the literature. Gupta covers a broad range of Stefan problems starting with an in-depth look at a classical 1-D, one-phase problem, and concluding with a multi-dimensional, n-phase, problem. Many authors, such as Welch and Wilson [9], present the Stefan problem as a test case to verify their solution methodology due to the availability of an analytical solution. As shown in section 6.1.2, when examined in detail, it becomes evident that the analytical solution does not solve the heat equation, but sets the temperature gradient to be linear at any point in time.

2.3 Computational Fluid Dynamics

2.3.1 Form of the Governing Equations

There are various approaches to solving the Navier-Stokes equations, including solutions for different primitive variables. The standard pressure-velocity formulation of the Navier-Stokes equation expresses the flow in terms of pressure and velocity. Therefore many discrete methods attempt to solve directly for these variables. On the other hand, the Vorticity-Stream function method does not use the standard formulation, but instead applies the curl operator to the Navier-Stokes equations, removing the pressure term and adding a vorticity term. The remaining vorticity-velocity formulation is solved. This brings about complications such as the necessity

for the vector-potential [10], and as will be shown, complicates boundary conditions.

2.3.2 Discretization Schemes

Discretizing partial differential equations can be accomplished by finite difference methods (FDM), finite volume methods (FVM), or finite element methods (FEM).

Finite Difference Method

The finite difference method is an approximation to the differential equation. Methods such as first-order forward difference in time and space (equations 2.5 and 2.5) are commonly used because of their simplicity. More complicated approximations such as second-order central second difference methods are also commonly employed (equation 2.7). Here i is the running index in the ζ direction and n is the running index in the t direction. For further background the reader is referred to [11].

First-order forward difference in time:

$$\frac{\partial T}{\partial t} = \frac{T_i^{n+1} - T_i^n}{\Delta t} \quad (2.5)$$

First-order forward difference in space:

$$\frac{\partial T}{\partial \zeta} = \frac{T_{i+1}^n - T_i^n}{\Delta \zeta} \quad (2.6)$$

Second-order central second difference in space:

$$\frac{\partial^2 T}{\partial \zeta^2} = \frac{T_{i+1}^n - 2T_i^n + T_{i-1}^n}{(\Delta \zeta)^2} \quad (2.7)$$

As standard FDM can only be applied to regular meshes, it is not ideal for complex geometries. Even so, because of the relative simplicity of FDM, many reserchers, including Fedkiw et al [12], use it with complex geometries through extended FDM

implementations such as the Ghost Fluid Method (GFM).

Notably, the Marker-and-Cell (MAC) method uses a finite difference discretization on a rectangular mesh and a shifted (staggered grid) mesh centred on the nodes of the basic mesh. This method was first introduced by Harlow and Welsh (1965) [13], [14], and is commonly referred to as the “finite volume method”.

Finite Volume Method

The finite volume method (FVM), also called the “finite difference scheme,” “cell centred difference scheme,” or “control volume finite difference method”, approximates the average integral value on a reference volume (i.e., it deals with flux values over surfaces instead of point values). These fluxes are commonly approximated using a finite difference approach on the boundaries of control volumes [13].

Recently, co-volume techniques (using a Voronoi¹-Delaunay² mesh pair) were applied to finite volume methods, allowing for complex geometries. Discrete exterior calculus is geometrically based on the co-volume finite volume method, except that it does not employ a finite difference scheme. Instead, it employs exterior calculus.

Finite Element Method

Unlike the finite difference method, the finite element method is an approximation to a solution and not the differential equation. Furthermore, unlike finite volume methods, finite element methods do not use staggered grids. Mattiussi states that finite element methods achieve a similar integration over an area by the use of weighting functions. Mattiussi also states that both finite volume and finite element methods have the same underlying mathematical structure if approached from algebraic topology [16]. This is a very important observation as DEC methods are based on applying calculus

---

¹A Voronoi region is the region that bounding grid points make around a central point.

²Delaunay triangulation is the formation of a triangular grid (by joining all neighbouring nodes by straight lines) [15].

directly to discrete algebraic topology and therefore can be considered as a more generalized form of finite volume and finite element methods.

In the finite element method, the problem is first given a weak or variational formulation and is then discretized on the domain of interest. Therefore, no thought is given to the underlying geometric structure when the variational formulation of the partial differential equation (PDE) is developed. It has become apparent that to increase the stability of the numerical solution, designing a numerical method compatible with the underlying geometric structures instead of just approximating them, is vastly more beneficial [17].

2.3.3 Reference Frame

Lagrangian algorithms, are formulations in which the computational mesh follows the material particle during motion. The Lagrangian formulation allows for easy tracking of free surfaces and interfaces between different materials. Unfortunately, this formulation is unable to follow large distortions in a flow without recourse to frequent re-meshing operations.

Eulerian algorithms are widely used in fluid dynamics. A Eulerian formulation has a mesh in space while the material points move with respect to it. In the Eulerian description, large distortions in the continuum motion can be handled easily, but interface tracking requires special care [18].

Arbitrary Lagrangian-Eulerian (ALE) methods have the advantages of both Lagrangian and Eulerian methods and alleviate the drawbacks of the mesh distortion in Lagrangian formulation [18] [19]. One such method is the particle-in-cell (PIC) method that was developed in 1950 by a group lead by F.H. Harlow [20]. The PIC scheme consists of a multi-step time-marching algorithm, also known as a time-splitting algorithm. In the PIC method, the values of the velocities and energy are calculated from known or assumed values of pressure gradients. The particles are then

advected in the flow. The necessity for high particle density, required for accurate flow prediction, is the major drawback of the PIC method [21] [22]. Previously, the resulting large computational power required to store the domain forced modifications to be made, resulting in the fluid-in-cell (FLIC) or coarse-large-particle method. In these modifications, the Lagrangian particle mesh was replaced with flux calculations across the Euler cell boundaries [23].

2.3.4 Interface Tracking

Navier-Stokes equations, and their numerical approximations, describe motions of specific fluids ranging from the simple distribution of static pressure to flows driven by surface tension. They do not, however, model and track interface conditions between fluids. To solve multi-phase problems (or free boundary problems), an interface tracking algorithm must be integrated into the solution.

Lagrangian methods are ideally suited if small displacements are expected, or the time penalty for re-meshing is acceptable, as the interface elements automatically track the boundaries. When using an Eulerian frame of reference, an interface tracking method must be used to keep track of moving fronts and free surfaces. One of the most common methods for tracking the liquid boundary during simulation uses a data structure called a “level set” which was introduced by Forster [24]. Level-set methods (LSM) implement a scalar function within the domain, setting the zero-level as the discontinuity [25].

2.3.5 Numerical Solutions

Of particular interest, Welch and Wilson present a volume of fluid method simulating flows with mass transfer due to phase change. To test their computational engine, they develop a test case which they name the “sucking” problem [9]. The sucking

problem is similar to the Stefan problem except that the latent heat of vaporization is drawn from the superheated liquid instead of the vapour, creating a thin thermal layer in the liquid. This thermal layer is responsible for mass transfer and travels with the interface, making it appear as if it is “sucking” the interface with it. The existence of this thin thermal layer makes the “sucking” problem computationally more challenging compared to the Stefan problem [9]. Using these 1D problems, Welch and Wilson demonstrate the effectiveness of the volume of fluids method to accurately compute temperature profiles in boiling flows. A similar approach is envisioned for the discrete exterior calculus method proposed. Therefore, the foundations for both the Stefan and “sucking” problems are presented here.

2.4 Geometry-based Discrete Exterior Calculus

As Tonti stated in 1976 [26], many physical theories exhibit a similar form even though they differ profoundly in physical content. He further states that every physical theory is tied to basic physical quantities, such as points, lines, surfaces and volumes [26]. If these geometric structures are expressed in terms of k -dimensional manifolds, operators based on geometry-based exterior calculus can directly act on these geometric structures, allowing one to solve many problems with one framework.

The mathematics of geometry-based discrete exterior calculus has its roots in differential geometry, algebraic topology, and homological algebra. However, these topics are outside the scope of this paper. For an in-depth review, the reader is directed to [27], [28] and especially [29].

Discrete exterior calculus (DEC), also referred to as exterior calculus of differential forms and differential geometry, was first introduced by Élie Cartan in 1945 [30]. DEC moves to extend the finite element method by developing discretizations which are compatible with the underlying geometric, topological, and algebraic structures [17].

Modern differential geometry has the ability to express analogues of classical operators such as gradient, curl and divergence through the “exterior derivative,” acting directly on differential forms. In actuality, these three first-order differential operators are replaced by one operator, the exterior derivative, and its application to a specific dimension of a form (specified as a k -form) results in the analogue of either the gradient, curl or divergence operator. These mathematical tools can then express Gauss’ theorem (also known as divergence theorem), Green’s theorem and Stokes’ theorem. As the gradient, curl and divergence operators are defined on the discrete geometry directly, the resulting solutions have no numerical dissipation, therefore allowing for exact solutions of Euler equations as well as maintaining validity over large time-scales. The implementation of discrete exterior calculus (DEC) into a numerical solver is quite simple as it reduces to matrix multiplication; the complications arise with the terminology gap, as DEC is not yet widely used in engineering, and the application of boundary conditions. Nevertheless, with these tools in-hand, the Navier-Stokes equations can be solved accurately and efficiently.

Chapter 3

Analytical Framework

Before examining the analysis of flow fields, a look into vector analysis and discrete exterior calculus is required, as the presented numerical approach combines theory from differential geometry and geometric algebra to define analogous operators to those used in vector analysis. The link between the formentioned topics and numerical analysis is discrete exterior calculus (DEC) or finite element exterior calculus (FEEC). Exterior calculus is the generalization of vector calculus to non-linear manifolds. Discrete exterior calculus is the discretization of exterior calculus for use in computations. It allows for a discrete, coordinate-free definition of the necessary operators and theorems from smooth theory, allowing full tensor analysis on manifolds of any dimension. It is important to note that these topics are very broad, and hence only an introduction to the necessities is discussed here.

3.1 Mathematic Definitions and Derivations

In vector analysis, several basic definitions must be considered. Throughout the classical operator definitions, an attempt is made to introduce the analogy to the corresponding exterior calculus operators that will be formally introduced in section 5.1.

Partial Derivative

In classical calculus, a partial derivative is defined as

$$\frac{\partial}{\partial y_i} f(\mathbf{y}) = \lim_{h \rightarrow 0} \frac{f(y_1, \dots, y_{i-1}, y_i + h, y_{i+1}, \dots, y_n) - f(y_1, \dots, y_n)}{h} \quad (3.1)$$

In exterior calculus, the exterior derivative is defined as the adjoint of the boundary operator of a k -simplex¹, and results in a $(k+1)$ -simplex. For example, the derivative of a curve at a point results in the tangent vector at that point. In a discrete setting, this curve is approximated through a discrete number of straight elements, and the tangent is approximated as the slope between the points bounding the element.

Nabla Operator

In a three-dimensional Cartesian coordinate system, the nabla operator is defined as the partial derivative with respect to each coordinate.

$$\nabla = \left(\frac{\partial}{\partial x_1}, \frac{\partial}{\partial x_2}, \frac{\partial}{\partial x_3} \right)^T \quad (3.2)$$

Material Derivative

The material derivative, also known as the substantial or Lagrangian derivative, is a derivative taken with respect to a moving coordinate system and defines the acceleration of a material point in a spatial frame. The resulting time rate of change for a fluid element is defined as

$$\frac{D\gamma}{Dt} = \frac{\partial \gamma}{\partial t} + \mathbf{v} \cdot \nabla \gamma, \quad (3.3)$$

where $\mathbf{v}$ is the velocity field, γ is a scalar or vector field and $\frac{\partial \gamma}{\partial t}$ is the rate of change

---

¹A simplex, or k -simplex, is the k -dimensional analogue of a triangle, and is used to describe the simplest mesh element of dimension k .

of γ at a fixed point in space.

Jacobian Matrix

The Jacobian matrix (J) is the partial derivative of the function f_i with respect to x_j and is represented as

$$J_F = \frac{\partial(y_1, \dots, y_m)}{\partial(x_1, \dots, x_n)} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \dots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \dots & \frac{\partial y_m}{\partial x_n} \end{bmatrix}. \quad (3.4)$$

Gradient

The derivative of a scalar field with respect to its coordinate, results in a vector field called the gradient. This represents the special case when a function is a scalar, therefore reducing the Jacobian into the gradient.

$$\text{grad}(f) = \nabla f = \left(\frac{\partial f}{\partial x_1}, \dots, \frac{\partial f}{\partial x_n} \right)^T \quad (3.5)$$

As mentioned above, in exterior calculus, the gradient is related to the exterior derivative of a 0-form.

Curl

The curl of $\mathbf{F}$ is another vector field that measures by how much the Jacobian fails to be a symmetric matrix. The symmetric and antisymmetric parts of the Jacobian

are given by

$$J_F^A = \frac{1}{2} (J_F + (J_F)^T) \quad (3.6)$$

$$J_F^S = \frac{1}{2} (J_F - (J_F)^T) \quad (3.7)$$

Therefore, any component in the resulting symmetric (J^S) or antisymmetric (J^A) matrix can be determined by

$$\left(\frac{\partial F_i}{\partial x_j} + \frac{\partial F_j}{\partial x_i} \right)_{ij} \quad \text{and} \quad \left(\frac{\partial F_i}{\partial x_j} - \frac{\partial F_j}{\partial x_i} \right)_{ij} \quad (3.8)$$

The antisymmetric part of the Jacobian contains the information of the curl. Its components are the differences of the three pairs of off-diagonal elements defined as

$$\text{curl}(\vec{\mathbf{F}}) = \nabla \times \vec{\mathbf{F}} = [(J_F^A)_{32}, (J_F^A)_{13}, (J_F^A)_{21}]^T = \begin{bmatrix} \frac{\partial F_x}{\partial y} - \frac{\partial F_y}{\partial z} \\ \frac{\partial F_x}{\partial z} - \frac{\partial F_z}{\partial x} \\ \frac{\partial F_y}{\partial x} - \frac{\partial F_x}{\partial y} \end{bmatrix} \quad (3.9)$$

In exterior calculus, the curl is related to the exterior derivative of a 1-form.

Kelvin's Circulation Theorem

The circulation, Γ , is defined as the integral of the velocity vector around a closed path

$$\Gamma = \oint_{\partial\sigma} \mathbf{u} \cdot d\mathbf{l} \quad (3.10)$$

Kelvin's circulation theorem states that the circulation around a closed loop moving

with an inviscid fluid remains constant with time.

$$\frac{d\Gamma}{dt} = 0 \quad (3.11)$$

Stokes' Theorem

The physical representation of curl is related to circulation by Stokes theorem. Following the fundamental theorem of calculus, Stokes' theorem states that

$$\int_{\sigma} \nabla \times \mathbf{u} \cdot d\sigma = \oint_{\partial\sigma} \mathbf{u} \cdot d\mathbf{l} \quad (3.12)$$

where σ is a $(k + 1)$ -simplex and $\mathbf{u}$ is the k -form. Therefore, Stokes' theorem states that the curl determines the flow around a region and thus the rotation of the fluid in the region. Thus, the curl of the flow results in the rotation or vorticity (ω) of the flow.

$$\omega = \nabla \times \mathbf{u} \quad (3.13)$$

It should be noted that Green's theorem is a special two-dimensional case of the more general Stokes' theorem.

Divergence

The divergence consists of the diagonal elements of the Jacobian, and is commonly determined by the trace of the J matrix. The divergence can also be represented by the dot product with the nabla operator as shown below.

$$\text{div}(\vec{\mathbf{F}}) = \nabla \cdot \vec{\mathbf{F}} = \left(\frac{\partial F_x}{\partial x} + \frac{\partial F_y}{\partial y} + \frac{\partial F_z}{\partial z} \right) \quad (3.14)$$

In exterior calculus, the divergence is related to the exterior derivative on a 2-form².

---

²k-forms are values stored on their respective k-dimensional geometric structures known as k-simplicies.

Gauss' Theorem

The Divergence Theorem, also known as Gauss' Theorem, states that the divergence at a point is equal to the sum of all the flows in and out of the neighbourhood of the point.

$$\int_V (\nabla \cdot \tilde{\mathbf{F}}) dV = \int_A (\tilde{\mathbf{F}} \cdot \tilde{\mathbf{n}}) dA \quad (3.15)$$

In this equation, $\tilde{\mathbf{n}}$ is the outward normal around the boundary dV . Therefore the divergence is equal to the difference of the flow through the area bounding the volume. In the limit as the volume is reduced to a point, it can be treated as the material generated or destroyed per unit volume at the point (source or sink). Thus, a divergence-free flow results in one definition for an incompressible fluid.

Laplacian

In vector analysis, it is common to apply the nabla operator twice in the form $\nabla \cdot \nabla$. Therefore, this operation is commonly known as the Laplacian and in Cartesian coordinates is written as

$$\Delta = \nabla \cdot \nabla = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2} = \nabla^2 \quad (3.16)$$

The Laplacian of a scalar field is the divergence of the gradient and therefore results in a scalar field.

The Laplacian can also be defined through the following vector identity:

Suppose $f : \mathbf{R}^3 \rightarrow \mathbf{R}$, and $\mathbf{F} : \mathbf{R}^3 \rightarrow \mathbf{R}^3$.

$$\begin{aligned}
\nabla \times (\nabla f) &= 0 \\
\nabla \cdot (\nabla \times \mathbf{F}) &= 0 \\
\nabla \cdot (\nabla f) &= \nabla^2 f \\
\nabla \times (\nabla \times \mathbf{F}) &= -\nabla^2 \mathbf{F} + \nabla (\nabla \cdot \mathbf{F}) \\
\nabla^2 \mathbf{F} &= \nabla (\nabla \cdot \mathbf{F}) - \nabla \times (\nabla \times \mathbf{F})
\end{aligned} \tag{3.17}$$

Therefore, the Laplacian can be represented as the following operator

$$\Delta = \nabla^2 = (\nabla \nabla \cdot - \nabla \times \nabla \times) \tag{3.18}$$

This is an important formulation, as the implementation of the DEC algorithm is based on this definition of the Laplacian.

3.2 Geometric Algebra

Even though the actual implementation of the DEC solver does not require any special algebraic consideration during implementation, the underlying algebraic theory stems from algebraic topology and homological algebra and can be broadly characterized as geometric algebra. These branches of math explore the underlying geometric structures (for example, 0-forms and 1-forms represent integrand values on 0-dimensional and 1-dimensional simplices). In classical mathematics, 0-dimensional values are equivalent to scalars and can form scalar fields. A vector field is a k-dimensional space with a vector attached to each point in the space. Algebraic topology and homological algebra extend this idea in a general k-space setting, defining general operators such as the exterior derivative (d), co-differential (δ), Hodge star ($\star$) and

wedge product ($\wedge$) which act on these k -dimensional forms. For example, the wedge product, also known as the exterior product, is a generalization of multiplication into k -dimensions.

The notion of k -dimensional forms can be considered an “underlying fabric” of mathematics and is described by chain complexes. Specifically, homological algebra explores the homology and cohomology of the underlying structures (chains and co-chains) while algebraic topology explores how the topological spaces on which these forms operate are constructed.

3.3 Differential Geometry

Differential Geometry is a branch of mathematics that applies the principles of differential and integral calculus to the study of curves and surfaces. As all physical problems are applied to geometry and can be described by PDEs, it seems natural that differential geometry should be used to describe these problems [26]. Applying differential geometry to a discrete setting results in the replacement of smooth curves and surfaces with meshes and simplicial complexes. A simplicial complex is a space constructed from points, lines, surfaces and their corresponding volumes referred to as a 0-simplex, 1-simplex, 2-simplex or 3-simplex, respectively. These k -dimensional simplices describe the simplest elements of a mesh of dimension k .

3.3.1 The Differential Structure

For completeness, the differential structure consists of the following seven operators:

d : the exterior derivative - extension of the differential of a function to differential forms

$\star$: the Hodge star - a metric for converting between primal and dual meshes

$\wedge$: the wedge (or exterior) product - multiplication extended to k-dimensions

$\flat$: the flat operator - relates k-dimensional vector fields to k-forms

$\sharp$: the sharp operator - relates k-forms to k-dimensional vector fields

$\lrcorner$: the interior product - dual to the exterior product

$\mathcal{L}_X$: the Lie derivative - derivative of a vector field over another vector field.

Out of these operators, the DEC algorithm for solving the incompressible Navier Stokes equations only directly implements the exterior derivative d . For a comprehensive examination of the other operators see [27] and [29].

3.3.2 Introduction to the Underlying Structure of DEC

As the DEC solver operates with k-forms on k-simplicies (the mesh), it intrinsically stores the integral values over the edges, faces, and volumes of the mesh geometry, resulting in all the values being represented as scalars with their inherent orientation defined by each corresponding k-simplex orientation. For example a

- 0-form is a scalar, and over a manifold can define a potential field such as temperature or pressure. As a scalar field, a 0-form has no orientation. The exterior derivative d^0 of a 0-form results in a 1-form and is analogous to the gradient $d^0 \Psi = \Phi$, where Ψ is a 0-form scalar field and Φ is the resulting 1-form;
- 1-form is stored as an integrand value over an edge (1-simplex) and therefore it can represent a vector with a coincident orientation of the associated edge (such as the gradient of a potential or a vector potential). The exterior derivative d^1 of a 1-form is analogous to the curl operator $d^1 \Phi = U$, where U represents a 2-form;

- 2-form is a scalar value integrated over a surface or face (2-simplex). Through the surface normal orientation, a 2-form can represent flux values. The exterior derivative d^2 of a 2-form results in the application of the Div operator $d^2 U = \nabla \cdot u$, and results in a 3-form;
- 3-form is a scalar field represented by the integral over a volume resulting in a scalar such as mass or charge.

Chapter 4

Governing Equations of Fluid Dynamics

The governing equations of fluid dynamics can be derived using the mathematical definitions presented in Chapter 3 as well as the principals of conservation of mass, momentum and energy.

4.1 Conservation of Mass

The conservation of mass states that matter cannot be made or destroyed, therefore the mass of a fluid element must remain constant. Consequently, if the density of a fluid element decreases, its volume must increase accordingly. Using a Lagrangian (material) frame of reference the conservation equation in the computational domain Ω can be written as

$$\frac{D\rho}{Dt} + \rho \nabla \cdot \mathbf{u} = 0 \quad (4.1)$$

Using the definition of the material time derivative from section 3.1, the conservation of mass can be rewritten using a Eulerian (spatial) frame of reference

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{u}) = 0 \quad (4.2)$$

where ρ is the discontinuous mass density (mass per unit volume), and $\mathbf{u}$ is the velocity

of the fluid. In the case of a divergence free flow, approximately true of most liquids including water, ρ does not vary within the specific fluid so the equation reduces to

$$\nabla \cdot \mathbf{u} = 0 \quad (4.3)$$

4.2 Conservation of Momentum

The conservation of momentum, or Newton's second law, states that the total momentum of any object must remain the same unless an outside force acts on the object. Using a Lagrangian (or material) formulation, it can be written as

$$\rho \frac{D\mathbf{u}}{Dt} = \nabla \cdot \sigma + \mathbf{b} \quad (4.4)$$

where $\mathbf{b}$ is the sum of the various external force densities and σ is the total viscous deviatoric and spherical elastic stress tensor given by

$$\sigma = \tau - p\mathbf{I} \quad (4.5)$$

where

$$\begin{aligned} \tau &= 2\mu\mathbf{D} \\ \mathbf{D} &= \frac{1}{2} \left(\nabla\mathbf{u} + (\nabla\mathbf{u})^T \right) \end{aligned} \quad (4.6)$$

Expanding to give the normal and shear stress components, equation 4.6 becomes

$$\tau_{xx} = \lambda (\nabla \cdot \mathbf{u}) + 2\mu \frac{\partial u_x}{\partial x} \quad (4.7)$$

$$\tau_{yy} = \lambda (\nabla \cdot \mathbf{u}) + 2\mu \frac{\partial u_y}{\partial y} \quad (4.8)$$

$$\tau_{zz} = \lambda (\nabla \cdot \mathbf{u}) + 2\mu \frac{\partial u_z}{\partial z} \quad (4.9)$$

$$\tau_{xy} = \tau_{yx} = \mu \left[\frac{\partial u_y}{\partial x} + \frac{\partial u_x}{\partial y} \right] \quad (4.10)$$

$$\tau_{xz} = \tau_{zx} = \mu \left[\frac{\partial u_z}{\partial x} + \frac{\partial u_x}{\partial z} \right] \quad (4.11)$$

$$\tau_{yz} = \tau_{zy} = \mu \left[\frac{\partial u_z}{\partial y} + \frac{\partial u_y}{\partial z} \right] \quad (4.12)$$

λ represents the second viscosity coefficient and ν is the molecular viscosity. Stokes suggested that

$$\lambda = -\frac{2}{3}\mu \quad (4.13)$$

for all Newtonian fluids, but this has not been proved to this day [31].

Applying the material time derivative from section 3.1 to equation 4.4 results in the Eulerian formulation of the momentum equation.

$$\rho \left(\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} \right) + \nabla p - \nabla \cdot \left(\mu \cdot \left(\nabla \mathbf{u} + (\nabla \mathbf{u})^T \right) \right) + \rho_0 \mathbf{g} \beta (T - T_0) - \mathbf{f} = 0 \quad (4.14)$$

The density term was expanded to include buoyancy and natural convection. The gravitational force $\mathbf{g}$ is multiplied by the discontinuous density ρ , which is a function of x and t , the coefficient of expansion β and the temperature difference. The term $\mathbf{f}$ is the surface tension force, and hence is only used when a solution across the interface

is sought after. The surface tension force is given by

$$\mathbf{f} = \sigma \kappa \hat{\mathbf{n}} \delta_\gamma \quad (4.15)$$

where $\hat{\mathbf{n}}$ is the normal vector to the interface Γ_{int} , κ is the curvature of the interface, and δ_γ is defined as a Dirac delta measure on the interface Γ_{int} .

4.3 Conservation of Energy

The conservation of energy (also known as the First Law of Thermodynamics) can be written as

$$\rho C_p \left(\frac{\partial T}{\partial t} + \mathbf{u} \cdot \nabla T \right) - \nabla \cdot (\mathbf{k} \nabla T) - Q - \Phi = 0 \quad (4.16)$$

where T is the temperature, C_p is the specific heat of the fluid at constant pressure, Q is the rate of internal heat generation, $\mathbf{k}$ is the thermal conductivity tensor, and Φ is the viscous dissipation in the fluid given by

$$\Phi = 2\mu D : D \quad (4.17)$$

where μ is the shear viscosity of the fluid. The density of the fluid is assumed to vary according to the coefficient of thermal expansion (β) and temperature (T) with the following constitutive equation

$$\rho = \rho_0 [1 - \beta(T - T_0)] \quad (4.18)$$

4.4 Direct Modelling of Boundary and Interface Conditions

As demonstrated in Figure 4.1, if two distinct fluids separated by an arbitrary interface exist in a domain Ω (where $\Omega = \Omega_1 \cup \Omega_2$) the discontinuities that arise on the interface Γ_{int} , including the density ρ , viscosity μ , and specific heat C_p , need to be defined in terms of the specific fluid at any given time.

Figure 4.1: Problem domain and boundary interface, [32].

$$(\rho(x), \mu(x), C_p(x)) = \begin{cases} (\rho_A, \mu_A, C_{pA}) & \forall \in \Omega_1 \text{ (in fluid A)} \\ (\rho_B, \mu_B, C_{pB}) & \forall \in \Omega_2 \text{ (in fluid B)} \end{cases} \quad (4.19)$$

Therefore, direct modelling of an interface between two different phases must include a method to track or capture the interface. Predominantly, interface tracking methods use deforming meshes and therefore explicitly track the interface. However, interface capturing methods use an auxiliary function to define the interface in a fixed domain. As large deformations are expected at the liquid gas interface, an interface capturing method is preferred.

The level-set method, commonly used as an interface capturing method, defines a Heaviside step function, (or simply the unit step function) to separate the two fluids and thus the sub-domains of different densities, viscosities, and specific heats.

If viscous fluids are being considered, a force equilibrium at the interface requires the stress vectors to be equal, where the normal stress component represents the pressure while the tangential component represents the “no slip” boundary condition.

$$\sigma_A = \sigma_B \quad \forall x \in \Gamma_{int} \quad (4.20)$$

Therefore, if a no mass flow simplification is taken across the interface, the normal and tangential velocities of both fluids have to match at the interface.

$$\mathbf{u}_A = \mathbf{u}_B \quad \forall x \in \Gamma_{int} \quad (4.21)$$

If vapourization is being included in the solution, the normal velocity components of the two phases differ by the mass flow rate across the interface. Please note that since fluids A and B are now being considered as different phases of the same fluid, the subscripts “A” and “B” have been replaced with “l” and “v” to represent the liquid and vapour phases respectively.

$$\dot{m} = \rho_v(\mathbf{u}_v - \mathbf{u}_s)A_{interface} \quad (4.22)$$

$$\dot{m} = \rho_l(\mathbf{u}_l - \mathbf{u}_s)A_{interface} \quad (4.23)$$

where u_s is the interface velocity.

To solve for the mass flowrate, an energy equation across the interface must be developed. Firstly, as energy is conserved across the interface, the energy required for vaporization must be conducted through the vapour and/or liquid phase to the interface.

$$\dot{m}h_{lv} = K_v \frac{\partial T_v}{\partial \hat{\mathbf{n}} \cdot \mathbf{x}_v} \Big|_{interface} - K_l \frac{\partial T_l}{\partial (\hat{\mathbf{n}} \cdot \mathbf{x}_l)} \Big|_{interface} \quad (4.24)$$

Where T represents the temperature and x represents the distance from the interface in the specific phase. It should also be noted that the thermal conductivity of the vapour (K_v) will vary significantly with the vapour pressure (p_v).

Assuming the bubble grows within a superheated liquid, it can be concluded that the energy required for vaporization is conducted through the liquid to the interface only (no energy flux transfer from the vapour phase), reducing equation 4.24 to

$$\dot{m} = - \frac{K_l}{h_{lv}} \frac{\partial T_l}{\partial (\hat{\mathbf{n}} \cdot \mathbf{x}_l)} \Big|_{interface} \quad (4.25)$$

Once the liquid or vapour velocity is known, applying equation 4.25 to equations 4.22 and 4.23 results in a solution for the remaining unknowns.

4.5 Simplified Spherical Bubble Model

Alternatively, bubbles can be modelled as heat sinks within a superheated liquid domain. A common approach could include modelling the bubbles using a simplified spherical model as that developed by Zwick & Plesset [33] [34] [35].

4.5.1 Simplifying Assumptions

Spherical Bubble

The bubble remains a perfect sphere. This assumption can be justified if the radial acceleration and velocity of the bubble remains negligible, resulting in a spherically symmetric external pressure field which will be stable under the action of surface tension.

Small bubble velocity

The buoyant forces caused by gravity become important if the bubble growth is followed for a period of time, allowing for a significant translational velocity to be acquired. The translational velocity will not only cause a deformation in shape but also increase the rate of heat inflow to the bubble through convection. According to research, this assumption holds true when bubble growth is not followed beyond a radius of approximately 1 mm.

No compressibility effects

Due to the low velocity of bubble growth compared to the velocity of sound in the liquid, the compressibility effects can be neglected due to the motion of the liquid produced by the bubble growth. Furthermore, the compressibility effects may be neglected in the bubble vapour as the velocity of sound in the vapour is high compared to the bubble growth velocity.

Constant pressure and temperature of liquid

Since the volume of the liquid is assumed to be infinite relative to the volume of the vapour bubble, it is assumed that any volume and temperature changes at the bubble will not affect the pressure and temperature of the liquid.

Vapour assumptions

Since the vapour density is small, it is assumed that vapour inertia can be neglected throughout the calculations. Furthermore, due to the low density in the vapour bubble, Bernoulli's equation states that the pressure within the vapour region may be taken as uniform. Since the velocity of sound in the vapour is large compared to the bubble wall velocity, the pressure within the bubble can be assumed to change

instantaneously. Lastly, since the pressures within the bubble are not extreme, it can be assumed that the vapour obeys the perfect gas law.

A full derivation of the 1-D spherical bubble is included in Appendix C.

4.6 Vorticity Formulation

The definition of vorticity is

$$\nabla \times \mathbf{u} = \boldsymbol{\omega} \quad (4.26)$$

Taking the curl of the momentum equation, which contains both the velocity and pressure, the pressure term can be eliminated, resulting in the velocity and vorticity formulation.

$$\begin{aligned} \frac{\partial \boldsymbol{\omega}}{\partial t} + (\mathbf{u} \cdot \nabla \boldsymbol{\omega} - \boldsymbol{\omega} \cdot \nabla \mathbf{u}) \boldsymbol{\omega} &= \nu \Delta \boldsymbol{\omega} + \nabla \times \mathbf{f} \\ \nabla \cdot \mathbf{u} &= 0 \end{aligned} \quad (4.27)$$

Instead of the scalar field p , the new unknown is the vorticity vector $\boldsymbol{\omega}$.

Chapter 5

Numerical Approach

To solve problems computationally, it is desirable that they be resolved to a discrete level with acceptable accuracy. To accomplish this, an understanding of the underlying geometric structure is of paramount importance, and results in the transfer from a continuous model to a discrete computational realization while preserving the defining physical properties. It should be noted that the basic tools necessary for the definition of discrete calculus exists through the work of Poincaré on cell decomposition of smooth manifolds. The purpose of this chapter is to explore the toolbox necessary to perform all the required operations in the discrete realm for which Poincaré laid the foundations. For a more in-depth look at the underlying mathematical principles, the reader is directed to works such as [29], [27], [36] and [28].

As stated by Arnold *et al.* [17], the stability of solutions to many PDEs is directly related to the ability of formulating the equations in a manner that respects the underlying geometrical, algebraic, topological, and homological structures. Exterior calculus intrinsically accomplishes this by directly reproducing and operating on the geometric structures, and not just approximating them. Extending the finite element method to incorporate exterior calculus on differentiable manifolds (as the integration scheme) results in the finite element exterior calculus (FEEC) or simply discrete exterior calculus (DEC) method. The resulting formulation embraces the underlying

geometric structure.

5.1 Finite Element Analysis with Discrete Exterior Calculus

Before taking a detailed look at the DEC machinery, a brief overview of the solution methodology is offered in an attempt to introduce the reader to the DEC solution that will be explored throughout this chapter.

The underlying principal of DEC is to define discrete analogues of the differential operators themselves, allowing for simple formulation of PDEs. Specifically, for computational fluid dynamics, DEC results in the following solution methodology:

- The DEC solution is started with an initial flow field U (kg/s) defined as flux values integrated over a corresponding mesh element surface or face ($kg/(m^2s) \times m^2$). These surfaces are defined to be part of the “primal” mesh, as a similar structure to staggered grids is used.
- From the flows defined on the primal mesh, the velocities at the centroids of the mesh elements are calculated. The mesh that is constructed by joining the centroids of the primal mesh elements is referred to as the “dual” mesh. Together, the primal and dual mesh make a Voronoi pair as demonstrated by Figure 5.3.
- The now calculated velocities on the dual mesh vertices are used to define the full velocity field throughout the solution domain.
- Using the now known velocity field, the vertices of the dual mesh elements (centroids of the primal mesh elements) are backtracked to find their location and velocity at the beginning of the time step.

- Using the backtracked centroid's velocity values, the flow along the “original deformed” dual edges is calculated. These flows are the discrete components of the circulation around a dual face. Summing the dual edges around a loop enclosing an area, forms the circulation around that area.
- Using Kelvin's circulation theorem (equation 3.11), the flows down the current non-deformed dual edges (before backtracking) are set to equal the flows down the backtracked dual edges preserving circulation. Therefore, the angular momentum is preserved during time step advection.
- Summing the flows around specific areas (specifically dual surfaces or faces), results in the discrete vorticities (Ω) on the dual faces.
- As the advected vorticity is now known throughout the domain, it is possible to diffuse the vorticity allowing for viscous flow solutions. Furthermore, it is possible to apply external forces as well. By separating these steps, the solution implementation is similar to operator splitting algorithms.
- From the resulting discrete vorticity, the updated flux values can be recalculated by through the use of the discrete vector potential (Φ).

$$\nabla \times \Phi = \Omega \tag{5.1}$$

$$U = d(\Phi) \tag{5.2}$$

5.1.1 Discrete Domain

To solve a complex model numerically, the geometry of the problem must be discretized through cell decomposition of the “space” or manifold. For simplicity, we will discretize the domain with a triangular mesh (2D models) or a tetrahedron mesh

(3D models). Since the numerical analysis will be performed on the discrete manifold, the discrete analogues of operators such as Div, Grad and Curl must also be defined.

Once discrete forms and continuous vector fields on the discrete geometry are defined, the discrete exterior calculus can be developed by defining the discrete exterior derivative (d), analogous to the del operator (∇), co-differential (δ) and Hodge star ($\star$) for operating on forms. Once these are done one can then define other useful operators such as the discrete Divergence ($\nabla \cdot$), Curl ($\nabla \times$) and Laplace (Δ) operators which can then be used to analyze the problem as described in the previous chapter.

Furthermore, the discrete wedge product ($\wedge$) for combining forms, discrete flat ($\flat$) and sharp ($\sharp$) operators for going between vector fields and 1-forms can also be defined but are not necessary in this implementation and hence are not discussed. For a further look at discrete exterior calculus, the reader is directed to [27] for an in-depth derivation of each of these operators from first principles.

Forms

It is only natural to try to preserve the underlying geometric structures on which physical quantities are constructed. By storing these properties as integral values over the elements of the mesh directly, these k -dimensional forms, or “ k -forms”, such as line, surface and volume integrals, should be stored in their intrinsic geometric k -simplices (locations such as nodes, edges, faces and tetrahedral respectively). Since each form is associated with a simplex, they are stored as vectors with their size equal to the number of associated simplices. Therefore, the dimension of the vectors containing the 0-form, 1-form, 2-form and 3-form values are equal to the number of vertices, edges, faces and tetrahedrons, respectively.

Discrete Velocity

As flux is defined as the amount that flows through a unit area per unit time, the normal component of the velocity is intrinsically stored through the use of the integral of the flux over a surface. Through the velocity's cardinality with the flux, the velocity is stored as a 2-form. The flow is calculated as an integrated value of the flux across a face (or 2-simplex), and mathematically is represented by the surface integral of the vector field $\mathbf{v}$

$$U_f = \int_{\sigma} \mathbf{v} \cdot d\sigma \quad (5.3)$$

where U_f represents the flow across a specific face and σ is the area of face f .

Furthermore, implementing velocity as a flow results in a direct implementation for Neumann boundary conditions. For example, a no-transfer boundary condition would simply force the flow across the face to zero.

Discrete Divergence

Stokes' theorem states that the divergence is the difference between the inflow and outflow for a specific volume. Applying this to the previous definition of velocity (flow across a face), results in the divergence being the sum of all the flows across each face of a volume. For a tetrahedron, the discrete divergence is equal the sum of the four flows on its four faces. The divergence is therefore stored as 3-form vector of dimension equal to the number of tetrahedron elements.

Discrete Vorticity

The discrete vorticity is a measure of the rotation of the fluid around an edge, and hence the flow around the edge. Since the flow is a measure of the flux, the vorticity can be calculated as the sum of the flows across the faces adjacent to the edge. [37] compares the edge with adjacent faces as a "paddle wheel" with the flow across each

face contributing to the torque that “spins” the edge. As the resulting vorticity is associated with edges, it is stored as a 1-form. As the vorticity replaces the pressure in the velocity-vorticity formulation of the Euler and Navier-Stokes equations, it performs a similar function to a pressure boundary condition at an outlet of a flow.

5.1.2 Primal Complex

K-Simplex

The k -simplex is a k -dimensional analogue of a triangle and is the simplest possible geometry in a space of R^k . Hence, it is also the simplest mesh element of dimension k . As shown in Figure 5.1, this analogue can be extended to any dimensions, and is referred to as a k -simplex made up of its underlying $(k + 1)$ -simplices. Specifically, for 3-dimensions, only vertices, edges, triangles and tetrahedrons are relevant. Since

Figure 5.1: Primal simplices which make up the simplicial complex

a 0-form is simply a vertex, it is evaluated at a point, while a 1-form is evaluated on a curve, 2-form on a plane, and a 3-form over a volume. The boundary of a k -simplex can only be a set of $(k - 1)$ -simplices. Using this rule, a simplicial complex can be constructed.

Simplicial Complex

A simplicial complex is a manifold constructed of k -simplices by “gluing” the $(k - 1)$ -simplices of the adjacent k -simplices together in an oriented fashion. For example, if working in R^3 , a manifold could be constructed from tetrahedrons by gluing the faces from adjacent tetrahedrons together. The result is the primal mesh.

Orientation of the Primal Complex

It should be noted that since the primal complex represents the geometry, an orientation for each element exists and must be specified when discretizing the domain. The assignment of the orientation is arbitrary, but must remain consistent throughout all the mathematical operations. For example, a vector or edge $\mathbf{a} = [a_1, a_2]$ could also be written as $\mathbf{b} = [a_2, a_1]$ resulting in a vector “pointing” in the opposite direction. Hence $\mathbf{a} = -\mathbf{b}$.

5.1.3 Dual Complex

Each k -simplex has a $(n-k)$ -dual simplex associated with it. Therefore, each primal vertex, edge, triangle and tetrahedron has an associated dual Voronoi cell, dual edge, dual face and dual vertex, respectively. Just as the k -simplices construct the primary mesh, the dual simplices make up a dual mesh. As shown in Figures 5.2 and 5.3, since the primal and their respective dual cells have the same cardinality, the existing primal mesh machinery can be used to store the necessary dual values.

The corresponding mesh created from a simplicial complex (simplest possible elements) and its corresponding dual mesh is shown in Figure 5.3. This is a generalization of standard staggered grid methods to an irregular Voronoi cell based staggered grid method.

Figure 5.2: Primal and corresponding dual simplices. **Primal simplices:** vertices, edges, triangles and tetrahedrons (top). **Dual simplices:** dual cells, dual faces, dual edges and dual vertices (bottom). [37]

Figure 5.3: 2D primal mesh constructed from a simplicial complex and its corresponding dual mesh

5.1.4 Discrete Exterior Derivative

The discrete d_k operator is analogous to the continuous exterior derivative as defined by Stokes' Theorem

$$\int_{\sigma} d\omega = \int_{\partial\sigma} \omega \quad (5.4)$$

This can be written in a discrete manner as

$$d\omega_{\sigma} = \sum_{s \in \partial\sigma} \omega_s \quad (5.5)$$

where ω represents the k -form which is being integrated over a specific $(k+1)$ -simplex (σ) . Therefore, through Stokes' Theorem it can be seen that the integral of $d\omega$ over a simplex can be evaluated through the use of ω over the simplex boundary $(d\sigma)$. More specifically, the $d\omega_{\sigma}$ (a $(k+1)$ -form on a $(k+1)$ -simplex) is equal to the sum of the k -forms over the bounding k -simplices of the $(k+1)$ -simplex.

Applying this to a single triangle element Δabc with known 1-forms f on the triangle edges, one can calculate the resulting 2-form over the triangle face using

$$\begin{aligned} \int_{\Delta abc} df &= \int_{\partial\Delta abc} f \\ &= \int_{[a,b]} f_{ab} + \int_{[b,c]} f_{bc} + \int_{[c,a]} f_{ca} \end{aligned} \quad (5.6)$$

Since k -forms are integrand values over the specific k -simplices, equation 5.6 simplifies to

$$dF = F_{ab} + F_{bc} + F_{ca} \quad (5.7)$$

Using this definition, the exterior derivative d can be represented by the multiplication of k -forms with the incidence matrix ∂ or simply d^T . The incidence matrix ∂ , which is known as the boundary operator, is constructed of k -simplices on $(k+1)$ -simplices and therefore maps the k -form to a $(k+1)$ -form. This matrix consists of

one row for each k -simplex and one column for each $(k + 1)$ -simplex, hence making a large sparse matrix.

$$\begin{aligned}\partial_0 &= d_0^T = [\# \text{ vertices} \times \# \text{ edges}] \\ \partial_1 &= d_1^T = [\# \text{ edges} \times \# \text{ faces}] \\ \partial_2 &= d_2^T = [\# \text{ faces} \times \# \text{ tets}]\end{aligned}\tag{5.8}$$

Each incident k -simplex on the $(k + 1)$ -simplex results in a “1” if the orientation is the same, and “−1” if not. If the simplices are not incident then the result is a “0”. For example, building the boundary matrices for a single triangle ($[123]$) made up from three vectors ($[1,2],[1,3],[3,2]$) all defined from the principle vertices would result in the following

Figure 5.4: 2D mesh example

$$\partial_0 = d_0^T = \begin{bmatrix} -1 & 0 & -1 \\ 1 & 1 & 0 \\ 0 & -1 & 1 \end{bmatrix} \text{ and } \partial_1 = d_1^T = \begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix}\tag{5.9}$$

5.1.5 Hodge Star

Formally, the Hodge star is defined as an operator (denoted by $\star$) that performs a linear mapping from k -forms to $(n - k)$ -forms within the vector space V . The Hodge star is a necessity for the DEC solution as an exterior derivative operation on a k -form results in a $k + 1$ -form, limiting its applicability. As the Hodge star permits the transfer of integrand values from the primal mesh to the dual mesh and visa-versa, resulting in a mapping of a k -form to $(n - k)$ -form, the exterior derivative can be applied recursively allowing the formulation of more complex operators such as the Laplacian.

$$\star : \omega^k(V) \rightarrow \omega^{n-k}(V) \quad (5.10)$$

The definition of the Hodge star presented here is identical to that defined by [37], which functions by preserving the underlying geometry, creating a consistent domain of integration.

$$\frac{\omega}{Vol(\sigma)} = \frac{\star\omega}{Vol(\tilde{\sigma})} \quad (5.11)$$

where σ and $\tilde{\sigma}$ represent the primal and dual k -simplices respectively. Solving for the Hodge star results in the following map from primal to dual simplices

$$\star = \frac{Vol(\tilde{\sigma})}{Vol(\sigma)} \quad (5.12)$$

The DEC implementation of the Hodge star results in four matrices representing the mappings from a 0-primal to 3-dual, 1-primal to 2-dual, 2-primal to 1-dual, and 3-primal to 0-dual mesh. Defining a unit volume for the the vertices for the primal and dual simplices, the volumes for the edges, faces and tetrahedrons can be calculated by summing the lower dimensional simplices that are decomposed from the original. At this point it should be noted that other definitions for the Hodge star exist but are not covered here.

5.1.6 Discrete Curl

As defined previously, the curl of the velocity field results in the vorticity (ω) at a point (x,y,z) .

$$\omega = \nabla \times \mathbf{u} \quad (5.13)$$

Assuming that the flow across a face is stored in the vector of flows $\mathbf{U}$, the circulation can be calculated by mapping the flow to its dual edges by using the Hodge star ($\star$) operator. The summation of the dual edges around a dual face is calculated by d^T and results in the discrete implementation of the curl such that the discrete vorticity Ω is

$$\Omega = d^T \star \mathbf{U} \quad (5.14)$$

Please note that as the discrete vorticity is stored as a scalar (integral of vorticity over a dual face), its orientation is taken as that of the edge. For further clarification the reader is directed to Figure 5.5, as it includes an overview of how all the continuous and discrete operators relate to each other.

5.1.7 Discrete Laplacian

The Laplacian can be represented as:

$$\Delta = (\nabla \nabla \cdot - \nabla \times \nabla \times) \quad (5.15)$$

and therefore, as given by [37], it can be represented in the discrete sense by

$$\begin{aligned} \Delta &= L = (\star d \star^{-1} d^T \star + d^T \star d) \\ &= [\star_2][d_1][\star_1^{-1}][d_1^T][\star_2] + [d_2^T][\star_3][d_2] \end{aligned} \quad (5.16)$$

5.1.8 Summary of Operators

At this point, all the basic operators have been defined. Due to the difficulty in describing the interdependence of the various operators, Figure 5.5 has been constructed in an attempt to clarify these concepts with a graphical representation.

Figure 5.5: Summary of operators and how they relate in the discrete domain

Once the DEC operators are defined, they can be employed into a solution through simple matrix multiplication. For example, starting with a flux field ($\mathbf{f}$) defined as a 2-form over the primal mesh, the discrete divergence (d_2) and discrete curl ($\star_1^{-1} d_2^T \star_2$) of the vorticity field can be calculated through

$$\Omega = \nabla \times [\mathbf{f}] = \star_1^{-1} \mathbf{d}_2^T \star_2 [\mathbf{f}] \quad (5.17)$$

For examples of the various DEC operator matrix construction, please refer to

Appendix A.

5.2 Decomposition of the Flow Field

As stated previously, in any given domain, a flow can be broken down into three components: the divergence-free, curl-free, and harmonic parts. Thinking of curl and divergence as quantities related to the vorticity, sources and sinks, the respective mathematical operator can be applied to separate these features from the rest of the flow. Therefore, the resulting three flow fields consist of vortices, sources and sinks, and finally the remainder, which is comprised of the incompressible and irrotational part of the flow that satisfies the normal flow boundary conditions.

5.2.1 Helmholtz-Hodge Decomposition

For smooth data, there is a well known way to decompose a vector field into both intuitive and useful components (specifically the scalar potential ψ , vector potential ϕ and harmonic h), through Helmholtz-Hodge decomposition. As Polthier [38] states, the various parts of the flow can be decomposed and represented by

$$\mathbf{u} = \nabla\psi + \nabla \times \phi + h \quad (5.18)$$

where $\nabla\psi$ and $\nabla \times \phi$ are the curl-free and divergence-free components of the velocity field respectively. The remainder h is called the harmonic and contains normal component of the flow boundary conditions.

To guarantee a unique solution to the decomposition, $\nabla\psi$ is forced to be zero on

the boundary while $\nabla \times \phi$ is forced to be tangential. Furthermore, through vector identities, it can be seen that

$$\nabla \times \nabla \psi = 0 \quad (5.19)$$

$$\nabla \cdot (\nabla \times \phi) = 0 \quad (5.20)$$

Assuming the harmonic component is zero and taking the curl of equation 5.18 results in the rotational component of the flow,

$$\nabla \times \mathbf{u} = \overbrace{\nabla \times \nabla \psi}^{\text{Zero (from eqn 5.19)}} + \nabla \times \nabla \times \phi + \overbrace{\nabla \times h}^{\text{Zero (by definition)}} \quad (5.21)$$

$$\nabla \times \mathbf{u} = \nabla \times \omega \quad (5.22)$$

while taking the divergence of 5.18 results in the sources and sinks.

$$\nabla \cdot \mathbf{u} = \nabla \cdot \nabla \psi + \overbrace{\nabla \cdot \nabla \times \phi}^{\text{Zero (from eqn 5.20)}} + \overbrace{\nabla \cdot h}^{\text{Zero (by definition)}} \quad (5.23)$$

$$\nabla \cdot \mathbf{u} = \nabla \cdot \text{Divergence} \quad (5.24)$$

Since, the Curl of the vorticity (ω) equals the vorticity while the Div of the Divergence is results in the Divergence of the flow, the remaining component h must therefore satisfy the normal flow boundary conditions.

5.3 Time Step Advection

Kelvin's circulation theorem states that the circulation around any closed loop with no dissipation must remain constant with time. Mathematically, this can be written as

$$\frac{D\Gamma}{Dt} = 0 \quad (5.25)$$

During time step advection, the vorticity variable is updated. The restriction on this update is that the circulation around any closed loop is maintained over the time step. This is accomplished by calculating the circulation at the initial time step and enforcing this circulation at the current time. In this implementation, vorticity is stored as a value on a primal edge; through the use of a Hodge star the vorticity is then associated with a dual face. Therefore, the natural choice for the circulation loop includes the dual edges bounding the dual face. Since Stokes' theorem states that the integral of vorticity over a dual face is the circulation, the discrete mechanism is already in place.

5.4 Solution Methodology

To initialize the DEC solutions, one must start with an initial flow field U (kg/s) throughout a domain (for example by solving for a divergence free flow over the domain). Note: a flux through an input face ($kg/(m^2s)$) is stored as an integral value over the face or U (kg/s).

1. From primal face (kg/s) $\rightarrow$ mass flow rate (U)
  - calculate velocity at the centroid (dual vertex) from known U 's in the direction of each face

- calculate velocities at primal vertices (interpolated from dual vertices)
  - find backtracked location and velocity of the centroids at the beginning of time step (velocities are found by interpolating between the velocities at the primal vertices)
  - calculate flow along the original “deformed” dual edges
  - set new dual edge value equal to calculated original “deformed” dual edge value
2. Dual edge ($kg/(s \cdot m)$) $\rightarrow$ circulation down an edge
    - $[d^T][\text{Dual edge}] = [\text{Dual Face}]$
  3. Dual face ($kg/(s \cdot m)$) $\rightarrow$ circulation around dual face (vorticity Ω)
    - solve $L \times \Phi = \Omega$
    - set primal edge value to the solved vector potential (Φ)
  4. Primal edge ($kg/(s \cdot m^2)$) $\rightarrow$ vector potential (Φ)
    - solve for new flow rate U ($U = d(\Phi)$)
  5. Primal face (should be kg/s)

Using Helmholtz-Hodge decomposition, the curl of the vector potential coupled with a harmonic field will result in the reconstructed velocity field. According to [37] the 2-form H (harmonic) need only be solved when the normal flow boundary conditions or external forces are modified. If this occurs, the harmonic component H is initially initialized to 0, and is updated at each time-step with the harmonic component of these forces and the normal flow boundary conditions.

5.4.1 Time-step Advection Loop

Below is an excerpt of the time-step advection loop from the prototype DEC solver written in Scilab. The full code is shown in Appendix D.

```

for i=1:num_time_steps
    // ——— CALCULATE FLOW (kg/s) AT DUAL VERTICIES ———
    full_dual_0form = Find_centroid_val(full_primal_2form, num_cubes, num_vertex_x,
        num_vertex_y, num_vertex_z, debug_output_level);

    // ——— CONVERT TO VELOCITIES ———
    full_dual_0form_vel = Find_velocity(full_dual_0form, material_density, vertex_spacing
    );

    // ——— INTERPOLATE TO FIND velocities AT PRIMAL VERTICIES
    full_brick_values_27node = Interpolate_centroid_val(full_dual_0form_vel,
        full_primal_2form);

    // ——— BACKTRACK CENTROID LOCATION TO BEGINNING OF TIMESTEP ———
    backtracked_locations = Backtrack_centroids(full_dual_0form_vel, time_step_size,
        vertex_spacing);

    // ——— FIND velocity AT BACKTRACKED LOCATION ———
    // converts to r,s,t -> find new value -> convert back to x,y,z
    backtracked_vel = Find_backtracked_value(backtracked_locations,
        full_brick_values_27node, vertex_spacing);

    // ——— CALCULATE circulations (flows) down BACKTRACKED EDGES, SET THIS
    // circulation TO CURRENT DUAL EDGES
    // NOTE: Advection of the circulation loops
    dual_1form = Calculate_circulation(backtracked_locations, backtracked_vel,
        full_primal_2form, full_brick_values_27node) * material_density;
    // SUM TO AROUND DUAL EDGES AND CALCULATE VORTICITY ———
    dual_2form = Calculate_vorticity(dual_1form, boundary_conditions);

    // ——— EXPORT PARTICLE, VELOCITY AND VORTICITY INFORMATION TO A FILE FOR
    // IMPORTING INTO TECPLOT ETC. ———
    if export_velocity == 1 then // velocity exporting was turned on ...
    if export_vorticity == 1 then // vorticity advection was turned on ...
    if export_particle == 1 then // particle advection was turned on ...

```

```

// ----- VISCOSITY STEP -----
// Backward difference method. left matrix divide (backslash) solves A*x=b,
// linsolve computes solutions to A*x+b=0.
// A_viscous = (I_viscous - (viscosity * material_density * time_step_size *
// L1_viscous));
// b = dual_2form;
dual_2form = A_viscous\dual_2form;

// ----- SOLVE Poisson's equation (Laplace * vector_potential = vorticity) -----
// Solve the formatted matrices
// --> linsolve computes all the solutions to A*x+b=0.
// Solve ( L1 * vector_potential = vorticity )
// ( L1 * primal_1form = dual_2form )
rot_primal_1form = linsolve(L1,-dual_2form);
rot_primal_2form = (dl * (rot_primal_1form ));

// set flow to be div-free
rot_primal_2form(input_face) = divfree_solution(input_face);
full_primal_2form = rot_primal_2form;// + harmonic_solution;
end; //for i=1:num_time_step;

```

5.5 Boundary Conditions

Boundary conditions for the DEC solver are enforced through the normal and tangential components of the flow on each face. Because the DEC solver uses the vorticity formulation, the boundary conditions must be set when calculating the vorticity. As shown on Figure 5.6 (a), on the internal mesh, the vorticity is calculated through the circulation around a primal edge. On a boundary, this circulation “loop” must be altered through the following steps. First, when calculating the average flow down a dual edge normal to the boundary, one boundary and one centroid value is used instead of the standard two centroids. Furthermore, the length of the dual edge must be shortened accordingly. When setting a tangential velocity, the dual edge is assumed to exist on the boundary itself (Figure 5.6 (b)). If a no-slip condition exists,

these tangential circulation components (edges “c” and “d”) are zero and the loop is summed around the remaining dual edges only (edges “a”, “b” and “e”).

Figure 5.6: Tangential velocity components during circulation calculation

5.6 Flow Discontinuities On Boundaries

Assuming an initial configuration with fluxes set on two boundary faces, as shown in Figure 5.7 (a), it can be seen that velocity discontinuities arise at the mesh boundaries (Figure 5.7 (b)). These discontinuities arise when a specific flow-rate is set on one cell boundary, while the neighbouring cell’s flow-rate is defined to be zero on the boundary. This results in a common vertex being assigned two different values depending on which cell one is working with.

5.6.1 Possible Solutions To Avoid Flow Discontinuities

To alleviate flow discontinuities, two solutions are possible. First, the introduction boundary cells results in the ability to resolve, admittedly poorly, a boundary layer, and therefore have a consistent flow-rate at each vertex. See Figure 5.8 (a). Unfortunately, unless special precautions are taken, this will result in a different overall flow-rate than set by the user on the specific face.

Figure 5.7: Flow discontinuities at vertices arising from using a linear flow profile on each face

The second possible solution is to use a higher order function to represent the velocity within a cell as shown in Figure 5.8 (b). The major drawback of this method is anticipated to be computational time as integrating the flow profile over a boundary and interpolating values within a cell are common operations during backtracking.

5.7 Interface Tracking

Since the level-set method has no difficulty handling the large deformation of the interface and the drastic topological changes as the bubble evolves, it has been determined that the finite element model should employ the level-set method to track the bubble interface. The drawback of the level-set method is that it does not inherently preserve the volume of each of the two fluids. Since the volume error is proportional to the numerical grid spacing, the straight forward way to minimize this error is to increase the grid resolution. Many other more “cost” effective ways to refine the

Figure 5.8: Possible methods of avoiding flow discontinuities at vertices

mesh exist. These include the implementation of p- or (h,p)-version refinement algorithms around the interface of the fluids. In addition, other extended finite element approaches exist to resolve this problem and are explored in [39] [40] [41].

5.7.1 The Level-Set Method

With the level-set method, the interface is represented as a zero level-set of a continuous function designed to be positive on one side of the curve and negative on the other. Therefore, letting $\phi(x, t)$ be defined as

$$\phi(x, t) = \begin{cases} > 0 & \forall \in \Omega_1 \\ = 0 & \forall \in \Gamma_{int} \\ < 0 & \forall \in \Omega_2 \end{cases} \quad (5.26)$$

The level-set function $\phi(x, t)$ stores the information about the closest distance to any interface separating the two fluids. If more than two fluids are being tracked

within the computational domain, vector is used to store the level-sets. In this case, each level-set is first evolved on its own, resulting in mismatches with the interface locations. These mismatches are then resolved using the projection method. For further information on level-sets and the projection the reader is directed to [42].

Furthermore, since the level-set function is advected by the velocity field, it must be continually updated. To update the level-set function, a signed distance function that determines how close a given point x is to the boundary Γ_{int} must be evaluated.

5.7.2 Signed Distance Function

The signed distance function has positive values on the inside of Γ_{int} , and decreases in value as x approaches the boundary, at which point the signed distance function is zero and then proceeds to take negative values outside of Γ_{int} . The signed function $d(x)$ can be defined as

$$d(x) = \min(x - \tilde{x}) \quad \text{for all } \tilde{x} \in \partial\Omega \quad (5.27)$$

where initially

$$\phi(x, t = 0) = d(x, t = 0) \quad (5.28)$$

To implement this in the discrete domain, a vector that stores the level-set value at each mesh point is created. This vector describes the mesh point distance to the closest boundary of a specific interface. If the mesh point is further than a specified value, the distance does not need to be recorded (to reduce computational time) and should be assumed to be “ $\pm$ large”. Once the “distance functions” are complete, the interface locations must be compared and updated such that the two distances sum to zero. From this point, the level-set functions are updated by the velocity field.

Knowing the underlying velocity field, the standard level-set update equation can

be applied to update the level-set interface location

$$\frac{\partial \phi}{\partial t} + (u) \cdot \nabla \phi = 0 \quad \forall x \in \Gamma_{int} \quad (5.29)$$

Once the level-set function ϕ is defined, the interface normal vector $\hat{\mathbf{n}}$ and the interface curvature κ can be defined as

$$\hat{\mathbf{n}} = \frac{\nabla \phi}{|\nabla \phi|}, \quad \kappa = \nabla \times \hat{\mathbf{n}} \quad (5.30)$$

Using a similar convention, the material properties on the entire domain Ω can be defined as a function of ϕ

$$\rho(\phi) = \begin{cases} \rho_1 & \phi \geq 0 \\ \rho_2 & \phi \leq 0 \end{cases} \quad \forall \in \Omega \quad (5.31)$$

$$\mu(\phi) = \begin{cases} \mu_1 & \phi \geq 0 \\ \mu_2 & \phi \leq 0 \end{cases} \quad \forall \in \Omega \quad (5.32)$$

$$C_v(\phi) = \begin{cases} C_{v1} & \phi \geq 0 \\ C_{v2} & \phi \leq 0 \end{cases} \quad \forall \in \Omega \quad (5.33)$$

Therefore, the density, viscosity, and specific capacity at any point in Ω can be written as

$$\begin{aligned} \rho(\phi) &= \rho_B + (\rho_A - \rho_B)H(\phi), \\ \mu(\phi) &= \mu_B + (\mu_A - \mu_B)H(\phi). \\ C_v(\phi) &= C_{vB} + (C_{vA} - C_{vB})H(\phi). \end{aligned} \quad (5.34)$$

where H is defined as the Heaviside (unit step) function.

Chapter 6

Results

A two-phase liquid/vapour bubble flow consists of boundary effects, mass transfer and property discontinuities across the fluid-bubble interface. Developing test cases that model the bubble interface directly (such as the Stefan and Sucking problems) allows the development of the tools necessary to further the goal of a 3D vapour bubble model. It should be noted that the numerical solutions for the Stefan and Sucking problems detailed herein were obtained using the VrSuite interface solver, a space-time formulation developed by Goldak Technologies Inc. Lastly, the DEC solver itself must be tested for functionality and viability to be used in large scale problems. This compilation of sub-problems demonstrate the tools necessary to solve a 3D vapour bubble.

6.1 Stefan Problem

As defined in Figure 6.1, the Stefan problem is a well-defined phase change problem. The driving energy for the vaporization is drawn through the wall on the left and conducted through the vapour to the interface. The velocity of the vapour is zero by definition, while the liquid's velocity is a function of the mass flow rate across the interface ($\dot{m}$). The vaporization is driven by the vapour temperature while the

liquid temperature is set at the saturation temperature ($T_l = T_{sat}$). This problem is referred to as a one-phase Stefan problem as the liquid is set to be at the equilibrium temperature (T_{sat}), resulting in heat conduction and temperature gradient in the vapour phase only.

Figure 6.1: Definition of the “Stefan” problem

6.1.1 Conservation of Energy

For a fixed spatial domain with material velocity $\mathbf{u}$, the energy equation is:

$$\rho C_p \left(\frac{\partial T}{\partial t} + \mathbf{u} \cdot \nabla T \right) - \nabla \cdot (\mathbf{K} \nabla T) - \Phi - Q = 0 \quad (6.1)$$

For a 1D analysis with no viscosity ($\Phi = 0$) and no internal heat generation ($Q = 0$), the energy equation reduces to

$$\rho C_p \left(\frac{\partial T}{\partial t} + \mathbf{u} \frac{\partial T}{\partial x} \right) - \mathbf{K} \frac{\partial^2 T}{\partial x^2} = 0 \quad (6.2)$$

Vapour Domain

In the vapour domain, the velocity of the vapour is defined as zero ($\mathbf{u} = 0$). Therefore, in the vapour domain the energy equation is reduced to the 1D heat equation.

$$\rho C_p \frac{\partial T}{\partial t} - \mathbf{K} \frac{\partial^2 T}{\partial x^2} = 0 \quad (6.3)$$

Liquid Domain

It can be easily shown that no solution to the energy equation is required in the liquid domain as the temperature of the liquid is set to be constant throughout the analysis.

Interface Conditions

From the law of conservation of energy, it can be seen that the energy required for vaporization must be conducted from the vapour to the liquid through the interface.

$$\dot{m} h_{lv} = K_v \left. \frac{\partial T_v}{\partial x} \right|_{interface} - K_l \left. \frac{\partial T_l}{\partial \zeta} \right|_{interface} \quad (6.4)$$

For the Stefan problem, the liquid temperature is set to be constant at T_{sat} . Therefore, the energy required for vaporization is conducted only through the vapour phase.

$$\dot{m} h_{lv} = K_v \left. \frac{\partial T_v}{\partial x} \right|_{interface} \quad (6.5)$$

6.1.2 Conservation of Mass

Through the conservation of mass it can be seen that across the interface

$$\dot{m} = \rho_v \frac{\partial \mathbf{u}_{(v-s)}}{\partial x} = \rho_l \frac{\partial \mathbf{u}_{(l-s)}}{\partial \zeta} \quad (6.6)$$

therefore

$$\dot{m} = \rho_v(\mathbf{u}_v - \mathbf{u}_s) \quad (6.7)$$

$$\dot{m} = \rho_l(\mathbf{u}_l - \mathbf{u}_s) \quad (6.8)$$

Since $\mathbf{u}_v = 0$

$$\dot{m} = \rho_v \mathbf{u}_s \quad (6.9)$$

$$\mathbf{u}_s = \frac{\dot{m}}{\rho_v} \quad (6.10)$$

Hence, the velocity of the liquid may be simply calculated as

$$u_l = \frac{\dot{m}}{\rho_l} + u_s \quad (6.11)$$

The analytical solution to the Stefan problem follows the following procedure:

As given by Gupta [8], equation 6.3 can be written in terms of an error function

$$x_s(t) = 2\delta(K_v t)^{\frac{1}{2}} \quad (6.12)$$

$$T_v = A \operatorname{erf} \left[\frac{x}{2(K_v t)^{\frac{1}{2}}} \right] + T_{wall} \quad (6.13)$$

As given by [8] and [9], δ and A are unknown constants that must be determined through the solution of

$$\delta \exp(\delta^2) \operatorname{erf}(\delta) = \frac{c_p(T_{wall} - T_{sat})}{h_{lg}\sqrt{\pi}} \quad (6.14)$$

$$A = \frac{T_{sat} - T_{wall}}{\operatorname{erf}(\delta)} \quad (6.15)$$

The solution to the transcendental equation 6.14 was obtained through graphical means as described in Appendix B.

Figure 6.2: Temperature profile for the analytical solution of the Stefan problem

6.1.3 Stefan Problem Results

The analytical and numerical results for the Stefan problem are shown in Figures 6.3 and 6.4. The numerical solution was calculated using VrSuite interface solver, a space-time formulation developed by Goldak Technologies Inc. For comparison to the analytical solution, the software was modified to instantaneously force a linear temperature gradient at any point in time in the solution. The temperature gradients in the solution are shown in Figure 6.3.

If one investigates the analytical solution (Figure 6.4), it becomes evident that the temperature profile assumed in the solution is linear (Figure 6.2). This is a distinct departure from a temperature profile that was observed in the numerical solution (before modification).

Due to the discrepancy between the analytical and numerical solutions, both solutions

Figure 6.3: Stefan problem interface location at 1600s

Figure 6.4: Analytical and numerical solutions for the interface position of the Stefan problem

are presented in Figure 6.4. It is obvious that the “constant temperature gradient” solution resembles the analytical solution, while the more realistic “heat equation solution” displacement is 2.6 times lower at 1600 seconds. Furthermore, an even more realistic solution would consider the thermal expansion of vapour.

6.2 Sucking Problem

As defined by Welch and Wilson [9], the Sucking problem is analogous to the 1D bubble problem. In the Sucking problem, the vapour velocity and temperature are defined to be zero and at the vapour saturation temperature respectively, while the liquid has a predefined superheated ($T_l > T_{sat}$) far-field temperature and constant velocity with respect to x . The interface velocity is therefore driven by the mass flow rate ($\dot{m}$) across the interface which is driven by the superheated liquid temperature. Welch and Wilson [9] dubbed this the Sucking problem because the “diffusive spreading of the thermal layer is counteracted somewhat by a sucking of the thermal layer towards the interface”. This counteracting effect results in a thin thermal layer.

Figure 6.5: Definition of the Sucking problem

At any point in time, ζ may be obtained through the following transformation

Figure 6.6: Initial conditions of the Sucking problem

$$\zeta = x - \int_0^t \mathbf{u}_s(t) dt. \quad (6.16)$$

6.2.1 Conservation of Energy

$$\rho C_p \left(\frac{\partial T}{\partial t} + \mathbf{u} \cdot \nabla T \right) - \nabla \cdot (\mathbf{K} \nabla T) - Q - \Phi = 0 \quad (6.17)$$

where

$$\Phi = 2\mu D : D \quad (6.18)$$

For the special case of the 1-D Sucking problem, the internal heat generation Q and viscous dissipation Φ are zero. Therefore, the energy equation reduces to

$$\rho_v C_p \left(\frac{\partial T_v}{\partial t} + \mathbf{u}_v \frac{\partial T_v}{\partial x} \right) - \mathbf{K}_v \frac{\partial^2 T_v}{\partial x^2} = 0 \quad (6.19)$$

$$\rho_l C_p \left(\frac{\partial T_l}{\partial t} + (\mathbf{u}_l - \mathbf{u}_s) \frac{\partial T_l}{\partial \zeta} \right) - \mathbf{K}_l \frac{\partial^2 T_l}{\partial \zeta^2} = 0 \quad (6.20)$$

where the subscripts “ v ” and “ l ” refer to the vapour and liquid phases respectively.

It should be noted that since the vapour is stationary and without temperature gradients, the energy equation for the vapour phase reduces to zero. Note that the advective term for liquid should include the difference in material velocity and the mesh velocity. In a Lagrangian formulation, the material velocity equals the mesh velocity. Therefore this term is zero. In an Eulerian formulation, the mesh velocity is zero, and therefore, the term has the material velocity. In a space-time formulation or arbitrary Eulerian-Lagrangian formulation, one needs both velocities.

6.2.2 Conservation of Mass

The conservation of mass states that mass cannot be created or destroyed. Generally, this can be written as

$$\frac{\partial \rho}{\partial t} + \nabla(\rho \mathbf{u}) = 0 \quad (6.21)$$

Assuming a 1-D divergence free flow, this reduces to

$$\frac{\partial \mathbf{u}}{\partial x} = 0 \quad (6.22)$$

as $\mathbf{u} = 0$ in vapour and $\mathbf{u} = \text{constant}$ in the liquid (with respect to x).

6.2.3 Interface Conditions

At the interface, both mass and energy must be conserved. The energy required for vaporization is conducted through the vapour and liquid to the interface

$$\dot{m}h_{lv} = K_v \frac{\partial T_v}{\partial x} \Big|_{interface} - K_l \frac{\partial T_l}{\partial \zeta} \Big|_{interface} \quad (6.23)$$

It should be noted that the thermal conductivity of the vapour (K_v) will change with the vapour pressure (ρ_v). For the 1D Sucking problem, the effects may be neglected

as gravity (and therefore pressure) terms are omitted. Furthermore, since there is no energy flux transfer from the vapour phase, it can be concluded that the energy required for vaporization is conducted through the liquid only.

$$\dot{m}h_{lv} = -K_l \frac{\partial T_l}{\partial \zeta} \Big|_{interface} \quad (6.24)$$

Rearranging for the mass flow rate yields

$$\dot{m} = -\frac{K_l}{h_{lv}} \frac{\partial T_l}{\partial \zeta} \Big|_{interface} \quad (6.25)$$

Through the conservation of mass it can be seen that across the interface

$$\dot{m} = \rho_v \frac{\partial \mathbf{u}_{(v-s)}}{\partial x} = \rho_l \frac{\partial \mathbf{u}_{(l-s)}}{\partial \zeta} \quad (6.26)$$

therefore

$$\dot{m} = \rho_v (\mathbf{u}_v - \mathbf{u}_s) \quad (6.27)$$

$$\dot{m} = \rho_l (\mathbf{u}_l - \mathbf{u}_s) \quad (6.28)$$

Since $\mathbf{u}_v = 0$ for the Sucking problem

$$\dot{m} = \rho_v \mathbf{u}_s \quad (6.29)$$

$$\mathbf{u}_s = \frac{\dot{m}}{\rho_v} \quad (6.30)$$

6.2.4 Sucking Problem Results

Combining the conservation of mass with the energy equation at the interface results in

$$\rho_v \mathbf{u}_s = \rho_l (\mathbf{u}_l - \mathbf{u}_s) = - \frac{K_l}{h_{lv}} \frac{\partial T_l}{\partial \zeta} \Big|_{interface} \quad (6.31)$$

The interface velocity can be calculated

$$\mathbf{u}_s = \frac{-K_l}{\rho_v h_{lv}} \frac{\partial T_l}{\partial \zeta} \Big|_{interface} \quad (6.32)$$

Rearranging the conservation of mass across the interface,

$$(\mathbf{u}_l - \mathbf{u}_s) = \frac{\rho_v}{\rho_l} (\mathbf{u}_s) \quad (6.33)$$

the energy equation can be written in terms of the interface velocity ($\mathbf{u}_s$)

$$\rho_l C_p \left(\frac{\partial T_l}{\partial t} + \frac{\rho_v}{\rho_l} (\mathbf{u}_s) \frac{\partial T_l}{\partial \zeta} \right) - \mathbf{K}_l \frac{\partial^2 T_l}{\partial \zeta^2} = 0 \quad (6.34)$$

The interface velocity can be solved using numerical methods by applying the initial conditions with a temperature discontinuity at the interface,

The numerical solution was obtained in an identical manner to that described for the Stefan problem. The interface positions and velocities are shown in Figures 6.7 and 6.8 respectively.

6.3 Prototype DEC Solver Results

Once the development of the Scilab prototype DEC solver was completed, results for Couette and Poiseuille flow test cases were obtained. Because of the test nature of the prototype solver, it has many limitations. These include a “hard-coded” geometry that is limited to a rectangular domain and poor memory usage that severely limits possible size of the domain. Furthermore, it should be noted that due to poor interpolation velocity values (especially apparent on boundaries), the solution from

Figure 6.7: Numerical solution for the interface position of the Sucking problem

Figure 6.8: Numerical solution for the interface velocity of the Sucking problem

the solver does not constrain a divergence-free flow. This problem could be mitigated through the implementation of the pressure Poisson equation. It should be noted that since the prototype solver does not converge to accurate solutions, it is not directly compared to any results but used as an aid to determine the characteristic profile of the flow.

Prototype Solver Setup

The prototype solver domain consists of a rectangular grid comprised of 7 elements wide, 3 elements deep and 11 elements long. Through experimentation, it was determined that a geometry no larger than that specified would allow for relatively quick solutions (within 30 minutes) on low-end portable hardware (1.6 GHz Pentium M). The selected element size was $0.01m^3$, resulting in a domain $7 \times 11 \times 3$ cm. The selected material properties were that of water at standard temperature and pressure. The input flux was set to $100kg/(m^2s)$. The prototype solver configuration is detailed in Figure 6.9.

6.3.1 Couette Flow

Couette flow is one of the few viscous flow problems which has an exact analytical solution. For this reason, it is an ideal problem for verifying numerical solutions. Couette flow also exhibits characteristics of more complicated boundary layer flows [43].

The Couette flow problem consists of a viscous fluid between two parallel plates separated by a given distance and moving with a given velocity with respect to each other, as shown in Figure 6.10.

The analytical solution for the Couette flow is derived from the x-momentum equation

Scilab

Toggle Menu

Please select a flow type: Poiseuille Jet **Couette**

Output divfree velocity/vorticity/particle data? Yes No

Output harmonic velocity/vorticity/particle data? Yes No

Advect particles in transient solution? Yes No

Transient particle advection
- Advect once every X time-steps: 1 2 5 10 100

Output velocity data for transient solution? Yes No

Transient velocity output
- Output once every X time-steps: 1 2 5 10 100

Output vorticity data for transient solution? Yes No

Transient vorticity output
- Output once every X time-steps: 1 2 5 10 100

Left boundary condition: Slip No-slip

Right boundary condition: Slip No-slip

Top boundary condition: Slip No-slip

Bottom boundary condition: Slip No-slip

Front boundary condition: Slip No-slip

Back boundary condition: Slip No-slip

Ok Cancel

(a) Configuration window 1

Scilab

Please enter initial conditions and material parameters:

Number of cubes in [X,Y,Z] directions: 7,3,11

Cube edge length: 0.01

Initial flux on input face (kg/(m² s)): 100

Material density (kg/m³): 1000

Kinematic viscosity (N s/m²) [value * 10⁽⁻⁶⁾]: 1.307

Time step size (s): 0.01

Number of time steps: 150

Ok Cancel

(b) Configuration window 2

Figure 6.9: Prototype Scilab DEC solver configuration

Figure 6.10: Couette flow schematic

$$\rho \frac{Du}{Dt} = -\frac{\partial p}{\partial x} + \frac{\partial \tau_{xx}}{\partial x} + \frac{\partial \tau_{yx}}{\partial y} + \frac{\partial \tau_{zx}}{\partial z} + \rho f_x \quad (6.35)$$

where ρf_x is the body force acting in the x direction on the fluid element resulting in the shear stress (measure of the deformation of the fluid element) is given by τ_{yx} . The normal stress given by τ_{xx} can be neglected following the assumption that a flow will remain incompressible if the flows Mach number is below 0.3.

Assuming the flow is steady-state, infinite in the $\pm x$ direction, and independent of x and t , the momentum equation can be reduced to

$$\frac{\partial \tau_{yx}}{\partial y} = 0 \quad (6.36)$$

Substituting equations 4.7 through 4.12 results in the following solution

$$\frac{\partial^2 u_x}{\partial y^2} = 0 \quad (6.37)$$

Integrating twice results in the linear velocity gradient

$$u_x = c_1 y + c_2 \quad (6.38)$$

where c_1 and c_2 are constants of integration. Setting the top and bottom plate velocities to u_{top} and zero respectively (as shown in Figure 6.10), allows one to solve for $c_1 = u_{top}/D$ and $c_2 = 0$. The final analytical Couette flow solution is

$$u_x = \frac{u_{top} y}{D} \quad (6.39)$$

As seen in Figure 6.11, the correct linear flow profile is starting to develop. Due to limitations in the domain size and the divergent nature of the solver, a fully developed flow is impossible with the Scilab prototype solver.

Figure 6.11: Couette velocity results from prototype DEC solver

6.3.2 Poiseuille Flow

For further verification of the DEC theory, a Poiseuille solution was attempted. Once again, even though there appear to be some issues with the inlet vorticities (seen on the bottom of Figure 6.12 b, c and d), the solution does follow what would be expected for a parabolic Poiseuille pipe flow given by

$$v = -\frac{1}{4\eta} \frac{\Delta P}{\Delta x} (R^2 - r^2) \quad (6.40)$$

where η is the viscosity, R is the tube radius and r is the distance from the centre of the tube. It should be noted that ΔP is directly related to the flowrate through the pipe.

Unfortunately, due to domain size limitations, a fully developed flow is deemed impossible with the prototype solver.

6.4 Vr Fluid Flow DEC Solver

With the aid of the Scilab prototype solver, Goldak Technologies Inc. implemented a DEC solver as an addition to their existing Vr Software Suite called Vr Fluid Flow. As Vr Software Suite is a full fledged FEM solver capable in solving large complex problems with excellent post processing and visualization capabilities, the Couette and Poiseuille test cases were solved with much higher mesh densities. More importantly, the availability of Vr Software Suite's pressure Poisson equation solver allowed for solution of these problems while maintaining a divergence free state.

Vr Fluid Flow Solver Setup

The domain configured for the Vr Fluid Flow solver consists of a rectangular section 9 elements wide, 9 element high, and 90 elements long. The solution domain and mesh

(a) 1 timestep (0.01s)

(b) 50 timesteps (0.50s)

(c) 100 timesteps (1.00s)

(d) 150 timesteps (1.50s)

Figure 6.12: Poiseuille vorticity results from prototype DEC solver

(a) 1 timestep (0.01s)

(b) 50 timesteps (0.50s)

(c) 100 timesteps (1.00s)

(d) 150 timesteps (1.50s)

Figure 6.13: Poiseuille velocity results from prototype DEC solver

Figure 6.14: Poiseuille particle advection results from prototype DEC solver

is shown in Figure 6.15 (a). A proposed more detailed solution domain (20 elements wide, 20 elements high and 200 elements long) is shown in 6.15 (b). The solution for both the Couette and Poiseuille flows was obtained by setting the input velocity at $1m/s$ with a time-step size of $0.001s$ for 100 time-steps. As the domain dimensions are $0.01m$ wide, $0.01m$ high and $0.1m$ long, the resulting element size is $0.0011m^3$, resulting in a Courant number below 1. As the outlet vorticity was not constrained, the outlet boundary condition is equivalent to a zero gauge pressure boundary.

Vr Fluid Flow Solver Results

As seen on Figures 6.16, 6.17, 6.18 and 6.19, fully developed flows are evident for both the Couette and Poiseuille cases. Furthermore, both Couette and Poiseuille flows are exhibiting their expected behaviours by tending to “linear” and “parabolic” velocity profiles for the Couette and Poiseuille flows, respectively. However, it is evident that the ideal linear Couette velocity profile is not obtained 6.17 (b), resulting in residual error. The cause of this error is not fully understood and further investigation is required. One possibility is the further need to refine the mesh as shown in Figure 6.15 (b).

(a) Actual solution mesh

(b) Proposed solution mesh

Figure 6.15: Vr Fluid Flow solution domain with mesh

(a) Couette flow vorticity

(b) Couette flow vorticity at outlet

Figure 6.16: Couette flow vorticity

(a) Couette flow velocity

(b) Couette flow velocity at outlet

Figure 6.17: Couette flow velocity

 Flow Direction

(a) Poiseuille flow vorticity

(b) Poiseuille flow vorticity at outlet

Figure 6.18: Poiseuille flow vorticity

(a) Poiseuille flow velocity

(b) Poiseuille flow velocity at outlet

Figure 6.19: Poiseuille flow velocity

Chapter 7

Conclusions

Because two-phase flows are commonly used as a means of high-efficiency heat transfer, their importance in the engineering community cannot be understated. Modelling 3D vapour-liquid interactions directly would allow researchers and engineers to gain insight into current and future designs, increasing the efficiency of many systems. To aid in this goal, the necessary tools to solve two-phase flow problems were presented throughout this thesis.

The first distinguishable characteristic of a two-phase flow problem is the interface between the vapour and liquid phase. Therefore, two interface problems were solved to develop the numerical tools necessary.

The classical Stefan problem, originally applied to ice formation, was modified to solve the necessary vaporization across an interface. Examination of the analytical solution (Figure 6.4), reveals a linear temperature profile (Figure 6.2). This is a distinct departure from both the expected and observed numerical solution temperature profile. This departure is attributed to the neglected addition of “cooler” vapour at the interface, allowing the maintenance of a linear temperature gradient (as the vapour velocity is zero).

The Sucking problem, as defined by Welch and Wilson [9], is analogous to a 1D bubble problem, and therefore is also of interest as it demonstrates the machinery

necessary to model bubble interface conditions. As expected, the interface velocity is driven by the mass flow rate across the interface, which is driven by the surrounding superheated liquid temperature.

The accurate capture of vortices shed from bubbles as they move through the liquid is necessary for meaningful results. Therefore, this research also involved the development of the tools necessary to implement a discrete exterior calculus vorticity based method, which intrinsically preserves angular momentum. This approach is unique in that it bridges a gap between the engineering discipline and discrete exterior calculus, a commonly overlooked mathematical field that is ideally suited for solving complex problems in a discrete domain. The Scilab prototype solver written by the student (see Appendix D) was tested on two problem cases (Couette and Poiseuille flow), and was also implemented into Vr Software Suite by Goldak Technologies Inc.

The DEC finite element solver presented here allows for a fast and efficient solution of the Navier-Stokes equations while preserving angular momentum intrinsically. Furthermore, its compatibility with the underlying geometric, topological, and algebraic structures results in well-posed problems and, therefore, stable solutions. For simplicity, both the prototype solver written by the student and the solver implemented by Goldak Technologies Inc. in Vr Software Suite used a structured brick mesh that is not ideally suited for complex geometries. However, the implementation of a primary tetrahedron mesh with a Voronoi dual mesh would significantly increase the ability to resolve complex shapes and interfaces.

The interface and DEC solvers presented in this research are key tools necessary to build a fully functional two-phase flow solution. As such, this research lays the foundation for future work in this area. Ongoing development should include the merging of the DEC solver with the proposed level-set interface tracking method, with the aim of developing a two-phase flow solver without thermodynamic interface conditions. Once the DEC solver is working seamlessly with the selected interface

tracking method, the addition of the interface solver would allow the solution of two-phase thermodynamic problems.

List of References

- [1] C. Ohl, A. Tijink, and A. Prosperetti, “The added mass of an expanding bubble,” *Journal of Fluid Mechanics*, vol. 482, pp. 271–291, 2003.
- [2] W. Gutkowski and T. A. Kowalewski, “Mechanics of the 21st century,” in *proceedings of the 21st International Congress of Theoretical and Applied Mechanics*, (Warsaw, Poland), pp. 7–15, Springer, August 2004.
- [3] “Millennium prize problems.” Retrieved 13 Dec. 2008 from the Clay Mathematics Institute website at <http://www.claymath.org/millennium>.
- [4] D. Bernoulli, “Hydrodynamic, A Work by Bernoulli.” Retrieved 13 Dec. 2008, from Encyclopedia Britannica Online at <http://www.britannica.com/EBchecked/topic/658890/Hydrodynamica>.
- [5] “History of theoretical fluid dynamics.” Retrieved 07 Dec. 2008 from the Centrum Wiskunde & Informatica website at <http://www.cwi.nl/en/fluidynamicshistory>.
- [6] J. Anderson Jr., *Modern Compressible Flow*, pp. 249–250. McGraw-Hill, 1990.
- [7] D. Lohse, “Bubble puzzles,” *Physics Today*, p. 36, February 2003.
- [8] S. W. Gupta, *The Classical Stefan Problem: Basic Concepts, Modelling and Analysis*. Sara Burgerhartstraat 25, Amsterdam, The Netherlands: Elsevier Science B.V., 2003.
- [9] S. Welch and J. Wilson, “A volume of fluid based method for fluid flows with phase change,” *Journal of Computational Physics*, vol. 160, pp. 662–682, 2000.
- [10] W. Minkowycz, *Advances in Numerical Heat Transfer*, pp. 297–300. Taylor and Francis, 2001.
- [11] J. Anderson Jr., *Computational Fluid Dynamics*, pp. 142–145. McGraw-Hill, 1995.

- [12] R. P. Fedkiw and X. dong Liu, "The ghost fluid method for viscous flows," 1998.
- [13] W. Habashi and M. Hafez, *Computational Fluid Dynamics Techniques*, p. 397. CRC Press, 1995.
- [14] F. Harlow and J. Welch, "Numerical calculation of time-dependent viscous incompressible flow of fluid with a free surface," *The Physics of Fluids*, vol. 8, pp. 2182–2189, 1996.
- [15] I. D. Mishev, "Finite volume methods on voronoi meshes," *Numerical Methods for Partial Differential Equations*, vol. 14, pp. 193–212, 1998.
- [16] C. Mattussi, "An analysis of finite volume, finite element, and finite difference methods using some concepts from algebraic topology," *Journal of Computational Physics*, vol. 133, pp. 289–309, 1997.
- [17] R. S. F. D. N. Arnold and R. Winther, *Acta Numerica*, vol. 15, ch. Finite Element Exterior Calculus, Homological Techniques, and Applications, p. 1155. Cambridge University Press, 2006.
- [18] R. d. B. Erwin Stein and T. J. Hughes, eds., *Encyclopedia of Computational Mechanics, Volume 1: Fundamentals*. John Wiley & Sons, Ltd., 2004.
- [19] C. Hirt, A. Amsden, and J. Cook, "An arbitrary lagrangian-eulerian computing method for all flow speeds," *Journal of Computational Physics*, vol. 14, pp. 227–253, 1974.
- [20] F. Harlow, "A machine calculation method for hydrodynamic problems," *Los Alamos Scientific Laboratory report LAMS-1956*, November 1955.
- [21] H. Okuda, "Nonphysical noises and instabilities in plasma simulation due to a spatial grid," *Journal of Computational Physics*, vol. 10, pp. 475–486, 1972.
- [22] D. Cardon, D. Cline, and P. Egbert., "Fluid flow for the rest of us: Tutorial of the marker and cell method in computer graphics." Retrieved 13 Nov. 2008 from <http://poseidon.cs.byu.edu/~cline/fluidFlowForTheRestOfUs.pdf>.
- [23] Y. N. Grigoryev, V. A. Vshivkov, and M. P. Fedoruk, *Numerical "particle-in-cell" Methods: Theory and Applications*, pp. 85–86. VSP, 2002.
- [24] N. Forster and R. Fedkiw, "Practical animation of liquids," in *the 28th annual conference on Computer graphics and interactive techniques*, (New York, NY, USA), pp. 23–30, ACM Press, 2001.

- [25] S. Osher and R. Fedkiw, *Level Set Methods and Dynamic Implicit Surfaces*, p. 7. Springer, 2003.
- [26] E. Tonti, “The reason for analogies between physical theories,” *Applied Mathematical Modelling*, vol. 1, pp. 37–50, June 1976.
- [27] A. Hirani, *Discrete Exterior Calculus*. PhD thesis, California Institute of Technology, USA, 2003.
- [28] S. Elcott and P. Schröder, “Building your own DEC at home,” in *Discrete Differential Geometry: An Applied Introduction*, pp. 55–59, SIGGRAPH, 2006.
- [29] P. Bamberg and S. Sternberg, eds., *A Course in Mathematics for Students of Physics*. Cambridge University Press, 1988.
- [30] E. Cartan and S. Finikov, *Riemannian geometry in an orthogonal frame: from lectures delivered by Elie Cartan at the Sorbonne in 1926-1927 / translated from Russian by Vladislav V. Goldberg; foreword by S. S. Chern*. World Scientific Publishing Co. Pte. Ltd., 2001.
- [31] J. Anderson Jr., *Computational Fluid Dynamics*, pp. 64–65. McGraw-Hill, 1995.
- [32] T. Belytschko and J. Chessa, “An extended finite element method for two-phase fluids,” *Journal of Applied Mechanics*, vol. 70, pp. 10–17, January 2003.
- [33] M. Plesset and S. Zwick, “A nonsteady heat diffusion problem with spherical symmetry,” *Journal of Applied Physics*, vol. 23, no. 1, pp. 95–98, 1952.
- [34] M. Plesset and S. Zwick, “The growth of vapor bubbles in superheated liquids,” *Journal of Applied Physics*, vol. 25, no. 4, pp. 493–500, 1954.
- [35] M. Plesset and S. Zwick, “On the dynamics of small vapor bubbles in liquid,” *Journal of Mathematical Physics*, vol. 33, pp. 308–330, 1954.
- [36] Y. Tong, S. Lombeyda, A. N. Hirani, and M. Desbrun, “Discrete multiscale vector field decomposition,” in *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*, (New York, NY, USA), pp. 445–452, ACM Press, 2003.
- [37] S. Elcott, “Discrete, circulation-preserving, and stable simplicial fluids,” Masters of Science, California Institute of Technology, USA, May 2005.
- [38] K. Polthier and E. Preu, “Variational approach to vector field decomposition.”

- [39] M. Herrmann, “Refined level set grid method for tracking interfaces,” in *Annual Research Briefs*, pp. 3–18, Center for Turbulence Research, 2005.
- [40] A. Gerstenberger and W. Wall, “An extended finite element method based approach for large deformation fluid-structure interaction,” in *European Conference on Computational Fluid Dynamics*, ECCOMAS CFD, September 2006.
- [41] A. Smolianski, “Finite-element/level-set/operator-splitting (felsos) approach for computing two-fluid unsteady flows with free moving interfaces,” *International Journal for Numerical Methods in Fluids*, vol. 48, no. 3, pp. 231–269, 2005. Institute of Mathematics, University of Zurich, Winterthurerstr. 190, CH-8057 Zurich, Switzerland.
- [42] F. Losasso, T. Shinar, A. Selle, and R. Fedkiw, “Multiple interacting liquids,” in *SIGGRAPH ’06: ACM SIGGRAPH 2006 Papers*, (New York, NY, USA), pp. 812–819, ACM Press, 2006.
- [43] J. Anderson Jr., *Computational Fluid Dynamics*, pp. 416–420. McGraw-Hill, 1995.

Appendix A

Examples of the DEC Machinery

To aid in the understanding of the DEC solver formulation, examples of the necessary discrete operators are shown below. It should be noted that all of the example operators are exhibited for a sample mesh constructed of two tetrahedrons.

A.1 Exterior Derivative

d0 matrix: The exterior derivative maps node values to edge values.

$$[\mathbf{e}] = [\mathbf{d_0}][\mathbf{v}] \tag{A.1}$$

For two tetrahedrons: [9-edges $\times$ 5-nodes]

$$\begin{bmatrix} e_0 \\ e_1 \\ e_2 \\ e_3 \\ e_4 \\ e_5 \\ e_6 \\ e_7 \\ e_8 \end{bmatrix} = \begin{bmatrix} -1 & 1 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 \\ 0 & -1 & 1 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 \\ 0 & -1 & 0 & 0 & 1 \\ 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & -1 & 1 \end{bmatrix} \begin{bmatrix} v_0 \\ v_1 \\ v_2 \\ v_3 \\ v_4 \end{bmatrix}$$

d1 matrix: Maps edge values to face values.

$$[\mathbf{f}] = [\mathbf{d}_1][\mathbf{e}] \tag{A.2}$$

For two tetrahedrons: [7-faces $\times$ 9-edges]

$$\begin{bmatrix} f_0 \\ f_1 \\ f_2 \\ f_3 \\ f_4 \\ f_5 \\ f_6 \end{bmatrix} = \begin{bmatrix} 1 & -1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & -1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & -1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & -1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 1 \end{bmatrix} \begin{bmatrix} e_0 \\ e_1 \\ e_2 \\ e_3 \\ e_4 \\ e_5 \\ e_6 \\ e_7 \\ e_8 \end{bmatrix}$$

d2 matrix: Maps face values to tetrahedron values.

$$[\mathbf{t}] = [\mathbf{d_2}][\mathbf{f}] \tag{A.3}$$

For two tetrahedrons: [2-tets $\times$ 7-faces]

$$\begin{bmatrix} t_0 \\ t_1 \end{bmatrix} = \begin{bmatrix} -1 & 1 & -1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 1 & -1 & 1 \end{bmatrix} \begin{bmatrix} f_0 \\ f_1 \\ f_2 \\ f_3 \\ f_4 \\ f_5 \\ f_6 \end{bmatrix}$$

A.2 Primal and Dual Volumes

For simplicity, the specific lengths, areas, and volumes used throughout the Hodge star calculations assume equilateral tetrahedrons.

$$\begin{aligned}
 f_H &= \frac{1}{2}(\sqrt{3} \cdot e_L) \\
 f_A &= \frac{1}{2}(e_L \cdot f_H) \\
 t_H &= \frac{1}{3}(\sqrt{6} \cdot e_L) \\
 t_V &= \frac{1}{3}(f_A \cdot f_H)
 \end{aligned} \tag{A.4}$$

The edge length, face height, face area, tetrahedron height and tetrahedron volume is represented by e_L , f_H , f_A , t_H , and t_V , respectively. Throughout the DEC solver, these geometric metrics are all referred to as “volumes”. In the specific case of two tetrahedrons, the primal and dual volumes would be:

Primal Volumes:

$$\begin{aligned}
 Vol(\sigma)_0 &= 1 \\
 Vol(\sigma)_1 &= e_L \\
 Vol(\sigma)_2 &= f_A \\
 Vol(\sigma)_3 &= t_V
 \end{aligned} \tag{A.5}$$

Dual Volumes:

$$\begin{aligned}
 Vol(\tilde{\sigma})_3 &= \frac{1}{4}(t_V) \\
 Vol(\tilde{\sigma})_2 &= \frac{1}{12}(\sqrt{3} \cdot f_H) \\
 Vol(\tilde{\sigma})_1 &= \frac{1}{4}(t_H) \\
 Vol(\tilde{\sigma})_0 &= 1
 \end{aligned} \tag{A.6}$$

A.3 Hodge Star

As shown in Figure 5.5, the Hodge Star transfers integrand values from the primal mesh to the dual mesh and visa-versa. Using this definition, the corresponding $\star_0$, $\star_1$, $\star_2$ and $\star_3$ matrices are:

$\star_0$: Maps primal 0-forms (verticies) to dual 3-form (voronoi cells)

$$\star_0 = \begin{bmatrix} 0.25 & 0 & 0 & 0 & 0 \\ 0 & 0.5 & 0 & 0 & 0 \\ 0 & 0 & 0.5 & 0 & 0 \\ 0 & 0 & 0 & 0.5 & 0 \\ 0 & 0 & 0 & 0 & 0.25 \end{bmatrix} \quad (\text{A.7})$$

$\star_1$: Maps primal 1-forms (edges) to dual 2-form (dual faces)

$$\star_1 = \begin{bmatrix} 0.125 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0.125 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.125 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.25 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.25 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.125 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0.25 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.125 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.125 \end{bmatrix} \quad (\text{A.8})$$

$\star_2$: Maps primal 2-forms (faces) to dual 1-forms (dual edges - edges connecting dual vertices)

$$\star_2 = \begin{bmatrix} 0.236 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0.236 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.236 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.471 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.236 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.236 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0.236 \end{bmatrix} \quad (\text{A.9})$$

$\star_3$: Maps primal 3-forms (tetrahedrons) to dual 0-form (dual vertices - circumcentres of the tetrahedrons)

$$\star_3 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad (\text{A.10})$$

A.4 Gradient, Curl and Divergence

It should be noted that each of these operators have two definitions, one starting with values on the dual mesh and one starting on the primal mesh. Depending if the initial flux values were defined over the primal or dual surfaces will dictate the required Gradient, Curl or Divergence definitions needed. Both definitions will results in the same physical interpretation of the solution, however, it is necessary to be consistent so that the metrics agree. For example, if one starts with fluxes on the primal faces, the circulation must be calculated down the dual edges, and therefore the vorticity is the sum of the circulation around a dual face.

Gradient:

$$\text{Primal grad (Primal 0-form to 1-form)} \rightarrow \nabla = G = [d_0] \quad (\text{A.11})$$

$$\text{Dual grad (Primal 3-form to dual 1-form)} \rightarrow \nabla = G = [d_2^T][\star_3] \quad (\text{A.12})$$

$$[d_2^T][\star_3] = \begin{bmatrix} -1 & 0 \\ 1 & 0 \\ -1 & 0 \\ 1 & -1 \\ 0 & 1 \\ 0 & -1 \\ 0 & 1 \end{bmatrix}$$

Curl:

$$\text{Primal curl (Primal 1-form to 2-form)} \rightarrow \nabla \times = C = [d_1] \quad (\text{A.13})$$

$$\text{Dual curl (Primal 2-form to dual 2-form)} \rightarrow \nabla \times = C = [d_1^T][\star_2] \quad (\text{A.14})$$

$$[d_1^T][\star_2] = \begin{bmatrix} 0.2357 & 0.2357 & 0 & 0 & 0 & 0 & 0 \\ -0.2357 & 0 & 0.2357 & 0 & 0 & 0 & 0 \\ 0 & -0.2357 & -0.2357 & 0 & 0 & 0 & 0 \\ 0.2357 & 0 & 0 & 0.4714 & 0.2357 & 0 & 0 \\ 0 & 0.2357 & 0 & -0.4714 & 0 & 0.2357 & 0 \\ 0 & 0 & 0 & 0 & -0.2357 & -0.2357 & 0 \\ 0 & 0 & 0.2357 & 0.4714 & 0 & 0 & 0.2357 \\ 0 & 0 & 0 & 0 & 0.2357 & 0 & -0.2357 \\ 0 & 0 & 0 & 0 & 0 & 0.2357 & 0.2357 \end{bmatrix}$$

Divergence:

$$\text{Primal div (Primal 2-form to 3-form)} \rightarrow \nabla \cdot = D = [d_2] \quad (\text{A.15})$$

$$\text{Dual div (Primal 1-form to dual 3 form)} \rightarrow \nabla \cdot = D = [d_0^T][\star_1] \quad (\text{A.16})$$

$$[d_0^T][\star_1] = \begin{bmatrix} -0.125 & -0.125 & -0.125 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0.125 & 0 & 0 & -0.25 & -0.25 & -0.125 & 0 & 0 & 0 \\ 0 & 0.125 & 0 & 0.25 & 0 & 0 & -0.25 & -0.125 & 0 \\ 0 & 0 & 0.125 & 0 & 0.25 & 0 & 0.25 & 0 & -0.125 \\ 0 & 0 & 0 & 0 & 0 & 0.125 & 0 & 0.125 & 0.125 \end{bmatrix}$$

A.5 Laplacian

Following a similar logic to the Gradient, Curl and Divergence operators, the Laplacian also has two different definitions (one starting with values on the dual mesh and one starting on the primal mesh).

A Laplacian acting on flows through primal faces (primal 2-forms) has the following formulation:

$$\Delta = (\nabla \nabla \cdot - \nabla \times \nabla \times) \quad (\text{A.17})$$

$$\begin{aligned} \Delta &= L = (d^T \star d + \star d \star^{-1} d^T \star) \\ &= [d_2^T][\star_3][d_2] + [\star_2][d_1][\star_1^{-1}][d_1^T][\star_2] \\ &= [7 \times 2][2 \times 2][2 \times 7] + [7 \times 7][7 \times 9][9 \times 9][9 \times 7][7 \times 7] \\ &= [7 \times 7] \end{aligned}$$

$$L = \Delta = \begin{bmatrix} 2.111 & -0.556 & 0.556 & -0.556 & 0.222 & 0. & 0. \\ -0.556 & 2.111 & -0.556 & 0.556 & 0. & 0.222 & 0. \\ 0.556 & -0.556 & 2.111 & -0.556 & 0. & 0. & 0.222 \\ -0.556 & 0.556 & -0.556 & 4.667 & -0.556 & 0.556 & -0.556 \\ 0.222 & 0. & 0. & -0.556 & 2.111 & -0.556 & 0.556 \\ 0. & 0.222 & 0. & 0.556 & -0.556 & 2.111 & -0.556 \\ 0. & 0. & 0.222 & -0.556 & 0.556 & -0.556 & 2.111 \end{bmatrix}$$

Alternatively, a Laplacian acting on fluxes through dual faces (dual 2-forms) has

the following formulation:

$$\Delta = (\nabla \nabla \cdot - \nabla \times \nabla \times) \quad (\text{A.18})$$

$$= [d_0][\star_0^{-1}][d_0^T][\star_1] + [\star_1^{-1}][d_1^T][\star_2][d_1]$$

$$= [9 \times 5][5 \times 5][5 \times 9][9 \times 9] + [9 \times 9][9 \times 7][7 \times 7][7 \times 9]$$

$$= [9 \times 9]$$

$$L = \Delta = \begin{bmatrix} 0.565 & -0.173 & -0.173 & 0.173 & 0.173 & -0.031 & 0. & 0. & 0. \\ -0.173 & 0.565 & -0.173 & -0.173 & 0. & 0. & 0.173 & -0.031 & 0. \\ -0.173 & -0.173 & 0.565 & 0. & -0.173 & 0. & -0.173 & 0. & -0.031 \\ 0.173 & -0.173 & 0. & 1.193 & -0.346 & -0.173 & 0.346 & 0.173 & 0. \\ 0.173 & 0. & -0.173 & -0.346 & 1.193 & -0.173 & -0.346 & 0. & 0.173 \\ -0.031 & 0. & 0. & -0.173 & -0.173 & 0.565 & 0. & -0.173 & -0.173 \\ 0. & 0.173 & -0.173 & 0.346 & -0.346 & 0. & 1.193 & -0.173 & 0.173 \\ 0. & -0.031 & 0. & 0.173 & 0. & -0.173 & -0.173 & 0.565 & -0.173 \\ 0. & 0. & -0.031 & 0. & 0.173 & -0.173 & 0.173 & -0.173 & 0.565 \end{bmatrix}$$

Appendix B

Transcendental Equation Solution

$$\delta \exp(\delta^2) \operatorname{erf}(\delta) = \frac{C_p(T_{wall} - T_{sat})}{h_{lg}\sqrt{\pi}} \quad (\text{B.1})$$

The solution to equation B.1 and 6.14 was found by graphical means and corresponds to a solution at $\delta = 0.003253$ for the constant properties defined below.

$$P_v \quad 101300 \text{ Pa}$$

$$c_p \quad 2029 \frac{\text{J}}{\text{kg}} \text{K}$$

$$h_{lv} \quad 2.4 \times 10^9 \frac{\text{J}}{\text{m}^3}$$

$$T_{wall} \quad 373 \text{ K}$$

$$T_{sat} \quad 398 \text{ K}$$

In figure B.1, the blue and pink lines denote the solution of the LHS and RHS of the transcendental equation respectively.

Figure B.1: Graphical solution of transcendental equation 6.14

Appendix C

1D Bubble Analysis

C.1 Summary of simplifying assumptions

The following assumptions are taken throughout the calculations:

1. Gases inside the bubble and the surrounding liquid move maintaining spherical symmetry.
2. Low velocity of bubble through the fluid caused by buoyant forces.
3. Compressibility and viscosity effects are neglected.
4. Constant pressure and temperature of liquid.
5. Gases inside the bubble obey the perfect gas law.
6. Vapour inertia may be neglected.
7. Uniform temperature and pressure within the vapour bubble.

C.2 Formulation of governing equations

The momentum equation, which is derived from the Navier-Stokes equation for motion, in spherical coordinates is given by

$$\frac{\partial u}{\partial t} + u \frac{\partial u}{\partial r} = \frac{-1}{\rho} \frac{\partial P}{\partial r} + \frac{\mu}{\rho} \left[\frac{1}{r^2} \frac{\partial}{\partial r} \left(r^2 \frac{\partial u}{\partial r} \right) - \frac{2u}{r^2} \right] \quad (\text{C.1})$$

Assuming no compressibility in the fluids, the continuity equation is given by

$$\frac{\partial}{\partial r} r^2 u = 0 \quad (\text{C.2})$$

Substituting (2) into (1) and integrating from the bubble surface R to infinity results in the equation of motion for R (the radius) of the bubble as a function of time.

$$R \frac{d^2 R}{dt^2} + \frac{3}{2} \left(\frac{dR}{dt} \right)^2 = \frac{p(R) - p_\infty}{\rho_l - \rho_v} \quad (\text{C.3})$$

Where ρ_l and ρ_v are the liquid and vapor densities, $p(R)$ is the pressure at the bubble boundary R , and p_∞ is the external pressure in the liquid. Assuming that $\rho_l \gg \rho_v$, the equation further reduces to

$$R \frac{d^2 R}{dt^2} + \frac{3}{2} \left(\frac{dR}{dt} \right)^2 = \frac{p(R) - p_\infty}{\rho_l} \quad (\text{C.4})$$

where

$$p(R) = p_v(T) - p_\sigma - p_\mu \quad (\text{C.5})$$

The forces acting on the bubble can be explained by a simple model. Assuming the gas satisfies the perfect gas law, the pressure p_v of the vapor in the bubble is given in terms of the temperature T , volume V of the spherical bubble, and n as the number of moles of gas in the bubble.

$$p_v(T) = \frac{n \mathcal{R} T}{V} \quad (\text{C.6})$$

The volume of a sphere is given by

$$V = \frac{4}{3}\pi R^3 \quad (\text{C.7})$$

As Plesset & Zwick [33] note, at this point one needs to make some assumptions about the nucleation process. The assumption that Plesset & Zwick [33] made, which will be followed here, is that the bubble contains initially no permanent gas or solid particle nucleus. Since this would result in no stable equilibrium for such a bubble, Plesset & Zwick suggest a relation for the radius R_o of the pure vapor bubble to the critical radius R_c for unstable equilibrium of a gas filled bubble at the same superheat.

$$R_o = \frac{3}{2}R_c \quad (\text{C.8})$$

Assuming that small surface tension σ variations with temperature are neglected, the pressure p_σ caused by the surface tension σ is

$$p_\sigma = \frac{2\sigma}{R} \quad (\text{C.9})$$

and pressure due to viscous effects p_μ can be defined as

$$p_\mu = 4\frac{\mu}{R}\frac{dR}{dt} \quad (\text{C.10})$$

Substituting equations (5) through (9) into equation (4) yields the Rayleigh-Plesset equation. This approximate equation is derived from the compressible Navier-Stokes equations and describes the motion of the radius of the bubble R with time.

$$R\frac{d^2R}{dt^2} + \frac{3}{2}\left(\frac{dR}{dt}\right)^2 = \frac{1}{\rho_l}\left(p_v(T) - p_\infty - \frac{2\sigma}{R} - 4\frac{\mu}{R}\frac{dR}{dt}\right) \quad (\text{C.11})$$

If one is to neglect viscous effects, the equation can be simplified to

$$R \frac{d^2 R}{dt^2} + \frac{3}{2} \left(\frac{dR}{dt} \right)^2 = \frac{1}{\rho_l} \left(p_v(T) - p_\infty - \frac{2\sigma}{R} \right) \quad (\text{C.12})$$

C.2.1 Equilibrium condition

If the vapor bubble is in equilibrium, the pressure equilibrium on the bubble interface can be written as

$$\underbrace{p_v(T)}_{\text{internal pressure}} = \underbrace{p(R) + p_\sigma + p_\mu}_{\text{external pressure}} \quad (\text{C.13})$$

Where $p_v(T)$ is the equilibrium vapor pressure for the temperature T , $p(R)$ is the pressure of the liquid at R , and p_σ and p_μ are the pressures caused by surface tension and viscous effects as defined by equations (8) and (9) respectively. Ignoring viscous effects, equations (9) and (13) reduce to

$$p_v(T) = p(R) + \frac{2\sigma}{R} \quad (\text{C.14})$$

It is possible to solve for R_o , the initial bubble radius at which the bubble is in equilibrium by neglecting small changes in σ and ρ due to temperature changes.

$$\frac{2\sigma}{R_o} = p_v(T_o) - p_o \quad (\text{C.15})$$

Since the bubble is in equilibrium, the initial temperature and pressure (T_o and P_o) are equal to the temperature and pressure of the superheated liquid. Therefore, R_o is

$$R_o = \frac{2\sigma}{p_v(T_\infty) - p_\infty} \quad (\text{C.16})$$

It should be noted that a bubble at rest with radius R_o is in unstable equilibrium for

the given initial conditions. Rearranging equation (16) for p_∞ yields

$$p_\infty = p_v(T_\infty) - \frac{2\sigma}{R_o} \quad (\text{C.17})$$

C.2.2 The Rayleigh solution

Specifically, in the Rayleigh solution, the cooling effect of evaporation is ignored.

Therefore

$$p_v(T) = p_v(T_\infty) \quad (\text{C.18})$$

Substituting equation (17) into (12), equation (12) can be written in terms of R_o as

$$R \frac{d^2 R}{dt^2} + \frac{3}{2} \left(\frac{dR}{dt} \right)^2 = \frac{1}{\rho_l} \left(p_v(T) - p_v(T_\infty) + \frac{2\sigma}{R_o} - \frac{2\sigma}{R} \right) \quad (\text{C.19})$$

Further applying the simplification assumptions from equation (18) results in the following differential equation:

$$R \frac{d^2 R}{dt^2} + \frac{3}{2} \left(\frac{dR}{dt} \right)^2 = \frac{1}{\rho_l} \left[\left(\frac{2\sigma}{R_o} \right) \left(1 - \frac{R_o}{R} \right) \right] \quad (\text{C.20})$$

As demonstrated by Brennen [?], this equation can be integrated by multiplying through by $2R^2 \frac{dR}{dt}$ and applying the initial condition of $\frac{dR}{dt}(0) = 0$ to give

$$\left(\frac{dR}{dt} \right)^2 = \left(\frac{R_o}{R} \right) \left(\frac{dR_o}{dt} \right)^2 + \left(\frac{4\sigma}{3\rho_l R_o} \right) \left[1 - \frac{R_o^3}{R^3} \right] - \left(\frac{2\sigma}{\rho_l R} \right) \left[1 - \frac{R_o^2}{R^2} \right] \quad (\text{C.21})$$

Assuming that $R \gg R_o$, the Rayleigh solution can be approximated to a constant growth of the bubble.

$$\left(\frac{dR}{dt}\right)^2 \approx \frac{4\sigma}{3\rho_l R_o} = \frac{2}{3\rho_l} [p_v(T_\infty) - p_\infty] \quad (\text{C.22})$$

Through experimentation, it has been determined that the actual solution deviates from the Rayleigh solution due to the cooling effect as the bubble expands.

C.2.3 Cooling effect

Since the heat of vaporization is defined as

$$Q = \int L \frac{dm}{dt} dt \quad (\text{C.23})$$

the latent heat L is time-dependent since pressure and volume are also changing with time. Therefore, the heat flux that must be supplied to the bubble per unit time is

$$\frac{dQ}{dt} = L \frac{dm}{dt} \quad (\text{C.24})$$

Where the mass m of the vapor bubble is given by

$$m = \rho_v \frac{4\pi}{3} R^3 \quad (\text{C.25})$$

the resulting heat flux caused by the cooling effect is

$$\frac{dQ}{dt} = L \frac{d}{dt} \left(\rho_v \frac{4\pi}{3} R^3 \right) \quad (\text{C.26})$$

The heat supplied to the bubble for the cooling effect is supplied by conduction from the liquid into the bubble. Therefore

$$\frac{dQ}{dt} = kA \left(\frac{\partial T}{\partial r} \right)_R \quad (\text{C.27})$$

where k is the thermal conductivity of the liquid and $(\partial T / \partial r)_R$ is the temperature

gradient in the liquid at the bubble boundary. Substituting the surface area for a sphere results in

$$\frac{dQ}{dt} = k(4\pi R^2) \left(\frac{\partial T}{\partial r} \right)_R \quad (\text{C.28})$$

Due to the conservation of energy (neglecting small variations of L and k with temperature) and assuming that all the energy is used for vaporization of the liquid and not cooling or heating the contents of the bubble, equations (26) and (28) can be equated, resulting in

$$L \frac{d}{dt} \left(\rho_v \frac{4\pi}{3} R^3 \right) = k(4\pi R^2) \left(\frac{\partial T}{\partial r} \right)_R \quad (\text{C.29})$$

Solving for the temperature gradient results in

$$\left(\frac{\partial T}{\partial r} \right)_R = \frac{L}{3} k R^2 \frac{d}{dt} (\rho_v R^3) \quad (\text{C.30})$$

Since the volume of the bubble increases by orders of magnitude while the vapor density changes very little (due to a small temperature drop), it can be assumed that the contribution of $d\rho_v/dt$ is much smaller than from dR^3/dt

$$\left(\frac{\partial T}{\partial r} \right)_R \approx \left(\frac{L\rho_v}{k} \right) \frac{dR}{dt} \quad (\text{C.31})$$

Equation (31) together with the specification of the temperature at infinity T_∞ determines the temperature field in the liquid. The solution for this temperature problem with the moving boundary $R(t)$ has been found by Plesset & Zwick [34] and is out of the scope of this analysis. The solution assumes that the drop in temperature from T_∞ to the value T at the bubble boundary takes place in a “thin thermal boundary layer” which has small thickness compared with $R(t)$. The approximate expression for the temperature at the bubble is given by

$$T = T_\infty - \left(\frac{D_l}{\pi}\right)^{\frac{1}{2}} \int_0^t \frac{[R(x)]^2 \left(\frac{\partial T}{\partial r}\right)_{r=R(x)}}{\left[\int_x^t R^4(y) dy\right]} dx \quad (\text{C.32})$$

Therefore, equations (12) and (32) fully define the bubble growth. Substituting equation (31) into (32) results in

$$T = T_\infty - \left(\frac{L\rho_v}{\rho_l c_{PL} D_l^{\frac{1}{2}}}\right) \left(\frac{1}{\pi}\right)^{\frac{1}{2}} \int_0^t \frac{[R(x)]^2 \left(\frac{dR}{dt}\right)}{\left[\int_x^t R^4(y) dy\right]} dx \quad (\text{C.33})$$

This result can be substituted into the Rayleigh-Plesset equation to generate an integro-differential equation for $R(t)$.

C.2.4 Solution to the equation of motion

Equations (12) and (33) are connected when the equilibrium vapor pressure is specified as a function of temperature. For superheats not too far above the boiling temperature T_b of the liquid at the external pressure P_∞ , the vapor pressure may be approximated by a linear function of the temperatures as demonstrated by Brennen [?].

$$\frac{p_v(T) - p_\infty}{\rho} = A(T - T_b) \quad (\text{C.34})$$

In the initial case, when T equals T_∞ , equation (34) gives

$$\frac{2\sigma}{\rho R_o} = A(T_\infty - T_b) \quad (\text{C.35})$$

Finally, bringing equations (12), (33) and (34) together yields

$$R \frac{d^2 R}{dt^2} + \frac{3}{2} \left(\frac{dR}{dt}\right)^2 = A \left[T_\infty - \left(\frac{L\rho_v}{\rho_l c_{PL} D_l^{\frac{1}{2}}}\right) \left(\frac{1}{\pi}\right)^{\frac{1}{2}} \int_0^t \frac{[R(x)]^2 \left(\frac{dR}{dt}\right)}{\left[\int_x^t R^4(y) dy\right]} dx \right] - T_b - \frac{2\sigma}{\rho R} \quad (\text{C.36})$$

Zwick & Plesset [35] have shown that the growth proceeds along three phases by solving each step by an asymptotic analysis. This is a very cumbersome procedure and the implementation of a numerical algorithm is usually favored.

The three steps of the bubble growth as described by Zwick & Plesset [35] are:

1. A latent period, the details and length of which depends on the particular way to destabilize the bubble, such as the heating rate of the liquid or the relative excess initial radius with respect to the equilibrium radius.
2. The initial growth.
3. The fully developed growth.

For calculation purposes, if a spherical vapor bubble is assumed to exist at critical size, the vapor pressure in the bubble nucleus is higher than that of the adjacent liquid and is balanced by surface tension. When making numerical computations, a small disturbance is required to initiate the vapor bubble growth from the critical size. This disturbance is usually imposed by increasing the initial vapor bubble size slightly above the known critical bubble size.

Appendix D

Prototype DEC Solver Code

D.1 Main Function

```
// Klimas DEC test implementation MAIN APPLICATION
// Completed as per requirements of M.A.Sc Thesis
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
// Prototype DEC implementation code
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
// INPUTS:
// - debug_output_level    -1: NO output
//                          0: informative output
//                          1: normal debugging output
//                          2: extreme debugging output
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

// clear all previously defined variables
clear;
debug_output_level = -1;
// 10 Million * 8 bytes = 76.29 Mbytes
// 100 000 000 * 8 bytes = 762.939453 megabytes
stacksize(100000000);
//disp(stacksize(),'*** Stacksize (Total/used) ***');

// Change to the working directory
currentdir = "/home/peter/school/Thesis/code/DEC_Solver";
cd "/home/peter/school/Thesis/code/DEC_Solver";
```

```

// Load DEC specific functions
getf('DEC_implementation.sci');
getf('DEC_operator_construction.sci');
getf('DEC_mesh_operations.sci');
getf('DEC_graphics.sci');
getf('DEC_interpolation_functions.sci');

// Ask for the flow type
l1=list('Please_select_a_flow_type:',1,['Poiseuille','Jet']);
l2=list('Output_divfree_velocity/vorticity/particle_data?',1,['Yes','No']);
l3=list('Output_harmonic_velocity/vorticity/particle_data?',1,['Yes','No']);
l4=list('Advect_particles_in_transient_solution?',1,['Yes','No']);
l5=list('Transient_particle_advection' + char(10) + "Advect_once_every_X_
time-steps:",4,['1','2','5','10','100']);
l6=list('Output_velocity_data_for_transient_solution?',1,['Yes','No']);
l7=list('Transient_velocity_output' + char(10) + "Output_once_every_X_
time-steps:",4,['1','2','5','10','100']);
l8=list('Output_vorticity_data_for_transient_solution?',1,['Yes','No']);
l9=list('Transient_vorticity_output' + char(10) + "Output_once_every_X_
time-steps:",4,['1','2','5','10','100']);
l10=list('Left_boundary_condition:",2,['Slip','No-slip']);
l11=list('Right boundary condition:",2,['Slip','No-slip']);
l12=list('Top_boundary_condition:",1,['Slip','No-slip']);
l13=list('Bottom_boundary_condition:",1,['Slip','No-slip']);
l14=list('Front boundary condition:",1,['Slip','No-slip']);
l15=list('Back_boundary_condition:",1,['Slip','No-slip']);

rep=x_choices('Toggle_Menu',list(l1,l2,l3,l4,l5,l6,l7,l8,l9,l10,l11,l12,l13,l14,l15))
;

if rep ~= [] then
    flow_type = rep(1);
    export_divfree = rep(2);
    export_harmonic = rep(3);
    export_particle = rep(4);
    select rep(5)
        case 1 particle_out_divider = 1;
        case 2 particle_out_divider = 2;
        case 3 particle_out_divider = 5;

```

```

    case 4 particle_out_divider = 10;
    case 5 particle_out_divider = 100;
end;

export_velocity = rep(6);
select rep(7)
    case 1 velocity_out_divider = 1;
    case 2 velocity_out_divider = 2;
    case 3 velocity_out_divider = 5;
    case 4 velocity_out_divider = 10;
    case 5 velocity_out_divider = 100;
end;

export_vorticity = rep(8);
select rep(9)
    case 1 vorticity_out_divider = 1;
    case 2 vorticity_out_divider = 2;
    case 3 vorticity_out_divider = 5;
    case 4 vorticity_out_divider = 10;
    case 5 vorticity_out_divider = 100;
end;

// value of 1 = slip
// value of 2 = wall
// 1=left boundary, 2=right boundary, 3=top boundary
// 4=bottom boundary, 5=front boundary, 6=back boundary
boundary_conditions(:) = rep(10:15);

else // Cancel button was pressed exit the program
    disp("Canceling!");
    return; // return to the calling function
end;

// Ask for output file locations
result_path=tk.getdir(currentdir, 'Please_select_result_output_directory ');

// Confirm the output directory and file name and set the result file name prefix
labels=["Result_output_directory:"; "Divergence_free_flow_filename:"; "Harmonic_flow_
filename:"; "Particle_advection_filename_prefix:"; "Velocity_output_filename_prefix
:"; "Vorticity_output_filename_prefix:"; "Mesh_filename:"; "Result_file_mask:" ];

```



```

[ok,result_path ,divfree_filename ,harmonic_filename ,result_particle_file ,
  result_velocity_file ,result_vorticity_file ,mesh_filename ,file_mask] = getvalue("
  Please_confirm_output_options_(eg._output:_\${PREFIX}_timestep#.plt):_",labels ,
  list("str",-1,"str",-1,"str",-1,"str",-1,"str",-1,"str",-1,"str",-1,"str",-1),[
  result_path;"divfree_flow";"harmonic_flow";"result_particle_flow";"
  result_velocity";"result_vorticity";"mesh";"plt"]);
// check to see if the users pressed OK or CANCEL.
if ~ok then // If the cancel button was pressed exit the program
  disp("Canceling!");
  return; // return to the calling function
end;

// Ask for the user input for number of x,y,z cubes and for initial flow conditions
// The getvalue function automatically checks that numbers have been entered
labels="Number_of_cubes_in_[X,Y,Z]_directions:";"Cube_edge_length:";"Initial_flux_
on_input_face_(kg/(m^2_s):";"Material_density_(kg/m^3):";"Kinematic_viscosity_(N_
s/m^2)-[value_*_10^(-6)]:";"Time_step_size_(s):";"Number_of_time_steps:"];

[ok,num_cubes_xyz ,vertex_spacing ,initial_flux ,material_density ,viscosity ,
  time_step_size ,num_time_steps] = getvalue("Please_enter_initial_conditions_and_
material_parameters:_",labels , list("vec",3,"col",1,"col",1,"col",1,"col",1,"col",
1,"col",1),["5,1,5";"0.01";"100";"1000";"1.307";"0.01";"50"]);
// check to see if the users pressed OK or CANCEL.
if ~ok then // If the cancel button was pressed exit the program
  disp("Canceling!");
  return; // return to the calling function
end;
viscosity = viscosity * 10^(-6);

// GLOBAL VARIABLE DECLARATIONS
// ***** CANNOT WRITE TO THESE VALUES IN OTHER FUNCTIONS *****
//for ease of reading
form_0 = 1;
form_1 = 2;
form_2 = 3;
form_3 = 4;

// number of vertices
num_vertex_x = num_cubes_xyz(1) + 1; // x direction
num_vertex_y = num_cubes_xyz(2) + 1; // y direction

```

```

num_vertex_z = num_cubes_xyz(3) + 1; // z direction

// We will pretend that a 20-node brick mesh exists (for interpolation)
// This way we can have 2 meshes at once and have no problems moving values between
nodes
num_vertex_x_27node = (2*num_vertex_x) -1;
num_vertex_y_27node = (2*num_vertex_y) -1;
num_vertex_z_27node = (2*num_vertex_z) -1;

// Build mesh
[num_vertex, vertex_mesh] = mesh_vertex(num_vertex_x, num_vertex_y, num_vertex_z,
    debug_output_level);
[num_edges, edge_mesh] = mesh_edges(num_vertex, vertex_mesh, debug_output_level);
[num_faces, face_mesh] = mesh_faces(num_vertex, vertex_mesh, num_edges, edge_mesh,
    debug_output_level);
[num_cubes, cube_mesh] = mesh_cubes(debug_output_level);

// build 27_node vertex mesh as we use it for exporting to file.
[num_vertex_27node, vertex_mesh_27node] = mesh_vertex(num_vertex_x_27node,
    num_vertex_y_27node, num_vertex_z_27node, debug_output_level);
//divide so the locations of every second point agree with the locations of the
standard mesh (ie. meshes are the same size)
vertex_mesh_27node = (vertex_mesh_27node/2);

brick_values_8node = zeros(num_vertex,3);
brick_values_27node = zeros(num_vertex_27node,3);

// Set the input/output face values (center of the front and back faces)
input_mass_flow_rate = initial_flux * vertex_spacing^2;
// Random value that represents an unknown flux
unknown = -999;

// FILE OUTPUTTING VARIABLES
rmdir(result_path + "/particle_advection","s"); //remove output directories if they
exist
rmdir(result_path + "/velocity_output","s"); //remove output directories if they
exist
rmdir(result_path + "/vorticity_output","s"); //remove output directories if they
exist

```

```

mkdir(result_path , "particle-advection"); //create new output directories
mkdir(result_path , "velocity-output"); //create new output directories
mkdir(result_path , "vorticity-output"); //create new output directories

// main outputting options
mesh_out = result_path + "/" + mesh_filename;
divfree_out = result_path + "/" + divfree_filename;
harmonic_out = result_path + "/" + harmonic_filename;
result_particle_out = result_path + "/particle-advection/" + result_particle_file;
result_velocity_out = result_path + "/velocity-output/" + result_velocity_file;
result_velocity_rot_out = result_path + "/velocity-output/" + result_velocity_file +
    "_rot";
result_vorticity_out = result_path + "/vorticity-output/" + result_vorticity_file;

// This is the number of particles that we will inject on the input face
particle_multi_number = 3;
percent_edge_offset = 0;
num_particle_x = (num_vertex_x - 1) * particle_multi_number;
num_particle_y = (num_vertex_y - 1); // * particle_multi_number;
start_particle_number = num_particle_x * num_particle_y;

if flow_type == 2 // "jet"
    // center input face
    input_face = Get_face_number([round(num_vertex_x/2),round(num_vertex_y/2),1;round(
        num_vertex_x/2)+1,round(num_vertex_y/2)+1,1],num_vertex_x,num_vertex_y ,
        num_vertex_z , debug_output_level);
    //output_face = Get_face_number([int(num_vertex_x/2),int(num_vertex_y/2),
        num_vertex_z;int(num_vertex_x/2)+1,int(num_vertex_y/2)+1, num_vertex_z],
        num_vertex_x,num_vertex_y,num_vertex_z , debug_output_level);
    f = 0;
    for y = 1 : num_vertex_y - 1
        for x = 1 : num_vertex_x - 1
            f = f + 1;
            output_face(f) = Get_face_number([x,y,num_vertex_z;x+1,y+1,num_vertex_z],
                num_vertex_x,num_vertex_y,num_vertex_z , debug_output_level);
        end; //for x = 1:num_vertex_x
    end; //for y = 1:num_vertex_y
    //These are the maximum bounds of the face where we want to be injecting particles
    into

```

```

input_position = Get_position_of_face(input_face,num_vertex_x,num_vertex_y,
num_vertex_z,debug_output_level) * vertex_spacing;

elseif flow_type == 1 //"pipe" - Poiseuille flow
    f = 0;
    // This is a duct/pipe flow input option (Poiseuille flow)
    for y = 1 : num_vertex_y - 1
        for x = 1 : num_vertex_x - 1
            f = f + 1;
            input_face(f) = Get_face_number([x,y,1;x+1,y+1,1],num_vertex_x,num_vertex_y,
num_vertex_z,debug_output_level);
            output_face(f) = Get_face_number([x,y,num_vertex_z;x+1,y+1,num_vertex_z],
num_vertex_x,num_vertex_y,num_vertex_z,debug_output_level);
        end; //for x = 1:num_vertex_x
    end; //for y = 1:num_vertex_y

    //These are the maximum bounds of the face where we want to be injecting particles
    into
    input_position(1,1) = (1 + 1*percent_edge_offset) * vertex_spacing; //x1
    input_position(1,2) = (1 + 1*percent_edge_offset) * vertex_spacing; //y1
    input_position(1,3) = (1) * vertex_spacing; //Z1
    input_position(2,1) = (num_vertex_x - 1*percent_edge_offset) * vertex_spacing; //x2
    input_position(2,2) = (num_vertex_y - 1*percent_edge_offset) * vertex_spacing; //y2
    input_position(2,3) = (1) * vertex_spacing; //Z2

end; // flow_type

//particle initialization locations based on the number of cubes we have in the x & y
directions
for y=1:num_particle_y
    for x=1:num_particle_x
        // particles should be evenly distributed
        start_particle_location(x+(y-1)*(num_particle_x),:) = [input_position(1,1) + ( (
            input_position(2,1)-input_position(1,1)) / (num_particle_x + 1)) * x,
            input_position(1,2) + ( (input_position(2,2)-input_position(1,2)) / (
            num_particle_y + 1) ) * y,input_position(1,3)];
    end;
end;

```

```

//start_particle_location(x+(y-1)*(num_particle_x),:) = [input_position(1,1) + (
    input_position(1,1)/(particle_multi_number+1)) + ( (input_position(2,1)-
    input_position(1,1)) / (num_particle_x)) * x, input_position(1,2) +
    input_position(1,2)/(particle_multi_number+1) + ( (input_position(2,2)-
    input_position(1,2)) / (num_particle_y) ) * y, input_position(1,3)];
end // for x
end; //for y

// Main outputting matrix
particle_tracking_matrix = [];
// any extra no-slip walls inside domain
internal_wall_cubes = [];

// END OF GLOBAL VARIABLE DECLARATIONS

// DEC SOLUTION CODE
// Initialize the timer to keep track of solution time
timer();
disp('***_Building_DEC_matrices_***' + string(timer()) + 's_***');
// —— Build the incident matrices ——
// Defines the exterior derivative operator
[d0,d1,d2] = Build_incident_matrices(num_cubes_xyz,edge_mesh,debug_output_level);
disp('——>_d0,_d1,_d2_matrix_construction_time:_ ' + string(timer()) + 's');

// —— Define the Hodge star ——
[star0,star1,star2,star3,primal_vol,dual_vol] = Build_star_operator(num_cubes_xyz,
    vertex_spacing,debug_output_level);

inv_star0 = inv(star0);
inv_star1 = inv(star1);
inv_star2 = inv(star2);
inv_star3 = inv(star3);

disp('——>_star0,_star1,_star2,_star3_matrix_construction_time:_ ' + string(timer()) +
    's_***');

```

```

// Build commonly used DEC operators (Grad, Curl, Div, Laplace)
C0 = d0' * star1;
C1 = d1' * star2;
C2 = d2' * star3;

// 2 laplace options exist (depends where you are starting from)
L1 = (star1 * d0 * inv_star0 * d0' * star1) + (d1' * star2 * d1); // starting at
    primal_1form
L2 = (star2 * d1 * inv_star1 * d1' * star2) + (d2' * star3 * d2); // starting at
    primal_2form
L1 = clean(L1);
L2 = clean(L2);

disp('——>_Grad,_Curl,_Div_and_Laplace_matrix_construction_time:_ ' + string(timer())
    + 's');

L1_viscous = (star1 * d0 * inv(star0) * d0') + (d1' * star2 * d1 * inv(star1)); //
    starting at dual_2form
I_viscous = eye(size(L1,1),size(L1,2)); //Identity matrix of size L1 (used to
    calculate viscosity)
// A_viscous = (I_viscous - (viscosity * material_density * time_step_size *
    L1_viscous));
A_viscous = (I_viscous - (viscosity * time_step_size * L1_viscous));

// If debugging is enabled, display extra debugging information
if debug_output_level > -1 then
    disp(full(C2), "Div:", full(C1), "Curl:", full(C0), "Grad:");
    disp(full(L2), "Laplacian_2:", full(L1), "Laplacian_1:");
end;

// ————— BEGIN SOLUTION —————

// Outputting mesh to file
disp("***_Exporting_mesh_...");
Export_mesh(mesh_out, file_mask);
disp('***_Time_to_export_mesh_to_file:_ ' + string(timer()) + 's_***');

```

```

// —— SET INITIAL CONDITIONS IN MESH
[initial_primal_2form,num_unknown] = Set_initial_values(input_face,output_face,
    internal_wall_cubes,input_mass_flow_rate,primal_vol,dual_vol,star2,
    debug_output_level);

// —— SOLVE for DIVERGENCE FREE FLOW) ——
disp('***_Staring_divergence_free_solution_***');
A = d2;
x = initial_primal_2form;
b = zeros(size(A,1),1);
divfree_solution = clean(Matrix_solve(A,x,b,unknown));
disp('***_Time_to_solve_for_divergence-free_flow:_'+ string(timer()) + 's_***');

//// —— SOLVE Laplaces equation for the harmonic flow) ——
//// the harmonic flow is the solution to laplaces equation
//// Solutions to Laplace's equation is by definition a harmonic function
//disp('*** Staring harmonic solution ***');
A = inv(star2) * L2;
x = initial_primal_2form;
x(output_face) = initial_primal_2form(input_face);
b = zeros(size(A,1),1);
harmonic_sol_direct = clean(Matrix_solve(A,x,b,unknown));

// HARMONIC SOLUTION WAS ONLY USED FOR TESTING PRUPOSES
harmonic_solution = harmonic_sol_direct;
full_primal_2form = divfree_solution;

// WRITE OUT THE INITIAL DIV-FREE FLOW TO THE FILE (FOR TECPLOT INPUT) – FOR
// DEBUGGING
if export_divfree == 1 then
    disp('***_Preparing_divergance_free_flow_for_export_***');

// —— CALCULATE FLOW (kg/s) AT DUAL VERTICIES ——
dual_0form = Find_centroid_val(divfree_solution,num_cubes,num_vertex_x,num_vertex_y
    ,num_vertex_z,debug_output_level);
disp('——>_Time_to_calculalte_flowrate_(kg/s)_at_dual_verticies_(centroids):_' +
    string(timer()) + 's');

```

```

// ———— CONVERT TO VELOCITIES ————
dual_0form_velocities = Find_velocity(dual_0form , material_density , vertex_spacing);

// ———— INTERPOLATE TO FIND velocities AT PRIMAL VERTICIES (should be least
      square estimation or barycentric interpolation?)
brick_values_27node = Interpolate_centroid_val(dual_0form_velocities ,
      divfree_solution);
disp('——>_Time_to_calculate_velocities_at_all_27_nodes_for_each_brick:_ ' + string(
      timer()) + 's');

// ———— FIND THE VORTICITY VALUES FOR THE DIV-FREE SOLUTION ————
divfree_dual_2form = dl' * star2 * divfree_solution;

disp('——>_Exporting_divergence_free_flow_data_...');
path = divfree_out + '_velocity.' + file_mask;
Export_velocity_to_file(brick_values_27node , vertex_mesh_27node , vertex_spacing , path
      ,0);
path_yz = divfree_out + '_vorticity' + '_yz.' + file_mask;
path_xz = divfree_out + '_vorticity' + '_xz.' + file_mask;
path_xy = divfree_out + '_vorticity' + '_xy.' + file_mask;
// The actual vorticity is stored on the primal edge as it must be scaled by area
Export_vorticity_to_file((inv_star1 * divfree_dual_2form) , vertex_mesh , edge_mesh ,
      vertex_spacing , path_yz , path_xz , path_xy ,0);
path = divfree_out + '_particles.' + file_mask;
Export_initial_particles(brick_values_27node , path , round(num_time_steps/
      particle_out_divider) , time_step_size*particle_out_divider , "Div-free_flow");

disp('——>_Time_to_output_divergence_free_solution_to_file:_ ' + string(timer()) + 's
      ');

end;

// WRITE OUT THE HARMONIC FLOW TO THE FILE (FOR TECPLOT INPUT) – FOR DEBUGGING
if export_harmonic == 1 then
      disp('***_Preparing_harmonic_flow_for_export_***');

// ———— CALCULATE FLOW (kg/s) AT DUAL VERTICIES ————
dual_0form = Find_centroid_val(harmonic_solution , num_cubes , num_vertex_x ,
      num_vertex_y , num_vertex_z , debug_output_level);
disp('——>_Time_to_calculate_flowrate_(kg/s)_at_dual_verticies_(centroids):_ ' +
      string(timer()) + 's');

```



```

// ----- CONVERT TO VELOCITIES -----
dual_0form_velocities = Find_velocity(dual_0form, material_density, vertex_spacing);

// ----- INTERPOLATE TO FIND velocities AT PRIMAL VERTICIES (should be least
      square estimation or barycentric interpolation?)
brick_values_27node = Interpolate_centroid_val(dual_0form_velocities,
      harmonic_solution);
disp('---->_Time_to_calculate_velocities_at_all_27_nodes_for_each_brick:_ ' + string(
      timer()) + 's');

// ----- FIND THE VORTICITY VALUES FOR THE HARMONIC SOLUTION -----
harmonic_dual_2form = dl' * star2 * harmonic_solution;

disp('---->_Exporting_harmonic_flow_data_...');
path = harmonic_out + '_velocity.' + file_mask;
Export_velocity_to_file(brick_values_27node, vertex_mesh_27node, vertex_spacing, path
      ,0);
path_yz = harmonic_out + '_vorticity' + '_yz.' + file_mask;
path_xz = harmonic_out + '_vorticity' + '_xz.' + file_mask;
path_xy = harmonic_out + '_vorticity' + '_xy.' + file_mask;
// The actual vorticity is stored on the primal edge as it must be scaled by area
Export_vorticity_to_file((inv_star1 * harmonic_dual_2form), vertex_mesh, edge_mesh,
      vertex_spacing, path_yz, path_xz, path_xy, 0);
path = harmonic_out + '_particles.' + file_mask;
Export_initial_particles(brick_values_27node, path, round(num_time_steps/
      particle_out_divider), time_step_size*particle_out_divider, "Harmonic_flow");
disp('---->_Time_to_output_harmonic_solution_to_file:_ ' + string(timer()) + 's');
end;

// THE BEGINNING OF THE MAIN SOLUTION
// ----- BEGIN TIMESTEP ADVECTION LOOP -----
for i=1:num_time_steps

disp('*****_SOLVING_FOR_FLOW_AT_TIMESTEP_' + string(i) + '_( ' + string(i*
      time_step_size) + 's)');

// ----- CALCULATE FLOW (kg/s) AT DUAL VERTICIES -----

```

```

full_dual_0form = Find_centroid_val(full_primal_2form , num_cubes , num_vertex_x ,
    num_vertex_y , num_vertex_z , debug_output_level);
disp('**_Time_to_calculate_flowrate_(kg/s)_at_dual_vertices_(centroids):_' +
    string(timer()) + 's_**');

// ———— CONVERT TO VELOCITIES ————
full_dual_0form_vel = Find_velocity(full_dual_0form , material_density , vertex_spacing
    );
disp('***_Converting_centroid_fluxes_to_velocities_***');

// ———— INTERPOLATE TO FIND velocities AT PRIMAL VERTICIES (should be least
    square estimation or barycentric interpolation?)
full_brick_values_27node = Interpolate_centroid_val(full_dual_0form_vel ,
    full_primal_2form);
disp('——>_Time_to_calculate_velocities_at_all_27_nodes_for_each_brick:_' + string(
    timer()) + 's');

// ———— BACKTRACK CENTROID LOCATION TO BEGINING OF TIMESTEP ————
// c_i <- PathTraceBackwards(c_i);
// This step only has to be done with the full flow
backtracked_locations = Backtrack_centroids(full_dual_0form_vel , time_step_size ,
    vertex_spacing);
disp('**_Time_to_backtrack_centroid_locations_to_begining_of_timestep:_' + string(
    timer()) + 's_**');

// ———— FIND velocity AT BACKTRACKED LOCATION ————
// v_i <- InterpolateVelocityField(c_i);
// converts to r,s,t -> find new value -> convert back to x,y,z
backtracked_vel = Find_backtracked_value(backtracked_locations ,
    full_brick_values_27node , vertex_spacing);
disp('**_Time_to_find_values_at_backtracked_locations:_' + string(timer()) + 's_**'
    );

// ———— CALCULATE circulations down BACKTRACKED EDGES, SET THIS circulation TO
    CURRENT DUAL EDGES, & SUM TO FIND CIRCULATION ————
// NOTE: Only backtracking the rotational part of the flow (that's what we are
    preserving)
dual_1form = Calculate_circulation(backtracked_locations , backtracked_vel ,
    full_primal_2form , full_brick_values_27node) * material_density;
// ANALYTICAL ALTERNATIVE WITHOUT BACKTRACKING OF CIRCULATION
//dual_2form = d1' * star2 * rot_primal_2form;

```

```

//full_dual_2form = d1' * star2 * full_primal_2form;
disp('**_Time_to_calculate_circulation_down_backtracked_edges:_ ' + string(timer())
    + 's_**');

dual_2form = Calculate_vorticity(dual_1form, boundary_conditions);

// ----- EXPORT PARTICLE, VELOCITY AND VORTICITY INFORMATION TO A FILE FOR
// IMPORTING INTO TECPLOT ETC. -----
if export_velocity == 1 then // velocity exporting was turned on
    if modulo(i, velocity_out_divider) == 0 then
        disp('—>_Exporting_velocity_values_...');
        path = result_velocity_out + "_" + string(round(i/velocity_out_divider)) + '.'
            + file_mask;
        Export_velocity_to_file(full_brick_values_27node, vertex_mesh_27node,
            vertex_spacing, path, i);
        disp('—>_Time_to_output_velocities_to_file:_ ' + string(timer()) + 's');
    else
        disp('—>_Skipping_velocity_output');
    end;
end;

if export_vorticity == 1 then // vorticity advection was turned on
    if modulo(i, vorticity_out_divider) == 0 then
        disp("—>_Exporting_vorticity_values_...");
        path_yz = result_vorticity_out + '_yz_' + string(round(i/vorticity_out_divider))
            + '.' + file_mask;
        path_xz = result_vorticity_out + '_xz_' + string(round(i/vorticity_out_divider))
            + '.' + file_mask;
        path_xy = result_vorticity_out + '_xy_' + string(round(i/vorticity_out_divider))
            + '.' + file_mask;

        // The actual vorticity is stored on the primal edge as it must be scaled by
        // area
        Export_vorticity_to_file((inv_star1 * dual_2form), vertex_mesh, edge_mesh,
            vertex_spacing, path_yz, path_xz, path_xy, i);
        disp('—>_Time_to_output_vorticity_to_file:_ ' + string(timer()) + 's');
    else
        disp('—>_Skipping_vorticity_output');
    end;
end; // if export_vorticity = 1 then

```

```

if export_particle == 1 then // particle advection was turned on
  if modulo(i,particle_out_divider) == 0 then
    disp('—>_Advecting_particles_in_flow_field_...');
    path = result_particle_out + '_' + string(round(i/particle_out_divider)) + '.'
      + file_mask;
    particle_tracking_matrix = Export_particles_to_file(particle_tracking_matrix,
      full_brick_values_27node,path,i,time_step_size,particle_out_divider);
    disp('—>_Time_to_output_timestep_to_file:_'+ string(timer()) + 's');
  else
    disp('—>Skipping_particle_advection');
  end;
end;

// ——— VISCOSITY STEP ——— Backward difference method
disp('—>_Adding_viscous_diffusion_to_flow_...');
// left matrix divide (backslash) solves A*x=b, linsolve computes solutions to A*x+
  b=0.
// A_viscous = (I_viscous - (viscosity * material_density * time_step_size *
  L1_viscous)); //CALCULATED ABOVE AS VALUE IS CONSTANT
// b = dual_2form;
// Dynamic viscosity of water: 1.307 * 10 -3
// Kinematic viscosity of water: 1.307 * 10 - 6
dual_2form = A_viscous\dual_2form;

// Forward Euler method - alternate method
// dual_2form = (I_viscous - time_step_size * viscosity * L1_viscous) * dual_2form
  ;
disp('***_Time_to_add_viscosity_damping_to_vorticity:_'+ string(timer()) + 's_***'
  );

// ——— PRESERVING ENERGY STEP ———
// inv_L2 (2*dual_2form + viscous_dissipation_2form) = 0
// 2*inv_l2 * dual_2form - inv_l2 * viscous_dissipation_2form

// ——— SOLVE Poisson's equation (Laplace x vector_potential = vorticity) ———
// Solve the formatted matrices
// —> linsolve computes all the solutions to A*x+b=0.
// Solve ( L1 * vector_potential = vorticity )
// ( L1 * primal_1form = dual_2form )

```

```

rot_primal_1form = linsolve(L1,-dual_2form);
rot_primal_2form = (d1 * (rot_primal_1form ));
disp('***_Time_solve_Possions_equation:_'+ string(timer()) + 's***');

// Export the velocity for just the rotational component
if export_velocity == 1 then // velocity exporting was turned on
    if modulo(i,velocity_out_divider) == 0 then
        disp('——>_Exporting_rotational_velocity_values_...');

        // —— CALCULATE FLOW (kg/s) AT DUAL VERTICIES ——
        dual_0form = Find_centroid_val(rot_primal_2form,num_cubes,num_vertex_x,
            num_vertex_y,num_vertex_z,debug_output_level);
        // —— CONVERT TO VELOCITIES ——
        dual_0form_velocities = Find_velocity(dual_0form,material_density,
            vertex_spacing);
        // —— INTERPOLATE TO FIND velocities AT PRIMAL VERTICIES (should be least
            square estimation or barycentric interpolation?)
        brick_values_27node = Interpolate_centroid_val(dual_0form_velocities,
            rot_primal_2form);

        path = result_velocity_rot_out + "_" + string(round(i/velocity_out_divider)) +
            '.' + file_mask;
        Export_velocity_to_file(brick_values_27node,vertex_mesh_27node,vertex_spacing,
            path,i);
        disp('——>_Time_to_output_velocities_to_file:_'+ string(timer()) + 's');
    else
        disp('——>Skipping_rotational_velocity_output');
    end;
end;

// —— ALL THE POSSIBLE OPTIONS FOR ADDING THE HARMONIC BACK IN ARE BELOW ——
// reduce floating point errors by zeroing all boundary values as no boundary
// normal values should be calculated by the vorticity solve

// set flow to be div-free
rot_primal_2form(input_face) = divfree_solution(input_face);
full_primal_2form = rot_primal_2form;// + harmonic_solution;

```

```
end; //for i=1:num_time_step;
```

D.2 DEC Operator Construction

D.2.1 Exterior Derivative

```
function [d0,d1,d2] = Build_incident_matrices(num_cubes_xyz,edge_mesh,
    debug_output_level)
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
// Passed parameters:
// num_cubes_xyz = [# cubes in x direction, # cubes in y direction, # cubes in z
    direction]
// - debug_output_level    -1: NO output
//                          0: informative output
//                          1: normal debugging output
//                          2: extream debugging output
// Return values:
// d1 - incident matrix for verticies to edges
// d2 - incident matrix for edges to faces
// d3 - incident matrix for faces to tets
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

// If debugging is enabled, display debugging information - extream level of
    debugging
if debug_output_level > 1 then
    disp("——_Extra_debugging_output_——");
    disp("Number_of_Z_verticies:_ " + string(num_vertex_z),"Number_of_Y_verticies:_ " +
        string(num_vertex_y),"Number_of_X_verticies:_ " + string(num_vertex_x));
    disp("Total_number_of_verticies:_ " + string(num_vertex));
    disp("Total_number_of_edges:_ " + string(num_edges));
    disp("Total_number_of_faces:_ " + string(num_faces));
    disp("Total_number_of_cubes:_ " + string(num_cubes));
end;

// Create d0,d1,d2 matrices full of zeroes
d0 = spzeros(num_edges,num_vertex);
d1 = spzeros(num_faces,num_edges);
d2 = spzeros(num_cubes,num_faces);
```

```

// set the edge, face and cube counters to 0
edge_num = 0;
face_num = 0;
cube_num = 0;

d0 = spzeros(num_edges,num_vertex);
for current_edge = 1:num_edges
    // find all vertices associated with each edge
    vertex1 = edge_mesh(current_edge,1);
    vertex2 = edge_mesh(current_edge,2);

    // starting vertex has value of -1, ending vertex of 1.
    d0(current_edge,vertex1) = -1;
    d0(current_edge,vertex2) = 1;
end;

d1_new = spzeros(num_faces,num_edges);
for current_face = 1:num_faces
    // find all edges associated with each face
    // as stated in function "mesh_faces", first 2 edges are always positive, next 2
    // edges are negative (by definition)
    // first edge is always edge starting at x1,y1,z1 and positive wrt face orientation

    d1(current_face,face_mesh(current_face,1)) = 1;
    d1(current_face,face_mesh(current_face,2)) = 1;
    d1(current_face,face_mesh(current_face,3)) = -1;
    d1(current_face,face_mesh(current_face,4)) = -1;

end;

d2 = spzeros(num_cubes,num_faces);
for current_cube = 1:num_cubes
    // Order of faces in cube mesh is: front,top,left, right,bottom,back
    // This order should correspond to lowest face ID -> highest face ID.

    // Face pointing into cube is positive
    d2(current_cube,cube_mesh(current_cube,1)) = 1; //Front
    d2(current_cube,cube_mesh(current_cube,2)) = 1; //Top
    d2(current_cube,cube_mesh(current_cube,3)) = 1; //Left
    d2(current_cube,cube_mesh(current_cube,4)) = -1; //Right

```

```

    d2(current_cube,cube_mesh(current_cube,5)) = -1; //Bottom
    d2(current_cube,cube_mesh(current_cube,6)) = -1; //Back
end;

// consistency check to make sure that matrices are created properly
// multiplying consecutive matrices should result in a matrix of zeroes
d1d0_chk = d1*d0;
d2d1_chk = d2*d1;

// compare check matrices with zero matrices of same size
if full(d1d0_chk) ~= full(spzeros(d1d0_chk)) then
    disp('Warning:_d0_and_d1_are_inconsistent!');
    disp('This_is_probably_a_bug_in_the_software._Displaying_d1xd0_matrix. ');
    disp(full(d1d0_chk));
// abort;
end;
if full(d2d1_chk) ~= full(spzeros(d2d1_chk)) then
    disp('Warning:_d1_and_d2_are_inconsistent!');
    disp('This_is_probably_a_bug_in_the_software._Displaying_d2xd1_matrix. ');
    disp(full(d2d1_chk));
// abort;
end;

// If debugging is enabled, display debugging information
// Normal debugging output
if debug_output_level > 0 then
    disp('——_BEGIN:_function_Build_incident_matrices_debugging_output_——');
    disp("Completed_constructing_d0,_d1,_d2_matrices.");
    disp('Incident_matrix_check_passed. ');
    disp('The_d0,_d1_and_d2_incident_matrices_are:')
    disp(full(d0),'d0_matrix: ');
    disp(full(d1),'d1_matrix: ');
    disp(full(d2),'d2_matrix: ');
    disp('——END:_function_Build_incident_matrices_debugging_output_——');
end;

endfunction;

```


D.2.2 Hodge Star

```

function [star0,star1,star2,star3,primal_vol,dual_vol] = Build_star_operator(
    num_cubes_xyz,vertex_spacing,debug_output_level)
// star is used for transfer from primal to dual mesh
// star' (inverse of star) is used for transfer from dual to primal mesh
// Required to set the "integral density" to be equal
// Note: Using equilateral tetrahedron, centroid is one-fourth of the way from base
// to opposing vertex
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
// Passed parameters:
// - debug_output_level    -1: NO output
//                          0: informative output
//                          1: normal debugging output
//                          2: extream debugging output
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
// star = vol(dual)/vol(primal)
//
// ----- Volumes of the primal simplices -----
// vertex - 1 (by definition)
// edge - length
// surface - area of triangle
// tet - volume of tet
//
// ----- Volumes of the dual simplices -----
// dual veroni cell -
// dual surface -
// dual edge - distance from circumcentres of adjacent tets (dual edge)
// dual vertex - 1 (by definition)
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

//for ease of reading
form_0 = 1;
form_1 = 2;
form_2 = 3;
form_3 = 4;

// number of vertices
num_vertex_x = num_cubes_xyz(1) + 1; // x direction
num_vertex_y = num_cubes_xyz(2) + 1; // y direction
num_vertex_z = num_cubes_xyz(3) + 1; // z direction

```

```

num_vertex = num_vertex_x * num_vertex_y * num_vertex_z;

// number of edges
num_edges_x = (num_vertex_x-1) * num_vertex_y * num_vertex_z;
num_edges_y = (num_vertex_y-1) * num_vertex_x * num_vertex_z;
num_edges_z = (num_vertex_z-1) * num_vertex_x * num_vertex_y;
num_edges = num_edges_x + num_edges_y + num_edges_z;

// number of faces
num_faces = Get_face_number([num_vertex_x-1,num_vertex_y-1,num_vertex_z;num_vertex_x,
    num_vertex_y,num_vertex_z],num_vertex_x,num_vertex_y,num_vertex_z,
    debug_output_level);

// number of cubes
num_cubes = num_cubes_xyz(1) * num_cubes_xyz(2) * num_cubes_xyz(3);

// Primal volumes of simple cubes
primal_vol(form_0) = 1; //vertex
primal_vol(form_1) = vertex_spacing ; //edge length
primal_vol(form_2) = vertex_spacing^2; //area of surface
primal_vol(form_3) = vertex_spacing^3; //volume of cube

// Dual volumes of simple cubes
dual_vol(form_0) = 1/8 * vertex_spacing^3 //dual volume
dual_vol(form_1) = 1/4 * vertex_spacing^2; //dual surface
dual_vol(form_2) = 1/2 * vertex_spacing; //dual edge
dual_vol(form_3) = 1; //dual vertex

// Build the star matrices
star0 = spzeros(num_vertex,num_vertex);
star1 = spzeros(num_edges,num_edges);
star2 = spzeros(num_faces,num_faces);
star3 = spzeros(num_cubes,num_cubes);

// Temporary per-cube stars
s0 = dual_vol(form_0)/primal_vol(form_0);
s1 = dual_vol(form_1)/primal_vol(form_1);
s2 = dual_vol(form_2)/primal_vol(form_2);
s3 = dual_vol(form_3)/primal_vol(form_3);

```

```

for z = 1:num_vertex_z
    for y = 1:num_vertex_y
        for x = 1:num_vertex_x

            // get vertex number
            i = Get_vertex_number([x,y,z], num_vertex_x, num_vertex_y, num_vertex_z,
                debug_output_level);
            // Much faster
            //i = x + ((y-1)*num_vertex_x) + (z-1)*num_vertex_x*num_vertex_y;

            // get edge numbers around vertex
            e(1) = Get_edge_number([x,y,z;x+1,y,z], num_vertex_x, num_vertex_y, num_vertex_z,
                debug_output_level);
            e(2) = Get_edge_number([x,y,z;x,y+1,z], num_vertex_x, num_vertex_y, num_vertex_z,
                debug_output_level);
            e(3) = Get_edge_number([x,y,z;x,y,z+1], num_vertex_x, num_vertex_y, num_vertex_z,
                debug_output_level);

            // get face numbers around vertex
            f(1) = Get_face_number([x,y,z;x+1,y+1,z], num_vertex_x, num_vertex_y, num_vertex_z,
                debug_output_level);
            f(2) = Get_face_number([x,y,z;x+1,y,z+1], num_vertex_x, num_vertex_y, num_vertex_z,
                debug_output_level);
            f(3) = Get_face_number([x,y,z;x,y+1,z+1], num_vertex_x, num_vertex_y, num_vertex_z,
                debug_output_level);

            // get cube numbers around vertex
            c(1) = Get_cube_number([x,y,z;x+1,y+1,z+1], num_vertex_x, num_vertex_y,
                num_vertex_z, debug_output_level);
            c(2) = Get_cube_number([x,y,z-1;x+1,y+1,z], num_vertex_x, num_vertex_y,
                num_vertex_z, debug_output_level);
            c(3) = Get_cube_number([x,y-1,z;x+1,y,z+1], num_vertex_x, num_vertex_y,
                num_vertex_z, debug_output_level);
            c(4) = Get_cube_number([x,y-1,z-1;x+1,y,z], num_vertex_x, num_vertex_y,
                num_vertex_z, debug_output_level);

            c(5) = Get_cube_number([x-1,y,z;x,y+1,z+1], num_vertex_x, num_vertex_y,
                num_vertex_z, debug_output_level);
            c(6) = Get_cube_number([x-1,y,z-1;x,y+1,z], num_vertex_x, num_vertex_y,
                num_vertex_z, debug_output_level);

```

```

c(7) = Get_cube_number([x-1,y-1,z;x,y,z+1],num_vertex_x,num_vertex_y,
    num_vertex_z,debug_output_level);
c(8) = Get_cube_number([x-1,y-1,z-1;x,y,z],num_vertex_x,num_vertex_y,
    num_vertex_z,debug_output_level);

// Find all cubes around a vertex (max 8 cubes per vertex)
for a = 1:8
    if (c(a) ~= -1) then
        star0(i,i) = star0(i,i) + s0;
    end;
end;

// Find all cubes around an edge (max 4 cubes per edge)
// X direction edge
if e(1) ~= -1 then
    if c(1) ~= -1 then star1(e(1),e(1)) = star1(e(1),e(1)) + s1; end;
    if c(2) ~= -1 then star1(e(1),e(1)) = star1(e(1),e(1)) + s1; end;
    if c(3) ~= -1 then star1(e(1),e(1)) = star1(e(1),e(1)) + s1; end;
    if c(4) ~= -1 then star1(e(1),e(1)) = star1(e(1),e(1)) + s1; end;
end;
// Y direction edge
if e(2) ~= -1 then
    if c(1) ~= -1 then star1(e(2),e(2)) = star1(e(2),e(2)) + s1; end;
    if c(2) ~= -1 then star1(e(2),e(2)) = star1(e(2),e(2)) + s1; end;
    if c(5) ~= -1 then star1(e(2),e(2)) = star1(e(2),e(2)) + s1; end;
    if c(6) ~= -1 then star1(e(2),e(2)) = star1(e(2),e(2)) + s1; end;
end;
// Z direction edge
if e(3) ~= -1 then
    if c(1) ~= -1 then star1(e(3),e(3)) = star1(e(3),e(3)) + s1; end;
    if c(3) ~= -1 then star1(e(3),e(3)) = star1(e(3),e(3)) + s1; end;
    if c(5) ~= -1 then star1(e(3),e(3)) = star1(e(3),e(3)) + s1; end;
    if c(7) ~= -1 then star1(e(3),e(3)) = star1(e(3),e(3)) + s1; end;
end;

//Find all cubes around a face (max 2 cubes per face)
// Face in X-Y plane
if f(1) ~= -1 then
    if c(1) ~= -1 then star2(f(1),f(1)) = star2(f(1),f(1)) + s2; end;
    if c(2) ~= -1 then star2(f(1),f(1)) = star2(f(1),f(1)) + s2; end;
end;

```

```

// Face in X-Z plane
if f(2) ~= -1 then
    if c(1) ~= -1 then star2(f(2),f(2)) = star2(f(2),f(2)) + s2; end;
    if c(3) ~= -1 then star2(f(2),f(2)) = star2(f(2),f(2)) + s2; end;
end;
// Face in Y-Z plane
if f(3) ~= -1 then
    if c(1) ~= -1 then star2(f(3),f(3)) = star2(f(3),f(3)) + s2; end;
    if c(5) ~= -1 then star2(f(3),f(3)) = star2(f(3),f(3)) + s2; end;
end;

// For each valid (positive) cube, define dual vertex
if c(1) ~= -1 then star3(c(1),c(1)) = s3; end;

end; // for x = 1:num_vertex_x
end; // for y = 1:num_vertex_y
end; // for z = 1:num_vertex_z

// If debugging is enabled, display debugging information
if debug_output_level > 0 then
    disp('-----_BEGIN:_function_Build_star_operator_debugging_output_
        -----');
    disp('The_star0,_star1,_star2_and_star3_matrices_are:')
    disp(star0,'Star0:');
    disp(star1,'Star1:');
    disp(star2,'Star2:');
    disp(star3,'Star3:');
    disp('-----_END:_function_Build_star_operator_debugging_output_
        -----');
end;

endfunction;

```

D.3 DEC Specific Functions

```

function [primal_2form,num_unknown] = Set_initial_values(input_face,output_face,
    internal_wall_cubes,input_mass_flow_rate,primal_vol,dual_vol,star2,
    debug_output_level);

```

```

// Set all the face fluxes to the "unknown" value
primal_2form(1:num_faces) = unknown;
s2 = dual_vol(form_2)/primal_vol(form_2)

// run through all the faces, find the boundary faces and record number of unknowns to
// solve for
num_unknown = 0;
for i=1:num_faces

    // Check to see if we have a boundary face
    if (star2(i,i) == s2) then
        primal_2form(i) = 0;
    else
        // the face has an unknown flux
        num_unknown = num_unknown + 1;
    end;

end;

// Set the constraints on the flow in the system
// Set the input flux * area of face (integral value over face) -> units are kg/s
primal_2form(input_face) = input_mass_flow_rate;

if flow_type == 1 // "pipe" - Poiseuille flow
    // all faces adjacent to a no-slip wall should have a half flux.
    firstface = [];
    lastface = [];

    if num_vertex_y == 2 then // we have a 2D flow!
        firstface = input_face(1);
        lastface = input_face(size(input_face,1));
    end;

    primal_2form(firstface) = 1/2 * input_mass_flow_rate;
    primal_2form(lastface) = 1/2 * input_mass_flow_rate;

end;

// Set the "unknown" value on the last face
primal_2form(output_face) = unknown;
// Add output faces to number of unknowns to solve for

```

```

unknown_outputs = size(output_face,1);
num_unknown = num_unknown + unknown_outputs;

///// ADDING any extra no-slip walls inside domain
//for i = 1:size(internal_wall_cubes,1)
// // top and bottom faces are zero and are already listed as known
// cube_faces = cube_mesh(internal_wall_cubes(i),:);
//
// // find the number of faces that are selected as unknowns and make them all known
// num_cube_unknown_faces = size(find(primal_2form(cube_faces)==unknown),2); //
// output from find is in row direction
// num_unknown = num_unknown - num_cube_unknown_faces;
// primal_2form(cube_faces) = 0;
//end;
disp('——>Time to set boundary conditions and find number of unknowns: ' + string(
    timer()) + 's');

endfunction;

// -----

function [primal_2form] = Set_input_boundary_val(primal_2form, primal_vol, dual_vol,
    star1);
s2 = dual_vol(form_2)/primal_vol(form_2)

// run through all the faces, find the boundary faces and overwrite with originals.
for i=1:num_faces
    // Check to see if we have a boundary face
    if (star2(i,i) == s2) then

        if find(input_face==i)==[] & find(output_face==i)==[] then // no input/output
            faces
            primal_2form(i) = 0;
        end;

    end;

end;
end;

```

```

primal_2form(input_face) = input_mass_flow_rate;

if flow_type == 1 /"pipe" - Poiseuille flow
    // all faces adjacent to a no-slip wall should have a half flux.
    firstface = [];
    lastface = [];

    if num_vertex_y == 2 then // we have a 2D flow!
        firstface = input_face(1);
        lastface = input_face(size(input_face,1));
    end;

    primal_2form(firstface) = 1/2 * input_mass_flow_rate;
    primal_2form(lastface) = 1/2 * input_mass_flow_rate;

end;
endfunction;

// -----

function [primal_2form] = Set_output_boundary_val(primal_2form, primal_vol, dual_vol,
    star1);
for i=1:size(output_face,1)
    f = output_face(i);
    // get the associated cube and set the output to be div-free!!
    [cube, face_number_on_cube] = find(cube_mesh==f);
    cube_faces = cube_mesh(cube,:);

    //make sure that the faces on the cube have a total flux that sums to zero!
    // d2 is defined as follows:
    // d2(cube, cube_mesh(cube, cube_face_1)) = 1; //Front
    // d2(cube, cube_mesh(cube, cube_face_2)) = 1; //Top
    // d2(cube, cube_mesh(cube, cube_face_3)) = 1; //Left
    // d2(cube, cube_mesh(cube, cube_face_4)) = -1; //Right
    // d2(cube, cube_mesh(cube, cube_face_5)) = -1; //Bottom
    // d2(cube, cube_mesh(cube, cube_face_6)) = -1; //Back

    // Therefore, we can solve the following equation:
    // [1 1 1 -1 -1 -1] * [fluxes] = 0;

```



```

// by setting the output face to a flux of zero, I can solve the above equation for
// the remaining flux
primal_2form(f) = 0;
output_value = full(d2(cube,cube_faces))*primal_2form(cube_faces);

// Set the output face to the new div-free solution
primal_2form(f) = output_value;
end;

// Now we need to scale the output to make it divfree with the input!
// display the discrepancy between input and output flow
total_input_flux = sum(full_primal_2form(input_face));
total_output_flux = sum(full_primal_2form(output_face));
dissipated_flux = total_input_flux - total_output_flux;
// what do I need to multiply my output fluxes by to make them equal my input flux?
flux_scale_factor = total_input_flux / total_output_flux;

//Apply the changes to the output flux
primal_2form(output_face) = primal_2form(output_face) * flux_scale_factor;

disp("Input_flux:_" + string(total_input_flux));
disp("Output_flux:_" + string(total_output_flux));
disp("Input/output_flux_difference:_" + string(dissipated_flux));

endfunction;

// -----

function [a_form_array] = Zero_boundary_values(a_form_array, primal_vol, dual_vol, star2,
    primal_form_number);
s2 = dual_vol(primal_form_number)/primal_vol(primal_form_number)

// run through all the faces, find the boundary faces and overwrite
for i=1:size(a_form_array,1);
    // Check to see if we have a boundary face
    if (star2(i,i) == s2) then
        a_form_array(i) = 0;
    end;
end;

```

```
end;
```

```
endfunction;
```

```
// _____
```

```
function Display_boundary_vel(brick_values_27node);
```

```
// make sure all velocity on the node is zero if on boundary
```

```
for z = 1 : num_vertex_z_27node
```

```
    for y = 1 : num_vertex_y_27node
```

```
        for x = 1 : num_vertex_x_27node
```

```
            if x == 1 | y == 1 | z == 1 | x == num_vertex_x_27node | y ==  
                num_vertex_y_27node | z == num_vertex_z_27node then
```

```
                // we have a boundary node
```

```
                v = Get_vertex_number([x,y,z], num_vertex_x_27node, num_vertex_y_27node,  
                    num_vertex_z_27node, debug_output_level);
```

```
                disp("Node_at_location_" + string(x) + "," + string(y) + "," + string(z) + "_  
                    has_a_velocity_of_" + string(brick_values_27node(v)) );
```

```
            end;
```

```
        end;
```

```
    end;
```

```
end;
```

```
endfunction;
```

```
// _____
```

```
function [x] = Matrix_solve(A,x,b,unknown);
```

```
// unknown = value that an unknown is labeled with
```

```
// scan through matrix x and find locations of unknowns
```

```
unknown_counter = 0;
```

```
for i = 1 : size(x,1)
```

```
    if x(i) == unknown then
```

```
        unknown_counter = unknown_counter + 1;
```

```
        // record the location of the unknown value to be able to remove the  
            appropriate column from the A matrix
```

```

        unknown_location(unknown_counter) = i;
    end;
end;

// Backup the original matrices as we will be working on them
A_knowns = A; x_knowns = x; b_knowns = b;
// Reformat the A matrix removing the columns corresponding to the unknown values
// This automatically removes columns numbered in the matrix unknown_locations
A_knowns(:, unknown_location') = [];
// Reformat x matrix removing the unknown rows;
x_knowns(unknown_location', :) = [];

// find the new value that will be subtracted from b and subtract ([L2_unknown *
    x_unknown] + [L2_known * x_known] = b)
b = b - (A_knowns * x_knowns);
// Build the unknown matrix
A_unknowns = A(:, unknown_location');

// Solve for the unknowns
M = size(A_unknowns, 1);
N = size(A_unknowns, 2)
R = rank(full(A_unknowns));
disp('Seeking a solution of Ax=b for matrices of size_' + string(M) + '_' +
    string(N) + '][x] = ' + string(size(b, 1)) + '_' + string(size(b, 2)) + ']'_and_rank_
    of:_' + string(R));
if M > N then // overdetermined system —> least squares solution
    X = A_unknowns\b;
    //X=pinv(full(A_unknowns))*b
else
    X=linsolve(full(A_unknowns), full(-b));
end;

// Add the unknowns back into the original passed matrix
x(unknown_location', :) = X

disp('—> Equation solving time:_' + string(timer()) + 's');

endfunction;

```

```

// -----

function [dual_0form] = Find_centroid_val(primal_2form,num_cubes,num_vertex_x,
    num_vertex_y,num_vertex_z,debug_output_level);

dual_0form = zeros(num_cubes,3);

for c=1:num_cubes
    // get face numbers around cube
    position = Get_position_of_cube(c,num_vertex_x,num_vertex_y,num_vertex_z,
        debug_output_level);
    x = position(1,1);
    y = position(1,2);
    z = position(1,3);

    f(1) = Get_face_number([x,y,z;x,y+1,z+1],num_vertex_x,num_vertex_y,num_vertex_z,
        debug_output_level);
    f(2) = Get_face_number([x,y,z;x+1,y,z+1],num_vertex_x,num_vertex_y,num_vertex_z,
        debug_output_level);
    f(3) = Get_face_number([x,y,z;x+1,y+1,z],num_vertex_x,num_vertex_y,num_vertex_z,
        debug_output_level);

    f(4) = Get_face_number([x+1,y,z;x+1,y+1,z+1],num_vertex_x,num_vertex_y,
        num_vertex_z,debug_output_level);
    f(5) = Get_face_number([x,y+1,z;x+1,y+1,z+1],num_vertex_x,num_vertex_y,
        num_vertex_z,debug_output_level);
    f(6) = Get_face_number([x,y,z+1;x+1,y+1,z+1],num_vertex_x,num_vertex_y,
        num_vertex_z,debug_output_level);
    // flow in x direction
    x_flow_sum = (primal_2form(f(1)) + primal_2form(f(4)))/2;
    y_flow_sum = (primal_2form(f(2)) + primal_2form(f(5)))/2;
    z_flow_sum = (primal_2form(f(3)) + primal_2form(f(6)))/2;

    dual_0form(c,:) = [x_flow_sum,y_flow_sum,z_flow_sum];
end; //for c=1:num_cubes

endfunction

```

```
function velocities = Find_velocity(flowrates , material_density , vertex_spacing);
```

```
velocities = flowrates * (1/material_density) * (1/(vertex_spacing^2));
```

```
endfunction;
```

```
function flowrates = Find_flowrate(velocities , material_density , vertex_spacing);
```

```
flowrates = velocities * material_density * vertex_spacing^2;
```

```
endfunction;
```

```
function [brick_values_27node] = Interpolate_centroid_val(dual_0form_values ,  
    primal_2form);
```

```
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
// This function takes the values at the cube centers and interpolates them onto a 27  
    node brick
```

```
// This way we can define a continuous velocity field throughout the domain
```

```
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
// INPUTS:
```

```
// - dual_0form_values = [array of x,y,z values at all cube centers] size(num_cubes  
    x 3);
```

```
// - primal_2form = face flux values
```

```
// OUTPUTS:
```

```
// - brick_values_27node = [array of x,y,z values on all 27-node cubes] size(  
    num_vertex_27node x 3)
```

```
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
// THIS FUCTION CAN PROBABLY BE REWRITTEN TO BE MORE EFFICIENT
```

```

// All the boundary wall vertices have a velocity of 0 unless it is an input/output
// face.

brick_values_27node = zeros(num_vertex_27node,3);
brick_values_8node = zeros(num_vertex,3);

for v = 1 : num_vertex
    // grab a node
    x = vertex_mesh(v,1);
    y = vertex_mesh(v,2);
    z = vertex_mesh(v,3);

    // pick a direction we will be interpolating in
    for direction = 1 : 3 // 1 = x, 2 = y, 3 = z
        // our current node
        v_27node(1) = Get_vertex_number([(x*2)-1,(y*2)-1,(z*2)-1],num_vertex_x_27node ,
            num_vertex_y_27node , num_vertex_z_27node , debug_output_level);

        select direction
        case 1 // —> if on x boundary (left or right), find adjacent y,z faces and
            add them to list
            // ——— LEFT OR RIGHT BOUNDARY NODE ———
            f(1) = Get_face_number([x,y-1,z-1;x,y,z],num_vertex_x,num_vertex_y ,
                num_vertex_z , debug_output_level); // +ve yz direction (sides)
            f(2) = Get_face_number([x,y-1,z;x,y,z+1],num_vertex_x,num_vertex_y ,
                num_vertex_z , debug_output_level); // +ve yz direction (sides)
            f(3) = Get_face_number([x,y,z-1;x,y+1,z],num_vertex_x,num_vertex_y ,
                num_vertex_z , debug_output_level); // +ve yz direction (sides)
            f(4) = Get_face_number([x,y,z;x,y+1,z+1],num_vertex_x,num_vertex_y ,
                num_vertex_z , debug_output_level); // +ve yz direction (sides)
            v_27node(2) = Get_vertex_number([(x*2)-1,(y*2)-1,(z*2)],num_vertex_x_27node ,
                num_vertex_y_27node , num_vertex_z_27node , debug_output_level); // +z
            direction
            v_27node(3) = Get_vertex_number([(x*2)-1,(y*2),(z*2)-1],num_vertex_x_27node ,
                num_vertex_y_27node , num_vertex_z_27node , debug_output_level); // +y
            direction
            v_27node(4) = Get_vertex_number([(x*2)-1,(y*2),(z*2)],num_vertex_x_27node ,
                num_vertex_y_27node , num_vertex_z_27node , debug_output_level); // face
            centered node
    end
end

```

case 2 // —> if on y boundary (top or bottom), find adjacent x,z faces and add them to list

// ——— TOP OR BOTTOM BOUNDARY NODE ———

```
f(1) = Get_face_number([x-1,y,z-1;x,y,z],num_vertex_x,num_vertex_y,
    num_vertex_z,debug_output_level); // +ve xz direction (top/bottom)
f(2) = Get_face_number([x,y,z-1;x+1,y,z],num_vertex_x,num_vertex_y,
    num_vertex_z,debug_output_level); // +ve xz direction (top/bottom)
f(3) = Get_face_number([x-1,y,z;x,y,z+1],num_vertex_x,num_vertex_y,
    num_vertex_z,debug_output_level); // +ve xz direction (top/bottom)
f(4) = Get_face_number([x,y,z;x+1,y,z+1],num_vertex_x,num_vertex_y,
    num_vertex_z,debug_output_level); // +ve xz direction (top/bottom)
v_27node(2) = Get_vertex_number([(x*2),(y*2)-1,(z*2)-1],num_vertex_x_27node,
    num_vertex_y_27node,num_vertex_z_27node,debug_output_level); // +x
    direction
v_27node(3) = Get_vertex_number([(x*2)-1,(y*2)-1,(z*2)],num_vertex_x_27node,
    num_vertex_y_27node,num_vertex_z_27node,debug_output_level); // +z
    direction
v_27node(4) = Get_vertex_number([(x*2),(y*2)-1,(z*2)],num_vertex_x_27node,
    num_vertex_y_27node,num_vertex_z_27node,debug_output_level); // face
    centered node
```

case 3 // —> if on z boundary (front or back), find adjacent x,y faces and add them to list

// ——— FRONT OR BACK BOUNDARY NODE ———

```
f(1) = Get_face_number([x-1,y-1,z;x,y,z],num_vertex_x,num_vertex_y,
    num_vertex_z,debug_output_level); // +ve xy direction (front/back)
f(2) = Get_face_number([x,y-1,z;x+1,y,z],num_vertex_x,num_vertex_y,
    num_vertex_z,debug_output_level); // +ve xy direction (front/back)
f(3) = Get_face_number([x-1,y,z;x,y+1,z],num_vertex_x,num_vertex_y,
    num_vertex_z,debug_output_level); // +ve xy direction (front/back)
f(4) = Get_face_number([x,y,z;x+1,y+1,z],num_vertex_x,num_vertex_y,
    num_vertex_z,debug_output_level); // +ve xy direction (front/back)
v_27node(2) = Get_vertex_number([(x*2),(y*2)-1,(z*2)-1],num_vertex_x_27node,
    num_vertex_y_27node,num_vertex_z_27node,debug_output_level); // +x
    direction
v_27node(3) = Get_vertex_number([(x*2)-1,(y*2),(z*2)-1],num_vertex_x_27node,
    num_vertex_y_27node,num_vertex_z_27node,debug_output_level); // +y
    direction
v_27node(4) = Get_vertex_number([(x*2),(y*2),(z*2)-1],num_vertex_x_27node,
    num_vertex_y_27node,num_vertex_z_27node,debug_output_level); // face
    centered node
```

```

end; // case statement

face_counter = 0;
face_value_sum = 0;
for i = 1 : size(f,1)
    if f(i) ~= -1 then
        if primal_2form(f(i)) == 0 then // we are on a boundary and should always set
            the node to have a value of zero
            face_value_sum = 0;
            face_counter = 1;
            // break out of the for loop as the velocity at the node must be zero to
            satisfy boundary condidtions
            break;
        else
            face_counter = face_counter + 1;
            face_value_sum = face_value_sum + primal_2form(f(i));
        end; // primal_2form(f(i)) == 0 then else
    end; // if f(i) ~= -1 then
end; // for loop

// x velocity of node -> average adjacent y,z faces and convert to velocity
brick_values_27node(v_27node(1),direction) = Find_velocity((face_value_sum/
    face_counter),material.density ,vertex.spacing);
brick_values_8node(v,direction) = brick_values_27node(v_27node(1),direction);

// now add the 27 brick node values in the +1 vnode2 and vnode3 directions only (
    this way we do not repeat doing each node in next step)
// in +vnode2 direction average of face above and below
if v_27node(2) ~= -1
    // reset all face counters and sums as we are doing the same thing as above but
    with 2 faces
    face_counter = 0;
    face_value_sum = 0;
    for i = 2 : 2 : 4
        if f(i) ~= -1
            if primal_2form(f(i)) == 0 then // we are on a boundary and should always
                set the node to have a value of zero
                face_value_sum = 0;
                face_counter = 1;

```



```

        // break out of the for loop as the velocity at the node must be zero to
        // satisfy boundary condidtions
        break;
    else
        face_value_sum = face_value_sum + primal_2form(f(i));
        face_counter = face_counter + 1;
    end; // if primal_2form(f(i)) == 0 then
end; // if f(i) ~= -1
end; //for i = 2 : 2 : 4

brick_values_27node(v_27node(2),direction) = Find_velocity((face_value_sum/
    face_counter),material.density ,vertex.spacing);
end; //if v_27node ~= -1

// in +vnode3 direction average of face left and right
if v_27node(3) ~= -1
    // reset all face counters and sums as we are doing the same thing as above but
    // with 2 faces
    face_counter = 0;
    face_value_sum = 0;
    for i = 3 : 1 : 4
        if f(i) ~= -1
            if primal_2form(f(i)) == 0 then // we are on a boundary and should always
                // set the node to have a value of zero
                face_value_sum = 0;
                face_counter = 1;
                // break out of the for loop as the velocity at the node must be zero to
                // satisfy boundary condidtions
                break;
            else
                face_value_sum = face_value_sum + primal_2form(f(i));
                face_counter = face_counter + 1;
            end; // if primal_2form(f(i)) == 0 then
        end; // if f(i) ~= -1
    end; //for i = 2 : 2 : 4

    brick_values_27node(v_27node(3),direction) = Find_velocity((face_value_sum/
        face_counter),material.density ,vertex.spacing);
end; //if v_27node ~= -1

```

```

// LASTLY, ADD THE VALUE FOR THE CENTER OF THE FACE (IF THE POSITIVE FACE IN
vnode(4) DIRECTION EXISTS)
if v_27node(4) ~= -1
    brick_values_27node(v_27node(4),direction) = Find_velocity((primal_2form(f(4)))
        ,material_density ,vertex_spacing);
    //disp("The face center value of face " + string(f(4)) + " in direction " +
        string(direction) + " is: " + string(brick_values_27node(v_27node(4),
        direction)) );
end;

end; // for direction = 1 : 3 // 1 = x, 2 = y, 3 = z
end; // for v = 1 : num_vertex

// LOOP THROUGH EDGES, FIND THEIR END POINTS, FIND THE LOCATION OF THE END POINTS,
// FIND THE CENTER LOCATION AND FILL IT
// NOW THAT ALL THE PRIMAL NODES ARE FILLED IN, A GOOD APPROXIMATION FOR THE HALF
// NODE IS THE AVERAGE BETWEEN THE TWO END NODES ON THE EDGE
// NOTE, ONLY VALID FOR THE VALUE IN THE NODE DIRECTION!
for e = 1 : num_edges
    vertices_of_edge = unique(edge_mesh(e,:)); // Unique removes duplicates - THESE
        ARE 8 BRICK NUMBERS! MUST CONVERT TO 27 BRICK NUMBERS!
    v_loc = (vertex_mesh(vertices_of_edge(:,:) *2) -1; // xyz converted to location
        in 27 brick mesh values
    verticies_of_edge_27node(1) = Get_vertex_number(v_loc(1,:),num_vertex_x_27node ,
        num_vertex_y_27node ,num_vertex_z_27node ,debug_output_level);
    verticies_of_edge_27node(2) = Get_vertex_number(v_loc(2,:),num_vertex_x_27node ,
        num_vertex_y_27node ,num_vertex_z_27node ,debug_output_level);

    v_cur_loc = round( sum(v_loc,1) / 2); // round to integer in case of round-off
        error;
    v_cur_num = Get_vertex_number(v_cur_loc,num_vertex_x_27node ,num_vertex_y_27node ,
        num_vertex_z_27node ,debug_output_level);

    // we only want to interpolate the values in the direction of the edge (
        brick_values_27node 1,2 or 3)
    // we need to know the direction of the edge!!!
    if v_loc(1,1) ~= v_loc(2,1) then // x diection
        brick_values_27node(v_cur_num,1) = (brick_values_27node(
            verticies_of_edge_27node(1),1) + brick_values_27node(verticies_of_edge_27node
            (2),1)) /2;
    end;
end;

```

```

elseif v_loc(1,2) ~= v_loc(2,2) then // y direction
    brick_values_27node(v_cur_num,2) = (brick_values_27node(
        vertices_of_edge_27node(1),2) + brick_values_27node(vertices_of_edge_27node
        (2),2)) / 2;
elseif v_loc(1,3) ~= v_loc(2,3) then // z direction
    brick_values_27node(v_cur_num,3) = (brick_values_27node(
        vertices_of_edge_27node(1),3) + brick_values_27node(vertices_of_edge_27node
        (2),3)) / 2;
end;
end; // for e = 1 : num_edges

```

```

//THE FACE CENTERS AREN'T BEING SET IN OTHER DIRECTION THEN THEIR FLUX DIRECTIONS!!
// SHOULD INTERPOLATE face primal vertices FOR OTHER DIRECTIONS NOT In FLUX

```

DIRECTION:

```

// LOOP THROUGH FACES
// —> FIND CENTER OF FACE AND IT'S LOCATION
// —> FIND THE FACE NODES AND THEIR VALUES
// —> INTERPOLATE IN BETWEEN THEM TO FIND THE FACE CENTERED VALUE
for f = 1 : num_faces
    vertices_of_face = unique(edge_mesh( face_mesh(f,:), :)); // Unique removes
        duplicates
    v_loc = (vertex_mesh(vertices_of_face(:),:) * 2) - 1; //xyz converted to location
        in 27 brick mesh values
    v_cur_loc = round( sum(v_loc,1) / 4); // round to integer in case of round-off
        error;
    v_cur_num = Get_vertex_number(v_cur_loc, num_vertex_x_27node, num_vertex_y_27node,
        num_vertex_z_27node, debug_output_level);

    vel_at_4_nodes = brick_values_8node(vertices_of_face,:);
    average_velocity = sum(vel_at_4_nodes,1) / 4;

    // find direction the face is oriented - must compare to the center node as
        vertices_of_face might not be in correct order
    if (v_loc(1,1) == v_cur_loc(1)) // x1 = x2 —> x direction flux through face
        is already set
        brick_values_27node(v_cur_num,2) = average_velocity(2);
        brick_values_27node(v_cur_num,3) = average_velocity(3);
    elseif (v_loc(1,2) == v_cur_loc(2)) // y1 = y2 —> y direction flux through face
        is already set

```

```

    brick_values_27node(v_cur_num,1) = average_velocity(1);
    brick_values_27node(v_cur_num,3) = average_velocity(3);
elseif (v_loc(1,3) == v_cur_loc(3)) // z1 = z2 → z direction flux through face
    is already set
    brick_values_27node(v_cur_num,1) = average_velocity(1);
    brick_values_27node(v_cur_num,2) = average_velocity(2);
end;
end; // f = 1 : num_faces

// Fill in brick_values_27node centroid locations. This is known directly from the
dual_0form values
for c = 1:num_cubes
    //find the vertex number at the center of the cube in the x,y,z positive
    directions (x,y,z values for 27 brick are all +1)
    vertices_of_cube = unique(edge_mesh( face_mesh( cube_mesh(c, :), :), :)); //
    Unique removes duplicates
    // min vertex will allways be front, top, left corner
    min_vertex_xyz_location = (vertex_mesh(min(vertices_of_cube), :) * 2) -1; //
    convert to 27node brick values automatically
    // +1 to the x,y and z coordinates
    cube_center_location = min_vertex_xyz_location + 1;
    v_27node_center = Get_vertex_number(cube_center_location, num_vertex_x_27node,
        num_vertex_y_27node, num_vertex_z_27node, debug_output_level);
    brick_values_27node(v_27node_center, :) = dual_0form_values(c, :);
end;

// Fill in the primal cube nodes not on the boundary
for z = 2 : num_vertex_z - 1
    for y = 2 : num_vertex_y - 1
        for x = 2 : num_vertex_x - 1

            v_8node = Get_vertex_number([x,y,z], num_vertex_x_27node, num_vertex_y_27node,
                num_vertex_z_27node, debug_output_level);
            // get vertex number for 27node brick mesh
            v_27node = Get_vertex_number([(x*2)-1,(y*2)-1,(z*2)-1], num_vertex_x_27node,
                num_vertex_y_27node, num_vertex_z_27node, debug_output_level);

            // get cube numbers around current vertex - getting the current cube (and
            hence and current dual velocity value)

```

```

c(1) = Get_cube_number([x,y,z;x+1,y+1,z+1],num_vertex_x,num_vertex_y,
    num_vertex_z,debug_output_level);
c(2) = Get_cube_number([x,y,z-1;x+1,y+1,z],num_vertex_x,num_vertex_y,
    num_vertex_z,debug_output_level);
c(3) = Get_cube_number([x,y-1,z;x+1,y,z+1],num_vertex_x,num_vertex_y,
    num_vertex_z,debug_output_level);
c(4) = Get_cube_number([x,y-1,z-1;x+1,y,z],num_vertex_x,num_vertex_y,
    num_vertex_z,debug_output_level);
c(5) = Get_cube_number([x-1,y,z;x,y+1,z+1],num_vertex_x,num_vertex_y,
    num_vertex_z,debug_output_level);
c(6) = Get_cube_number([x-1,y,z-1;x,y+1,z],num_vertex_x,num_vertex_y,
    num_vertex_z,debug_output_level);
c(7) = Get_cube_number([x-1,y-1,z;x,y,z+1],num_vertex_x,num_vertex_y,
    num_vertex_z,debug_output_level);
c(8) = Get_cube_number([x-1,y-1,z-1;x,y,z],num_vertex_x,num_vertex_y,
    num_vertex_z,debug_output_level);

// set all vertex values to zero. replace only the ones that exist
vertex_values = zeros(8,3);
for i = 1:8
    // Check if the cube exists (if not we are at a wall cube and value is zero
    )
    if c(i) ~= -1 then dual_vertex_values(i,:) = dual_0form_values(c(i),:); end
    ;
end; // for i=1:8

// NOW WE WILL EXTRAPOLATE ONTO THE 27 NODE-BRICK FROM THE BRICK CENTROIDS
// AND VERTEX VALUES
// First, do a 4-node interpolation for the 3 mid nodes (in the +ve direction
// ) next to the current vertex
// interpolate from the 8 dual values: rst = 0,0,0 for dual centroid (primal
// vertex) - this is the middle of the dual-brick
brick_values_27node(v_27node,:) = interpolate_8node([0,0,0],
    dual_vertex_values);
brick_values_8node(v_8node,:) = brick_values_27node(v_27node,:);

end; //for x = 1:num_vertex_x
end; //for y = 1:num_vertex_y
end; //for z = 1:num_vertex_z

```

```

// value of 1 = slip
// value of 2 = wall
// 1=left boundary
// 2=right boundary
// 3=top boundary
// 4=bottom boundary
// 5=front boundary
// 6=back boundary

// make sure all velocity on the node is zero if on boundary

//for z = 1 : num_vertex_z_27node
//  for y = 1 : num_vertex_y_27node
//    for x = 1 : num_vertex_x_27node

//      if x == 1 & boundary_conditions(1) == 2 then //Left boundary is a no-slip
//        condition (x=1)
//          // Fill in the nodes not on the boundary
//          v = Get_vertex_number([x,y,z], num_vertex_x_27node, num_vertex_y_27node,
//            num_vertex_z_27node, debug_output_level);
//          brick_values_27node(v,2) = 0;
//          brick_values_27node(v,3) = 0;
//
//      elseif x == num_vertex_x & boundary_conditions(2) == 2 then //right boundary
//        is a no-slip condition (x=num_vertex_x_27node)
//          v = Get_vertex_number([x,y,z], num_vertex_x_27node, num_vertex_y_27node,
//            num_vertex_z_27node, debug_output_level);
//          brick_values_27node(v,2) = 0;
//          brick_values_27node(v,3) = 0;
//      end;

//      if y == 1 & boundary_conditions(3) == 2 then //top boundary is a no-slip
//        condition (y=1)
//          // Fill in the nodes not on the boundary
//          v = Get_vertex_number([x,y,z], num_vertex_x_27node, num_vertex_y_27node,
//            num_vertex_z_27node, debug_output_level);
//          brick_values_27node(v,1) = 0;
//          brick_values_27node(v,3) = 0;
//

```

```

//      elseif y == num_vertex_y & boundary_conditions(4) == 2 then //bottom boundary
//      is a no-slip condition (y=num_vertex_y_27node)
//      v = Get_vertex_number([x,y,z], num_vertex_x_27node, num_vertex_y_27node,
//      num_vertex_z_27node, debug_output_level);
//      brick_values_27node(v,1) = 0;
//      brick_values_27node(v,3) = 0;
//      end;

//      if (z == 1) & (boundary_conditions(5) == 2) then //front boundary is a no-
//      slip condition (z=1)
//      // Fill in the nodes not on the boundary
//      v = Get_vertex_number([x,y,z], num_vertex_x_27node, num_vertex_y_27node,
//      num_vertex_z_27node, debug_output_level);
//      brick_values_27node(v,1) = 0;
//      brick_values_27node(v,2) = 0;
//
//      elseif (z == num_vertex_z) & (boundary_conditions(6) == 2) then //back
//      boundary is a no-slip condition (z=num_vertex_z_27node)
//      v = Get_vertex_number([x,y,z], num_vertex_x_27node, num_vertex_y_27node,
//      num_vertex_z_27node, debug_output_level);
//      brick_values_27node(v,1) = 0;
//      brick_values_27node(v,2) = 0;
//      end;

//      end;
//      end;
//end;

endfunction;

// -----

function [backtracked_location] = Backtrack_centroids(dual_velocities_8node, time_step
, vertex_spacing);
// Major simplification used!!

```

// Currently assuming that backtracked_flow is constant throughout timestep and cells

```
backtracked_location = zeros(num_cubes,3);
```

```
for c=1:num_cubes
```

```
    position = Get_position_of_cube(c,num_vertex_x,num_vertex_y,num_vertex_z,
```

```
        debug_output_level);
```

```
    centroid_position(1,:) = ((position(1,:) + position(2,:)) / 2) * vertex_spacing;
```

```
    // find the new backtracked location
```

```
    // This step should be expanded (maybe not just a linear backtrack) to be more  
    accurate.
```

```
    backtracked_location(c,:) = centroid_position(1,:) - (dual_velocities_8node(c,:) *  
        time_step);
```

```
end;
```

```
endfunction;
```

```
// _____
```

```
function [backtracked_value] = Find_backtracked_value(location,brick_values_27node,  
    vertex_spacing);
```

```
// Should convert to r,s,t -> find new backtracked_flow -> convert back to x,y,z
```

```
// ——— FIND VELOCITIES AT BACKTRACKED LOCATION ———
```

```
for c = 1:num_cubes
```

```
    //disp(location(c,:), "The location at which i am finding the backtracked velocity  
    at:");
```

```
    [backtracked_value(c,1),backtracked_value(c,2),backtracked_value(c,3)] =
```

```
        find_value_at_xyz(location(c,:),brick_values_27node,vertex_spacing);
```

```
end;
```

```
if debug_output_level > -1 then
```

```
    disp(backtracked_value, "This_is_the_value_at_interpolated_dual_backtracked_  
        vertices");
```



```

end;

endfunction;

// -----

function [r,s,t] = xyz_to_rst(cube_num,location_in_cube);
    // r -> x direction
    // s -> y direction
    // t -> z direction

    x = location_in_cube(1); y = location_in_cube(2); z = location_in_cube(3);

    cube_position = Get_position_of_cube(cube_num,num_vertex_x,num_vertex_y,
        num_vertex_z,debug_output_level) * vertex_spacing;

    r = (x - (1/2*(cube_position(1,1) + cube_position(2,1))) ) / ((cube_position(2,1) -
        cube_position(1,1))/2);
    s = (y - (1/2*(cube_position(1,2) + cube_position(2,2))) ) / ((cube_position(2,2) -
        cube_position(1,2))/2);
    t = (z - (1/2*(cube_position(1,3) + cube_position(2,3))) ) / ((cube_position(2,3) -
        cube_position(1,3))/2);

    // If debugging is set to extreme level, display debugging information
    if debug_output_level == 2 then
        disp ("The_x,y,z_location:_" + string(x) + "," + string(y) + "," + string(z) +
            ")");
        disp ("is_in_cube_" + string(cube_num) + "_and_results_in_the_following_r,s,t_
            mapping:_" + string(r) + "," + string(s) + "," + string(t) + ")");
    end;

endfunction

// -----

function [x_value, y_value, z_value] = find_value_at_xyz(location,brick_values_27node
    ,vertex_spacing);

```

```

// find which cube our location(x,y,z) is in.
// location holds the x,y,z coordinates of all the backtracked centroids
x = int(location(1) / vertex_spacing);
y = int(location(2) / vertex_spacing);
z = int(location(3) / vertex_spacing);

// NOTE, CHECKS HAVE TO BE PERFORMED TO MAKE SURE WE ARE NOT OUTSIDE OUR FLOW ON A
// USER DEFINED "INPUT" CUBE
// Currently simple checks are performed
if x > num_vertex_x-1 then
    x = num_vertex_x-1;
    location(1) = num_vertex_x * vertex_spacing;
elseif x < 1 then
    x = 1;
    location(1) = vertex_spacing;
end;

if y > num_vertex_y-1 then
    y = num_vertex_y-1;
    location(2) = num_vertex_y * vertex_spacing;
elseif y < 1 then
    y = 1;
    location(2) = vertex_spacing;
end;

if z > num_vertex_z-1 then
    z = num_vertex_z-1;
    location(3) = num_vertex_z * vertex_spacing;
elseif z < 1 then
    z = 1;
    location(3) = vertex_spacing;
end;

cube_num = Get_cube_number([x,y,z;x+1,y+1,z+1],num_vertex_x,num_vertex_y,
    num_vertex_z,debug_output_level);
if cube_num == -1 then
    disp(cube_num, "is_in_cube_number:", location(:), "The_location:");
    disp(location, "XYZ_location:");
end;

```

```

// // If debugging is set to extreme level, display debugging information
// if debug_output_level == 2 then
//     disp(cube_num, "is in cube number: ", location(:), "The location: ");
//     disp(location, "XYZ location:");
// end;

// SINCE WE ARE USING 27 NODE BRICK, FIND THE NEW X,Y,Z for the top-left-front
// corner of THE 27 NODE BRICK
x = (x*2)-1; y = (y*2)-1; z = (z*2)-1;

// Find vertex values for all 27 nodes are stored in brick_values_27node
// BELOW ARE THE LOCATIONS AND ORDER OF THE NODES TO BE PASSED TO THIS FUNCTION
//
//
//                                     / +ve (t)
//      19-----20-----21          /
//      /|          /|          /|      ———> +ve (r)
//      / |          / |          / |      |
//      10-----11-----12 |      \ / +ve (s)
//      /| |          / |          /| |
//      / | 22-----/-----23-----/-----24
//      1-----2-----3 | /|
//      | | / |          | / |          | | / |
//      | 13-----|-----14-----|-----15 |
//      | /| |          | / |          | /| |
//      | / | 25-----/-----26-----/-----27
//      4-----5-----6 | /
//      | | /          | /          | | /
//      | 16-----|-----17-----|-----18
//      | /          | /          | /
//      | /          | /          | /
//      7-----8-----9

// we want to make sure we never have more than 27 nodes
v = zeros(27,1);
node_num = -2;
for i = 0:2
    for j = 0:2
        node_num = node_num + 3;
        v(node_num) = Get_vertex_number([x,y+j,z+i],num_vertex_x_27node,
            num_vertex_y_27node,num_vertex_z_27node,debug_output_level);
    end
end

```

```

    if v(node_num) == -1 then
        disp(cube_num, "is_in_cube_number:", location(:), "The_location:");
        disp("The_27_node_XYZ_values_are:" + string(x) + ", " + string(y) + ", " +
            string(z));
    end;

    v(node_num+1) = v(node_num) + 1; // in x direction, + 1 x
    v(node_num+2) = v(node_num) + 2; // in x direction, + 1 x again
end;
end;

// if v(:) == -1 | cube_num == -1 then
//     disp("X,Y,Z coordinates " + string(x) + "," + string(y) + "," + string(z) + " are
//         inconsistent.");
//     disp("Error. Cube " + string(cube_num) + " does not have valid coordinates");
// end;

// ——— Interpolate to find the flowrate at the specified point ———
// Find rst location from xyz location in cube number "c"
[r,s,t] = xyz_to_rst(cube_num, location(:));

// Interpolate flowrate
// Note, the vertex values are not the same as the primal_flow values
// The vertex numbers around a cube are not in the same order as Primal_flow
vertex_values = zeros(27,3);
vertex_values = brick_values_27node(v(:),:); // 1 -> x; 2 -> y; 3 -> z;

values = interpolate_27node([r,s,t], vertex_values);

x_value = values(1,1);
// Interpolate Y flowrate
y_value = values(1,2);
// Interpolate Z flowrate
z_value = values(1,3);

endfunction;

// —————

```

```

function [dual_edge] = Calculate_circulation(locations,dual_0form_values,
    primal_2form_values,brick_values_27node);
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
// Calculate circulation down dual edges
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
// INPUTS:
// - locations = [array of x,y,z locations of all backtracked cube centers] size(
    num_cubes x 3);
// - dual_0form_values = [array of x,y,z velocities of all backtracked cube centers]
    size(num_cubes x 3);
// - primal_2form_values = [array of primal face values - flux] size(num_faces x 1);
// - brick_values_27node = [array of x,y,z velocities for all nodes used in 27node
    brick interpolation] size(num_vertex_27node x 3)
// OUTPUTS:
// - dual_edge = [array of flows down dual edges] size(num_primal_faces x 1)
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
// BOUNDARY CONDITION OPTIONS
// 1 = SLIP
// 2 = NO-SLIP
// boundary_conditions(1) = Left boundary condition
// boundary_conditions(2) = Right boundary condition
// boundary_conditions(3) = Top boundary condition
// boundary_conditions(4) = Bottom boundary condition
// boundary_conditions(5) = Front boundary condition
// boundary_conditions(6) = Back boundary condition

// for each cube we have vertex of the dual edge location and flowrate at that point.
dual_edge_flowrate = [];
// dual edge numbering should be identical to primal face numbering as each dual edge
    is associated to one primal face
num_dual_edge = num_faces;
dual_edge = zeros(num_dual_edge);

// FOR EACH DUAL EDGE
// For every adjacent cube, there will be a dual edge joining the cube centers
// Boundary dual edges must be handled with care - circulation on boundary = 0 with
    viscous flows
for i = 1:num_dual_edge

    // Find position of dual edge

```

```

//      Primal
//      face
//      |
//      x---|---x dual edge
//      |
// Since each dual edge is associated with a primal face, by finding the location
// of the primal face it is easy to find the location of the dual edge. :-)
face_position = Get_position_of_face(i,num_vertex_x,num_vertex_y,num_vertex_z,
    debug_output_level);

// For ease of reading
X1 = face_position(1,1);
Y1 = face_position(1,2);
Z1 = face_position(1,3);
X2 = face_position(2,1);
Y2 = face_position(2,2);
Z2 = face_position(2,3);

// find the flow through the center of our current face
// first find the x,y,z 27 node integer location of the face center node
v_loc_face_center = round( sum((face_position*2) -1,1) / 2); // round to closest
    integer in case of round-off error;
// find the node number at the location of the face center
v_num_face_center = Get_vertex_number(v_loc_face_center ,num_vertex_x_27node ,
    num_vertex_y_27node ,num_vertex_z_27node , debug_output_level);
// look up the velocity of the face center node (x,y,z values)
v_vel_face_center(1,:) = brick_values_27node(v_num_face_center ,:);

// I am dealing with a rectangular domain - makes checking for boundary easier
// Need to check if this is a boundary face
if X1 == X2 then // X1 = X2 of primal face (we are on a face in the Y-Z direction
)
// therefore , the dual edge is in the X direction
// Check to see that X1 is 1 or num_vertex_x

select X1
case 1 then // X1=1
    // left boundary dual edge
    // calculate the (x,y,z) location on boundary - center of face
    boundary_location = [X1,(Y1+Y2)/2,(Z1+Z2)/2] * vertex_spacing;

```

```

c = Get_cube_number([X1,Y1,Z1;X1+1,Y1+1,Z1+1],num_vertex_x,num_vertex_y,
    num_vertex_z,debug_output_level);
dual_edge_vector = locations(c,:) - boundary_location;

// Find average flowrate down dual edge
if boundary_conditions(1) == 2 then // no-slip condition
    // flow is only allowed in the face direction
    dual_edge_flowrate = 1/2 * ([v_vel_face_center(1),0,0] + dual_0form_values
        (c,:));
    //          dual_edge_flowrate = 1/2 * ([0,0,0] + dual_0form_values(c,:));
else
    // flow in all directions is allowed
    dual_edge_flowrate = 1/2 * (v_vel_face_center + dual_0form_values(c,:));
end;

case num_vertex_x then
    // right boundary dual edge
    boundary_location = [X1,(Y1+Y2)/2,(Z1+Z2)/2] * vertex_spacing;;
    c = Get_cube_number([X1-1,Y1,Z1;X1,Y1+1,Z1+1],num_vertex_x,num_vertex_y,
        num_vertex_z,debug_output_level);
    dual_edge_vector = boundary_location - locations(c,:);
    // Find average flowrate on dual edge

    if boundary_conditions(2) == 2 then // no-slip condition (tangential)
        // flow is only allowed in the face direction
        dual_edge_flowrate = 1/2 * ([v_vel_face_center(1),0,0] + dual_0form_values
            (c,:));
        //          dual_edge_flowrate = 1/2 * ([0,0,0] + dual_0form_values(c,:));
    else
        // flow in all directions is allowed
        dual_edge_flowrate = 1/2 * (v_vel_face_center + dual_0form_values(c,:));
    end;

else
    // normal dual edge in X direction - internal dual edge between 2 bricks/
    // cubes
    c0 = Get_cube_number([X1-1,Y1,Z1;X1,Y1+1,Z1+1],num_vertex_x,num_vertex_y,
        num_vertex_z,debug_output_level);
    c1 = Get_cube_number([X1,Y1,Z1;X1+1,Y1+1,Z1+1],num_vertex_x,num_vertex_y,
        num_vertex_z,debug_output_level);
    dual_edge_vector = locations(c1,:) - locations(c0,:);

```

```

    dual_edge_flowrate = 1/2 * (dual_0form_values(c0,:) + dual_0form_values(c1,:))
    );
end;

elseif Y1 == Y2 then // Y1 = Y2 (we are on a face in the X-Z direction)
    // Y direction dual edge

    select Y1
    case 1 then
        // top boundary dual edge
        // calculate the (x,y,z) location on boundary
        boundary_location = [(X1+X2)/2,Y1,(Z1+Z2)/2] * vertex_spacing;
        c = Get_cube_number([X1,Y1,Z1;X1+1,Y1+1,Z1+1],num_vertex_x,num_vertex_y,
            num_vertex_z,debug_output_level);
        dual_edge_vector = locations(c,:) - boundary_location;

        // Find flowrate through the top boundary face
        if boundary_conditions(3) == 2 then // no-slip condition (tangential)
            // flow is only allowed in the face direction
            dual_edge_flowrate = 1/2 * ([0,v_vel_face_center(2),0] + dual_0form_values
                (c,:));
            //
            dual_edge_flowrate = 1/2 * ([0,0,0] +
                dual_0form_values(c,:));
        else
            // flow in all directions is allowed
            dual_edge_flowrate = 1/2 * (v_vel_face_center + dual_0form_values(c,:));
        end;

    case num_vertex_y then
        // bottom boundary dual edge
        boundary_location = [(X1+X2)/2,Y1,(Z1+Z2)/2] * vertex_spacing;
        c = Get_cube_number([X1,Y1-1,Z1;X1+1,Y1,Z1+1],num_vertex_x,num_vertex_y,
            num_vertex_z,debug_output_level);
        dual_edge_vector = boundary_location - locations(c,:);

        // Find flowrate through the bottom boundary face
        if boundary_conditions(4) == 2 then // no-slip condition (tangential)
            // flow is only allowed in the face direction
            dual_edge_flowrate = 1/2 * ([0,v_vel_face_center(2),0] + dual_0form_values
                (c,:));

```



```

//                                dual_edge_flowrate = 1/2 * ([0,0,0] +
dual_0form_values(c,:));
else
// flow in all directions is allowed
dual_edge_flowrate = 1/2 * (v_vel_face_center + dual_0form_values(c,:));
end;

else
// normal dual edge in Y direction
c0 = Get_cube_number([X1,Y1-1,Z1;X1+1,Y1,Z1+1],num_vertex_x,num_vertex_y,
num_vertex_z,debug_output_level);
c1 = Get_cube_number([X1,Y1,Z1;X1+1,Y1+1,Z1+1],num_vertex_x,num_vertex_y,
num_vertex_z,debug_output_level);
dual_edge_vector = locations(c1,:) - locations(c0,:);
dual_edge_flowrate = 1/2 * (dual_0form_values(c0,:) + dual_0form_values(c1,:))
);
end;

else // Z1 = Z2 (we are on a face in the X-Y direction)
// Z direction dual edge

select Z1
case 1 then

// front boundary dual edge
boundary_location = [(X1+X2)/2,(Y1+Y2)/2,Z1] * vertex_spacing;
c = Get_cube_number([X1,Y1,Z1;X1+1,Y1+1,Z1+1],num_vertex_x,num_vertex_y,
num_vertex_z,debug_output_level);
dual_edge_vector = locations(c,:) - boundary_location;

// Find flowrate through the front boundary face
if boundary_conditions(5) == 2 then // no-slip condition (tangential)
// flow is only allowed in the face direction
dual_edge_flowrate = 1/2 * ([0,0,v_vel_face_center(3)] + dual_0form_values
(c,:));
//                                dual_edge_flowrate = 1/2 * ([0,0,0] +
dual_0form_values(c,:));

else
// flow in all directions is allowed
dual_edge_flowrate = 1/2 * (v_vel_face_center + dual_0form_values(c,:));

```

```

end;

case num_vertex_z then
    // Find flowrate through the back boundary face
    //f = Get_face_number([X1,Y1,Z1;X1+1,Y1+1,Z1],num_vertex_x,num_vertex_y,
        num_vertex_z,debug_output_level);
    //face_value = Find_velocity(primal_2form_values(f),material_density,
        vertex_spacing);

    // back boundary dual edge
    boundary_location = [(X1+X2)/2,(Y1+Y2)/2,Z1] * vertex_spacing;
    c = Get_cube_number([X1,Y1,Z1-1;X1+1,Y1+1,Z1],num_vertex_x,num_vertex_y,
        num_vertex_z,debug_output_level);
    dual_edge_vector = boundary_location - locations(c,:);

    // Find flowrate through the back boundary face
    if boundary_conditions(6) == 2 then // no-slip condition (tangential)
        // flow is only allowed in the face direction
        dual_edge_flowrate = 1/2 * ([0,0,v_vel_face_center(3)] + dual_0form_values
            (c,:));
        //
        dual_edge_flowrate = 1/2 * ([0,0,0] +
            dual_0form_values(c,:));

    else
        // flow in all directions is allowed
        dual_edge_flowrate = 1/2 * (v_vel_face_center + dual_0form_values(c,:));
    end;

else
    // normal dual edge in Z direction
    c0 = Get_cube_number([X1,Y1,Z1-1;X1+1,Y1+1,Z1],num_vertex_x,num_vertex_y,
        num_vertex_z,debug_output_level);
    c1 = Get_cube_number([X1,Y1,Z1;X1+1,Y1+1,Z1+1],num_vertex_x,num_vertex_y,
        num_vertex_z,debug_output_level);
    dual_edge_vector = locations(c1,:) - locations(c0,:);
    dual_edge_flowrate = 1/2 * (dual_0form_values(c0,:) + dual_0form_values(c1,:)
        );
end;

end;

```

```

    // Calculate circulation down dual edge (dot product)
    //  $\mathbf{a} \cdot \mathbf{b} = \mathbf{a}^T \mathbf{b}$ 
    dual_edge(i) = dual_edge_flowrate * dual_edge_vector';

end;

endfunction;

// -----

function [dual_2form] = Calculate_vorticity(dual_1form, boundary_conditions);
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
// Calculate vorticity on dual_faces
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
// INPUTS:
// - dual_1form = [array of flows down dual edges] size(num_primal_faces x 1)
// - boundary_conditions = [array of 1 or 2 (slip/no-slip conditions)] size(6 x 1);
// OUTPUTS:
// - dual_face = [array of vorticity values on dual faces] size(num_primal_edges x
1)
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
// BOUNDARY CONDITION OPTIONS
// 1 = SLIP
// 2 = NO-SLIP
// boundary_conditions(1) = Left boundary condition
// boundary_conditions(2) = Right boundary condition
// boundary_conditions(3) = Top boundary condition
// boundary_conditions(4) = Bottom boundary condition
// boundary_conditions(5) = Front boundary condition
// boundary_conditions(6) = Back boundary condition

// for ease of reading
x = 1; y = 2; z = 3;

// each primal face = dual edge
// each primal edge = dual face
// must sum all the dual edge values with direction in mind (right hand rule) - aka.
paddel wheel with primal faces
// Steps:
// 1. Grab a primal edge (akin to dual_face)

```

```

// 2. Find the 4 associated primal faces (akin to dual_edges)
// 3. Using right hand rule, sum the dual_edge values and store the total on the
    dual_face

dual_2form = zeros(num_edges,1);
for cur_primal_edge = 1:num_edges //this should be a primal edge! as it's associated
    with a dual face!.. and find all the primal faces as they are associated with dual
    edges!
    // find the associated primal faces (2-4 faces)
    // face_mesh has all edges associated with each face
    // - search face_mesh for all faces with current edge

    // face_mesh is a 2D array of 1 to num_faces x num_edges_per_face
    // - each row has the 4 edge numbers associated with the face represented by the
        row number
    [in_rows,in_columns] = find(face_mesh == cur_primal_edge)
    number_of_paddle_faces = size(in_rows,2);
    face_numbers = in_rows; // for ease of reading

    // using right hand rule, sum all the values of the of the dual_edges associated
        with the primal_faces
    // all dual edges are considered positive in the following direction
    //
    //      +          USING RHR:
    //      /          - edge in x dir ( $x1 \triangleleft x2$ ) will have negative faces in the z+1 and
        y-1 directions
    //      0---> +    - edge in y dir ( $y1 \triangleleft y2$ ) will have negative faces in the x+1 and
        z-1 directions
    //      |          - edge in Z dir ( $z1 \triangleleft z2$ ) will have negative faces in the x-1 and
        y+1 directions
    //      +

    // find the edge vertices
    // edge_mesh is a 2D array of 1 to num_edges x 2 (vertices of each edge)
    // vertex_mesh is a 2D array of 1 to num_vertices x 3 (x,y,z coordinates of each
        vertex)
    edge_vertices_xyz = vertex_mesh(edge_mesh(cur_primal_edge,:),:);

for face = 1:number_of_paddle_faces

```

```

// Get_position_of_face return the top corner xyz and bottom corner xyz locations
// of the face
face_position = Get_position_of_face(face_numbers(face), num_vertex_x, num_vertex_y
, num_vertex_z, debug_output.level);

// face will always be positive in one direction (the direction of the edge)
// want to find if the face is in the -x/+x/-y/+y/-z/+z direction
// face_direction(face, x/y/z dir) = +1 or 0;

// Get position of face always returns the "top left" corner first
if face_position(1,:) - edge_vertices_xyz(1,:) == [0,0,0] // our face moves away
in a positive direction from the top edge corner
    face_direction(x) = face_position(2,x) - face_position(1,x);
    face_direction(y) = face_position(2,y) - face_position(1,y);
    face_direction(z) = face_position(2,z) - face_position(1,z);

else // our face is "
    negative"
    face_direction(x) = face_position(1,x) - face_position(2,x);
    face_direction(y) = face_position(1,y) - face_position(2,y);
    face_direction(z) = face_position(1,z) - face_position(2,z);
end;

// find direction of edge
// find the direction of face
// find which face is in which direction and build matrix of (1 and -1)'s to know
how to add dual edges
if edge_vertices_xyz(1,x) <> edge_vertices_xyz(2,x)
    // edge in x direction

    // edge in x dir (x1<>x2) will have negative faces in the z+1 and y-1
    directions
    if (face_direction(z) == 1) | (face_direction(y) == -1)
        d_face(face) = -1;
    else
        d_face(face) = 1;
    end;

elseif edge_vertices_xyz(1,y) <> edge_vertices_xyz(2,y)

```

```

// edge in y direction
// edge in y dir (y1<>y2) will have negative faces in the x+1 and z-1
// directions
if (face_direction(x) == 1) | (face_direction(z) == -1)
    d_face(face) = -1;
else
    d_face(face) = 1;
end;

elseif edge_vertices_xyz(1,z) <> edge_vertices_xyz(2,z)
// edge in z direction
// edge in Z dir (z1<>z2) will have negative faces in the x-1 and y+1
// directions
if (face_direction(x) == -1) | (face_direction(y) == 1)
    d_face(face) = -1;
else
    d_face(face) = 1;
end;

end; // if/elseif

// Check if we are on a dual edge adjacent and parallel to a boundary (primal
// face perpendicular to boundary) and override d_face value with a zero
// only if primal edge is on the boundary do I need to check if my face is
// perpendicular to the boundary
// 1. Check if primal edge is on a domain boundary
// 2. Check which boundary primal edge is on (primal edges along domain edges
// can be on 2 boundaries)
// 2. Check if we have a slip/no-slip condition on the boundary
// 3. If no-slip do nothing, else continue
// 4. Check if our current face is perpendicular to boundary
// 5. If perpendicular, set the value of d_face to 0

//[rows,columns] = find(face_position == 1)
//disp(face_position,"Face-position:")
//disp(rows,"Rows with face_position = 1");
//disp(columns,"Columns with face_position = 1");

```

```

    dual_2form(cur_primal_edge) = dual_2form(cur_primal_edge) + (d_face(face) *
        dual_1form(face_numbers(face)));

    end; //for face = 1:number_of_paddle_faces
end; //for cur_primal_edge = 1:num_edges

// // We have now calculated the flow down each dual edge. We need to now walk
// through all boundary condidtions and add or subtract the circulation corectly.
// // Maybe we should just walk through all the dual edges around each dual face and
// sum them up manually?
// //This section does not work correclty yet
// // if any of the conditions below exist, the effective circulation down the
// edge should be zero as it's a slip condition
// // 1. check which faces we are parallel too (4 of 6)
// // 2. check which boundaries we are next too
// // - for 2d flow we will be always next to the top and bottom flows
// // - need to ignore this and set vorticity to zero on these boundries)
// // 3. set the flow down the edge to zero (overwrite circulation down edge)
// if (X1 == 1) & (boundary_conditions(1) == 1) & (X1 ~= X2)
//     dual_edge(i) = 0;
// elseif (X2 == num_vertex_x) & (boundary_conditions(2) == 1) & (X1 ~= X2)
//     dual_edge(i) = 0;
// end;
//
// if (Y1 == 1) & (boundary_conditions(3) == 1) & (Y1 ~= Y2)
//     dual_edge(i) = 0;
// elseif (Y2 == num_vertex_y) & (boundary_conditions(4) == 1) & (Y1 ~= Y2)
//     dual_edge(i) = 0;
// end;
//
// if (Z1 == 1) & (boundary_conditions(5) == 1) & (Z1 ~= Z2)
//     dual_edge(i) = 0;
// elseif (Z2 == num_vertex_z) & (boundary_conditions(6) == 1) & (Z1 ~= Z2)
//     dual_edge(i) = 0;
// end;

// Sum circulation around dual faces using global DEC operator.
// Find the final circulations around a dual face (stored on the dual faces)
//dual_face_old = full(d1)' * dual_1form;

```

```

// disp(dual_2form,"Edge by edge:");
// disp(dual_face_old,"The DEC multiplication:");
endfunction;

```

D.4 Mesh Construction and Operation Functions

```

function [vertex_num] = Get_vertex_number(position , num_vertex_x , num_vertex_y ,
num_vertex_z , debug_output_level)
    x = position(1);
    y = position(2);
    z = position(3);

    if (x > num_vertex_x) | (y > num_vertex_y) | (z > num_vertex_z) | ..
        (x < 1) | (y < 1) | (z < 1) then

        // Not a valid vertex position
        vertex_num = -1;
        return;
    end;

    vertex_num = x + ((y-1)*num_vertex_x) + (z-1)*num_vertex_x*num_vertex_y;

    // If debugging is enabled, display debugging information - extream level of
    debugging
    if debug_output_level > 1 then
        disp("(" + string(x) + "," + string(y) + "," + string(z) + ")_is_vertex_number:_ " +
            string(vertex_num));
    end;

endfunction;

// -----

```

```

function [edge_num] = Get_edge_number(position , num_vertex_x , num_vertex_y , num_vertex_z
, debug_output_level)
    x1 = position(1,1);
    y1 = position(1,2);
    z1 = position(1,3);
    x2 = position(2,1);
    y2 = position(2,2);

```



```

z2 = position(2,3);

if (x2 > num_vertex_x) | (y2 > num_vertex_y) | (z2 > num_vertex_z) | ..
    (x1 < 1) | (y1 < 1) | (z1 < 1) then

    // Not a valid vertex position
    edge_num = -1;
    return;
end;

edge_num = 0;
// Is this edge in the x,y or z direction?
// First, add the current line of edges to the current edge
if (z1 == z2) & (y1 == y2) & (x1 ~= num_vertex_x) then
    // This is an edge in the X direction

    if (z1 == num_vertex_z) & (y1 == num_vertex_y) then
        // edge on back bottom corner
        edge_num = x1;

    elseif y1 == num_vertex_y then
        // edge on bottom plane
        edge_num = (x1 * 2) - 1;

    elseif z1 == num_vertex_z
        // edge is on back plane
        edge_num = (x1 * 2) - 1;

    else
        edge_num = (x1 * 3) - 2;
    end;

elseif (z1 == z2) & (x1 == x2) & (y1 ~= num_vertex_y) then
    // This is an edge in the Y direction

    if (x1 == num_vertex_x) & (z1 == num_vertex_z) then
        // edge on right back corner
        edge_num = (x1 * 2) - 1;

    elseif x1 == num_vertex_x then
        // edge on right plane

```

```

    edge_num = (x1 * 3) - 2;

elseif z1 == num_vertex_z
    // edge is on back plane
    edge_num = (x1 * 2);

else
    edge_num = (x1 * 3) - 1;
end;

elseif (x1 == x2) & (y1 == y2) & (z1 ~= num_vertex_z) then
    // This is an edge in the Z direction

    if (x1 == num_vertex_x) & (y1 == num_vertex_y) then
        // edge on right bottom corner
        edge_num = (x1 * 2) - 1;

    elseif x1 == num_vertex_x then
        // edge on right plane
        edge_num = (x1 * 3) - 1;

    elseif y1 == num_vertex_y
        // edge is on bottom plane
        edge_num = (x1 * 2);

    else
        edge_num = (x1 * 3);
    end;

else
    edge_num = -1;
    return;
end;

//Add number the rows of edges above the current row of edges
if z1 ~= num_vertex_z then
    // Not a back plane
    edge_num = edge_num + ((y1-1) * (3*num_vertex_x - 1));
else
    edge_num = edge_num + ((y1-1) * (2*num_vertex_x - 1));
end;

```

```

//Add number of x * y edge planes before this one.
edge_num = edge_num + ((z1-1) * ( (3 * num_vertex_x - 1) * (num_vertex_y - 1) + (2 *
    num_vertex_x - 1) ));

    // If debugging is enabled, display debugging information - extream level of
    debugging
    if debug_output_level > 1 then
        disp( "(" + string(x1) + "," + string(y1) + "," + string(z1)+")->(" + string(
            x2) + "," + string(y2) + "," + string(z2) + ")_is_edge_number:_ " + string(
                edge_num));
    end;

endfunction;

// -----

function [face_num] = Get_face_number(position ,num_vertex_x ,num_vertex_y ,num_vertex_z
    ,debug_output_level)
    x1 = position(1,1);
    y1 = position(1,2);
    z1 = position(1,3);
    x2 = position(2,1);
    y2 = position(2,2);
    z2 = position(2,3);

    if (x2 > num_vertex_x) | (y2 > num_vertex_y) | (z2 > num_vertex_z) | ..
        (x1 < 1) | (y1 < 1) | (z1 < 1) then

        // Not a valid vertex position
        face_num = -1;
        return;
    end;

    face_num = 0;
    // Is this face in the x,y or z direction?
    // First, add the current line of faces to the current face
    if (z1 == z2) & (x1 ~= num_vertex_x) & (y1 ~= num_vertex_y) then

```

```

// This is a face in the X plane
if z1 ~= num_vertex_z then
    face_num = (x1 * 3) - 2;
else
    face_num = x1;
end;

elseif (y1 == y2) & (x1 ~= num_vertex_x) & (z1 ~= num_vertex_z) then
    if y1 ~= num_vertex_y then
        face_num = (x1 * 3) - 1;
    else
        face_num = x1;
    end;

elseif (x1 == x2) & (y1 ~= num_vertex_y) & (z1 ~= num_vertex_z) then
    if x1 ~= num_vertex_x then
        face_num = (x1 * 3)
    else
        face_num = (x1 * 3) - 2;
    end;

else
    face_num = -1;
    return;
end;

//Add number of rows of faces above this row
if z1 ~= num_vertex_z then
    face_num = face_num + ((y1-1) * (3*num_vertex_x - 2));
else
    face_num = face_num + ((y1-1) * (num_vertex_x - 1));
end;

//Add number of x * y face planes before this one.
face_num = face_num + ((z1-1) * ( (3 * num_vertex_x - 2) * (num_vertex_y - 1) + (
    num_vertex_x - 1) ));

// If debugging is enabled, display debugging information - extream level of
debugging
if debug_output_level > 1 then

```

```

    disp( "(" + string(x1) + "," + string(y1) + "," + string(z1) + ")" -> "(" + string(
        x2) + "," + string(y2) + "," + string(z2) + ")"_is_face_number:_ " + string(
        face_num));
end;

endfunction;

// -----

function [cube_num] = Get_cube_number(position, num_vertex_x, num_vertex_y, num_vertex_z
, debug_output_level)

    x1 = position(1,1);
    y1 = position(1,2);
    z1 = position(1,3);
    x2 = position(2,1);
    y2 = position(2,2);
    z2 = position(2,3);

    if (x2 > num_vertex_x) | (y2 > num_vertex_y) | (z2 > num_vertex_z) | ..
        (x1 < 1) | (y1 < 1) | (z1 < 1) then

        // Not a valid cube
        cube_num = -1;
        return;
    end;

    cube_num = 0;
    // Add the current line of cubes to cube_num
    cube_num = x1;
    // Add the rows above but in the same plane to cube_num
    cube_num = cube_num + ((num_vertex_x - 1) * (y1 - 1));
    // Add the x by y plane of cubes before the current
    cube_num = cube_num + ((num_vertex_x - 1) * (num_vertex_y - 1) * (z1 - 1));

    // If debugging is enabled, display debugging information - extream level of
    debugging
    if debug_output_level > 1 then
        disp( "(" + string(x1) + "," + string(y1) + "," + string(z1) + ")" -> "(" + string(
            x2) + "," + string(y2) + "," + string(z2) + ")"_is_cube_number:_ " + string(
            cube_num));
    end;
end;

```

```

    end;

endfunction;

// -----

function [position] = Get_position_of_face(face_num,num_vertex_x,num_vertex_y,
    num_vertex_z,debug_output_level)
    current_face = face_num;
    // Faces per row of vertices
    row_face_num = (num_vertex_x * 3) - 2;

    // Faces per row * column (plane) of vertices
    plane_face_num = (row_face_num * (num_vertex_y - 1)) + (num_vertex_x - 1)

    // First check to see if we are not on the right bottom face (X-Z)
    if modulo(face_num,plane_face_num) == 0 then
        x1 = num_vertex_x - 1;
        x2 = x1 + 1;
        y1 = num_vertex_y;
        y2 = y1;
        z1 = int(face_num/plane_face_num);
        z2 = z1 + 1;

        position(1,1) = x1
        position(1,2) = y1;
        position(1,3) = z1;
        position(2,1) = x2;
        position(2,2) = y2;
        position(2,3) = z2;

        // If debugging is enabled, display debugging information - extream level of
        // debugging
        if debug_output_level > 1 then
            disp(position,"The_position_of_face_number_" + string(current_face) + "_is:");
        end;

        return(position);
    else
        // Check to see which plane we are on
        z1 = int(face_num/plane_face_num) + 1;

```

```

    face_num = face_num - (plane_face_num * (z1 - 1));
end;

// Double check for the special case of this face being on the very back of the
// domain
if z1 == num_vertex_z then
    // We are on the very back of the domain
    // Only faces in X-Y direction exist
    z2 = z1;

    if modulo(face_num, (num_vertex_x-1)) == 0 then
        // we are on last face in row
        y1 = int(face_num/(num_vertex_x-1));
    else
        y1 = int(face_num/(num_vertex_x-1)) + 1;
    end;

    face_num = face_num - ((num_vertex_x-1) * (y1 -1));
    y2 = y1 + 1;
    x1 = face_num;
    x2 = x1 + 1;

    position(1,1) = x1
    position(1,2) = y1;
    position(1,3) = z1;
    position(2,1) = x2;
    position(2,2) = y2;
    position(2,3) = z2;

    // If debugging is enabled, display debugging information - extream level of
    // debugging
    if debug_output_level > 1 then
        disp(position, "The_position_of_face_number_" + string(current_face) + "_is:");
    end;

    return(position);
end;

```

```

// Double check for the special case of this face being on the very right of the
domain
if modulo(face_num,row_face_num) == 0 then
// We are on the right side of domain (Y-Z)
    z2 = z1 + 1;
    y1 = int(face_num/row_face_num);
    y2 = y1 + 1;
    x1 = num.vertex_x;
    x2 = x1;

    position(1,1) = x1
    position(1,2) = y1;
    position(1,3) = z1;
    position(2,1) = x2;
    position(2,2) = y2;
    position(2,3) = z2;

    // If debugging is enabled, display debugging information - extream level of
    debugging
    if debug_output_level > 1 then
        disp(position,"The_position_of_face_number_" + string(current_face) + "_is:");
    end;

    return(position);
else
    // Check to see which row we are in
    y1 = int(face_num/row_face_num) + 1;
    face_num = face_num - (row_face_num * (y1 -1));
end;

// Double check for the special case of this face being on the very bottom of the
domain
if y1 == num.vertex_y then
    // We are on the very bottom of the domain
    // Only faces in X-Z direction exist
    z2 = z1 + 1;
    y2 = y1;
    x1 = face_num;
    x2 = x1 + 1;

```



```

    position(1,1) = x1
    position(1,2) = y1;
    position(1,3) = z1;
    position(2,1) = x2;
    position(2,2) = y2;
    position(2,3) = z2;

    // If debugging is enabled, display debugging information - extream level of
    // debugging
    if debug_output_level > 1 then
        disp(position, "The_position_of_face_number_" + string(current_face) + "_is:");
    end;

    return(position);
end;

// We are somewhere in the normal domain
// This means the face can be in any direction.

// This should be a case structure (does scilab support them?)
if modulo(face_num,3) == 0 then
    // Face is in the Y-Z direction (left side of cube)
    z2 = z1 + 1;
    y2 = y1 + 1;
    x1 = int(face_num/3);
    x2 = x1;
elseif modulo(face_num,3) == 2 then
    // Face is in the X-Z direction (top of cube)
    z2 = z1 + 1;
    y2 = y1;
    x1 = int(face_num/3) + 1;
    x2 = x1 + 1;
else
    // Must be a X-Y direction face (front of cube)
    z2 = z1;
    y2 = y1 + 1;
    x1 = int(face_num/3) + 1;
    x2 = x1 + 1;
end;

```

```

    position(1,1) = x1
    position(1,2) = y1;
    position(1,3) = z1;
    position(2,1) = x2;
    position(2,2) = y2;
    position(2,3) = z2;

    // If debugging is enabled, display debugging information - extream level of
    // debugging
    if debug_output_level > 1 then
        disp(position, "The_position_of_face_number_" + string(current_face) + "_is:");
    end;
endfunction;

// -----

function [position] = Get_position_of_cube(cube_num, num_vertex_x, num_vertex_y,
    num_vertex_z, debug_output_level)

    // (solve for z) Add the x by y planes of cubes before the current
    if modulo(cube_num, ((num_vertex_x - 1) * (num_vertex_y - 1))) == 0 then
        // We are on the last cube of a plane

        z1 = cube_num / ((num_vertex_x - 1) * (num_vertex_y - 1));
        y1 = num_vertex_y - 1;
        x1 = num_vertex_x - 1;

        position(1,1) = x1
        position(1,2) = y1;
        position(1,3) = z1;
        position(2,1) = x1+1;
        position(2,2) = y1+1;
        position(2,3) = z1+1;

        // If debugging is enabled, display debugging information - extream level of
        // debugging
        if debug_output_level > 1 then

```

```

        disp("Cube_number_" + string(cube_num) + "_is_at_location:_" + string(x1) + "
            ," + string(y1) + "," + string(z1)+")->_" + string(x1+1) + "," + string(y1
            +1) + "," + string(z1+1) + ")");
    end;

    return(position);
else
    z1 = int(cube_num/((num_vertex_x - 1) * (num_vertex_y - 1))) + 1
    new_cube_num = cube_num - ((num_vertex_x - 1) * (num_vertex_y - 1) * (z1 - 1));
end;

// (solve for y) Add the rows above but in the same plane to cube_num
if modulo(new_cube_num,(num_vertex_x - 1)) == 0 then
    // We are on the last cube of a row
    y1 = new_cube_num/(num_vertex_x - 1);
    x1 = num_vertex_x - 1;

    position(1,1) = x1
    position(1,2) = y1;
    position(1,3) = z1;
    position(2,1) = x1+1;
    position(2,2) = y1+1;
    position(2,3) = z1+1;

    // If debugging is enabled, display debugging information - extream level of
    debugging
    if debug_output_level > 1 then
        disp("Cube_number_" + string(cube_num) + "_is_at_location:_" + string(x1) + "
            ," + string(y1) + "," + string(z1)+")->_" + string(x1+1) + "," + string(y1
            +1) + "," + string(z1+1) + ")");
    end;

    return(position);
else
    y1 = int(new_cube_num/(num_vertex_x - 1)) + 1;
    new_cube_num = new_cube_num - ((num_vertex_x - 1) * (y1 - 1));
end;

x1 = new_cube_num;

```

```

    position(1,1) = x1
    position(1,2) = y1;
    position(1,3) = z1;
    position(2,1) = x1+1;
    position(2,2) = y1+1;
    position(2,3) = z1+1;

    // If debugging is enabled, display debugging information - extream level of
    debugging
    if debug_output_level > 1 then
        disp("Cube_number_" + string(cube_num) + "_is_at_location:_" + string(x1) + "," +
            + string(y1) + "," + string(z1)+")_>_" + string(x1+1) + "," + string(y1+1)
            + "," + string(z1+1) + ")");
    end;

endfunction;

// -----

function [num_vertex,vertex_mesh] = mesh_vertex(num_vertex_x,num_vertex_y,
    num_vertex_z,debug_output_level);
    // build the vertex mesh
    num_vertex = 0;

    for z = 1:num_vertex_z
        for y = 1:num_vertex_y
            for x = 1:num_vertex_x
                num_vertex = num_vertex + 1;
                vertex_mesh(num_vertex,:) = [x y z];
            end; //for x = 1:num_vertex_x
        end; //for y = 1:num_vertex_y
    end; //for z = 1:num_vertex_z

endfunction;

// -----

function [num_edges,edge_mesh] = mesh_edges(num_vertex,vertex_mesh,debug_output_level
);
    // build the edge mesh

```

```

num_edges = 0;

// run through all the vertices and record neighbouring vertices as edges
for vertex_1 = 1 : num_vertex-1
    // Scan through remaining vertices to see if any are neighbouring

    for vertex_2 = vertex_1+1 : num_vertex

        if ((vertex_mesh(vertex_1,1)+1 == vertex_mesh(vertex_2,1)) & ..
            (vertex_mesh(vertex_1,2) == vertex_mesh(vertex_2,2)) & ..
            (vertex_mesh(vertex_1,3) == vertex_mesh(vertex_2,3)) ..
            ) | ..
            ((vertex_mesh(vertex_1,1) == vertex_mesh(vertex_2,1)) & ..
            (vertex_mesh(vertex_1,2)+1 == vertex_mesh(vertex_2,2)) & ..
            (vertex_mesh(vertex_1,3) == vertex_mesh(vertex_2,3)) ..
            ) | ..
            ((vertex_mesh(vertex_1,1) == vertex_mesh(vertex_2,1)) & ..
            (vertex_mesh(vertex_1,2) == vertex_mesh(vertex_2,2)) & ..
            (vertex_mesh(vertex_1,3)+1 == vertex_mesh(vertex_2,3)) )

            //disp(string(vertex_mesh(vertex_1,:)) + " —> " + string(vertex_mesh(
                vertex_2,:)) );

            num_edges = num_edges + 1
            edge_mesh(num_edges,:) = [vertex_1 vertex_2];
            //disp(edge_mesh(num_edges,:));

        end; // super if
    end; // for vertex_2 = vertex_1 : num_vertex
end; // vertex_1 = 1 : num_vertex - 1

endfunction;

// -----

function [num_faces, face_mesh] = mesh_faces(num_vertex, vertex_mesh, num_edges,
    edge_mesh, debug_output_level);
    num_faces = 0;
    // find 4 edges that make a face

```

```

num_faces = Get_face_number([num_vertex_x-1,num_vertex_y-1,num_vertex_z;
    num_vertex_x ,num_vertex_y ,num_vertex_z ], num_vertex_x ,num_vertex_y ,num_vertex_z ,
    debug_output_level);

// Go around each face (starting at position x1,y1,z1) in the direction of face
    orientation
// face is oriented in the direction of lower to higher cube # (ie, right +ve; down
    +ve; back +ve)
for i = 1:num_faces
    face_position = Get_position_of_face(i ,num_vertex_x ,num_vertex_y ,num_vertex_z ,
        debug_output_level);
    //disp(face_position,"Face Position");

    // For ease of reading
    x1 = face_position(1,1);
    y1 = face_position(1,2);
    z1 = face_position(1,3);
    x2 = face_position(2,1);
    y2 = face_position(2,2);
    z2 = face_position(2,3);

    // For each face there are 4 edges that must be found
    // first we need to know in which direction the face is facing
    // note, it is possible to just search the edge_mesh for the edge number, but
        this should yeild better performace
    // searching through edge_mesh means minimal dependence on actual mesh
        orientation

    if z1 == z2 then
        // Face in X-Y direction

        // Top edge
        position(1,:) = [x1 y1 z1];
        position(2,:) = [x2 y1 z1];
        edge_num1 = Get_edge_number(position ,num_vertex_x ,num_vertex_y ,num_vertex_z ,
            debug_output_level)
        // right edge
        position(1,:) = [x2 y1 z1];
        position(2,:) = [x2 y2 z1];
        edge_num2 = Get_edge_number(position ,num_vertex_x ,num_vertex_y ,num_vertex_z ,
            debug_output_level)

```

```

// bottom edge (note, direction is flipped)
position(1,:) = [x1 y2 z1];
position(2,:) = [x2 y2 z1];
edge_num3 = Get_edge_number(position, num_vertex_x, num_vertex_y, num_vertex_z,
    debug_output_level)
// left edge (note, direction is flipped)
position(1,:) = [x1 y1 z1];
position(2,:) = [x1 y2 z1];
edge_num4 = Get_edge_number(position, num_vertex_x, num_vertex_y, num_vertex_z,
    debug_output_level)

elseif y1 == y2 then
// Face in X-Z direction

// Front edge
position(1,:) = [x1 y1 z1];
position(2,:) = [x1 y1 z2];
edge_num1 = Get_edge_number(position, num_vertex_x, num_vertex_y, num_vertex_z,
    debug_output_level)
// right edge
position(1,:) = [x1 y1 z2];
position(2,:) = [x2 y1 z2];
edge_num2 = Get_edge_number(position, num_vertex_x, num_vertex_y, num_vertex_z,
    debug_output_level)
// back edge (note, direction is flipped)
position(1,:) = [x2 y1 z1];
position(2,:) = [x2 y1 z2];
edge_num3 = Get_edge_number(position, num_vertex_x, num_vertex_y, num_vertex_z,
    debug_output_level)
// left edge (note, direction is flipped)
position(1,:) = [x1 y1 z1];
position(2,:) = [x2 y1 z1];
edge_num4 = Get_edge_number(position, num_vertex_x, num_vertex_y, num_vertex_z,
    debug_output_level)

elseif x1 == x2 then
// Face in Y-Z direction

// Front edge
position(1,:) = [x1 y1 z1];

```

```

    position(2,:) = [x1 y2 z1];
    edge_num1 = Get_edge_number(position , num_vertex_x , num_vertex_y , num_vertex_z ,
        debug_output_level)
    // bottom edge
    position(1,:) = [x1 y2 z1];
    position(2,:) = [x1 y2 z2];
    edge_num2 = Get_edge_number(position , num_vertex_x , num_vertex_y , num_vertex_z ,
        debug_output_level)
    // back edge (note, direction is flipped)
    position(1,:) = [x1 y1 z2];
    position(2,:) = [x1 y2 z2];
    edge_num3 = Get_edge_number(position , num_vertex_x , num_vertex_y , num_vertex_z ,
        debug_output_level)
    // top edge (note, direction is flipped)
    position(1,:) = [x1 y1 z1];
    position(2,:) = [x1 y1 z2];
    edge_num4 = Get_edge_number(position , num_vertex_x , num_vertex_y , num_vertex_z ,
        debug_output_level)

    end;

    face_mesh(i,:) = [edge_num1 edge_num2 edge_num3 edge_num4];

end; // for i = 1:num_faces

// disp(face_mesh);
endfunction;



---


function [num_cubes, cube_mesh] = mesh_cubes(debug_output_level);
    num_cubes = Get_cube_number([num_vertex_x-1,num_vertex_y-1,num_vertex_z-1;
        num_vertex_x , num_vertex_y , num_vertex_z] , num_vertex_x , num_vertex_y , num_vertex_z ,
        debug_output_level);

    for i = 1:num_cubes
        // Fine all the faces around each cube
        // Order of faces is: front, top, left , right , bottom, back
        // This order should correspond to lowest face ID -> highest face ID.

```



```

cube_position = Get_position_of_cube(i , num_vertex_x , num_vertex_y , num_vertex_z ,
    debug_output_level);
// For ease of reading
x1 = cube_position(1,1);
y1 = cube_position(1,2);
z1 = cube_position(1,3);
x2 = cube_position(2,1);
y2 = cube_position(2,2);
z2 = cube_position(2,3);

// Front Face
position(1,:) = [x1 y1 z1];
position(2,:) = [x2 y2 z1];
face_num1 = Get_face_number(position , num_vertex_x , num_vertex_y , num_vertex_z ,
    debug_output_level)
// Top Face
position(1,:) = [x1 y1 z1];
position(2,:) = [x2 y1 z2];
face_num2 = Get_face_number(position , num_vertex_x , num_vertex_y , num_vertex_z ,
    debug_output_level)
// Left Face
position(1,:) = [x1 y1 z1];
position(2,:) = [x1 y2 z2];
face_num3 = Get_face_number(position , num_vertex_x , num_vertex_y , num_vertex_z ,
    debug_output_level)
// Right Face
position(1,:) = [x2 y1 z1];
position(2,:) = [x2 y2 z2];
face_num4 = Get_face_number(position , num_vertex_x , num_vertex_y , num_vertex_z ,
    debug_output_level)
// Bottom Face
position(1,:) = [x1 y2 z1];
position(2,:) = [x2 y2 z2];
face_num5 = Get_face_number(position , num_vertex_x , num_vertex_y , num_vertex_z ,
    debug_output_level)
// Back Face
position(1,:) = [x1 y1 z2];
position(2,:) = [x2 y2 z2];
face_num6 = Get_face_number(position , num_vertex_x , num_vertex_y , num_vertex_z ,
    debug_output_level)

```

```

cube_mesh(i,:) = [face_num1 face_num2 face_num3 face_num4 face_num5 face_num6];

end; //for i = 1:num_cubes

endfunction;

```

D.5 FEM Interpolation Functions

```

function [result] = interpolate_4node(rs_location,vertex_values);
// NOTE: vertex values must match up in order with rst values

// BELOW ARE THE LOCATIONS AND ORDER OF THE NODES TO BE PASSED TO THIS FUNCTION
//
//
//      1-----2      --> +ve
//      |           |      |
//      |           |      |
//      |           |      \ / +ve
//      |           |
//      |           |
//      3-----4
//
// NODE 1 = (-1,-1)
// NODE 4 = (+1,+1)

// r -> x direction
// s -> y direction

r = rs_location(1); s = rs_location(2);

R = [1/4*(1-r)*(1-s),1/4*(1+r)*(1-s),1/4*(1-r)*(1+s),1/4*(1+r)*(1+s)];

result = R * vertex_values;

endfunction;

// -----

```

```

function [result] = interpolate_8node(rst_location,vertex_values);
// NOTE: vertex values must match up in order with rst values

// BELOW ARE THE LOCATIONS AND ORDER OF THE NODES TO BE PASSED TO THIS FUNCTION
//
//
//          5-----6          / +ve
//          /|              /|          /
//          / |              / |          /
//          / |              / |          /
//          1-----2          | \ / +ve
//          | |              | |
//          | 7-----8
//          | /              | /
//          | /              | /
//          | /              | /
//          3-----4
//
// NODE 1 = (-1,-1,-1)
// NODE 8 = (+1,+1,+1)

// r -> x direction
// s -> y direction
// t -> z direction

r = rst_location(1); s = rst_location(2); t = rst_location(3);
R = [1/8*(1-r)*(1-s)*(1-t), 1/8*(1+r)*(1-s)*(1-t), 1/8*(1-r)*(1+s)*(1-t), 1/8*(1+r)
    *(1+s)*(1-t), 1/8*(1-r)*(1-s)*(1+t), 1/8*(1+r)*(1-s)*(1+t), 1/8*(1-r)*(1+s)*(1+t)
    , 1/8*(1+r)*(1+s)*(1+t)];

result = R * vertex_values;

endfunction;

// -----

function [result] = interpolate_20node(rst_location,vertex_values);
// NOTE: vertex values must match up in order with rst values

```

// BELOW ARE THE LOCATIONS AND ORDER OF THE NODES TO BE PASSED TO THIS FUNCTION

```
//
//                                     / +ve (t)
//      13-----14-----15      /
//      /|                      /|  ———> +ve (r)
//      9 |                      10| |
//      / 16                      / 17 |
//      1-----2-----3   |   \ / +ve (s)
//      |   |                   |   |
//      | 18-----19-----20
//      4 /                      5 /
//      | 11                      | 12
//      | /                      | /
//      6-----7-----8
//
// NODE 01 = (-1,-1,-1)
// NODE 20 = (+1,+1,+1)
```

```
// r -> x direction
// s -> y direction
// t -> z direction
```

```
r = rst_location(1); s = rst_location(2); t = rst_location(3);
```

```
// Corner nodes:  $N_i = 1/8*(1+r_0)*(1+s_0)*(1+t_0)*(r_0+s_0+t_0-2)$ ;
```

```
// Mid side nodes:  $N_i = 1/4*(1-r^2)*(1+s_0)*(1+t_0)$ 
```

```
R = [1/8*(1-r)*(1-s)*(1-t)*(-r-s-t-2);
      1/4*(1-r^2)*(1-s)*(1-t);
      1/8*(1+r)*(1-s)*(1-t)*(r-s-t-2);
      1/4*(1-s^2)*(1-r)*(1-t);
      1/4*(1-s^2)*(1+r)*(1-t);
      1/8*(1-r)*(1+s)*(1-t)*(-r+s-t-2);
      1/4*(1-r^2)*(1+s)*(1-t);
      1/8*(1+r)*(1+s)*(1-t)*(r+s-t-2);
      1/4*(1-t^2)*(1-r)*(1-s);
      1/4*(1-t^2)*(1+r)*(1-s);
      1/4*(1-t^2)*(1-r)*(1+s);
      1/4*(1-t^2)*(1+r)*(1+s);
      1/8*(1-r)*(1-s)*(1+t)*(-r-s+t-2);
```

```

1/4*(1-r^2)*(1-s)*(1+t);
1/8*(1+r)*(1-s)*(1+t)*(r-s+t-2);
1/4*(1-s^2)*(1-r)*(1+t);
1/4*(1-s^2)*(1+r)*(1+t);
1/8*(1-r)*(1+s)*(1+t)*(-r+s+t-2);
1/4*(1-r^2)*(1+s)*(1+t);
1/8*(1+r)*(1+s)*(1+t)*(r+s+t-2)];

result = R' * vertex_values;

endfunction;

// -----

function [result] = interpolate_27node(rst_location, vertex_values);
// NOTE: vertex values must match up in order with rst values

// BELOW ARE THE LOCATIONS AND ORDER OF THE NODES TO BE PASSED TO THIS FUNCTION
//
//
//          19-----20-----21          / +ve (t)
//          /|          /|          /|          /
//          / |          / |          / |          --> +ve (r)
//          / |          / |          / |          |
//          10-----11-----12 |          \ / +ve (s)
//          /| |          / |          /| |
//          / | 22-----/---23-----/---24
//          1-----2-----3 | /|
//          | | / |          | / |          | | / |
//          | 13-----|---14-----|---15 |
//          | /| |          | / |          | /| |
//          | / | 25-----|/---26-----|/---27
//          4-----5-----6 | /
//          | | /          | /          | | /
//          | 16-----|---17-----|---18
//          | /          | /          | /
//          | /          | /          | /
//          7-----8-----9
//
//
// NODE 01 = (-1,-1,-1)
// NODE 20 = (+1,+1,+1)

```

```

// r -> x direction
// s -> y direction
// t -> z direction

r = rst_location(1);   s = rst_location(2);   t = rst_location(3);

// Corner nodes:   Nc = 1/8*r0*(r0-1)*s0*(s0-1)*t0*(t0-1);
// Mid edge nodes: Ne = -1/4*(r^2-1)*s0*(s0-1)*t0*(t0-1);
// Mid face nodes: Nf = 1/2*(r^2-1)*(s^2-1)*t0*(t0-1);
// centroid node:  NCentroid = -(r^2-1)*(s^2-1)*(t^2-1);

//Shape function are created by expansion of Lagrangian polynomials

R = [ 1/8*r*(r-1)*s*(s-1)*t*(t-1);           // Node (1) - Corner node
      -1/4*(r^2-1)*s*(s-1)*t*(t-1);          // Node (2) - Edge node
      1/8*r*(r+1)*s*(s-1)*t*(t-1);           // Node (3) - Corner node
      -1/4*r*(r-1)*(s^2-1)*t*(t-1);          // Node (4) - Edge node
      1/2*(r^2-1)*(s^2-1)*t*(t-1);           // Node (5) - Face node
      -1/4*r*(r+1)*(s^2-1)*t*(t-1);          // Node (6) - Edge node
      1/8*r*(r-1)*s*(s+1)*t*(t-1);           // Node (7) - Corner node
      -1/4*(r^2-1)*s*(s+1)*t*(t-1);          // Node (8) - Edge node ** N13
      1/8*r*(r+1)*s*(s+1)*t*(t-1);           // Node (9) - Corner node
      -1/4*r*(r-1)*s*(s-1)*(t^2-1);          // Node (10) - Edge node
      1/2*(r^2-1)*s*(s-1)*(t^2-1);           // Node (11) - Face node
      -1/4*r*(r+1)*s*(s-1)*(t^2-1);          // Node (12) - Edge node
      1/2*r*(r-1)*(s^2-1)*(t^2-1);           // Node (13) - Face node
      -(r^2-1)*(s^2-1)*(t^2-1);              // Node (14) - Centroid node
      1/2*r*(r+1)*(s^2-1)*(t^2-1);           // Node (15) - Face node
      -1/4*r*(r-1)*s*(s+1)*(t^2-1);          // Node (16) - Edge node
      1/2*(r^2-1)*s*(s+1)*(t^2-1);           // Node (17) - Face node
      -1/4*r*(r+1)*s*(s+1)*(t^2-1);          // Node (16) - Edge node
      1/8*r*(r-1)*s*(s-1)*t*(t+1);           // Node (19) - Corner node
      -1/4*(r^2-1)*s*(s-1)*t*(t+1);          // Node (20) - Edge node
      1/8*r*(r+1)*s*(s-1)*t*(t+1);           // Node (21) - Corner node
      -1/4*r*(r-1)*(s^2-1)*t*(t+1);          // Node (22) - Edge node
      1/2*(r^2-1)*(s^2-1)*t*(t+1);           // Node (23) - Face node
      -1/4*r*(r+1)*(s^2-1)*t*(t+1);          // Node (24) - Edge node
      1/8*r*(r-1)*s*(s+1)*t*(t+1);           // Node (25) - Corner node
      -1/4*(r^2-1)*s*(s+1)*t*(t+1);          // Node (26) - Edge node
      1/8*r*(r+1)*s*(s+1)*t*(t+1) ];         // Node (27) - Corner node

```

```

    result = R' * vertex_values;

endfunction;

function interpolate_domain();

    x = linspace(0.5*vertex_spacing + vertex_spacing, 0.5*vertex_spacing + num_cubes_xyz
        (1)*vertex_spacing, num_cubes_xyz(1));
    y = linspace(0.5*vertex_spacing + vertex_spacing, 0.5*vertex_spacing + num_cubes_xyz
        (2)*vertex_spacing, num_cubes_xyz(2));
    z = linspace(0.5*vertex_spacing + vertex_spacing, 0.5*vertex_spacing + num_cubes_xyz
        (3)*vertex_spacing, num_cubes_xyz(3)); // interpolation grid

    dual_0form_velocities;
    V = zeros(num_cubes_xyz(1), num_cubes_xyz(2), num_cubes_xyz(3));
    for i = 1 : num_cubes_xyz(1)
        for j = 1 : num_cubes_xyz(2)
            for k = 1 : num_cubes_xyz(3)
                c_num = Get_cube_number([i, j, k; i+1, j+1, k+1], num_vertex_x, num_vertex_y,
                    num_vertex_z, debug_output.level)
                V(i, j, k) = dual_0form_velocities(c_num, k)
            end;
        end;
    end;

    //V = f(X, Y, Z);
    t1 = splin3d(x, y, z, V);
    //test point to see if it worked
    vp_interp = interp3d(3*vertex_spacing, 3*vertex_spacing, 3*vertex_spacing, t1)

endfunction;

function [velocity_at_face_center] = interpolate_from_flux(given_primal_2form,
    vel_at_4_nodes, vertex_spacing);
    // -----
    // %%%%%%%%%%

```

```

// This function takes the volume_flow_rate at a face, nodal face velocities, and
// calculates the required velocity at the center face node
// This way we can define a continuous velocity field throughout the domain that
// matches the fluxes on each face
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
// INPUTS:
// - vol_flow_rate = in the direction normal to the face
// - vel_at_4_nodes = 4 scalars (in the direction of the face normal only!)
// - vertex_spacing = space between vertices (length and width of pyramid and
// rectangle)
// OUTPUTS:
// - velocity_at_face_center = scalar (in the direction of the face normal)
// %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

//V1-->|\
//      | \
//      |  \
//      |---> velocity_at_face_center
//      | /
//      | /
//V2 >|/

vol_flow_rate = given_primal_2form / material_density;

// First calculate the average "height" (and therefore volume) from the four nodes and
// then the associated volume for the rectangle (h x l x w)
h_node = 0.25 * sum(vel_at_4_nodes);
vol_square = h_node * vertex_spacing^2;

// Volume flow rate must equal v_node + v_pyramid --> solve for h_pyramid
// v_pyramid = 1/3(l * w * h)
h_pyramid = 3*(vol_flow_rate - vol_square) / (vertex_spacing^2);

// final velocity is the sum of the velocities (represented as heights)
// since this method will ALWAYS over estimate, use only 3/4 of calculated pyramid
// height
// the area of a parabola bounded from -1 to 1 with a height of 1 is 4/3
//// the height of a triangle with an area of 4/3 and width of 2 (from -1 to 1) is
// 4/3
//// therefore, 4/3 T_height = 1 P_height --> T_height = 3/4 P_height

```



```

velocity_at_face_center = h_node + (0.75 * h_pyramid);

endfunction;

```

D.6 Output Functions

```

function Export_mesh(path,file_mask);
// export the selected mesh to a file to be imported by tecplot
path = path + '.' + file_mask;

current_mesh_vertex = 0;
for z = 1:num_vertex_z
    for y = 1:num_vertex_y
        for x = 1:num_vertex_x

            current_mesh_vertex = current_mesh_vertex + 1;
            mesh_out(current_mesh_vertex,:) = [x,y,z];

            if z ~= num_vertex_z then
                current_mesh_vertex = current_mesh_vertex + 1;
                mesh_out(current_mesh_vertex,:) = [x,y,z+1];
                current_mesh_vertex = current_mesh_vertex + 1;
                mesh_out(current_mesh_vertex,:) = [x,y,z];
            end;

            if y ~= num_vertex_y then
                current_mesh_vertex = current_mesh_vertex + 1;
                mesh_out(current_mesh_vertex,:) = [x,y+1,z];
                current_mesh_vertex = current_mesh_vertex + 1;
                mesh_out(current_mesh_vertex,:) = [x,y,z];
            end;

        end; //for x = 1:num_vertex_x

    end; //for y = 1:num_vertex_y

end; //reset to the begining of next line (more efficient then doing everytime x
      increments)
current_mesh_vertex = current_mesh_vertex + 1;
mesh_out(current_mesh_vertex,:) = [x,y+1,z];

```

```

    end;

    end; //for y = 1:num_vertex_y

    if z ~= num_vertex_z then
        // reset to the begining of next plane (more efficient then doing everytime x
        // increments)
        current_mesh_vertex = current_mesh_vertex + 1;
        mesh_out(current_mesh_vertex,:) = [x,y,z+1];
        current_mesh_vertex = current_mesh_vertex + 1;
        mesh_out(current_mesh_vertex,:) = [x,1,z+1];
    end;

    end; // for z = 1:num_vertex_z

    // Resize the mesh with the proper vertex spacing
    mesh_out = mesh_out * vertex_spacing;

    // Number of rows in the matrix
    // Note, this is an N x M output
    I = size(mesh_out);
    I = I(1);
    // NOTE: char 10 = 'newline', char 34 = '"'
    //export mesh to file
    fprintfMat(path,mesh_out,"%f","VARIABLES=" + char(34) + "X" + char(34) + "," +
        char(34) + "Y" + char(34) + "," + char(34) + "Z" + char(34) + char(10) + char
        (10) + "ZONE_T=" + char(34) + "Mesh" + char(34) + "," + char(34) + "F=POINT," + string(I));

endfunction;

// -----

function Export_initial_particles(brick_values_27node,path,num_time_steps,
    time_step_size,flow_name);
    // divfree export is a modified export_timestep_to_file function as we do not need to
    // recalculate the position of the previous points (only the ones for the next time
    // step)
    matrix_out = start_particle_location;

```

```

for i = 1:num_time_steps
    // grab the last set of particles , and advect them in the div-free flow
    // replaced for loop with just the last particle (for j = 1:number_of_particles
    (1))
    number_of_particles = size(matrix_out,1);
    // allocate space for the new particles
    last_particle_set_start = number_of_particles - start_particle_number + 1;

    for j = last_particle_set_start:number_of_particles
        location = matrix_out(j,:);
        [x_flow,y_flow,z_flow] = find_value_at_xyz(location,brick_values_27node,
            vertex_spacing);
        velocities = [x_flow,y_flow,z_flow]; //Find_velocity([x_flow,y_flow,z_flow],
            material_density,vertex_spacing);
        // advect the point to it's new location
        matrix_out(j+start_particle_number,:) = matrix_out(j,:) + (velocities *
            time_step_size);
    end; // for j = last_particle_set_start:number_of_particles(1)

end; //for i = 1:num_time_steps

// Number of rows in the matrix
// Note, this is an N x M output
I = size(matrix_out,1);
// NOTE: char 10 = 'newline', char 34 = '"'
fprintfMat(path,matrix_out,"%f", "VARIABLES=_"+char(34)+"X"+char(34)+",_"+
    char(34)+"Y"+char(34)+",_"+char(34)+"Z"+char(34)+char(10)+char
    (10)+"ZONE_T="+char(34)+flow_name+char(34)+",_F=POINT,_I="+string(I))
;

endfunction;

// _____

function Export_vorticity_to_file(dual_2form,vertex_mesh,edge_mesh,vertex_spacing,
    path_yz,path_xz,path_xy,time_step_number);
// export the selected mesh to a file to be imported by tecplot

```

```

yz_counter = 0;
xz_counter = 0;
xy_counter = 0;
for e = 1:size(edge_mesh,1);
    vertices_of_edge = unique(edge_mesh(e,:));
    v_loc = (vertex_mesh(vertices_of_edge(:),:));

    if (v_loc(1,1) ~= v_loc(2,1)) // x1 ~= x2 —> x direction edge holds vorticity
        values for y-z plane
        yz_counter = yz_counter + 1;
        mesh_out_yz(yz_counter,1:3) = ((v_loc(1,:) + v_loc(2,:)) / 2) * vertex_spacing;
        mesh_out_yz(yz_counter,4) = dual_2form(e);
    elseif (v_loc(1,2) ~= v_loc(2,2)) // y1 ~= y2 —> y direction edge holds vorticity
        values for x-z plane
        xz_counter = xz_counter + 1;
        mesh_out_xz(xz_counter,1:3) = ((v_loc(1,:) + v_loc(2,:)) / 2) * vertex_spacing;
        mesh_out_xz(xz_counter,4) = dual_2form(e);
    elseif (v_loc(1,3) ~= v_loc(2,3)) // z1 ~= z2 —> z direction edge holds vorticity
        values for x-y plane
        xy_counter = xy_counter + 1;
        mesh_out_xy(xy_counter,1:3) = ((v_loc(1,:) + v_loc(2,:)) / 2) * vertex_spacing;
        mesh_out_xy(xy_counter,4) = dual_2form(e);
    end;
end;

I_yz = num_vertex_x - 1;
J_yz = num_vertex_y;
K_yz = num_vertex_z;

I_xz = num_vertex_x;
J_xz = num_vertex_y - 1;
K_xz = num_vertex_z;

I_xy = num_vertex_x;
J_xy = num_vertex_y;
K_xy = num_vertex_z - 1;

// NOTE: char 10 = 'newline', char 34 = '"'
//export mesh to file

```

```

fprintfMat(path_yz, mesh_out_yz, "%f", "VARIABLES_=" + char(34) + "X" + char(34) + ",_
" + char(34) + "Y" + char(34) + ",_" + char(34) + "Z" + char(34) + ",_" + char
(34) + "Vorticity_in_YZ_plane" + char(34) + char(10) + char(10) + "ZONE_T=" +
char(34) + "Vorticity_YZ_@_t=" + string(time_step_number*time_step_size) + char
(34) + ",_F=POINT,_I=" + string(I_yz) + ",_J=" + string(J_yz) + ",_K=" + string(
K_yz));

fprintfMat(path_xz, mesh_out_xz, "%f", "VARIABLES_=" + char(34) + "X" + char(34) + ",_
" + char(34) + "Y" + char(34) + ",_" + char(34) + "Z" + char(34) + ",_" + char
(34) + "Vorticity_in_XZ_plane" + char(34) + char(10) + char(10) + "ZONE_T=" +
char(34) + "Vorticity_XZ_@_t=" + string(time_step_number*time_step_size) + char
(34) + ",_F=POINT,_I=" + string(I_xz) + ",_J=" + string(J_xz) + ",_K=" + string(
K_xz));

fprintfMat(path_xy, mesh_out_xy, "%f", "VARIABLES_=" + char(34) + "X" + char(34) + ",_
" + char(34) + "Y" + char(34) + ",_" + char(34) + "Z" + char(34) + ",_" + char
(34) + "Vorticity_in_XY_plane" + char(34) + char(10) + char(10) + "ZONE_T=" +
char(34) + "Vorticity_XY_@_t=" + string(time_step_number*time_step_size) + char
(34) + ",_F=POINT,_I=" + string(I_xy) + ",_J=" + string(J_xy) + ",_K=" + string(
K_xy));

endfunction;

// -----

function Export_velocity_to_file(brick_values_27node, vertex_mesh_27node,
    vertex_spacing, path, time_step_number);
// export the selected mesh to a file to be imported by tecplot

mesh_out = [vertex_mesh_27node*vertex_spacing, brick_values_27node]
//disp(mesh_out);

I = num_vertex_x_27node;
J = num_vertex_y_27node;
K = num_vertex_z_27node;

// NOTE: char 10 = 'newline', char 34 = '"'
//export mesh to file

```

```

fprintfMat(path,mesh_out,"%f", "VARIABLES_=" + char(34) + "X" + char(34) + "," +
char(34) + "Y" + char(34) + "," + char(34) + "Z" + char(34) + "," + char(34) +
"Velocity_X" + char(34) + "," + char(34) + "Velocity_Y" + char(34) + "," + char
(34) + "Velocity_Z" + char(34) + char(10) + char(10) + "ZONE_T=" + char(34) + "
Velocity_@_t=" + string(time_step_number*time_step_size) + char(34) + ",_F=POINT,
_I=" + string(I) + ",_J=" + string(J) + ",_K=" + string(K));

endfunction;

// -----

function matrix_out = Export_particles_to_file(matrix_out,brick_values_27node,path,
time_step_number,time_step_size,timestep_out_divider);

    particle_timestep = round(time_step_number / timestep_out_divider);
    // Add new particles on input face
    // Putting in a loop to allow for changing particle number easily
    for new_particle=1:start_particle_number
        matrix_out(start_particle_number*(particle_timestep-1) + new_particle,:) =
            start_particle_location(new_particle,:);
    end;

    // grab each particle and find which cube it's in
    number_of_particles = size(matrix_out,1);

    for j = 1:number_of_particles

        // matrix_out(j,:) is the location of the particle
        [x_flow,y_flow,z_flow] = find_value_at_xyz(matrix_out(j,:),brick_values_27node
            ,vertex_spacing);
        velocities = [x_flow,y_flow,z_flow]; //Find_velocity([x_flow,y_flow,z_flow],
            material_density,vertex_spacing);
        // advect the point to it's new location (in x,y and z)
        matrix_out(j,:) = matrix_out(j,:) + (velocities * time_step_size *
            timestep_out_divider);

    end;

```

```

// Number of rows in the matrix
// Note, this is an N x M output so we want to take the N value (number of rows)
// I = size(matrix_out,1);
// NOTE: char 10 = 'newline', char 34 = ''

fprintfMat(path,matrix_out,"%f","VARIABLES_=_"+char(34)+"X"+char(34)+",_"+
char(34)+"Y"+char(34)+",_"+char(34)+"Z"+char(34)+char(10)+char
(10)+"ZONE_T="+char(34)+"Particles_@_t="+string(time_step_number*
time_step_size)+char(34)+",_F=POINT,_I="+string(number_of_particles));

endfunction;

```