Department of Mechanical and Aerospace Engineering

University of Strathclyde

Development and validation of a novel hybrid CFD slurry model

Alasdair Mackenzie

A thesis presented for the degree of

Doctor of Philosophy

2019

Declaration of author's rights

This thesis is the result of the author's original research. It has been composed by the author and has not been previously submitted for examination which has led to the award of a degree.

The copyright of this thesis belongs to the author under the terms of the United Kingdom Copyright Acts as qualified by University of Strathclyde Regulation 3.50. Due acknowledgement must always be made of the use of any material contained in, or derived from, this thesis.

Alasdair Mackenzie

April 2019

Abstract

Erosion from slurry flows is a large problem in the mining and oil and gas industries, hence the interest from Weir group. Centrifugal slurry pumps manufactured by the Weir group are considered to be some of the world's best, however the harsh environments they operate in can erode hardened steel parts in a matter of weeks. One way to mitigate the damage caused to the parts is better design. This is achieved by simulating the flow with computational fluid dynamics (CFD), using a two phase model: one phase for the fluid, one phase for the particles. The two phase models can, broadly speaking, be separated into two categories: Euler-Euler (EE), and Euler-Lagrange (EL). EE models the fluid and particles as two continua, is fast to run, but lacks particle impact data which is required for erosion modelling. EL models a fluid phase and a particulate phase, is more computationally expensive than the EE model, but has the benefit of having particle impacts at the wall.

A hybrid slurry model was developed combining both EE and EL models, taking advantage of each model's strengths. The hybrid model proved that a combined EE/EL solver was capable of modelling slurry flows in less time than a pure EL model, but with more particle impact data at the wall than a EE model. Issues with combining two computational models in one domain were identified and overcome, buy including a transition baffle, where the solver changed from EE to EL.

Experiments were carried out to validate the computational simulation results on a submerged jet impingement test. Particle image velocimetry was carried out to obtain data from the experiments, and images were correlated to output particle velocity data.

Acknowledgements

I would like to express my gratitude to my primary and secondary supervisors for their help throughout my years at the University of Strathclyde; Dr Matthew Stickland and Dr Bill Dempster. My primary supervisor, Dr Stickland, was my fourth year undergraduate project supervisor and the opportunity to continue under his guidance was one advantage for continuing to do a PhD. Dr Stickland was not only a great source of help and knowledge with regards to my project, but was also very supportive of my aviation related activities inside and outside of university.

I am thankful to the Weir Group for taking me on as a PhD student in the Weir Advanced Research Centre and supporting me financially throughout the 3 years. This enabled me to go on more courses and conferences than the average PhD student. I must thank Alan Bickley, the office director, for allowing me to take a 3 month break from my PhD to obtain my pilot's license: something I have always wanted to do. I would also like to thank those in the office who helped me throughout my PhD, particularly Drs. Alejandro Lopez and Konstantinos Ritos for their help with OpenFOAM and programming during the first year, and Anthony Kinsella for keeping the office fun.

Professor Hrvoje Jasak and his student Vanja Skuric were also a great help to me in my third year, as they kindly invited me out to Croatia for assistance with my programming issues. Without their help I might still be trying to figure out what to do.

Finally, I must thank all those who proofread for me. If there are any grammar mistakes: I'll blame them.

Nomenclature

Equation Variables

α_p Volume Fraction of Particulate Phase

α_l Volume Fraction of Liquid Phase

α_s Volume Fraction of Solid Phase

μ_f Fluid Viscosity

ρ Density

ρ_p Particle Density

τ_p Characteristic Time

A_{cell} Area of Cell Face

C_d Coefficient of Drag

D Pipe Diameter

D_p Particle Diameter

f Coefficient of Viscosity

$f(\gamma)$ Function of Impact Angle

F_B Buoyancy Force

F_D Drag Force

F_G Gravity Force

g Gravitational Constant

k Particle Impact Rate

m_p Particle Mass

nP Number of Particles

St Stokes number

U Average Velocity

u_f Fluid Velocity

u_p Particle Velocity

V_p^n Particle Impact Velocity

V_{normal} Normal Velocity Component

Vol_p Volume of Particle

Abbreviations

CFD Computational Fluid Dynamics

DBM Discrete Bubble Model

DDM Discrete Droplet Model

DDPM Dense Discrete Phase Model

EE Euler-Euler

EL Euler-Lagrange

GUI Graphical User Interface

JIT Jet Impingement Test

LDV	Laser Doppler Velocimetry
LPT	Lagrangian Particle Tracking
PIV	Particle Image Velocimetry
PTV	Particle Tracking Velocimetry
RANS	Reynolds-Averaged Navier-Stokes
RCM	Region Coupling Method
SG	Specific Gravity
SOI	Start Of Injection
UDF	User Defined Function
VOF	Volume-Of-Fluid

Contents

1 Introduction **1**

1.1 Motivation and background 1

1.2 Multiphase flows 3

1.3 Eulerian/Lagrangian modelling approaches 3

1.4 Overview of thesis 7

2 Literature Review **9**

2.1 Erosion caused by slurry 9

2.1.1 Stokes number 11

2.2 CFD modelling of erosion 14

2.2.1 Popular approaches of erosion modelling 15

2.3 Hybrid models 16

2.4 Experimental methods for modelling slurry erosion 21

2.4.1 Jet Impingement Test 21

2.5 Conclusion 23

3 Solver development **25**

3.1 Choosing software and models 25

3.1.1 Euler-Euler Model 26

3.1.2 Euler-Lagrange Model 27

3.2 Creating the new Hybrid model 28

3.2.1 Geometry setup: Mesh/Baffles/Regions 28
3.2.2 Solver creation and interpolation 30
3.2.3 Addition of particles to solver 32
3.3 New injection model 35
3.3.1 Modifications to injection model 36
3.3.2 Modifications to KinematicLookupTableInjection.C 38
3.3.3 Modifications to solver 41
3.4 Discussion 43
3.4.1 Baffle position 43
3.4.2 Injection method 44
3.4.3 Recent developments in OpenFOAM 45

4 Experimental equipment 47

4.1 Introduction 47
4.1.1 Particle Image Velocimetry 48
4.2 Design of rig 51
4.3 Samples and sample holder 57
4.4 Hardware and software used 58
4.4.1 Hardware 58
4.4.2 Software 59

5 Results and validation of CFD model 64

5.1 First phase comparison of standard models (Jet Impingement Test) 64
5.2 Results from the Hybrid Model Tutorial (Pipe bend) 67
5.3 First phase comparison with CFD: Hybrid model (Jet Impingement Test) 71
5.3.1 Cone results 75
5.3.2 Hemisphere results 75
5.3.3 Cylinder results 78
5.4 Second phase comparison with CFD- Hybrid model (Jet Impingement Test) . . . 82

5.4.1 Time to solve comparison 82
5.4.2 Particle impact comparison 83
5.4.3 Visual comparison 85
5.5 Comparison of CFD and experimental work 87

6 Conclusions and further work **95**

References **99**

Appendices **108**

A Tutorial on the development of the Hybrid Model, published at Chalmers University **109**

B Figures of experimental rig components **155**

C Stitched particle images **159**

D Modified files from solver **162**

E PatchPostProcessing.C **205**

List of Figures

1.1 Author's images of impellers from a gold mine in Australia (2015), showing wear that can occur in as little as two weeks. 2

1.2 Separated/stratified flow on left, dispersed multiphase flow on right. [1] 3

1.3 Erosion modelling pyramid. 8

2.1 Effects of a turbulent eddy (solid line) on particle trajectory (dashed line) for different Stokes-number limits [1]. 11

2.2 Stokes number illustration [2]. 13

2.3 Erosion degree for various Stokes numbers. Fluid direction is from the horizontal section of the pipe to the vertical [3]. 14

2.4 Figures from Messa *et al.* [4] 17

2.5 Intersection of control volumes between spray and engine mesh by Vujanovic *et al.* 18

2.6 Transition from spray to engine by Vujanovic *et al.* 19

2.7 RCM shown in between VOF and LPT modelling of liquid diesel spray by Yu *et al.* 20

2.8 Submerged JIT showing velocity vectors obtained by PIV (taken by author). . . . 22

2.9 Figure from Mansouri's [5] experimental test, showing particles and nozzle. . . . 23

2.10 PIV setup for submerged jet impingement tests from Mansouri's paper [5]. 24

3.1 Wireframe geometry showing baffle in red. 29

3.2 Depiction of iterative loop. 31

3.3 Velocity contour plot of Hybrid model showing cells with same values either side of baffle (red writing). 33
3.4 Depiction of transition between two models at the baffle. P is particulate phase, and F is the fluid phase. 37
3.5 Screenshot of Meld- showing comparison of edited and original InjectionModel.C. 41
3.6 Boundary issue. 45

4.1 A 2D schematic of the initial test rig 48
4.2 An example of an experimental setup for PIV in a wind tunnel [6]. 49
4.3 Frame straddling technique [7] (time along x-axis). 51
4.4 Detailed drawing of ejector pump, flow left to right, showing particle feed coming from the top [8]. 52
4.5 Test rig: view from left. 53
4.6 Test rig: view from right. 54
4.7 Velocity contours (m/s) of original slurry pipe location showing asymmetric impingement velocity profile. 55
4.8 Pipe bend drawing showing steel slurry pipe insert. 56
4.9 Test rig: nozzle and new sample holder. 60
4.10 Samples manufactured for testing (sizes in mm). 61
4.11 Shadow created by cone. 61
4.12 Original sample holder base. 62
4.13 New sample holder assembled. 62
4.14 Basic PIV setup [9]. 62
4.15 Frac sand size distribution [10]. 63

5.1 Averaged vector plot from PIV software: Flow Manager. 65
5.2 Radial velocity 1mm above surface. 66
5.3 Axial velocity 1mm above surface. 66
5.4 Velocity magnitude 1mm above surface of cylinder on JIT from CSIRO conference. 67

5.5 Hybrid model showing 2D slice of 2nd Eulerian phase velocity contours, and Lagrangian particles in second region: both coloured by velocity magnitude. . . . 69

5.6 EL particle impacts. 70

5.7 Hybrid model particle impacts. 70

5.8 Cartesian mesh of cone, showing regions one and two in blue and green respectively. 72

5.9 Velocity (m/s) contours of all three geometries (slight discontinuity shows baffle location). 73

5.10 Left image shows baffle location and two white sample lines on CFD velocity magnitude contours. Right image shows same sample lines in blue on the velocity vector results from PIV data. 74

5.11 Comparison of experimental and computational results displaying contours of vertical (top image) and horizontal (bottom image) velocity components (experimental results shown inside dashed squares). Data is non-dimensional. 76

5.12 Cone results: 5mm below nozzle exit velocity profiles. 77

5.13 Cone results: 9.5mm below nozzle exit velocity profiles. 77

5.14 Comparison of experimental (left) and computational (right) results displaying contours of vertical (top image) and horizontal (bottom image) velocity components: Vmax is the maximum centreline vertical velocity component. Data is non-dimensional. 78

5.15 Hemisphere results: 5mm below nozzle exit velocity profiles. 79

5.16 Hemisphere results: 9.5mm below nozzle exit velocity profiles. 79

5.17 Comparison of experimental (left) and computational (right) results displaying contours of vertical (top image) and horizontal (bottom image) velocity components: Vmax is the maximum centreline vertical velocity component. Data is non-dimensional. 80

5.18 Cylinder results: 5mm below nozzle exit velocity profiles. 81

5.19 Cylinder results: 5mm above sample velocity profiles. 81

5.20 Comparison of particle positions 0.028 seconds after injection. Particles coloured by velocity magnitude, with quarter model shown of the fluid domains coloured by fluid velocity magnitude. 84
5.21 Cylinder impingement surface divided into 1mm bins [11]. 85
5.22 Particle impact velocities on cylinder sample. 85
5.23 Particles simulation on cone sample, showing impingement surface and omitting fluid: Hybrid model on left, EL model on right. (EL particles do not seem to bounce as much as the Hybrid model particles, however the same boundary conditions were used in both cases.) 86
5.24 Particles simulation on cone sample, showing impingement surface coloured by pressure : Hybrid model on left, EL model on right. 88
5.25 Particles simulation on cylinder sample, showing impingement surface coloured by pressure : Hybrid model on left, EL model on right. 88
5.26 Particles simulation on hemisphere sample, showing impingement surface coloured by pressure : Hybrid model on left, EL model on right. 89
5.27 Single image from set of 500 showing one particle track over 1/500th of a second, with arrow showing distance travelled. 90
5.28 Plan view of sample coloured in blue, with green dashed line illustrating area where particles will be analysed by laser. 91
5.29 Comparison of particle tracks on cylindrical shaped sample. 93
5.30 Comparison of particle tracks on conical shaped sample. 93
5.31 Comparison of particle tracks on hemisphere shaped sample. 94

6.1 Green laser light shining through tank onto cylinder sample 96

List of Tables

1.1 Comparison of EE and EL solvers. 4
1.2 Types of coupling 7
2.1 Summary of liquid-based JIT erosion testing [2]. 12
3.1 Fields in each solver. 28
5.1 Comparison of execution times from 0-0.39 seconds with different mass concentrations (MC). 68
5.2 Comparison of execution times on the pipe bend. 82
5.3 Comparison of execution times on the cone shaped sample. 82

Chapter 1

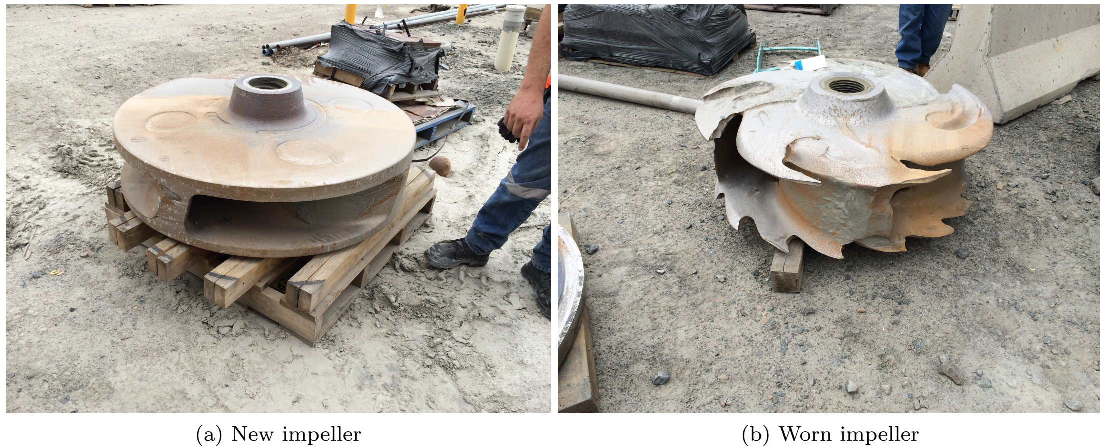
Introduction

> Time flies like an arrow; fruit flies like a banana.
>
> Author unknown

1.1 Motivation and background

The Weir Group [12] is one of the world's largest producers of equipment for the mining, and the oil and gas industries. The Weir Advanced Research Centre was established in 2011 in collaboration with the University of Strathclyde, Glasgow to undertake research on the company's products and areas of interest.

One main area of research interest has been in the erosion in centrifugal slurry pumps. One such area of focus is in a gold/copper processing circuit where the ore is initially transported dry on conveyor systems. After the ore has been graded into different diameters and put through the mill to break it down into smaller pieces, water is introduced to create a slurry which is capable of being pumped. Slurries can vary in characteristics a lot, from 2% to 40% mass concentration. This slurry is then pumped into hydro-cyclones where the separation process of the minerals from the rock begins [13] (the video referenced shows the process in detail). The components in these slurry pumps are subject to great abuse and impellers can be completely worn out in as little as two weeks of continuous use (see Figure 1.1b).



(a) New impeller (b) Worn impeller

Figure 1.1: Author's images of impellers from a gold mine in Australia (2015), showing wear that can occur in as little as two weeks.

In addition to erosion causing a waste of energy and materials, it also impacts on work efficiency and leads to downtime in that the section of the mine being refitted cannot produce any revenue whilst the pump is out of order. Energy savings are important for businesses and the environment; this is evident as for the first time in more than a decade, energy use in the mining sector in Australia (2014-15) fell, despite general growth of output [14]. This is due to the fact that engineers have made an effort to increase the efficiency of the process. Mining still accounts for 8.8 % of Australia's total energy consumption, and so increasing the efficiency of the slurry pumps which run continuously by a small fraction could be a significant saving.

The overall thesis is that if the pump designer has the ability to model erosion by using computational fluid dynamics (CFD), then they will be able to change the design of the pump to minimise material loss over time. Improving the efficiency of the erosion modelling CFD code, will enable more simulations to be done in a fixed unit of time. This will affect the whole industry and their current design process, as they will be able to simulate more pumps per week, which will allow more design iterations, and thus allow development to progress at a faster rate. This, in combination with materials research, will mean components will last for longer and run more efficiently: saving time, energy and money. This should exemplify the importance of improving the efficiency of CFD erosion modelling code.

1.2 Multiphase flows

Multiphase flows are arguably the most common type of flow found on the planet: ‘multiphase’ meaning more than one phase (liquid-solid in this thesis). There are many examples of multiphase flows in the earth: in nature there is weather such as rain and snow, in human bodies there is digestive systems and blood, and in engineering there is coal furnaces and slurry pumps. Flows can either be described as separated or dispersed, as shown in Figure 1.2.

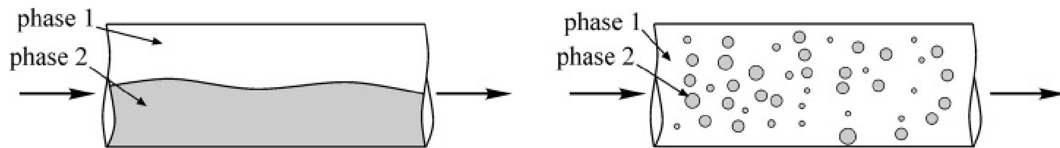


Figure 1.2: Separated/stratified flow on left, dispersed multiphase flow on right. [1]

It is dispersed flows that will be considered here, with a particular interest in densely dispersed liquid-solid flows (slurries), as this is what is found in slurry pumps. There is a large variation in particle size and density depending on the application, but the ranges typically witnessed in a slurry pump are from 30-50% concentration (by mass) and particles sizes from 150-800 μm in diameter. Although the new CFD model was developed in this thesis with slurry pumps as the main application, the hybrid model is applicable to other areas with some minor modifications. For example, the model was developed on a submerged jet impingement test geometry: which has similarities with other injection systems like the diesel injection system for an internal combustion engine. The diesel is injected as a liquid into the air filed chamber, and then breaks up into smaller droplets suitable for combustion. This is a similar process to the modelling in this thesis, although particles are modelled, not diesel.

1.3 Eulerian/Lagrangian modelling approaches

CFD modelling requires utilising either the Eulerian or the Lagrangian reference frame. Both describe different ways of looking at/modelling the flow, which can be illustrated by picturing the flow of a river: the Eulerian frame can be understood by visualising sitting on the river bank

and watching the river flow past a fixed location, whereas the Lagrangian can be visualised by sitting on a boat floating down the river, and looking at the flow next to the boat.

This quotation from the in-depth book by Crowe *et al.* states: '*With the discrete element and discrete parcel methods, individual particles or parcels of particles are tracked through the field and the local properties of the cloud are determined by the properties of the particle or parcel as they pass the point in the field. This is the Lagrangian approach. The two-fluid approach, in which a set of algebraic conservation equations are solved simultaneously for each node in the field, is the Eulerian approach.*' [15]. In application to multiphase flows the continuous phase (fluid) is normally in the Eulerian frame, and the dispersed phase (solid) is normally in the Lagrangian: a slurry model of this type would be named Euler-Lagrange (EL).

However, although the Lagrangian approach is normally used for particles, the Eulerian can also be used to model particles. In densely packed flows the particle collisions happen so frequently that particles can behave like a fluid (like moving sand on a riverbed), enabling the Euler-Euler (EE) method to be used. Both the EL and the EE methods of multiphase modelling will be used in this thesis as both have their advantages and disadvantages, as detailed in Table 1.1 below.

Euler-Euler	**Euler-Lagrange**
Two fluid model	Fluid/particle model
Computationally efficient	Computationally expensive
Averaged cell value at the wall	Particle/parcel impact data at wall

Table 1.1: Comparison of EE and EL solvers.

The main disadvantages of EE are that it is an averaged solution, and also that there are no particle data at the wall. Fluid values are calculated at the cell centres, meaning that the accuracy of the particle velocity vectors at the wall depend on the mesh density at the wall. If a cell near the wall is 4mm x 4mm, then the velocity vector for the particulate phase will be 2mm from the surface of the wall. The closest velocity vectors will be the adjacent cells, which will be at minimum 4mm away. The accuracy of particle erosion modelling depends on the

amount of particle impacts/unit area: therefore modelling it in EE renders it mesh dependant. This is not true regarding Lagrangian particles, as their velocity values can be different from the cell centre value.

The main downside with EL is its large computational effort required to simulate high numbers of particles. As the number of particles in EL simulations increase, so do the solution times. This is because there are more equations to solve, since there are a predetermined number of equations for each particle; whereas EE treats both phases as continua, so there is no increase in computational effort with increased particle density. One way of mitigating this problem is to solve for 'parcels' of particles, thus reducing the number of equations to solve, and therefore computational time. Parcels will be further discussed in Chapter 2.

Summary of approaches

EE: Both phases are treated as inter-penetrating continuum, the dispersed phase is averaged over the control volume, each phase is governed by a similar set of conservation equations. Modelling is required for interaction between the phases, turbulent dispersion of particles, and collisions between the particles and the walls.

$$\alpha_l + \alpha_s = \alpha_1 + \alpha_2 = 1 \tag{1.1}$$

Equation 1.1 shows the phase fraction equation for a mixture of liquids (α_l) and solids (α_s). The equation shows the assumption that must be made for EE solvers, that the liquid and solid volume fraction must sum to equal 1 in each cell. These volume fractions are in each of the Eulerian equations. EE models solve the mass continuity equations, the momentum equations and the energy equation for each phase. If the volume fraction increases, only the α value changes, and the same number of equations exist. This means the solver is largely independent of volume fraction, which is the opposite of the EL solving technique.

More details of the EE solver used can be found in the Chalmers University tutorial by Manni [16].

EL: The fluid flow is found by solving the RANS (Reynolds-averaged Navier-Stokes) equations with a turbulence model. The dispersed phase is simulated by tracking a large number of representative particles/parcels. Modelling is required for collisions between particles and the walls, and particle/particle collisions.

$$m_p \frac{du_p}{dt} = \sum F = F_D, F_G, F_B, +... \tag{1.2}$$

Newton's second law of motion (Equation 1.2) describes each of the forces acting on a particle; e.g drag (F_D), gravity (F_G), buoyancy (F_B) etc [17]. Each of these forces has to be accounted for, for each parcel of particles. Drag force (Equation 1.3) dominates the motion of the parcels as it can account for 80% of the total, therefore the drag force is a controlling factor:

$$F_D = C_d \frac{\pi D_p^2}{8} \rho f (u_f - u_p)|u_f - u_p| \tag{1.3}$$

where C_d is the coefficient of drag, D_p is the diameter of the particle, ρ is the density of the fluid, f is the coefficient of viscosity, u_f is the velocity of the fluid, and u_p is the velocity of the particle [18]. Each other force considered will need its own equation, which will increase computational time linearly.

EE-EL Comparison: The equations above which must be solved for each parcel of particles are what slow EL solvers down, compared to EE counterparts. The more parcels the EL solver has, the more equations it must solve, and so the longer it will take to solve. This is not the case with EE, as it is based on phase fractions.

These key differences are why a hybrid model was to be created: to keep the accuracy at the wall of Lagrangian particles, but to exploit the runtime of the more computationally efficient Eulerian approach.

Coupling between the phases is also an important factor in multiphase flows, as this has to be replicated in simulations. The type of coupling describes how the phases affect each other, and to what degree, as seen in Table 1.2. When modelling a multiphase flow with CFD a decision has to be made on what type of coupling will be used, usually based upon the concentration of

particles or the dispersed phase volume fraction.

Coupling	Description
One-way	Continuous phase affects dispersed phase only
Two-way	As one-way, but continuous phase is affected by dispersed phase
Four-way	As two-way, but dispersed particles also interact with each other

Table 1.2: Types of coupling

1.4 Overview of thesis

This thesis is the collection of 3.5 years' work. A new hybrid CFD model was developed and validated with a test rig which was built for purpose.

The main aim of the thesis was to enable better erosion modelling capabilities by improving the computational efficiency of the whole process. This was done by addressing the fluid/particle flow aspects of the problem, and not the surface removal etc. Erosion modelling is at the top of the pyramid as it relies upon the lower three sections: particle impact data/erosion equations, particle flow modelling, and fluid flow modelling (see Figure 1.3). Before a geometry-independent erosion model may be created, these three lower sections need to be accurate and reliable. Since the particle flow modelling is the most time consuming part of the process, the aim was to reduce the computational effort required there. If this step is made more efficient, it will directly improve the efficiency of the whole erosion process. It should be noted that if geometry changes are added as effects of the erosion modelling, then this pyramid structure will turn into a loop since the geometry changes will affect the fluid flow which will affect the particle flow and so on.

Developing a new computational model on a commercial CFD package has a number of pitfalls, with the biggest one being the lack of access to the underlying code. ANSYS Fluent® for example, one of the largest commercial CFD programs, does not allow the user access to the code but instead allows the user to create user defined functions (UDF) to modify existing solvers at runtime. The lack of access is due to the program having commercial financial

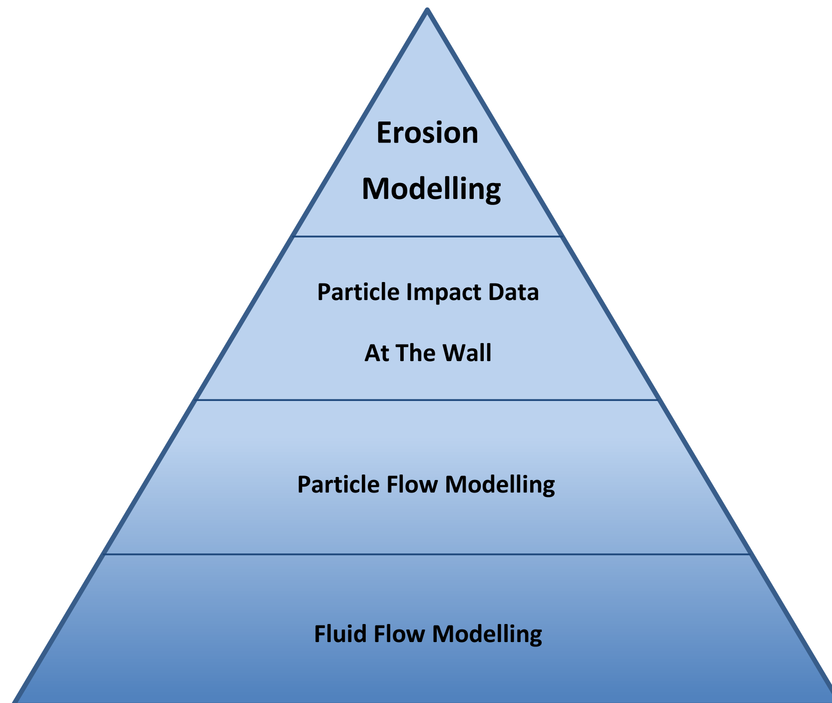


Figure 1.3: Erosion modelling pyramid.

interests. Open source programs do not have this issue, and are by definition, open to modify and edit as the user pleases. OpenFOAM [19] is the most popular and robust open source CFD code currently available [20], and so it was chosen for development purposes. OpenFOAM 3.0.x (the foundation version) was used, as it was the most recent version at the time of commencing the PhD. Before development was started ANSYS Fluent® and StarCCM+® were used to model a simple jet (Chapter 5); however the vast majority of software used throughout the 3 years was open source.

Chapter 2

Literature Review

> Give me six hours to chop down a tree
>
> and I will spend the first four
>
> sharpening the axe.
>
> — Abraham Lincoln

The industries affected by slurry erosion have been aware of the problem for a considerable period of time. One of the first and most cited academic research papers published on the subject was by Finnie [21] in 1960, which set the ball rolling for more detailed research into the field. He realised that a sound theoretical knowledge of the fundamental processes of how material was removed by erodents was lacking. The following sections will give a brief overview of the work done in the areas which are relevant to this thesis. It should be noted that papers often bridge more than one of these headings.

2.1 Erosion caused by slurry

The bulk of research papers published on slurry erosion are concerned with material properties and their effects on erosion rate. Although CFD is the focus of this thesis, the knowledge contained in their papers can help better understand the issues with erosion.

3D CFD simulations were in their infancy when erosion papers started to be published, therefore the papers of that era did not contain many simulations. The papers were primarily

focused on gaining a better understanding of the method of erosion: for example considering the factors that affect erosion rate: angle of impingement, particle velocity, particle shape, concentration of particles, surface shape, surface properties etc. [22, 23, 24]. All of these factors were found to have an effect on the erosion rates with the two most significant being particle impact velocity and impact angle. This is evident in the fact that most erosion equations are based upon these two factors, as in Equation 2.1 from Kim *et al.* [25] (which is based on Finnie's)

$$E = kV_p^n f(\gamma) \tag{2.1}$$

Where E is the dimensionless mass removed, k is particle impact rate, V_p^n is particle impact velocity, and $f(\gamma)$ is a dimensionless function of impact angle. The value of n, the powered term for velocity, is usually between two and three [26] and can therefore be one of the most influencing parameters for erosion rate. The value of n can be ascertained empirically.

It is then evident that having accurate particle data at the wall is necessary for accurate erosion modelling, since the magnitude of erosion depends so heavily on impact velocity and angle.

There were two papers published by Clark [27, 28], one in 1992 the other in 2004, looking into the phenomenon of the so called 'squeeze film effect'. The effect is observed when a thin film of fluid creates a barrier in-between the wall and incoming solid objects. Clark found that particle impact velocities were sometimes 50% lower than expected which would translate to a reduction of erosion rate by 75% (if $n = 2$). This energy absorbing layer not only has implications for mass removal calculations but also for rebound values. Most CFD codes have rebound coefficients for walls and so the 'squeeze film effect' should be kept in mind. Clark also mentions that the particle shape had a big influence, as sharp edged particles could cut through the film.

2.1.1 Stokes number

The Stokes number describes an important phenomenon to consider. It determines how closely coupled the particles are to the flow: if the Stokes number is near zero, the particles will tend to follow the flow, whereas if the number is over one, the particle tracks will start to deviate from the fluid path lines (dominated by their inertia). This is shown in Figure 2.1.

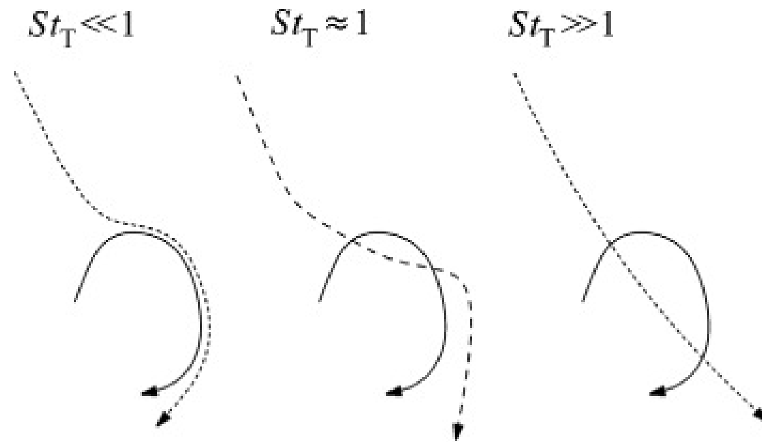


Figure 2.1: Effects of a turbulent eddy (solid line) on particle trajectory (dashed line) for different Stokes-number limits [1].

This demonstrates how important the Stokes number is when modelling liquid/particulate flows, as it cannot be assumed that the particles always follow the fluid flow (which is what EE models can do). It is these differences between the EE and EL models which makes creating a hybrid model difficult and challenging. The Stokes number and how it relates to CFD is discussed further in Chapter 3.

The paper by Frosell *et al.* [2] investigates the effect slurry concentration has on solid particle erosion for jet impingement tests. Table 2.1 is taken from the paper and shows a summary of some of the most recent erosion papers which use the JIT, where 'H' is the height of the nozzle above the sample, 'D' is the diameter of the pipe, and Vol% is the volume fraction of the slurry. It should be noted how the Stokes number varies largely within the papers reviewed, and how this might affect the erosion process. The Stokes number is defined in Equation 2.2 and the characteristic time (τ_p) in Equation 2.3.

Author	H (mm)	D (mm)	H/D	U (m/s)	d_p (μm)	Vol%	St
Giourntas	5	3	1.7	24	550	0.01	356
Mansouri	12.7	7	1.8	14	300	0.4	27
Mansouri	12.7	8	1.6	14	150	0.2	0.4, 5.8
Nguyen	12.7	6.4	2.0	15–30	150	0.5	12–23
Sugiyama	25	3	8.3	10	80	0.4	3.1
Wang	9	4.8	1.9	14	300	1–3.2	39
Turenne	N/A	4.8	N/A	17	250	0.4–8.6	33
Gnanavelu	5	7	0.7	5, 7.5	250	0.4	6.6, 10
Li	38.1	3.2	11.9	18–23	24, 150	N/A	.02–28
Mansouri	12.7	7	1.8	14	300	0.4–6.4	1.3–27
Current work	19–47	6.4	3.0–7.4	29.9	150	2.1 – 11.4	15

Table 2.1: Summary of liquid-based JIT erosion testing [2].

$$St = \frac{\tau_p U}{D} \tag{2.2}$$

$$\tau_p = \frac{\rho_p d_p^2}{18\mu_f} \tag{2.3}$$

Where U is the average exit velocity, D is nozzle diameter, d_p is particle diameter, ρ_p is particle density and μ_f the fluid viscosity. It can be seen from the equations that the inertia and the drag force on the particle are the two main factors which describe what trajectory the particle will follow.

The importance of the Stokes number in relation to CFD and PIV is to do with how closely the particles follow the fluid flow field, as illustrated in Figure 2.2. The lower the Stokes number, the more closely the particles follow the flow. This phenomenon is used when carrying out PIV on the liquid phase as small, low density particles ($St < 1$) are seeded to follow the flow field. Since they follow the fluid streamlines so well, the data taken from them can be assumed to be the same as what the actual fluid is doing. Thus, a discontinuous series of particles are used to measure the continuous flow of liquid.

The relationship between Stokes number and erosion location was investigated in a paper

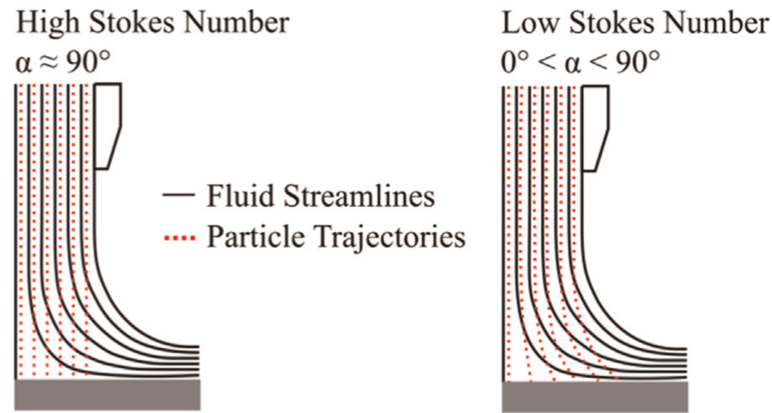


Figure 2.2: Stokes number illustration [2].

by Peng *et al.* [3]. They carried out erosion CFD simulations on a 90 degree circular pipe bend, and varied the Stokes number of the water/sand mixture to see what affect this had on the locations of maximum erosion. Figure 2.3 shows the varying degrees erosion for different Stokes numbers. Location A is the inner side of the elbow, and Location B is the outer side of the elbow. In a quote from the paper, they describe the relationship according to their results: '*The maximum erosion location will dynamically change as the Stokes number increases. Location A is exposed to greater erosion when the Stokes number is small. However, as Stokes number increases, this maximum erosion area will move to location B. When Stokes number is small, the drag force is dominant. The secondary flow plays an important role in the movement of particles. The secondary flow first drives the particles in the circumferential direction from the outer side to the inner side of the elbow and then drives the particles to the side wall of the elbow and as a result, causing severe erosion in this area. For a larger Stokes number, the inertia force plays a larger role on the particles movement. The particles have enough momentum to flow across eddies, thus the fluid velocity and the fluid flow direction have a smaller influence on the particles. As a result, the solid particles deviate from the fluid streamlines and impact on the outer wall.*'.

The evidence for this statement can clearly be seen from their figure, where the maximum erosion location shifts from location A when there is a low Stokes number, to location B when there is a high Stokes number. The importance of this in regard to this thesis is that care must

be taken with the Stokes number when comparing experimental results with computational results, as there can be significant differences in results if the Stokes number is not kept the same. For further detailed information about the different paths the particles take, and the relationship with erosion, consulting the paper is recommended.

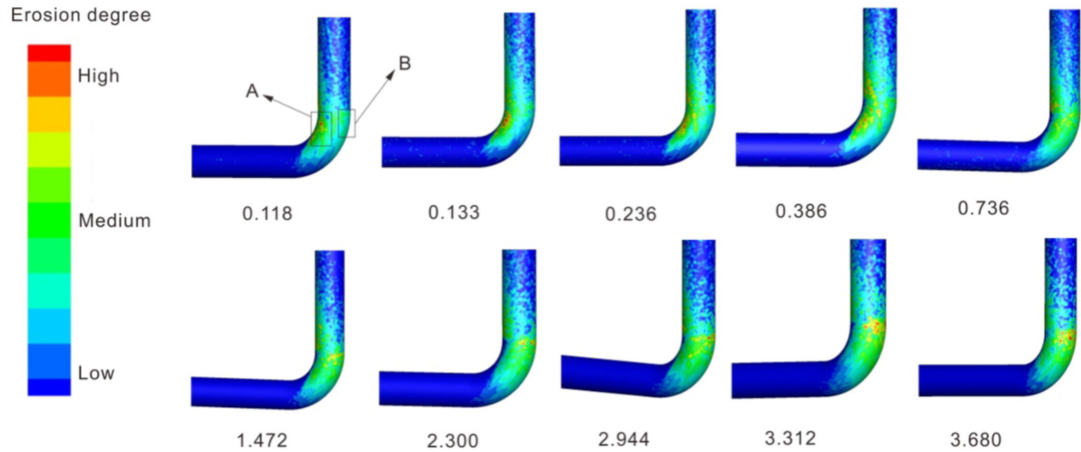


Figure 2.3: Erosion degree for various Stokes numbers. Fluid direction is from the horizontal section of the pipe to the vertical [3].

2.2 CFD modelling of erosion

Humphrey (1990) [29] states in his extensive review of over two hundred publications: *'many researchers continue to interpret and attempt to understand particle impact erosion almost exclusively in terms of the material properties involved. Little attention has been given to clarifying the influence of fluid motion, especially in the turbulent flow regime'*. To date, there has been little research into the affect that the flow field has on erosion. Since the fluid can be modelled on CFD and the codes have improved at modelling multiple phases; the research into erosion modelling subsequently increased. Although there is more simulation work being done in the field now than there was then, the author thinks there is still room for more: as the importance of better design as opposed to better material properties is often overlooked.

In 2007 Zhang *et al.* [30] published a paper comparing experimental and computational particle velocities. The introduction quotes Meng *et al.* [31] saying: 'no *single* (emphasis added)

erosion model is general and accurate for practical use'. Although this quote is from over 20 years ago, there are still a lot of papers published which use erosion models with arbitrary constants added in to suit the particular geometry being modelled [32, 33, 34]. If a single erosion model is ever going to be created there cannot be any factors added in, or constants to suit the geometry. Zhang realises that the particle impact velocity must be used instead of the flow velocity, since they are not necessarily the same. At low solid particle concentrations the particles do not significantly influence the flow and therefore the particle tracking can be carried out after the fluid simulation. This thesis is interested in high particle concentrations, and so this approach cannot be utilised. More details on the Stokes number can be found in Section 2.1.1.

Zhang states the three general steps that the majority of CFD based erosion modelling consists of: flow modelling, particle tracking, and erosion equations (based upon particle impacts). This idea was used by the author [35] of this thesis to compare different CFD packages' abilities to model the liquid phase of a submerged jet impingement test, as it was found that the liquid phase was frequently assumed to be correct rather than being tested. Both papers used the jet impingement test, but Zhang used Laser Doppler Velocimetry (LDV) whilst the current author used Particle Image Velocimetry (PIV) (more can be found on PIV in Chapter 4).

These findings and their effects must be considered by the CFD engineer trying to simulate the process. Although the main reason of the Hybrid model being developed in this thesis is with the application to erosion modelling, as mentioned previously, the main aim is to improve the flow modelling efficiency, and therefore mechanisms of erosion will not be further discussed. A large review on solid particle erosion by Parsi *et al.* [36] has been carried out, in which they cover the majority of published literature up until 2014. For a comprehensive overview this paper should be consulted.

2.2.1 Popular approaches of erosion modelling

Erosion is modelled via CFD in various different ways, with each way having associated advantages and disadvantages. A paper by Solnordal *et al.* [37] regarding erosion in pipe bends states

: 'Most commonly, researchers used standard Reynolds-averaged Navier-Stokes (RANS) solvers to predict Eulerian fluid flow through the elbow, followed by Lagrangian tracking of particles the so called Euler-Lagrange approach'. This backs up the idea that particulate modelling is a post-processing step. CFD modelling in elbows is common as there are a lot of experimental data on pipe bends due to the ease of experimentation [38, 39, 40, 41, 42]. In most of these papers, the main CFD model is a Eulerian one.

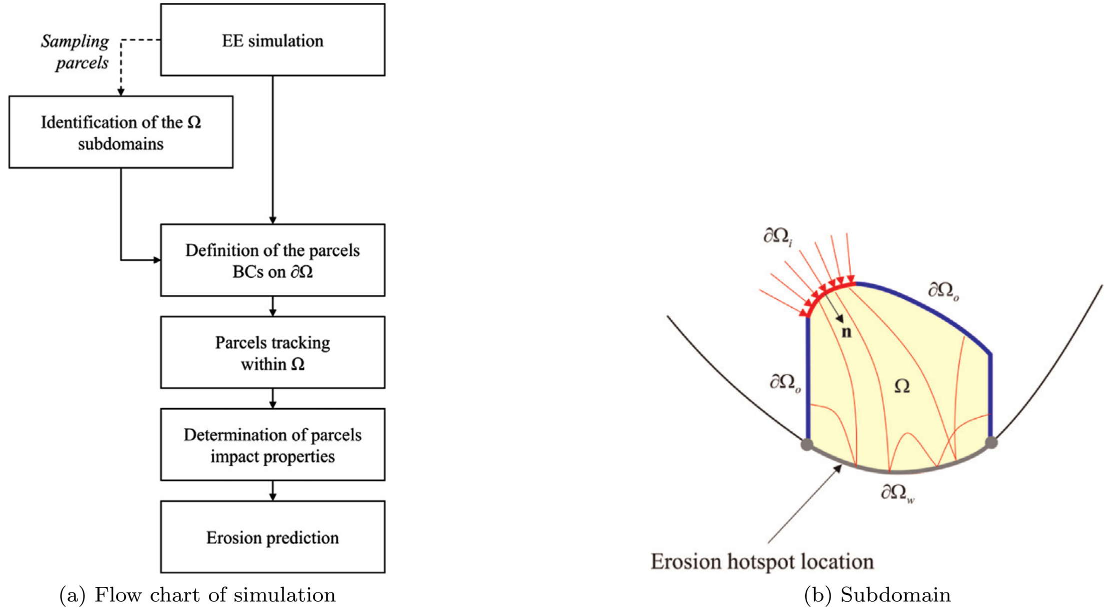
The main modelling approach for the submerged jet impingement test is a usually running a eulerian model through the domain first, and then injecting particles which follow the streamlines of the eulerian model as a post process step. In some papers, as in Lopez' [11], the eulerian simulation was run until convergence and then particles were injected as a transient simulation. More information on the jet impingement test can be obtained in section 2.4.1.

2.3 Hybrid models

The bulk of papers on erosion and slurry modelling are concentrated on the erosion modelling accuracy and not on the computational efficiency of the model. The purpose of combining two models into a hybrid is to reduce computational time without sacrificing accuracy. Attempts have been made to improve erosion predictions from the Eulerian reference frame (two fluid) however they still come across the problem that the particle phase is modelled as a continuum and thus have no velocity values *at the wall* as shown in the paper by Lu *et al.* : 'By taking the velocity vectors of the solid phase near the target wall' [43]. This poses a problem for erosion modelling since the erosion equations all need velocity values of the particles at the wall. This is something that EE models fail to accomplish.

The paper which is most similar to the work in this thesis is by Messa *et al.* entitled 'A mixed Euler-Euler/Euler-Lagrange approach to erosion prediction' [4]. Realising the problem with running EL at high particle concentrations, their solution is to first run EE throughout the whole domain and then run EL in a sub-domain of interest as a post process step; using the commercial code PHONETICS. Since their test case had low particle loading, they only

ran one phase (Euler) instead of the more expensive two phase Euler-Euler. Figure 2.4a shows a flow chart of the order that the simulation takes place, and Figure 2.4b depicts one of the subdomains. After the EE simulation has taken place, subdomains are chosen where high erosion rates are expected and then parcels of particles are injected to calculate the erosion rate. This requires the engineer to select the places they think erosion will be highest, based on the first simulation's results.



(a) Flow chart of simulation

(b) Subdomain

Figure 2.4: Figures from Messa *et al.* [4]

The new model was applied to two cases, one of which is the jet impingement test. The results of the hybrid model were similar to the pure EL ($<$ 15% difference), with a reduction of total simulation time of 30%. They conclude that the model is successful but only when low loading or one way coupling is available, and for short simulations so that geometry changes are negligible.

The introduction of Lagrangian particles can be seen to be a post processing step and not a runtime one. The downside of this is that the model is not two-way coupled, and therefore cannot deal with transient simulations or geometry changes for example. At the start of their paper they highlight the lack of work done in geometry changes due to erosion, other than the paper from Nguyen *et al.* [44]. Since this publication Lopez has published his PhD thesis

which details his method of mesh deformation [8]. The model in this thesis was created with the future plan to combine these two ideas.

A similar paper by Vujanovic *et al.* [45] combines an EE and EL simulation for an internal combustion engine. They use two control volumes which have an overlapping region, as shown in Figure 2.5. The blue region in the middle is where the control volumes from the spray on the left, and the engine on the right overlap. The spray starts off modelled as EE, and transitions into discrete particles (DDM: discrete droplet model), as can be seen in Figure 2.6. The EE simulation is run, and source terms are mapped onto the inlet boundary of the engine, for dispersed particles to be injected. The results of the paper were promising, with 'good agreement with the experimental data observed'. The main advantage of their approach was the improved CPU efficiency, characteristic of the EE method, combined with increased accuracy, characteristic of the EL method.

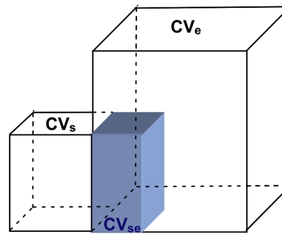


Figure 2.5: Intersection of control volumes between spray and engine mesh by Vujanovic *et al.*

Another approach witnessed in the field of bubbles/cavitation was by Dr Aurelia Vallier [46] in her PhD thesis. She studied the modelling approaches to cavitation on a hydrofoil and managed to combine two models: one to model small bubbles, and another to model large bubbles. The VOF (volume-of-fluid) model is used for large bubbles, however it cannot be used for structures that are smaller than the cell size, like the DBM (discreet bubble method) can. A quote from Chapter 5 of the thesis explains the new model: *'The new multi-scale approach uses the strength of the VOF method to model the large structures and the strength of the DBM approach to model the small structures. The bubbles are modelled in a Lagrangian framework, and the VOF representation is also referred to as the Eulerian framework.'*. The

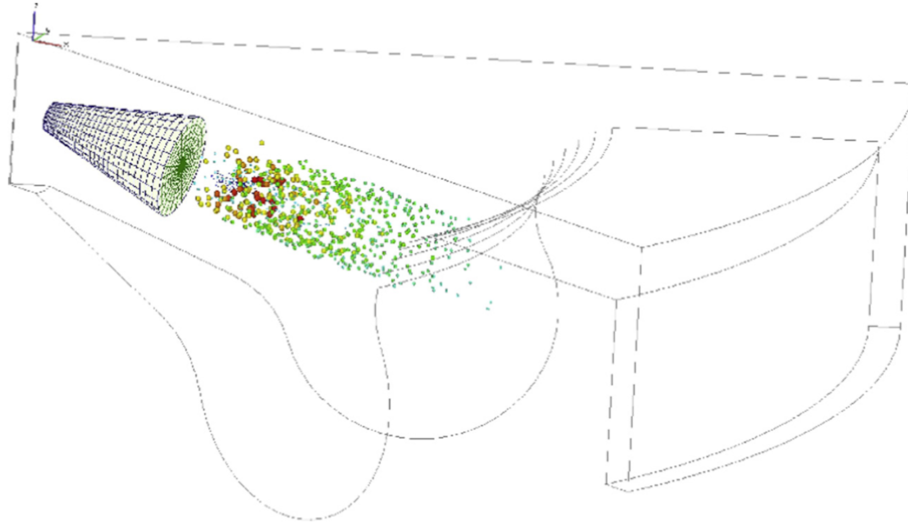


Figure 2.6: Transition from spray to engine by Vujanovic *et al.*

method presented by Vallier transitions from the Eulerian frame to Lagrangian, and back again. The downside with this approach is that two meshes are required for one geometry, and so this creates more work for the engineer who wants to simulate their problem.

The field of diesel spray modelling has also seen some attempts at hybrid models [45, 47, 48, 49]. One of the most recent papers is by Yu, titled 'A parallel volume of fluid-Lagrangian Parcel Tracking coupling procedure for diesel spray modelling' [50]. The breakup of the jet is captured by the VOF method, and in regions where the phase interface cannot sufficiently be resolved small scale liquid structures are described by a Lagrangian Parcel Tracking (LPT) approach. They combine the two grids required for the two approaches in an inter facial region, called the Region Coupling Method (RCM) shown in Figure 2.7.

Figure 2.7 shows the two grids required for this technique, the VOF and the LPT, and their overlapping region in the middle. The results confirmed that the RCM gave high fidelity results of the evolution from a liquid spray to dispersed secondary droplets. Although this technique looks promising for the future in the application of diesel sprays, there are some difficulties which make it hard to apply to the field of erosion modelling. One difficulty is the need for different grid sizes and for an overlapping region. If complex geometries are present, it is more difficult to create a uniform 'coupling region' as in Figure 2.7. One other difference is that

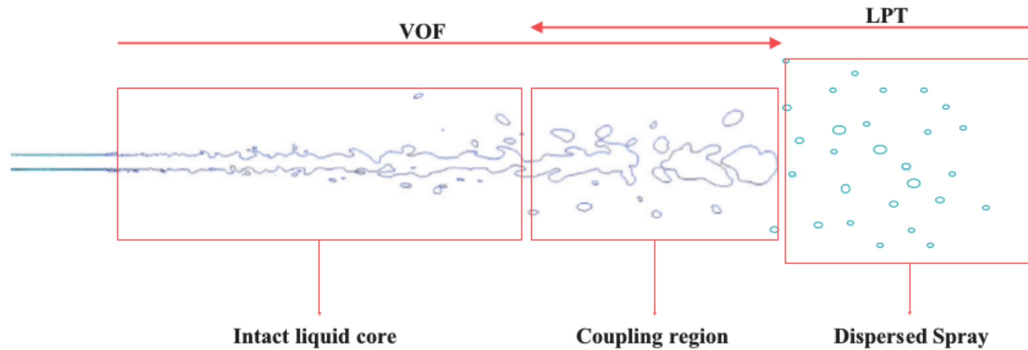


Figure 2.7: RCM shown in between VOF and LPT modelling of liquid diesel spray by Yu *et al.*

slurries are not usually modelled with the VOF method, since it is more suited to free surface liquids; it is therefore uncertain if the above technique would work with slurry erosion. The phenomenon is slightly different, as the RCM technique models the break up of the fluid flow, whereas a hybrid slurry model must transform the dispersed liquid (particles) phase into a dispersed solid (particles) phase.

Another hybrid modelling paper by Wardle *et al.* [51] which mentions the associated benefits and drawbacks is also based on the VOF solver. In the introduction they state: '*For 'dispersed' flows in which one phase is continuous and the other is distributed in fine droplets, one can use Lagrangian particle tracking for small phase fractions (less than 10%) in which each individual fluid particle is followed through the fluid in response to local flow conditions.*'. It is this 'fact' which hybrid models aim to solve, by enabling higher particle concentrations (and therefore higher phase fractions), through reducing the computational burden by partially solving the domain through the EE technique.

As shown by the papers cited, the hybrid modelling approach has been applied to various engineering problems with success. The models have all had the aim to reduce computational effort without sacrificing the accuracy of the result: which is the aim of the new hybrid model in this thesis. The benefits of the models have all been reduced runtime or reduced CPU load, however these benefits do not come without incurring a number of problems. The transitions between the coupled models proved to be difficult in some cases, and the ability to scale up to more complex geometries seems to be a challenge for the future. The trade off between saving

computational time and reducing accuracy is also an ongoing issue.

2.4 Experimental methods for modelling slurry erosion

Slurries are not very easy to measure experimentally due to their opaque and distributed nature: however there are a few solutions available which will be discussed here, with focus on the methods used in this thesis.

2.4.1 Jet Impingement Test

By far the most popular experimental method for erosion verification is some form of the jet impingement test (JIT) [2, 4, 11, 23, 25, 34, 35, 52, 53, 54, 55]. The main reason for this is that the JIT gives a wide range of particle impact speeds and angles in a small geometrical area. Since erosion is primarily affected by speed and impact angle, this is an ideal experiment. The experiments in this thesis were all done on the submerged jet impingement test with water and sand particles. Other common test rig set-ups are the centrifugal accelerator [56] and the pipe bend [57]. The first of these isn't as repeatable as the JIT, and the second is harder to analyse (in real-time and in post processing) due to the curved nature of the pipe.

Figure 2.8 shows an image of a nozzle and flat impingement surface (laser sheet shining from the right), with seeded particles for analysing. The green arrows show a scaled average representation of velocity vectors, using the data from a series of images. The arrows appear to go through the sample, however this is because they are scaled up for clarity, not because there are any velocity values in the sample.

Data acquisition in the JIT Another benefit with the JIT is the ease with which it can be compared to CFD data. A 2D light sheet can be created with a laser for particle image velocity (PIV) to be carried out, as in Mansouri or Mackenzie [5, 34, 35]. The paper by Mansouri in 2016 uses a similar technique to the one adopted in this paper on a submerged jet impingement test. Although there have been a lot of erosion studies carried out on the JIT, very few exist which have PIV as the main way of gathering fluid data. The material loss method [26, 56, 58],

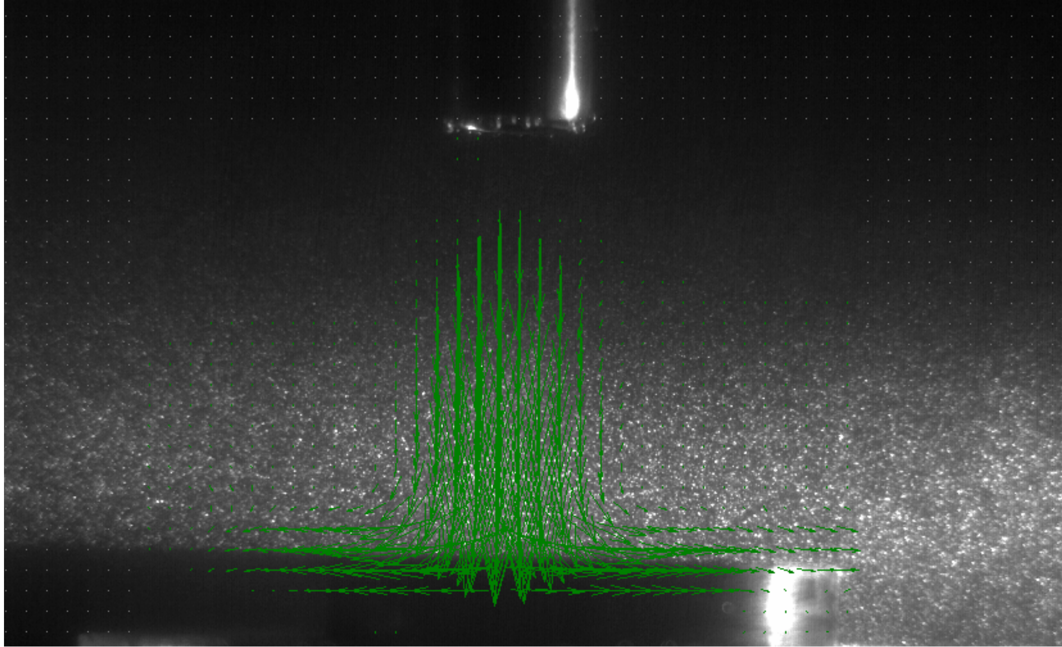


Figure 2.8: Submerged JIT showing velocity vectors obtained by PIV (taken by author).

which is common, does not give direct results of the flow field and therefore must include some assumptions which makes PIV a superior method for data acquisition (material loss method is where the accuracy of the erosion model is purely based upon how much material it predicts will be lost. This does not take into account the particles' impact velocity or angle, but is based on constants being inserted into the equations.). The Mansouri paper analysed particle impacts of sand in a water mixture with average diameter of 300μm, and jet exit velocity of 8m/s (see Figure 2.9). The data were processed with the particle tracking velocimetry (PTV) technique, grouping the data into cuboid shaped cells near the wall. Average velocities were then attributed to each cell, allowing direct comparison with the CFD code. They found that CFD results matched experimental data very well.

Figure 2.10 shows the setup from Mansouri's paper which is similar to setup in this thesis (Chapter 4) and contains the equipment required for any PIV experiment. The PIV technique is explained in more detail in section 4.1.1.

Another measurement technique used is Laser Doppler Velocimeter analysis [59]. Two downsides with this method are that there are fixed data measurement points (unlike PIV which

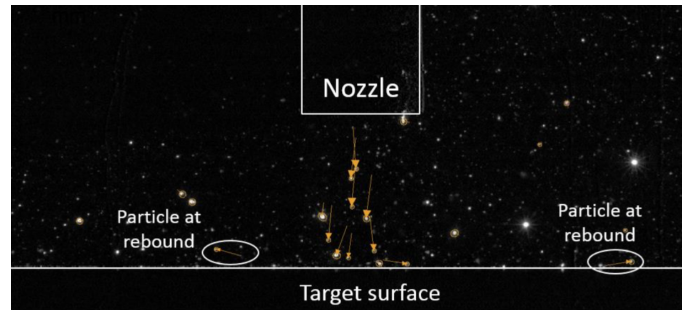


Figure 2.9: Figure from Mansouri's [5] experimental test, showing particles and nozzle.

creates a continuious 2D sheet), and that near-wall results are difficult to obtain due to the volume of the laser beams, as Okita *et al.* [52] states 'The particle speeds could not be measured near or on the target surface because of interference of the measuring volume of the laser beams of LDV system and the wall'. Although the PIV technique close to the wall also has its problems, it was chosen as the method for data acquisition, as the author already had access and experience with the equipment necessary, and as has been discussed: the technique worked well in other papers.

The papers all cited who used the jet impingement test for data acquisition used the same impingement surface shape, a flat surface. The author realised that there had not been any analysis on different sample shapes, which would be more difficult for the computational model to replicate accurately. If a geometry independent erosion model was to be developed, the CFD codes modelling the JIT needed to be accurate on a range of different shapes, and not just on a flat surface.

2.5 Conclusion

The problem of erosion modelling as viewed in the current literature has been presented. Extensive research has been carried out on trying to model erosion with various CFD models, however as shown, there has not been the same effort made to improve the efficiency of these erosion models. The Hybrid model developed in this thesis will attempt to bridge some of the gaps in the literature, in the aim of making a geometry independent erosion model, capable of

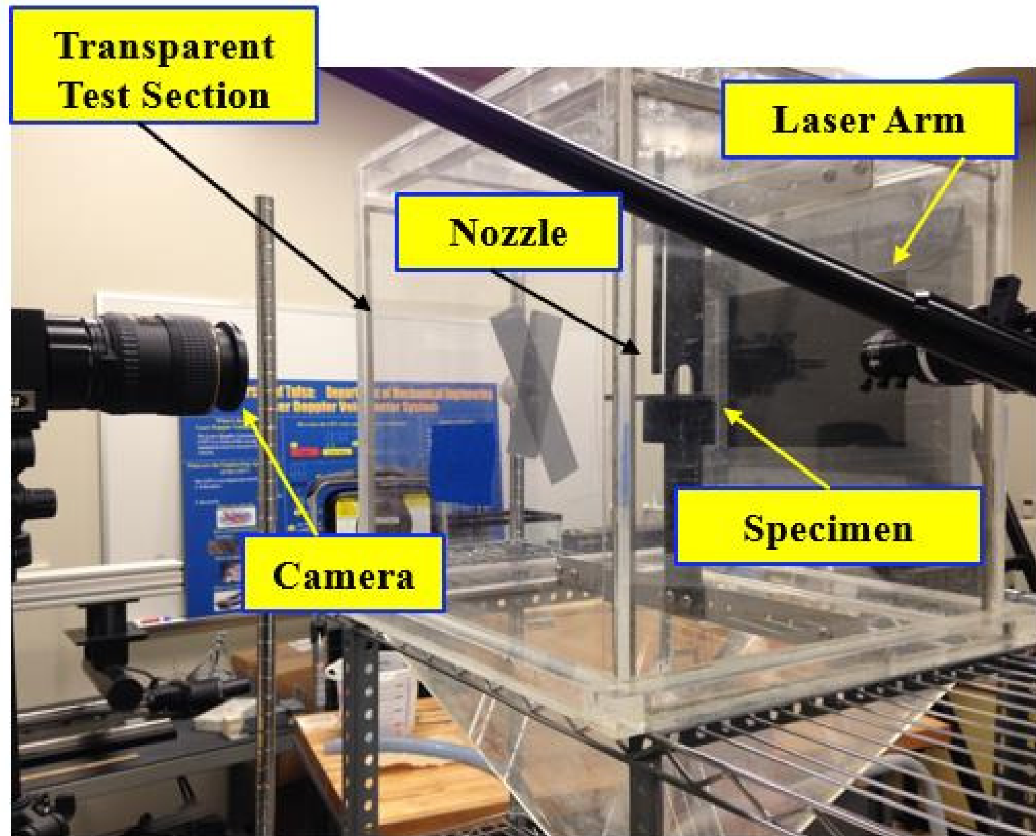


Figure 2.10: PIV setup for submerged jet impingement tests from Mansouri's paper [5].

modelling dense slurries.

Chapter 3

Solver development

> At its heart, engineering is about using science to find creative, practical solutions. It is a noble profession.
>
> Queen Elizabeth II

A solver which modelled dense slurry flows more efficiently but kept particle impact data at the walls was developed. The method was to combine and modify two existing solvers from the open source CFD package, OpenFOAM [19]. EE models can model high mass concentration slurries with less computational effort than equivalent EL models. However, since particles are modelled as a fluid, there are no particle impact data available at the wall. The idea was to combine an EE model and an EL model, having the EL model near the walls (where particle impact data is required) and the EE model in the rest of the domain where computational efficiency is more important.

This chapter will detail why each model was chosen and how they were combined.

3.1 Choosing software and models

As mentioned previously, one of the main problems with simulating a high number of particles (as in dense slurry flows) is the high computational cost. This is due to the way the Lagrangian solver stores each parcel's location and velocity each time step. Dense slurries can also be

modelled as a fluid in the Eulerian reference as this is a much more efficient process. The idea was to combine these two modelling methods into one hybrid model, reducing computational time but keeping accurate results.

With this task in mind, a CFD code was required which contained two suitable models and was easily modifiable. OpenFOAM 3.0.x was chosen as it met these two requirements, and was the most current version in 2015 (when the development of the model started). Furthermore, since the program is open source, one of the greatest benefits was the ability to have complete control over the code: something which was not possible with commercial alternatives.

A comprehensive tutorial [60] was published by the author on the development of this model on the Chalmers University website. It was written in a pedagogical manner to enable the interested person to replicate the work; however the model was not fully complete at the time of publication (January 2017) and three main issues still existed.

- The volume fraction of the Lagrangian particles had no effect on the fluid phase.
- The lookupTable was only read when the solution was initialised, and ignored thereafter.
- The code wasn't reading the values from the lookupTable correctly.

As well as explaining the solution to these issues the development of the solver will also be explained in this chapter. The reasoning behind the approaches will be emphasised rather than the methodology: as most of the methodology steps have already been covered in the tutorial already referenced.

3.1.1 Euler-Euler Model

A multiphase model that modelled both the fluid and the particles as fluids (or in the Eulerian frame) was found in the set of existing multiphase OpenFOAM solvers, called

`reactingTwoPhaseEulerFoam` [61]. The description given by OpenFOAM is:

```
Solver for a system of 2 compressible fluid phases with a common pressure,
but otherwise separate properties. The type of phase model is run time
selectable and can optionally represent multiple species and in-phase
reactions. The phase system is also run time selectable and can optionally
represent different types of momentum, heat and mass transfer.
```

Although this model description is not exactly what is required, small modifications were made to make it suitable for the application. This technique of modifying an existing solver to suit the user's needs is usually what is required in OpenFOAM. Reactions between phases and mass transfer were turned off, and both 'liquids' were made incompressible. A description of the model can be read in the tutorial by Thummala [62], and the governing equations in the paper by Bhusare [63]. The minor modifications made and further details on the solver are in Section 2.2 of the Chalmers Tutorial.

3.1.2 Euler-Lagrange Model

The most widely used dispersed particulate solver in OpenFOAM is `DPMFoam` [64] (Discrete Particulate Modelling Foam). OpenFOAM's description is as follows:

```
Transient solver for the coupled transport of a single kinematic particle
cloud including the effect of the volume fraction of particles on the
continuous phase.
```

It should be noted that the code already existed for including the volume fraction of particles on the continuous phase, however the author had not yet implemented it. The trajectories of the particles/parcels are based upon Newton's equations of motion. The motion of a parcel is found by integrating the force balance on it, and it is treated as a point mass therefore torque is omitted [65]. The below equations show how the parcel's velocity is found:

$$\frac{dx_p}{dt} = u_p \tag{3.1}$$

$$m_p \frac{du_p}{dt} = F_p \tag{3.2}$$

where x_p is the position of the parcel, u_p is the parcel's velocity, m_p is the particle mass, and F_p is the force on the parcel. Although the parcels are treated as point masses from a dynamics point of view, their influence on the volume fraction of the continuous phase is included in the volume fraction equation as will be shown later. More information regarding the equations in DPMFoam can be found in the paper by Li *et al.* [66].

3.2 Creating the new Hybrid model

This section will describe how the Hybrid model was created, and will be divided into subsections on the way the model was developed, similar to the OpenFOAM workshop paper by the author [67]. The reader should also be aware that this methodology of combining two solvers can be applied to other applications.

3.2.1 Geometry setup: Mesh/Baffles/Regions

The aim was to have the EL model near the wall, and the EE model everywhere else. Having two solvers running in one geometrical space created some issues as previously discussed in the Literature review. The method chosen for switching solvers was to create a geometrical boundary which acted as an outlet for the EE and an inlet for the EL. The boundary had two purposes: firstly, it provided a location where the second Eulerian phase would be 'removed' (or turned off), and secondly it was where the particles could be injected from. This boundary should not significantly affect the flow field, and so the values of the fields from the 'outlet' side needed to be mapped onto the 'inlet' side. The fields in each solver are detailed in Table 3.1.

Description of variable	**EE** reactingTwoPhaseEulerFoam	**EL** DPMFoam
Volume fraction of fluid	alpha.water	alpha.water
Volume fraction of solids	alpha.particles	-
Thermal diffusivity (unused)	alphat.water	-
Thermal diffusivity (unused)	alphat.particles	-
Turbulence kinetic energy	k.water	k.water
Turbulence viscosity (calculated)	nut.water	nut.water
Turbulence viscosity (calculated)	nut.particles	-
Dynamic viscosity (calculated)	-	mu.water
Turbulence specific dissipation rate	omega.water	omega.water
Pressure	p	p
Mass flow through cell (calculated)	phi.water	phi.water
Mass flow through cell (calculated)	phi.particles	-
Pressure ÷ density	p_rgh	-
Granular temperature	Theta.particles	-
Velocity of fluid	U.water	U.water
Velocity of solids	U.particles	-

Table 3.1: Fields in each solver.

It should be noted that although some fields do not exist in both models, these fields are

not so influential as to affect the flow; or they have another definition in the other model. For example 'alpha.particles' does not exist for the EL model however the volume of the Lagrangian particles are taken into effect and are subtracted from 'alpha.water'; and so in cells where particles are present: 'alpha.water' $\neq 1$ ($1 = 100\%$ volume fraction). 'nut.water/particles' are not used, as they are only used for the 'SpalartAllmaras' turbulence models. There is also a temperature field in the EE model, but since the temperature equation was switched off, this had no effect on the results.

A simple geometry was chosen for development purposes, and meshed with OpenFOAM's native mesher `blockMesh`. A square pipe bend was chosen from the tutorial by Lopez [68] as it is complex enough to prove the concept but it is simple enough to converge easily. A coarse mesh was used at this stage (3000 cells) as quick solution time was much more important than accurate results.

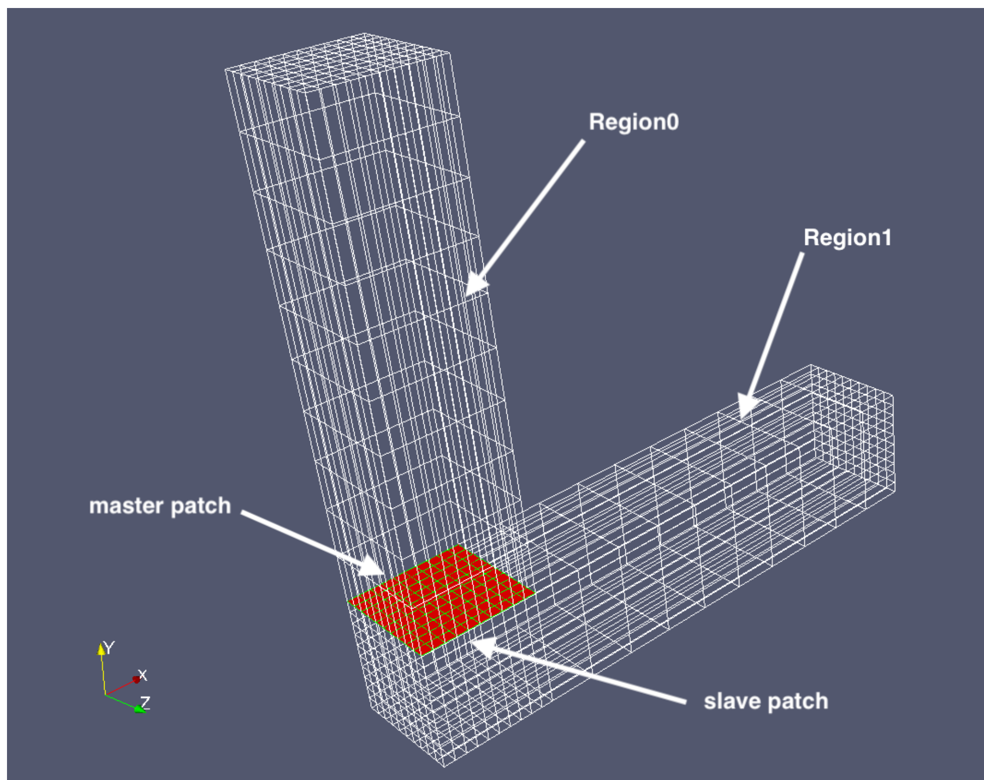


Figure 3.1: Wireframe geometry showing baffle in red.

To create the boundary, an application in OpenFOAM was used called `createBaffles`.

This utility transforms internal faces into boundary faces: which allows boundary conditions to be applied. For the baffle to be created, an .stl surface must be drawn at a suitable location. This was chosen near the corner of the pipe bend as shown in Figure 3.1. This location is where high gradients are expected to be found and therefore will test out the solver's capability more than if an 'easier' location was chosen, for example further upstream in the pipe. Two patches were created, the top one called 'master' and the bottom one called 'slave'. The inlet of the pipe is at the top and the outlet at the bottom right, and all other faces are walls.

With the baffle in place, another utility called `splitMeshRegions` was used to split the mesh into two separate regions. This application splits the mesh along the patch line, which enables the first region to contain EE and the second region to contain EL. Since the geometry has two regions they can be solved one after another in a loop, which is what is done in the solver `chtMultiRegionFoam` as it solves fluid regions before solving for solid regions.

3.2.2 Solver creation and interpolation

The new solver was created by using `reactingTwoPhaseEulerFoam` as the template, since the code which deals with particles from `DPMFoam` could easily be added to it. The method of creation was to first get the EE model running correctly and unaffected by the baffles through the whole domain (both regions). Another OpenFOAM utility `patchToPatchInterpolation` was used to achieve this, by transferring the fields from the outlet of the first region to the inlet of the second region. This utility can also transfer data between dissimilar patches which could prove useful in future for more complex geometries.

To create two regions the following two lines were added from `chtMultiRegionFoam`:

```
#include "createFluidMeshes.H" // This is the first region mesh
#include "createSolidMeshes.H" // This is the second region mesh
```

The names were kept from the original solver, therefore the fluid mesh is the first region, and the solid mesh is the second region. These 'include' statements reference the header files with the same names; and it is in these files that the meshes are created.

With the regions created, the necessary code from `patchToPatchInterpolation` was added

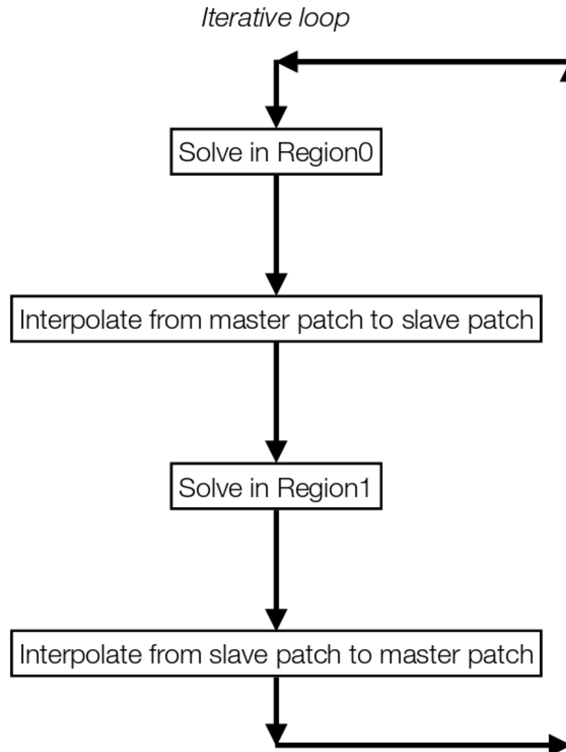


Figure 3.2: Depiction of iterative loop.

to the main solver file. An iterative loop was written as seen in Figure 3.2. This translates to the top section of pipe being solved first, then the results are interpolated from the top section to the bottom. The bottom region is then solved, and any changes are interpolated back to the top section. This loop is carried out continuously throughout the solver process. This method of interpolating back to the first region ensures that any geometry changes caused by erosion will affect the upstream flow, or any flow changes in the second region will affect the flow in the first region. This makes the solver suitable for erosion work where large deformations in the mesh are common, which in turn create changes in the flow field.

The patches created by the baffles were labelled so they could easily be used by the interpolation code. An include statement referencing a header file was added containing all of the interpolations from the master to the slave patch. The interpolation code for one of the fields is included below:

```
//U1 interpolator    note:U1=U.particles                (master to slave)
    patchToPatchInterpolation interpolatorU1
    (
```

```
        mesh.boundaryMesh()[master], // from patch
        mesh.boundaryMesh()[slave], // to patch
        intersection::HALF_RAY,
        intersection::VECTOR //
    );

// interpolate from outlet to inlet
        vectorField interpolatedInletU1 =
        interpolatorU1.faceInterpolate <vector>

        (
        U1.boundaryField()[master]
        );

        if(U1.boundaryField()[slave].type() !=
        fixedValueFvPatchVectorField::typeName)

        FatalError << "inlet patch should be fixedValue!" << exit(FatalError);

        U1.boundaryField()[slave] == interpolatedInletU1;

Info<< "inletU1InterpolDomain1 = " << interpolatedInletU1 << endl;
```

This is for the field ‘U.particles’, and it is interpolating from the master to the slave patch. This is done for each field in the solver, and a similar file is created with the fields going back from the slave to master patch. More detail on the interpolation code is in the tutorial.

At this stage the solver runs almost as if the baffles do not exist and there is just one region/mesh. The solution is almost identical to that of the standard EE model, with the largest error being the cells adjacent to the baffle: clearly shown in Figure 3.3. This error must be accepted as a downside of the Hybrid model, and therefore the aim should be to reduce, instead of eliminate, any discrepancies near the baffle.

More analysis of this geometry can be found in chapter 4 of the OpenFOAM workshop paper by the author [67]. To conclude this section: the EE model was giving almost the same results as if there was one mesh, however it was split over two, and the important outlet/inlet patches existed for particle injections. The solver was ready to incorporate particle injections.

3.2.3 Addition of particles to solver

To conserve mass continuity when the Lagrangian particles were added, the second Eulerian phase (particles) of the EE model was removed from the second region. This was simply done

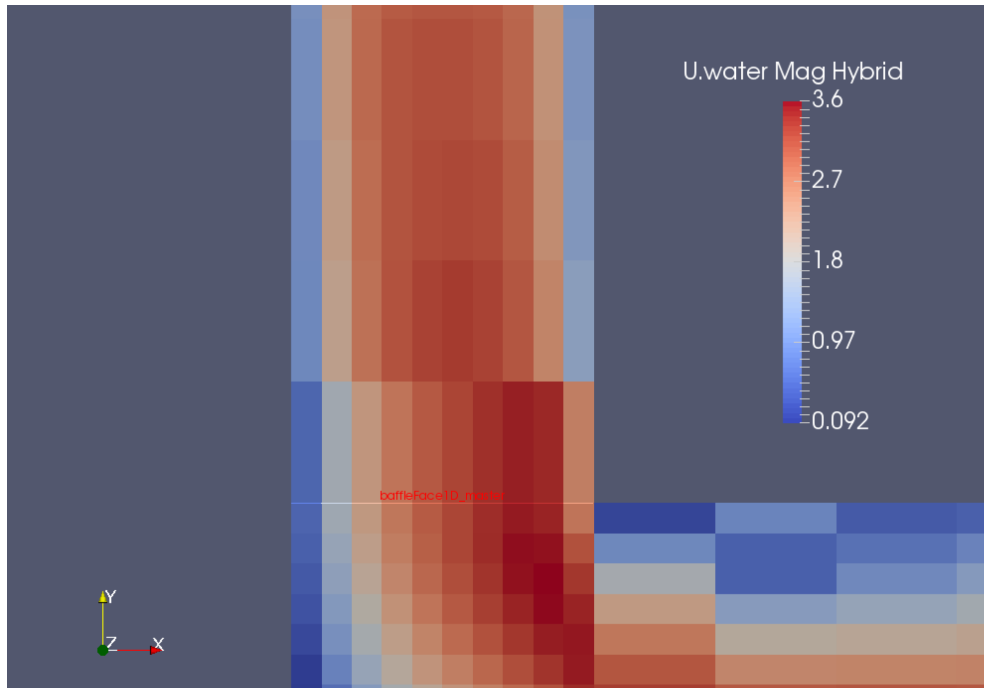


Figure 3.3: Velocity contour plot of Hybrid model showing cells with same values either side of baffle (red writing).

by removing the alpha.particles and U.particles (velocity of particles) interpolation code (thus stopping interpolation), and overwriting both inlet boundary conditions to equal 0 for the second region. This left values of alpha.particles and U.particles at the master patch (outlet for the first region) which the particle injections could be based on, whilst still removing all of the particulate Eulerian phase from the second region.

The necessary sections of code from DPMFoam to introduce particles was added to the main solver file and is shown below (note that the code shown is inserted into the code at the correct locations: see files in the appendix or the Tutorial if more detail is required).

```
argList::addOption //  gives option to change cloud name
   (
     "cloudName",
     "name",
     "specify alternative cloud name. default is 'kinematicCloud'"
   );



Info<< "Evolving " << kinematicCloud.name() << endl;
      kinematicCloud.evolve();
      kinematicCloud.updateMesh();
```

```
// Update continuous phase volume fraction field
alpha2 = max(1.0 - kinematicCloud.theta() - alpha1, alpha2Min);
alpha2.correctBoundaryConditions();
alpha2f = fvc::interpolate(alpha2);
alphaPhi2 = alpha2f*phi1;

fvVectorMatrix cloudSU(kinematicCloud.SU(U2));
volVectorField cloudVolSUSu
(
        IOobject
        (
                "cloudVolSUSu",
                runTime.timeName(),
                mesh
        ),
        mesh,
        dimensionedVector
        (
                "0",
                cloudSU.dimensions()/dimVolume,
                vector::zero
        ),
        zeroGradientFvPatchVectorField::typeName
);
cloudVolSUSu.internalField() = -cloudSU.source()/mesh.V();
cloudVolSUSu.correctBoundaryConditions();
cloudSU.source() = vector::zero;
```

As well as adding this code to the main file, other sections of code need to be added to createFields.H, the header file where different fields are created. The above code is in three sections: first is the option to call the cloud a new name, second is the 'evolve' function which contains all the functions to inject/move particles, and lastly is the code to update the continuous phase volume fraction field, using this equation (which is an existing equation in OpenFOAM [69]):

```
alpha2 = max(1.0 - kinematicCloud.theta() - alpha1, alpha2Min);
```

Where `kinematicCloud.theta()` is defined as `theta[cellI] += p.nParticle()*p.volume();` i.e. `theta()` is the number of particles multiplied by their volume within a particular cell. This solved one of the problems remaining from the Hybrid model tutorial as mentioned in Section 3.1.

The code for Lagrangian particles from DPMFoam was now in the solver and so could be run with the test case created. However, no standard injection method from OpenFOAM suited

the requirements of the Hybrid model: therefore, one had to be created.

3.3 New injection model

There was a requirement for particles to be injected into the second region (from the slave patch), using the results from the first region (master patch). The injection method needed to cope with geometry changes and therefore had to be coupled two-way with the first region. This meant that input values of particle injections had to dynamically change with the output values from the first region. If this did not happen, then the whole process of injecting particles becomes a post-processing step, and not a run-time one. The requirements for the injection model were:

- The injections needed to be based on the concentration and velocity of the Eulerian particulate phase from the first region.
- The injection values needed to update each time step, to account for flow changes in the first region, and to keep two way coupling.
- Each injection site had to control the number of particles to inject individually: on a cell by cell basis.

Out of the eleven OpenFOAM injection models, none of them fitted these requirements. The closest one was `kinematicLookupTableInjection`, which has the description:

```
Particle injection sources read from look-up table. Each row corresponds to
an injection site.

(
    (x y z) (u v w) d rho mDot   // injector 1
    (x y z) (u v w) d rho mDot   // injector 2
    ...
    (x y z) (u v w) d rho mDot  // injector N
);

where:
    x, y, z = global cartesian co-ordinates [m]
    u, v, w = global cartesian velocity components [m/s]
    d       = diameter [m]
    rho     = density [kg/m3]
    mDot    = mass flow rate [kg/m3]
```

This lookupTable allows the user to specify the position of the injection site, the velocity of the particle injected, as well as its diameter, density and mass flow rate. As with all of OpenFOAM's injection methods it is a static input, and so does not change with time and therefore was one of the main things that needed to be edited. It should also be noted that the `mDot` variable does not actually control the mass flow rate, but rather is used to calculate the volume of particles injected. Since there was still only one particle injected from each injection site, this also needed to be modified.

3.3.1 Modifications to injection model

The solution was to add another column to the table called `numParticles` which defined the number of particles to inject. The particles are injected as a parcel, and the number of particles per parcel is based upon the amount of volume fraction of the second Eulerian phase which exits the outlet of the first region. Each cell across the baffle face has its own row in the table, and the parcel is injected from the cell centres. The equation for `numParticles` is:

$$nP = \frac{(\alpha_p * A_{cell} * V_{normal})}{(Vol_p * \Delta T^{-1})} \tag{3.3}$$

where α_p is the volume fraction of the particulate phase, A_{cell} is the area of the face of the cell on the baffle, V_{normal} is the normal velocity component (to the baffle) of the particulate phase, Vol_p is the volume of one particle, and ΔT^{-1} is the number of time-steps per second. So, the top line gives the volume flow rate of the Eulerian phase, and the bottom line gives the equivalent volume flow rate of the Lagrangian phase. The new columns in the lookupTable have the following format:

```
(x y z) (u v w) d rho mDot numParticles
```

This calculation is done per cell on the baffle patch: with the result of the equation defining the number of particles to insert into each parcel. The `abs` function is used on the formula to force the answer to be an integral value, as injecting a fraction of a particle is not possible. This technique creates some small rounding errors, however they will be negated since there is

rounding up as well as down.

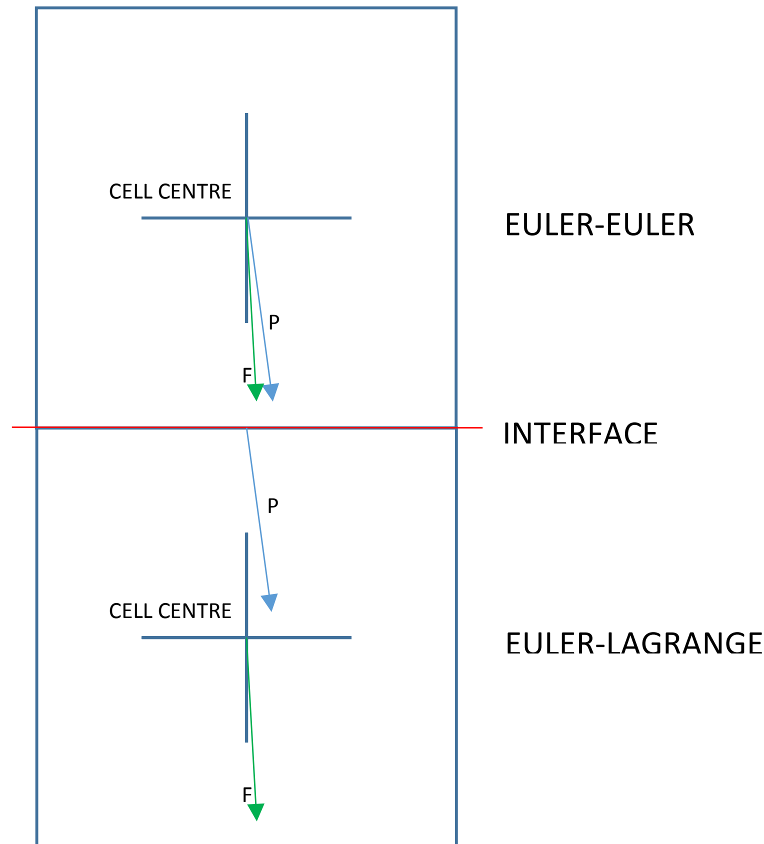


Figure 3.4: Depiction of transition between two models at the baffle. P is particulate phase, and F is the fluid phase.

Figure 3.4 shows the baffle/injection sites in more detail, and what takes place. The blue arrow with P represents the particulate phase, and the green arrow with F represents the fluid phase. The blue squares represent cells with the cross in the middle showing where the cell centre is, and the red line indicating the location of the baffle/interface. OpenFOAM uses cell centre values for calculations, as can be seen by both the vectors in the EE region. As described in Section 3.2.2, the fluid velocity vector from the EE is translated down to the EL cell centre: thus replicating the cell from the 'fluid' perspective. The particulate phase is slightly different as in region 1 it is modelled as a fluid, and in region 2 is modelled as a parcel. The solver first calculates the volume flow rate of the particulate phase *through the cell face*, which requires obtaining the velocity normal to the baffle (see Section 5.2 of the Tutorial for more detail).

This volume flow rate is then translated into a number of particles (Equation 3.3), which are then added to the parcel to be injected. The injection location is then set as the cell centre of the slave patch (on the interface), as shown by the P vector in region 2. The velocity value for the injection is taken from the EE P velocity vector.

The technique's advantages and disadvantages are discussed in Section 3.4.

3.3.2 Modifications to KinematicLookupTableInjection.C

A large number of modifications were made to the injection model main file: 'KinematicLookupTableInjection.C'. Not every detail will be explained here as the modified injection model folder (along with files) is included in the appendix and some details already exist in the Tutorial. The major modifications will be explained going through the file from top to bottom.

Firstly, the IOobject [70] `injectors_` was edited from saying 'NO_WRITE' to 'AUTO_WRITE'. This enabled the object `injectors_` to re-write automatically when requested by the objectRegistry, which was crucial for allowing the model to adapt each time step. Spaces needed to be added to the file 'kinematicParcelInjectionDataIO.C', as without them each of the scalars were printed to file without any gaps, rendering them indistinguishable from one another.

A new object called `currentInjector_` was constructed alongside the other injector objects. This object allows access to the specific injector required by the injection model which is crucial for the correct particle distribution. This feature was not required with the previous methodology, since every injector was treated the same.

The volume total (`volumeTotal_`) equation was changed to use `numParticles` instead of `mDot`, as the variable `mDot` is not used or updated in the lookupTable. The equation was changed from:

```
this->volumeTotal_ += injectors_[i].mDot()/injectors_[i].rho();
```

to:

```
this->volumeTotal_ += injectors_[i].numParticles()*
(((pi)*pow3(injectors_[i].d()))/6)*(timestepsPerSecond_);
```

The new equation is the number of particles, multiplied by their volume, multiplied by how

many injections per second (which is the same as time steps per second): which gives a volume flow rate equivalent to the original equation.

The original formula for the number of parcels to inject (`parcelsToInject`) looked up a scalar defined by the user in the 'kinematicCloudProperties' file in the constant directory. However, this scalar is defunct and so the equation was changed to equal the number of injector cells, meaning each cell has the ability to inject one parcel. It should be noted that if the number of particles in a certain parcel is 0, then no parcel will be injected- as one would expect.

`setPositionAndCell` sets the position of the injectors and what cell they will be in. This section was unedited, since the standard code does what is required and selects the position based on what is in the lookupTable.

`setProperties` sets the properties of each parcel: velocity, diameter, density and the current injector number (newly added). `currentInjector_` is defined here, to allow use for defining the number of particles per parcel needed in `setNumberOfParticles`.

Lastly, the code from 'InjectionModel.C' file was copied into 'KinematicLookupTableInjection.C' called `setNumberOfParticles`, overwriting what is in the 'InjectionModel.C' file. This key function defines how many particles are in each parcel, as shown below:

```
template<class CloudType>
Foam::scalar Foam::KinematicLookupTableInjection<CloudType>::setNumberOfParticles
(
const label parcels,
const scalar volumeFraction,
const scalar diameter,
const scalar rho
)
{
    scalar nP = 0.0;
    scalar nPTotal = 0;
    nP = injectors_[currentInjector_].numParticles();
    nPTotal += nP;

    Info<< nl
        << "Injector #" << currentInjector_ << " --> has numParticles = "
    << injectors_[currentInjector_].numParticles()<< endl;;
    return nP;
}
```

`nP` (number of particles) looks into `injectors_`, which is populated from the lookupTable, and takes the value of `numParticles` associated with the correct injector number (or row in the

table). For example, below is text snipped from a lookupTable file. If the current injector is number 3 and the lookupTable below is used, `nP` looks into the object `injectors_` (which is the same as the lookupTable) and takes the value in the last column of row 3, which is `numParticles`: so `nP`=2.

```
constant/kinematicLookupTableInjection
(0.0005 0.01 -0.0005) (0.0252 -0.486 -0.009529) 5.5e-05 2750 0.005 3
(0.0015 0.01 -0.0005) (0.0718 -1.011 -0.008696) 5.5e-05 2750 0.005 1
(0.0025 0.01 -0.0005) (0.1149 -1.327 -0.008962) 5.5e-05 2750 0.005 2
...
```

`nPTotal` adds up the total number of particles in each injection (over all cells). Lastly there is an info statement which prints to the terminal each time step/injection.

Other necessary modifications were made to relevant header files and data files associated with 'KinematicLookupTableInjection.C' (like 'kinematicParcelInjectionData.C', and these changes can be found in the appendix.

A small modification was made to 'InjectionModel.C' which made the injection model print the number of particles (as well as the standard number of parcels) to the terminal whilst the solver was running. This addition to the code was useful for knowing on average how many particles were in each parcel. To compare these edited files from their originals the standard linux command `diff` can be used, however the GUI program 'Meld' [71] was prefered due to its ease of use. A screenshot is shown in Figure 3.5, showing the edited file on the left with the original on the right. The highlighted green sections show where lines have been added, other functions can be found on the referenced website [71].

The large modifications made to the file KinematicLookupTableInjection.C and its family were integral to the operation of the new hybrid model. They were necessary since the injection model was created for steady state, invariable injection parameters: whereas the Hybrid model required a dynamic injection model which changed each time step. The changes highlighted above enabled the injection method to be used in conjunction with the rest of the Hybrid model.

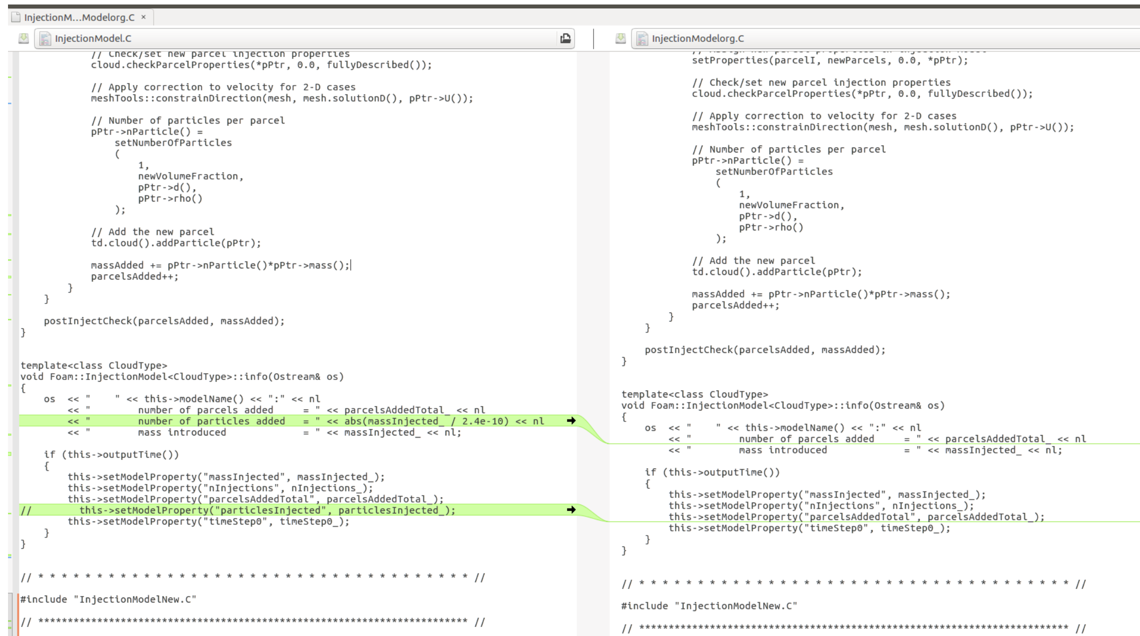


Figure 3.5: Screenshot of Meld- showing comparison of edited and original InjectionModel.C.

3.3.3 Modifications to solver

Modifications to the main solver.C file will be explained in this subsection. After the interpolation from master to slave, the following code was added to update the lookupTable:

```
if ((runTime.timeOutputValue()+0.002) > SOI)
{
        vectorField normalSlaveVector = mesh.Sf().boundaryField()[slave];
        scalarField dotProduct = U1.boundaryField()[master] & normalSlaveVector;
        scalarField uNormal = dotProduct/mag(normalSlaveVector);
        vectorField centres = mesh.Cf().boundaryField()[slave] - (0.00005);


        kinematicParcelInjectionDataIOList& injectors =
                const_cast<kinematicParcelInjectionDataIOList&>
    (
                mesh.lookupObject<kinematicParcelInjectionDataIOList>
                ("kinematicLookupTableInjection")
    );

        forAll(injectors, i)
    {
        injectors[i].x() = centres[i];

        injectors[i].U() = U1.boundaryField()[slave][i];

        injectors[i].numParticles() = abs((alpha1.boundaryField()[master]
        *uNormal[i])/(((((pi)*pow3(injectors[i].d()))/6))
        *(-1)*timestepsPerSecond));
```

```
    }

    injectors.write();

}
```

The top line 'if' statement was added to save computational effort, so that the lookupTable was only being updated when the `SOI` (start of injection) was approaching the runtime. This was useful because often the user converges the problem with only the fluid first, and then adds particles. If this code was not present, the lookupTable would be written without being read up until the injection time, thus wasting computational effort.

`injectors` was defined and then over-writes the object in the mesh called `kinematicParcelInjectionDataIOList`, which points to the data held for the lookupTable. `x`, `U` and `numParticles` are then updated in memory by using the values calculated by the solver in runtime. The memory data is then written to the lookupTable so that if the simulation is stopped, it can be restarted again with the correct initial conditions. This is slightly different from how it worked in the Tutorial, as the model then was directly creating and populating the text file every timestep and expecting the injection model to re-read, whereas now the data in the memory for the model is overwritten directly, and *then* printed to file. It should be noted that one downside of this approach is that the solver does not have the ability to create the lookupTable with the correct amount of columns. This requires the user to look into the polymesh/boundary file and check how many cells the injection baffle has and then create a table accordingly (so if the baffle has 250 cells, there should be 250 columns in the lookupTable). However this is only done once for each mesh, and therefore is not too burdensome.

Another difference is that the 'number of particles per parcel' variable was removed from the equation since this is no longer a fixed value. The idea in the tutorial was to have different numbers of injections per cell with a fixed number of particles/parcel: rather than having one injection per cell with a variable number of particles/parcel as it is now. Each method has its benefits, however it was easier to successfully implement the latter.

The scalar `timestepsPerSecond` was added to the solver file and is defined in createFields to look into the constant/kinematicCloudProperties file. The scalar must be defined by the user, to ensure the solver reads the correct number. This could be updated in the future to calculate `timestepsPerSecond` from 1/deltaT, read from the controlDict.

3.4 Discussion

The creation and development of the Hybrid model and new injection model has been shown in this chapter. The aim was to develop a model which could manage high density slurries and model them accurately. The EE and EL model were coupled together in run time, which is a big advantage over a post processing injection of particles as in Messa *et al.* [4]. This characteristic enabled the Hybrid model to cope with geometry changes and transient flow condition changes, as the particle injections were based upon the EE conditions over the baffle.

3.4.1 Baffle position

The engineer must consider carefully where to position the baffle, as this will affect the performance of the solver and the accuracy of the model. If the baffle is closer to the impingement/erosion surface, the solver will take less computational time as there will be less Lagrangian particles to solve in the domain. The disadvantage of this is that the particles will have less time to disperse before hitting the surface, which results in less of a scatter- and potentially less like the real impact scar.

Overall, the fact that the engineer must position the baffle manually can be seen as a disadvantage as it is an added step the engineer must take before simulation; or it can be seen as an advantage as it gives more control over the whole simulation. There is only one mesh in the Hybrid model solver, and this is to be seen as a big advantage over other similar hybrid models. Having two meshes of differing cell size often becomes a problem with complex geometries. If a large cell sized mesh is used, small features may not be picked up from the geometry. Having one continuous mesh which can be created from any meshing software circumvents these potential

issues.

3.4.2 Injection method

Some more detail on the theory of the injection method will be discussed here, since it is one of the most important parts of the Hybrid model. Firstly, as shown in Figure 3.3 and discussed there, the values of the cells either side of the baffle are the same. This problem seems to be unavoidable, and so for the purpose of this model should be reduced instead of eliminated. Reducing the error can be done by placing the baffle in a location with low velocity gradients through the flow, and by making the cells small either side of the baffle. Creating small cells carries with it some related issues. Firstly, the benefit is that the error of duplicated cells will be smaller in the geometrical sense, since the cells will physically make up a smaller percentage of the domain. Ideally only the length of the cells in the flow direction would be reduced, however this would affect the aspect ratio, and so the length of cells across the baffle face will also have to be reduced. This means that there must be more cells across the baffle: which means greater accuracy, but crucially slower runtime since there are more interpolations taking place (and more cells globally). Since each cell contains an injection site, this also needs to be taken into account. The engineer must choose wisely where to position the baffle and how to mesh around it. The model will be most efficient when the EE section is as large as possible, minimising the region where Lagrangian particles are to be modelled. The solver cannot currently handle 'bounce back' of particles from the wall through the baffle region, and so this is a downside that should be addressed in the future. In the meantime, the baffle should be positioned far enough away from any impingement surfaces which may cause 'bounce back'.

Another aspect to consider is the transition from the Eulerian particulate phase into the Lagrangian parcels. Figure 3.6 shows the transition between EE and EL as in Figure 3.4, however the particulate phase in the first region is shown as the equivalent Lagrangian particles (or what the particles would look like in a pure EL simulation).

Although there are three individual particles with their own velocity vectors, when transferred into the EL region, these three particles are grouped together into one parcel, with one

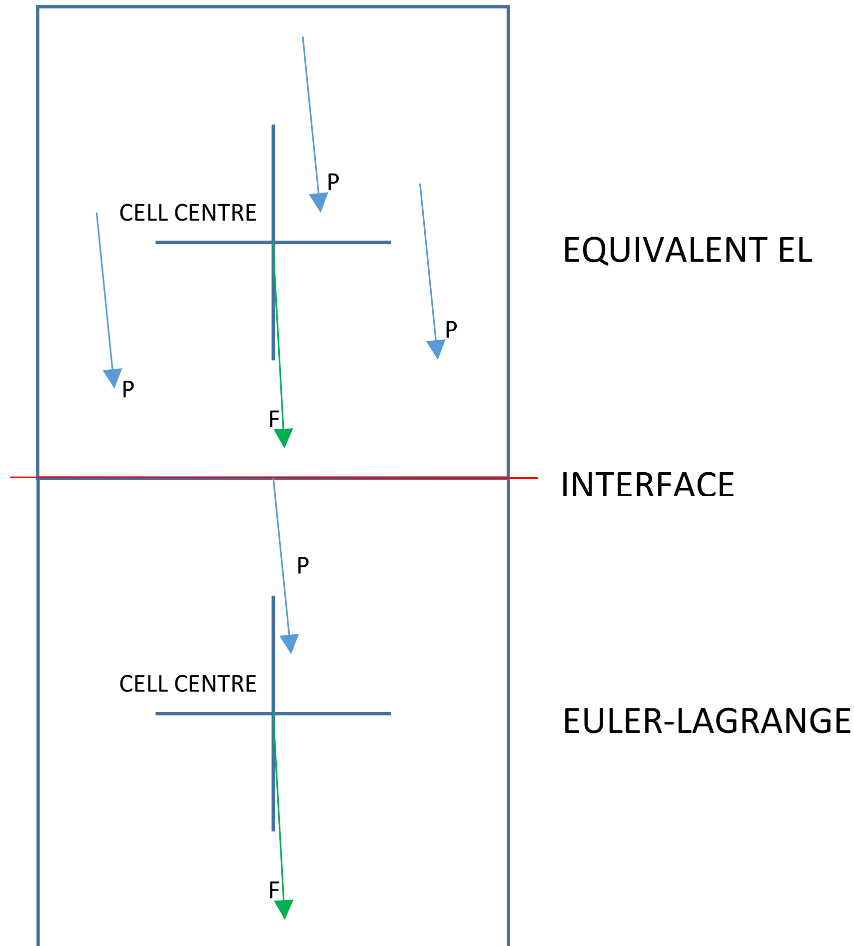


Figure 3.6: Boundary issue.

velocity vector. The Hybrid model therefore cannot give the same result as a pure EL simulation, even from a conceptual viewpoint. This is because of the fundamental fact that at the interface there is a lack of information about each particle's trajectory since they are modelled in the Eulerian reference.

3.4.3 Recent developments in OpenFOAM

In late 2016 a new function object and a Lagrangian injection model was incorporated in OpenFOAM v1612. The function object is called `extractEulerianParticles`, and is in some respects similar to the Hybrid model created in this chapter. Particle data is derived from fluid elements traversing a face zone during multiphase flow calculations, from the Eulerian to

Lagrangian frame. The data is stored as a Lagrangian cloud, with parameters such as velocity and position stored.

One key difference between this and the model described in the chapter is that the function object must be used after the simulation of the Eulerian phase has taken place.

The data from the function object can then be used in cooperation with the injection model `injectedParticleInjection`, to inject the stored data as Lagrangian particles. The success of this model is not known, as the author was not using the version of OpenFOAM which it was available on, and there were no published papers at the time of writing the thesis.

It shows that the contents of this thesis are relevant to the users of OpenFOAM, since similar developments are happening in parallel within the developing community in OpenFOAM.

Chapter 4

Experimental equipment

> The proper method for inquiring after the properties of things is to deduce them from experiments.
>
> Isaac Newton

4.1 Introduction

To validate the solver and gain a better understanding of particle dynamics in slurries, experimental work was carried out. Although there are data available for erosion scars due to slurry flows, there are little on both impact and field velocities. Since it was important for the Hybrid model to accurately describe the flow in the far wall area, and near wall, bespoke experiments were deemed necessary.

An initial rig (Figure 4.1) was created to analyse the submerged jet impingement test, with the results published in a conference paper [35] presented by the author in Australia. The initial design was very simple as shown, however the basic principles were kept and used on a new rig which was set up and shared with Dr Alejandro Lopez, a (then) PhD student at the Weir Advanced Research Centre. This rig had water as the working fluid, and a small pump to feed the header tank with water. Seeding particles were added to the header tank, and a valve was opened to allow water from the header tank to impinge on the sample in the tank below. Whilst this was done, the laser illuminated the seeding particles to allow PIV analysis to take

place.

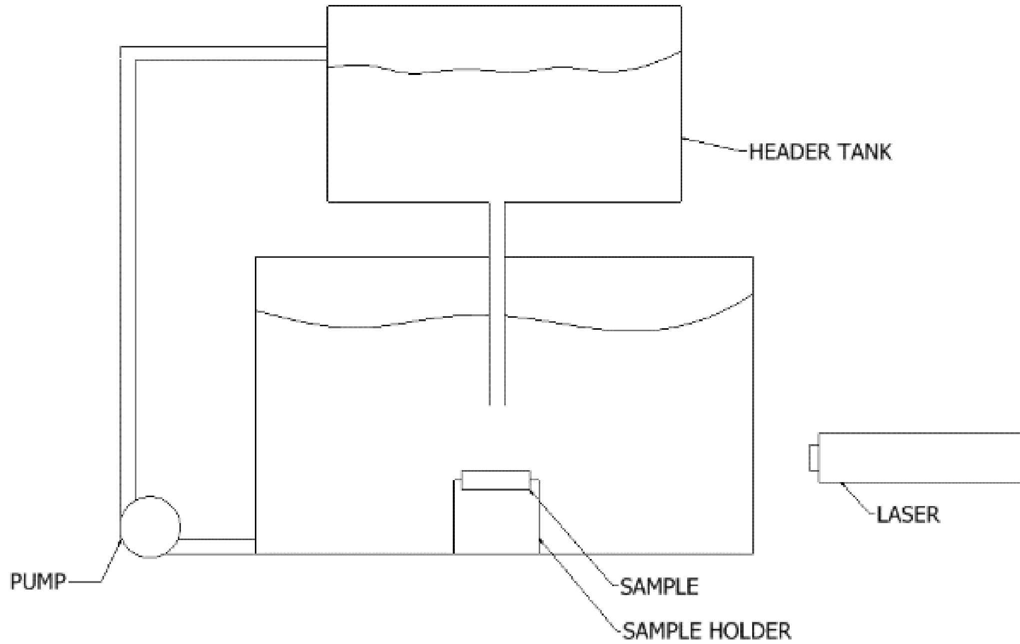


Figure 4.1: A 2D schematic of the initial test rig

The main major improvement on the new rig was using a pump to pressurise the nozzle rather than relying on gravity to energise the flow. The drawback of using a header tank with no pump (as in the initial rig) was the limited range of low outlet velocities attainable.

The new rig was designed and built to examine flow characteristics and carry out flow validation work. This chapter will explain the principles of Particle Image Velocimetry, the design of the new rig and the hardware and software used.

4.1.1 Particle Image Velocimetry

As mentioned, PIV is one of the best ways to gather data about the flow field, both for the continuous and the dispersed phase. The principle is to compare two images, taken within a short period of time, of the flow field to ascertain the velocity. The velocity of a single particle can be calculated since the distance and time between images is known: **t** and **t'** in Figure 4.2. This can be done for all particles in the area of interest, enabling the velocity of the flow field

to be established. Tracer (or seeding) particles which have the same density as the fluid and have a small diameter (low Stokes number) can be added to analyse the continuous phase flow field whereas the dispersed phase can be measured directly with larger particles.

The laser light sheet is present for two main reasons. Firstly, it provides a near-2D plane (it is less than 1mm thick) for analysing the data. If there was no 2D sheet, the camera or imaging optics would pick up every particle which was in view or in focus. This would make validation difficult as the user would not know where they were taking data from. The second reason for using a laser is to create a sheet bright enough for the imaging equipment (camera in this case) to pick up. The images need to be close enough together to contain the same particles for analysing, which requires short frame rates, meaning the sensor does not have much time to be subject to light. Laser light is capable of high power, it can be easily formed into a thin light sheet and it is monochromatic: for these reasons it is frequently the light source of choice for PIV.

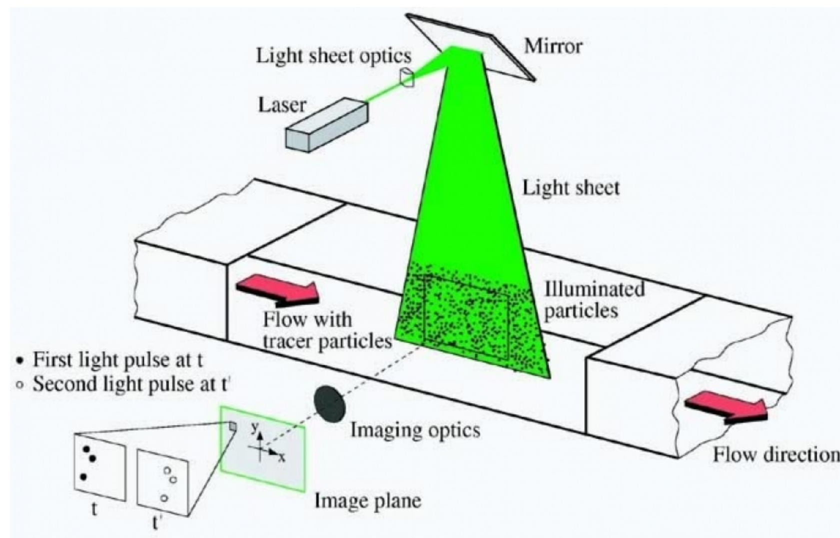


Figure 4.2: An example of an experimental setup for PIV in a wind tunnel [6].

Various recording techniques for PIV were used, one of which was frame straddling. Frame straddling is used when the normal two frame/single exposure method cannot capture the flow due to the time step between the two images being too large. One way to get around this is to pulse the laser twice within the same frame. This creates a 'double exposure' which needs to

be analysed using auto-correlation. One issue with auto-correlation is that the user/software cannot tell whether a particle is from the first pulse or from the second, and therefore the general direction of flow needs to be known *a priory.* Frame straddling still uses two frames, but decreases the time between laser pulses.

The camera used was set on its highest frame rate which could be used at maximum resolution(500fps at 1024x1024 resolution) and so one of the only ways to get the two images closer together is to strategically time the pulse of the laser. Figure 4.3 shows what takes place: immediately before Frame 1 closes the laser pulses, and immediately after Frame 2 opens the laser pulses again. As can be seen, the two pulses of the laser are much closer than the middle of Frames 1 and 2 (which would be the time step without this technique), and since the images will be recorded when the laser illuminates the particles there is a much smaller time step achievable. In the University, the time step was brought from 2000μs to 67μs, theoretically allowing velocities with four times the magnitude to be captured. The downsides of this technique are that the laser and camera equipment need to be coupled together via a synchroniser to ensure proper straddling, and laser power regulation needs to be taken into account. If the power was not regulated, the first pulse would be much brighter than the second one, as the laser has more time to charge [72]. If particles are travelling out of the plane of the laser, the PIV results will give deceptively low velocities. This is because the PIV software assumes a 2D flow field, whereas the distance measurements will be measuring the components of vectors rather than the true vector. Therefore, care ought to be taken to ensure that the laser pulse is long enough to ensure the particles are illuminated sufficiently, but short enough not to capture those particles with an out-of-plane velocity component. The width of the laser beam should also be kept to a minimum to reduce this phenomenon.

For an in-depth review of PIV, the book 'Particle Image Velocimetry' [73] is recommended.

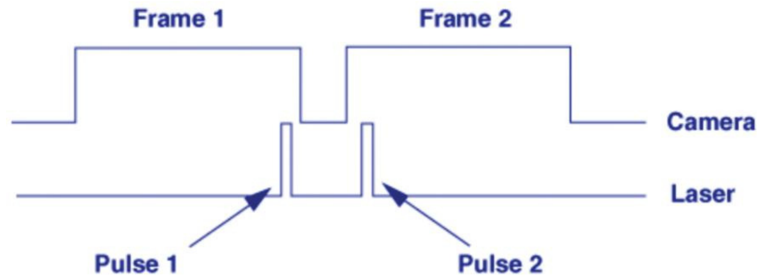


Figure 4.3: Frame straddling technique [7] (time along x-axis).

4.2 Design of rig

The usefulness of the submerged jet impingement test (JIT) was already discussed in 2.4.1, which is why this was the type of experimental set up chosen for validation work. Other authors' designs were taken into account during initial stages. The main aims/requirements of the rig were that it should:

- allow PIV to take place on different geometries in the submerged JIT
- have the ability to inject seeding particles and sand particles
- be easy to use
- allow the user to vary the flow rate of the jet
- allow quick changing of the samples between tests

The seemingly obvious solution to increase jet velocities was to use a pump to increase the pressure before the nozzle, however since Lopez was using the rig for erosion purposes there was a strong preference that particles did not pass through the pump as they would erode it. Bypassing the pump meant that it lasted longer, and that the particles retained their shape/roundness.

The author and Lopez tried various strategies to get around this problem including designing different ejector pump suction sections, the last of which is shown in Figure 4.4. This brass example was the last iteration of 3-4 different venturi sections. It was split along the lateral axis to allow easy manufacture of the inside and to allow the downpipe to be inserted. A gasket

was inserted between the two faces and a series of nuts and bolts were used to crimp the two sections together. Not only were different designs tried with different geometries, but also a range of materials and manufacturing techniques. The first one for example was 3D printed from plastic in two sections (again along the lateral axis) and glued together. 3D printing was chosen as it allowed more flexibility at the design stage, allowing a very small convergent diameter which would not have been manufacturable by using traditional lathes. The downside of the 3D printed model was that the printing method left some porosity in the material and when pressurised, the ejector pump leaked.

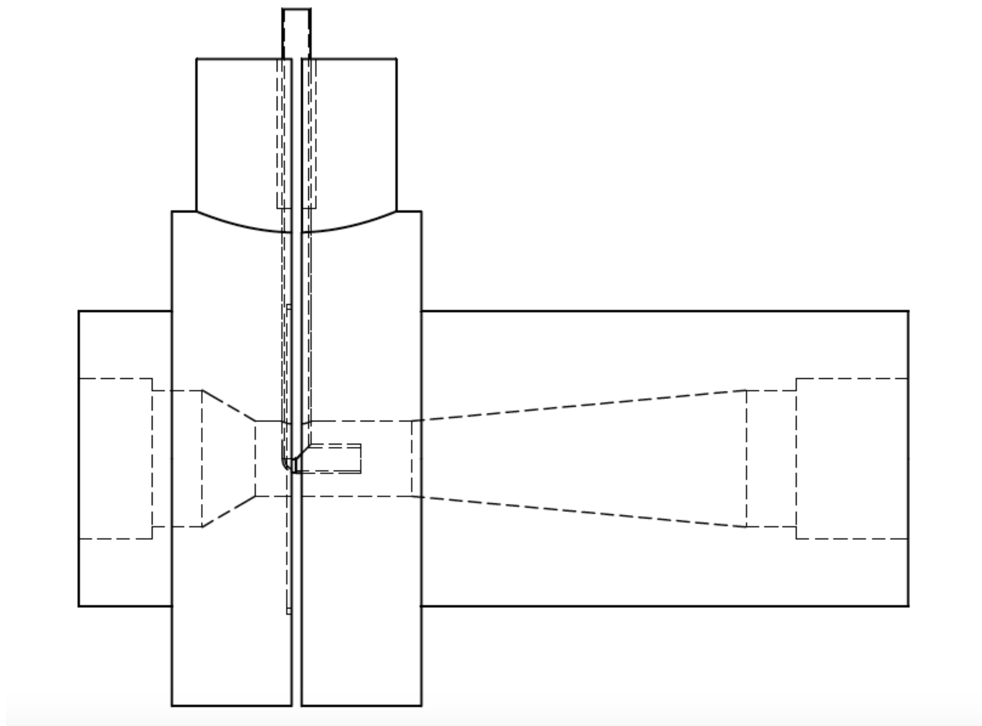


Figure 4.4: Detailed drawing of ejector pump, flow left to right, showing particle feed coming from the top [8].

The idea was to have the venturi positioned downstream of the pump to introduce particles to the system, thus bypassing the pump. The contraction in the venturi would increase the flow rate of the fluid and following Bernoulli's principle this would simultaneously decrease the pressure. If this pressure was less than atmospheric, particles could then be sucked up into the system. The problem was that the contraction from the main pipe diameter to the outlet pipe diameter created a large back pressure, which increased the pressure in the venturi; so

that instead of sucking, the inlet pipe of the venturi ejected water. If the injection nozzle was removed from the end of the system, the venturi worked as intended confirming the theory. Solutions were attempted by changing the geometry of the nozzle exit and by changing the venturi parameters, however no configuration was successful. The methodology in a commercial abrasive water jet cutter was examined with the idea of replicating the technique, however it was found they use very high nozzle velocities, and their venturi/contraction is in the same location as the outlet nozzle (pressures can be as high as 6000bar resulting in velocities of Mach 4 [74]) meaning they do not have the same issue with back pressure.

Following this, the decision was eventually made to use a smaller sacrificial pump for the sand (erodent) particles. This kept the main pressure supplying pump in good condition, and still allowed sand particles to be used. The issue of introducing particles to the system without putting them through a pump still seems to exist in the literature [2], however as experimentation times were small the issue of pump wear was not an issue in this study.

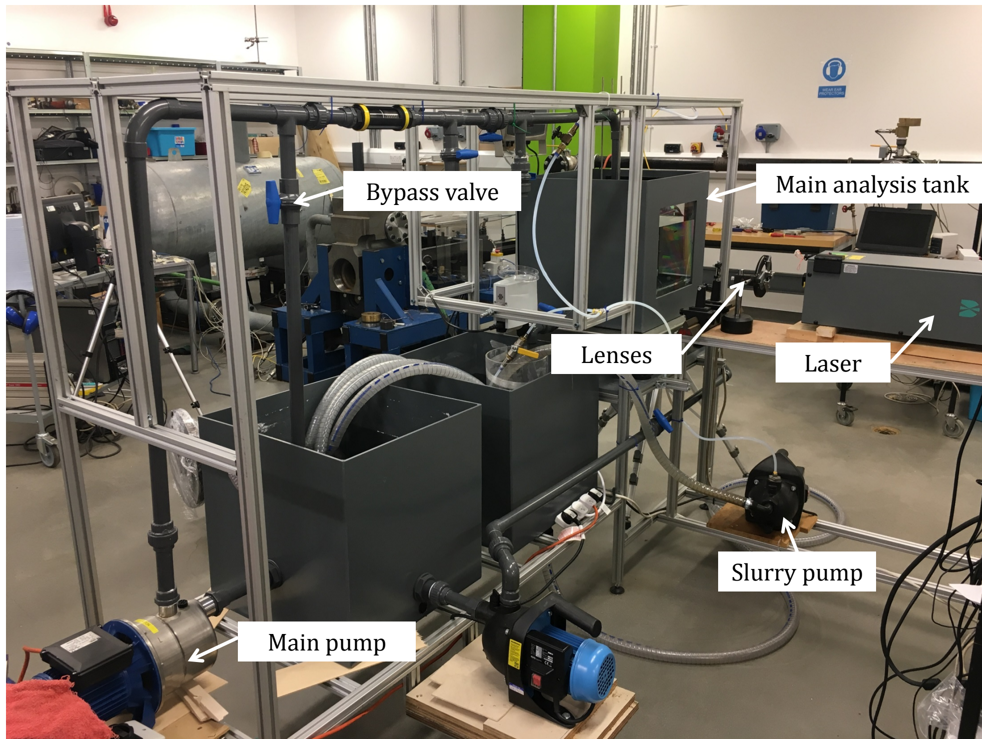


Figure 4.5: Test rig: view from left.

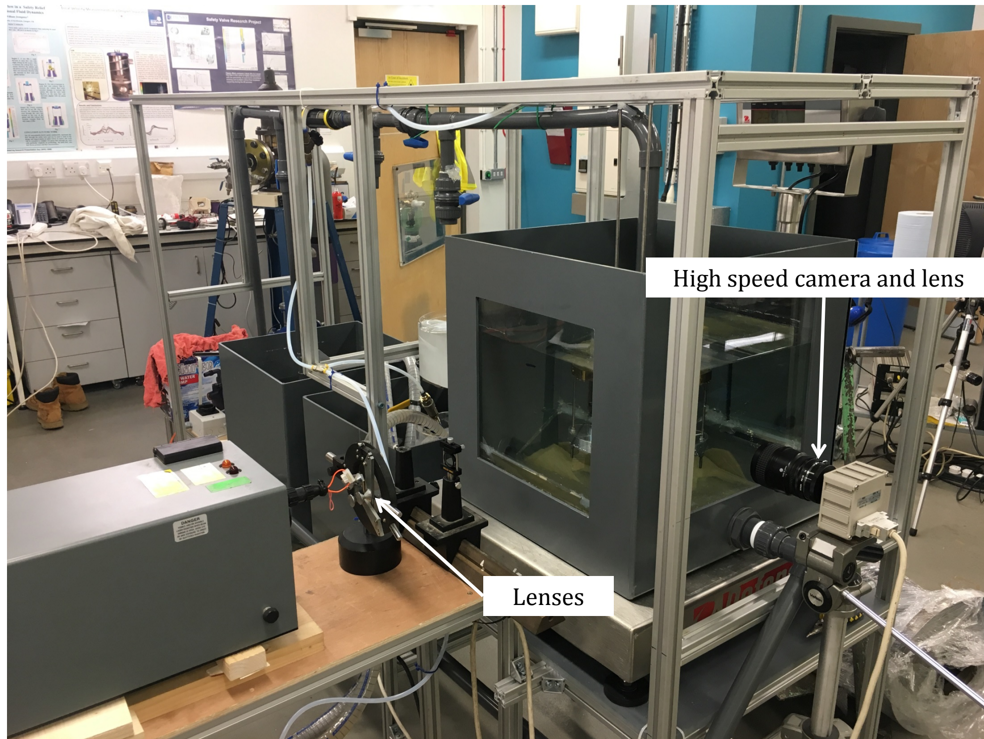


Figure 4.6: Test rig: view from right.

Figures 4.5 and 4.6 show the test rig viewed from two different angles: Figure 4.5 showing the pumping section and Figure 4.6 showing the optics and camera.

The main pump takes water from the closest tank in Figure 4.5, pumps it along the top pipe and down into the analysis tank through the custom made nozzle. The water is then gravity fed into the middle tank, and back to the closest one through a straight section of pipe. The flow rate to the nozzle can be controlled by the bypass valve which directs water back into the closest tank. All pipes are 32mm diameter with the nozzle exit diameter of 9mm. A conical piston flow meter was used downstream of the bypass valve, however debris in the system continued to jam the flow meter which resulted in false readings, rendering it useless. For future work it is recommended to use an external flow meter, or one which is less sensitive to small debris in the flow. A weighing scale was used underneath the main analysis tank, as shown in Figure 4.6, to allow the flow rate to be calculated gravimetrically. This was done by closing the outlet to the tank and measuring the increase in mass over a set period of time.

When particle tracking was required the slurry pump was used in combination with the main pump to add sand particles to the loop. Sand was submerged in the middle tank, sucked out with the slurry pump, then injected into the pipe bend above the nozzle. The stainless steel pipe inserted into the PVC pipe is detailed in Figure 4.8, where it can be seen inserted at an angle to avoid the frame directly above the radius of the pipe. The steel pipe was inserted into the radius far away from the nozzle exit so that the sand had time to distribute evenly throughout the pipe and so that the slurry addition didn't affect the jet velocity distribution. An earlier design (Figure 4.7) had the slurry pipe joining much closer to the nozzle exit, which after being analysed with CFD (Fluent) showed to give an asymmetric jet profile, which is not good for PIV analysis: especially for non-flat samples, since only one side can be analysed. (Note: Figure 4.8 is shown to illustrate the asymmetric jet profile only)

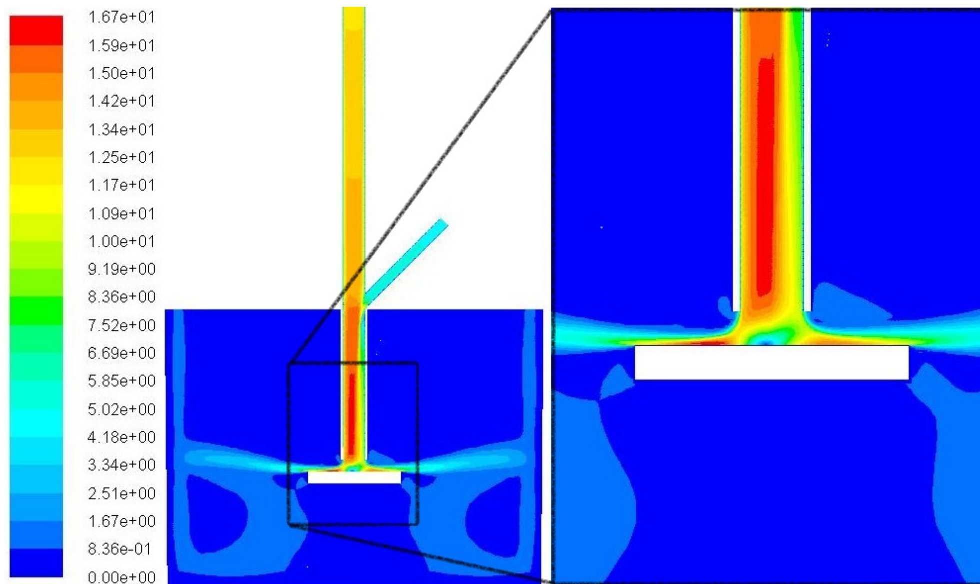


Figure 4.7: Velocity contours (m/s) of original slurry pipe location showing asymmetric impingement velocity profile.

Figure 4.9 shows the nozzle, straight pipe section, sample, and sample holder: all designed and manufactured in the university. The reducing nozzle was manufactured from brass and reduced the diameter from 32mm to 9mm over a distance of 110mm, providing a smooth acceleration profile for the flow. A straight copper pipe was positioned downstream, and was more than 10 diameters (an accepted industry rule of thumb) long to ensure flow was settled

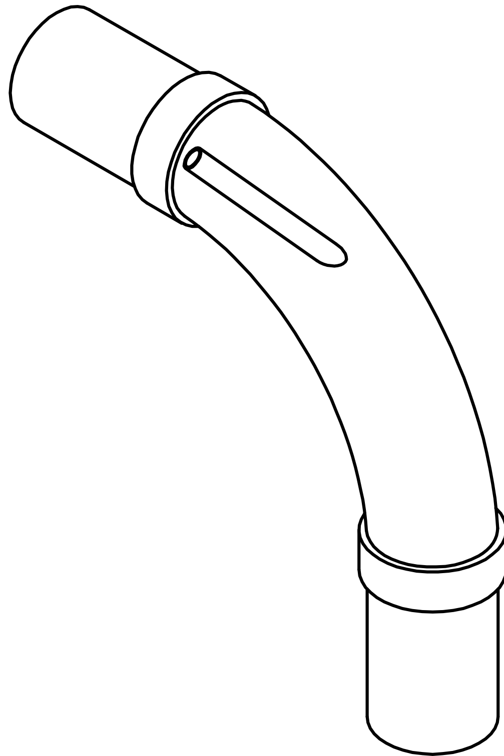


Figure 4.8: Pipe bend drawing showing steel slurry pipe insert.

and established before exiting and impinging on the sample.

The analysis tank was designed with the purpose of carrying out PIV. It was manufactured from 5mm thick PVC, fitted with two large glass panels which were chosen for their optical properties. Clear perspex was initially going to be used, however perspex carries the risk of melting if the laser is incorrectly focused into a thin sheet when passing through the perspex. The analysis tank was raised above the other tanks for two reasons: firstly, so that it had some 'head pressure' available to empty it, and secondly so that the analysis could take place at a convenient height for the users. This saved bending or stooping when changing samples, and also provided an easier view when aligning the laser. Keeping the laser below eye level when standing was also a requirement, as this minimised the risk of accidental exposure, which could damage the retina of the eye.

4.3 Samples and sample holder

Three different shaped aluminium samples were manufactured for PIV analysis; a cylinder, cone, and hemisphere (see Figure 4.10). The author realised there was a gap in the literature for different shapes as typically it is only the cylinder which is considered in the submerged JIT [2,26,53]. One possible reason for this is that papers on the JIT are usually only concerned with erosion work and the flat impingement surface which the cylinder provides is ideal for surface scanning equipment. Since the flow field is the main focus in this work, there was not the same requirement for the flat surface. A downside with the hemisphere and cone is that only half of the light sheet over the sample can be analysed, since the geometry obstructs the other half from being illuminated. This can be seen in Figure 4.11, where the darker green depicts the shadow cast by the geometry.

The original sample holder was designed only for cylindrical samples which would be subject to testing for a long period of time. Experiments commenced on the various different shaped samples bringing simulation times to only a few minutes, and so the ability to change samples quickly became critical. A new sample holder allowing easy access and expeditious changing times was therefore designed and manufactured.

Figure 4.12 and 4.13 show the old and new holders respectively. The main problem with the old one was the depth of the seat that the sample was sitting in was greater than the distance between the top of the samples and the nozzle. This meant that the base of the holder had to be lowered by using the 4 nuts on the bottom of the assembly. These nuts were difficult to access and raising the bed back up also required the use of two hands.

The new design had a larger eccentric hole, with a retaining bolt to hold the sample in place. Two cover plates, manufactured from perspex, were added over the hole to reduce any adverse effect on the jet impingement surface as there was an asymmetry in the base created by the eccentric hole. The process of changing sample was much simpler and quicker: first, the top two nuts (the closest two in Figure 4.13) were loosened and the perspex cover removed; then the retaining bolt was loosened, leaving the sample free to slide out. This redesign saved a lot

of time in the experimental laboratory. More detailed drawings are in Appendix A.

4.4 Hardware and software used

A brief summary will be given of the equipment and software used for the experiments carried out in this project.

4.4.1 Hardware

Other than the test rig, the main components of the PIV setup were the laser and optics, the tracer and sand particles, the camera and lens, the PC and synchroniser, and weighing scales.

Figure 4.14 shows the components and how they are connected in a standard PIV set up. The high speed camera used was the *FASTCAM ultima1024*, from Photron LTD. [75]. The experiment and Figure 4.14 are slightly different since the synchronizer is a *National Instruments timer card* located inside the PC, not an external piece of hardware.

The images were initially taken by a camera using a 60mm f/2.8 prime lens with an extension tube. This was positioned 600mm back from the laser sheet to enable correct focusing, however there were two problems: 1) the area of the image taken was too large for accurate PIV analysis, and 2) the images were too dark. As mentioned previously, the laser cannot produce as much power on its second burst due to the short charging time, which limits the power of the laser considerably, making the images too dark. To get around this issue, a prime zoom lens (105mm f/2.8) was purchased by the author, and it was positioned 300mm from the laser sheet. Calculations were necessary beforehand to ensure the correct lens was purchased. Light intensity is described by the inverse squared law and so moving the camera in to half the distance the intensity/brightness increases fourfold, and with the zoom lens a smaller area was captured for post processing accuracy.

The laser used was a *LDP-100MQG Nd:YAG* manufactured by Lee Laser inc. [76], emitting light with a wavelength of 532nm and a pulse energy of 5mJ per pulse at 10kHz. The laser outputs a beam with a circular cross section, and so for this to be turned into a light sheet

a series of lenses needed to be used. A sheet with rectangular cross section of thickness 2mm was first made by using a planar convex lens, and then thinned into a sheet with less than 0.5mm thickness by another planar convex lens (f=100mm). This sheet was thin enough to only illuminate the particles travelling in the plane-wise direction.

Standard seeding particles with size $50\mu m$ and specific gravity of 1 were used to analyse the liquid phase. For the erodent, 500μm, diameter frac sand was used, with specific gravity of 2.65. Frac sand is primarily used in hydraulic fracturing due to its size, shape and mechanical strength, and it was used here due to its spherical shape and the quality of the diameter distribution as shown in Figure 4.15.

Finally, the main tank where the PIV took place sat on a measuring scale. This was to measure the flow rate through gravimetric analysis, thus having a reference value for the PIV.

More information and background about the hardware (of the laser) can be found in the paper by Stickland *et al.* [72].

4.4.2 Software

The two pieces of software used for the experiments were for the camera, *Photon FASTCAM viewer* and the laser pulse delay program for frame straddling and data post-processing, *DynamicStudio 5.1* [77]. A small piece of software from Dantec using the *National Instruments* card synchronised the pulses of the laser and the frame rate of the camera, which allowed the user to control the frame rate of the camera and the delays for the laser pulses, required for frame straddling. The *FASTCAM viewer* was used to visualise the images and record the data.

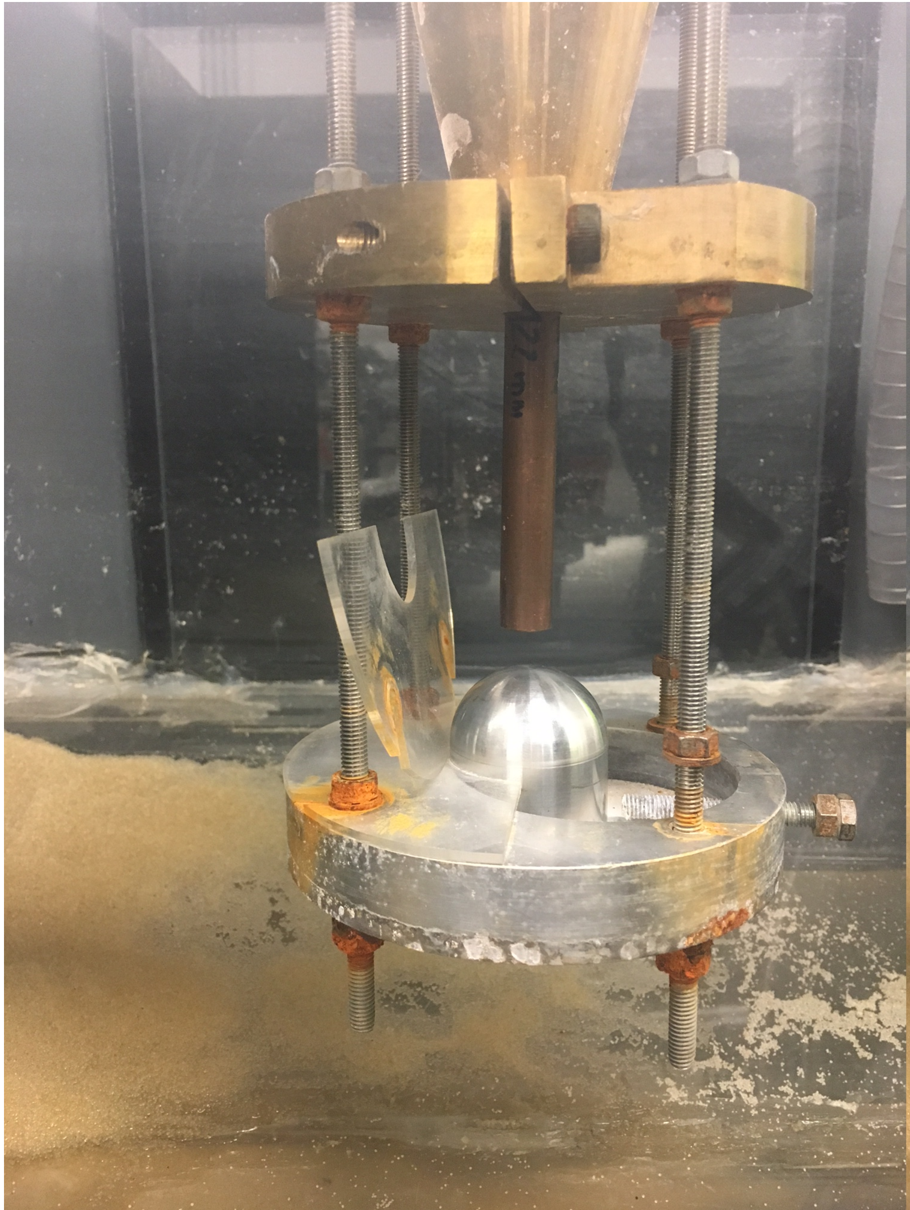


Figure 4.9: Test rig: nozzle and new sample holder.

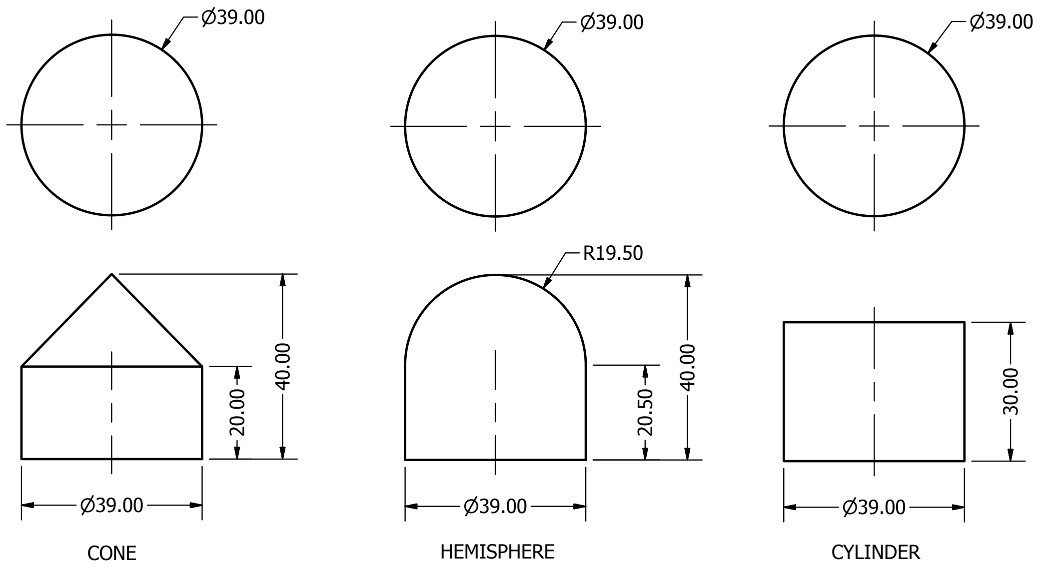


Figure 4.10: Samples manufactured for testing (sizes in mm).

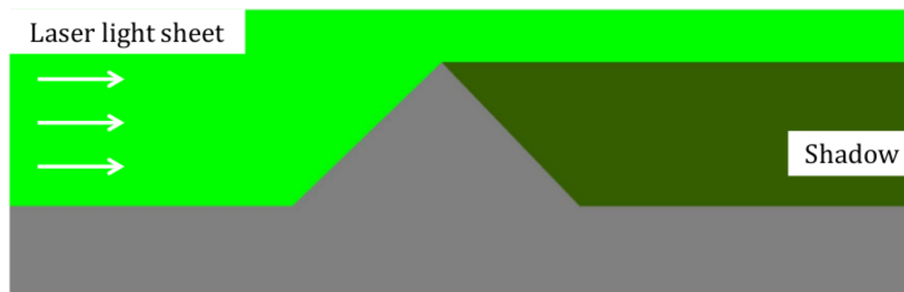


Figure 4.11: Shadow created by cone.

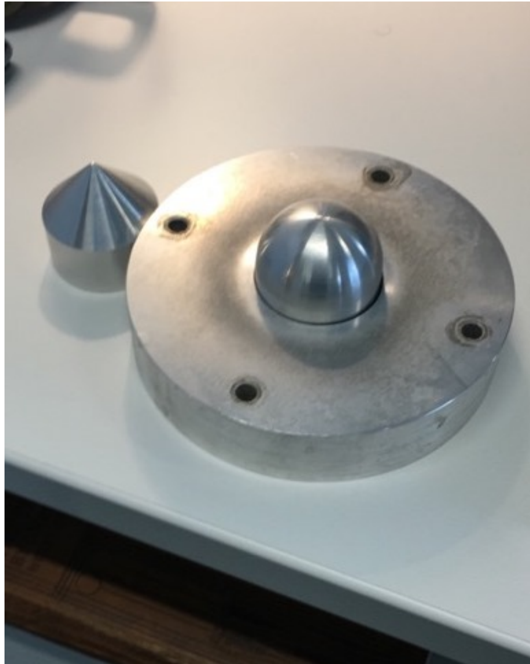


Figure 4.12: Original sample holder base.

Figure 4.13: New sample holder assembled.

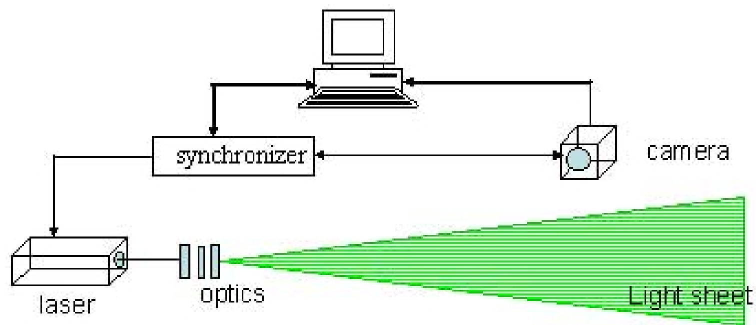


Figure 4.14: Basic PIV setup [9].

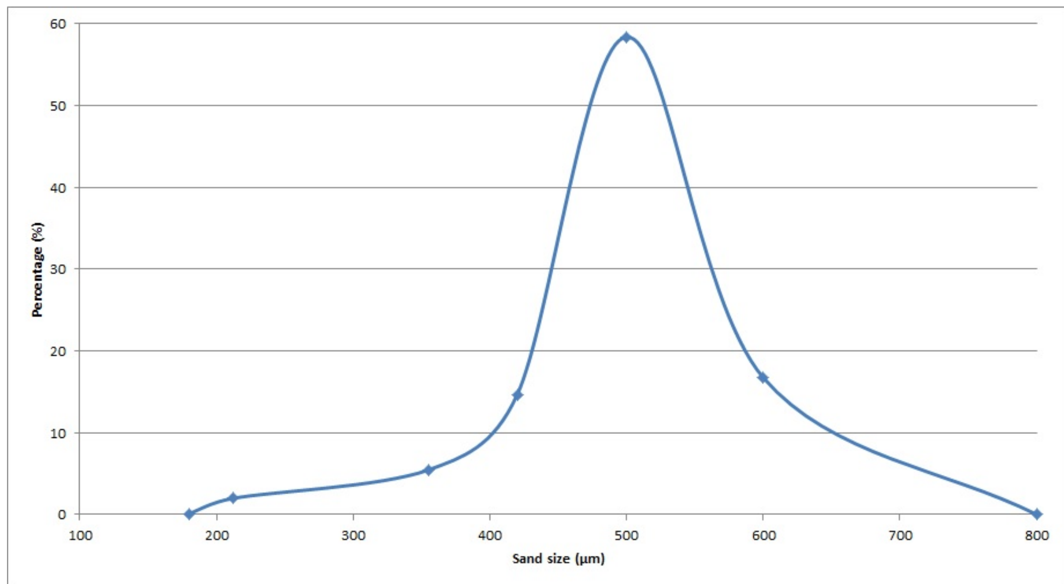


Figure 4.15: Frac sand size distribution [10].

Chapter 5

Results and validation of CFD model

People love chopping wood. In this activity one immediately sees results.

Albert Einstein

This chapter will present the most relevant results which were obtained with the experimental rig and from the CFD code. The solver was tested throughout the development which took over one year, with the results and related issues published in various conference proceedings, an online tutorial and in a chapter of the 11th OpenFOAM workshop book [35,60,67,78,79,80].

5.1 First phase comparison of standard models (Jet Impingement Test)

The first paper from the CSIRO conference in Melbourne, Australia (2015) [35] compared three different CFD software packages' (ANSYS Fluent, Star-CCM+ and OpenFOAM) abilities to model the fluid phase of the submerged jet impingement test. This paper was presented by the author of this thesis as his first conference paper. The standard k-epsilon turbulence model was

used in all packages and the k-omega SST (shear stress transport) was tried in Star-CCM+ since various papers used it to model the JIT [81, 82]. CFD results were compared against an experimental test, using PIV data from the initial test rig which was introduced in the previous chapter.

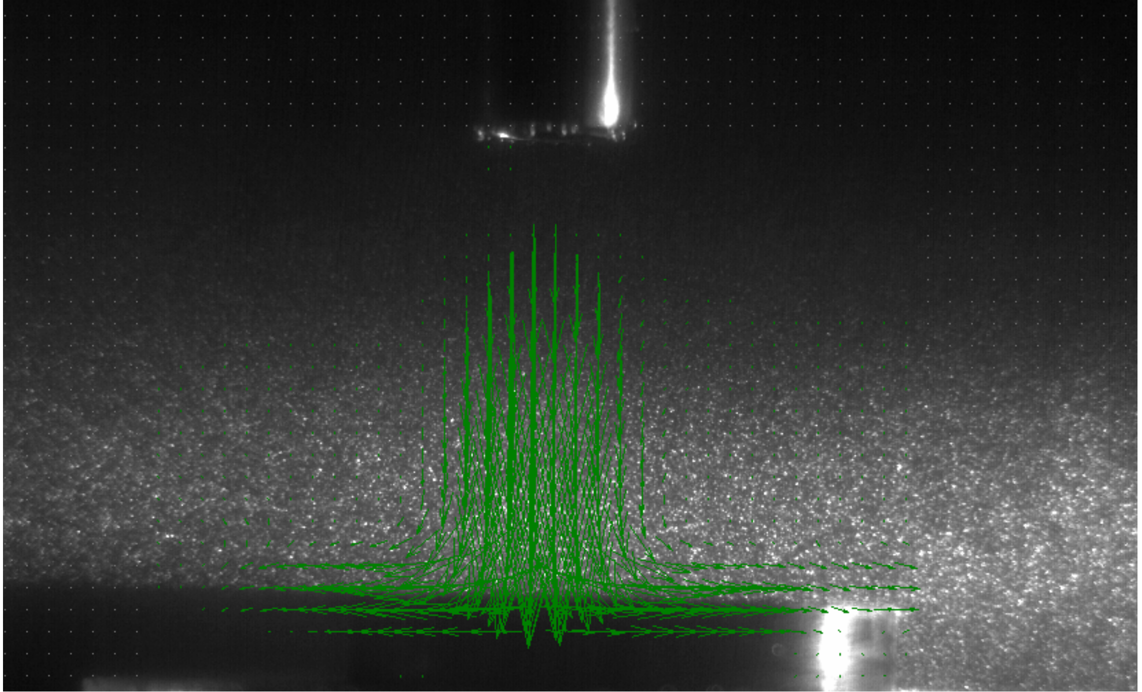


Figure 5.1: Averaged vector plot from PIV software: Flow Manager.

Figure 5.1 shows the average velocity vectors of the fluid from the PIV software. The fluid was seeded with $20\mu m$ diameter particles, and specific gravity of 1, which resulted in them following the flow (low Stokes number). The camera was set to 500fps, and the frame straddling technique was used to lower the ΔT to $100\mu s$. The pipe can be seen at the top of the image, with the cylindrical sample below (laser is shining from right in the image). The green vectors depict the averaged flow field vectors from 50 sets of images, with a vector range of $\pm 3m/s$.

Figures 5.2 and 5.3 below show the radial and axial velocities of the different solvers, and Figure 5.4 gives the velocity magnitude. PIV data was taken 1mm above the surface of the sample, and a line was drawn in the CFD simulations in the same location to compare results. The blue line represents the experimental data set, taken from the software 'FlowManager'.

As shown, the CFD simulation result that resembles the experimental PIV result in regards

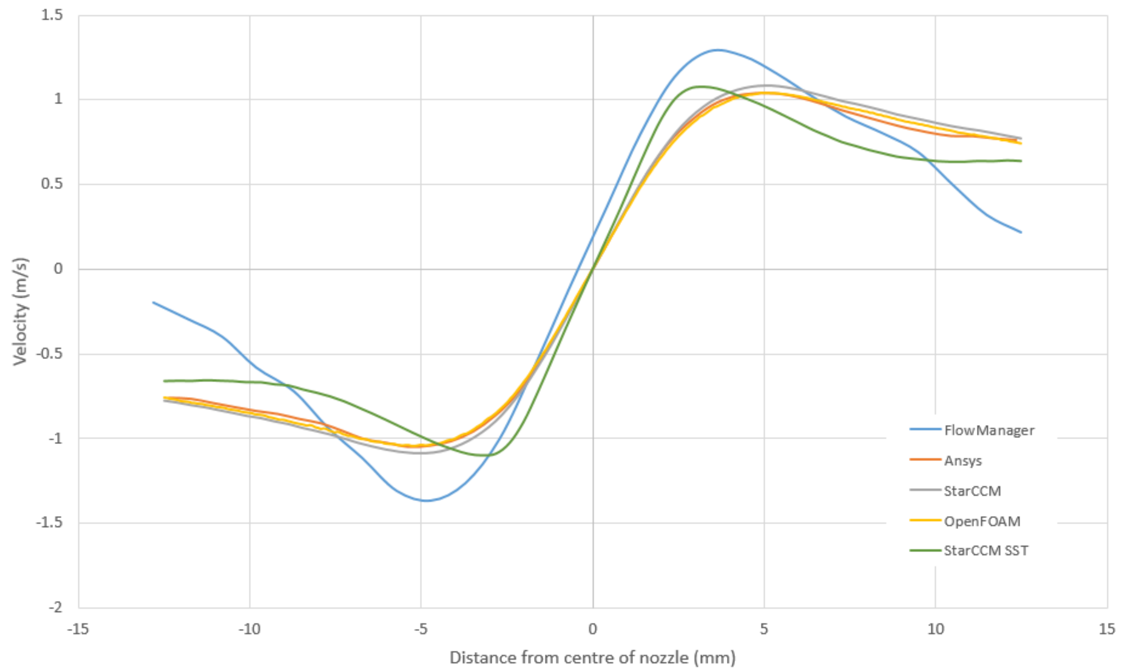


Figure 5.2: Radial velocity 1mm above surface.

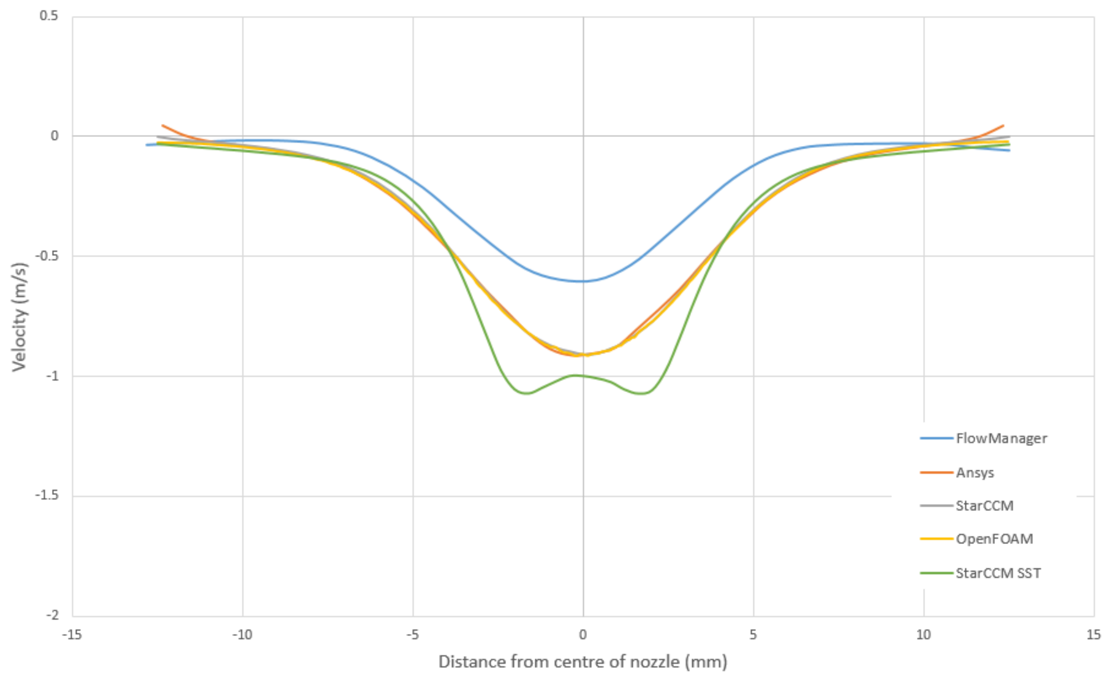


Figure 5.3: Axial velocity 1mm above surface.

to the location and magnitude of the peak velocities the most is the k-omega SST StarCCM result, coloured in green. The results in the conference paper by the author also showed lit-

tle discrepancy between the different software which proves that although OpenFOAM has no licence costs and is open source, it is no less accurate than its commercial counterparts. Following this, OpenFOAM in combination with the k-omega SST turbulence model was selected for future work on the submerged JIT. The paper also showed that care needed to be taken to ensure CFD is accurately picking up the fluid phase flow, as the velocity of particles can highly depend on this. This idea is discussed in the paper by Gnanavelu [53], where the differences between particle and fluid trajectories are contrasted. Both turbulence models and all three software packages manage to capture the general trend and shape of the graph well. More information can be found in the conference paper [35].

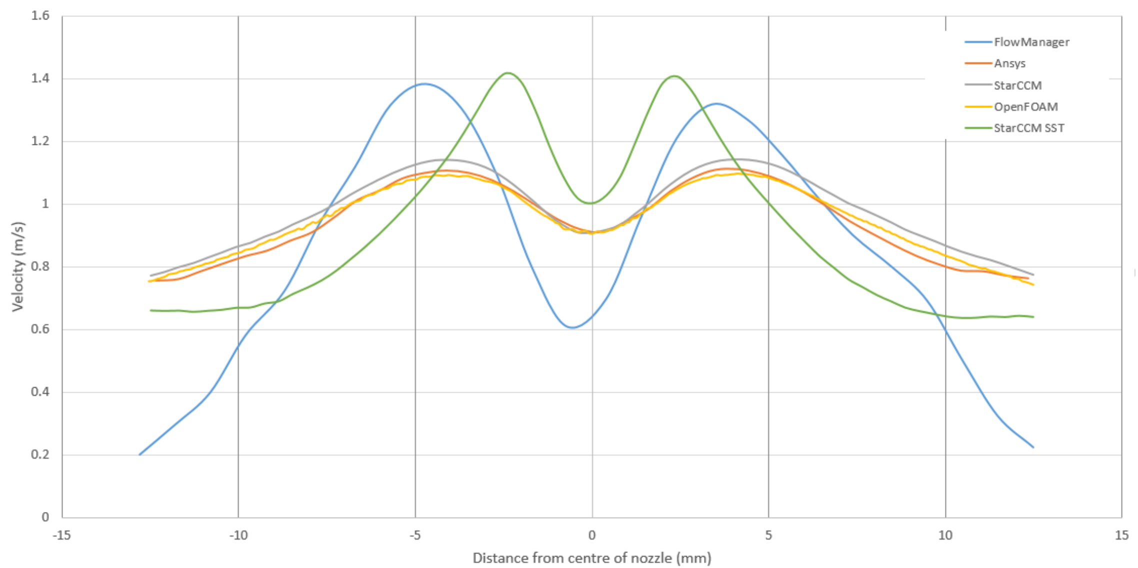


Figure 5.4: Velocity magnitude 1mm above surface of cylinder on JIT from CSIRO conference.

5.2 Results from the Hybrid Model Tutorial (Pipe bend)

Although the model was not complete at the time of writing the tutorial published in Chalmers University [60] by the author, the initial results are shown below to show there were signs of progress at an early stage. If the concept of a Hybrid model did not work on a simple geometry, further work would not have carried on.

The boundary conditions for the case were:

Parameter	Value
alpha.particles	0.039 (10% mass concentration)
alpha.water	0.961
epsilon.water	51
k.water	0.36
nut.particles	0 (no turbulent viscosity)
nut.water	0
p	1e5 Pa
Theta.particles	1e-7 (granular temperature)
T.particles	300 Celsius
T.water	300 Celsius
U.particles	5 m/s
U.water	5 m/s

The y+ was not considered important at this stage, as the purpose of the simulation was not to produce accurate/realistic results, but more to prove the concept worked. Convergence criteria was 10^{-3} for all simulations in this thesis.

The solver was run on the mesh shown in Figure 3.1 and compared against the standard EE and EL solvers, `reactingTwoPhaseEulerFoam` and `DPMFoam`. Lagrangian particles were injected from the inlet in the DPMFoam solver, and the Hybrid model was split in the same manner as in Chapter 3. All solvers were run from 0-0.39 seconds, with the start of injection (SOI) at 0.29 seconds. The same mesh was used and the same boundary conditions were set for each solver as far as possible, and two different mass concentrations were simulated; 1% and 2%. Table 5.1 below shows the execution times of the three different solvers with the different mass concentrations. As expected the EE model was the fastest. The EL model was the slowest, with the Hybrid model taking half the time to solve as the EL. This was a good indication that the Hybrid model was a solution to reduce long execution times. It should be noted that the solver was not running properly at this stage, as the particle injection values did not change over time. This was corrected later on, so that the particle injection values were updated every time step, which resulted in a small penalty in execution time.

Model	**Execution Time (s)**	
	1% MC	2% MC
Hybrid model	225	298
Euler-Lagrange	420	585
Euler-Euler	102	105

Table 5.1: Comparison of execution times from 0-0.39 seconds with different mass concentrations (MC).

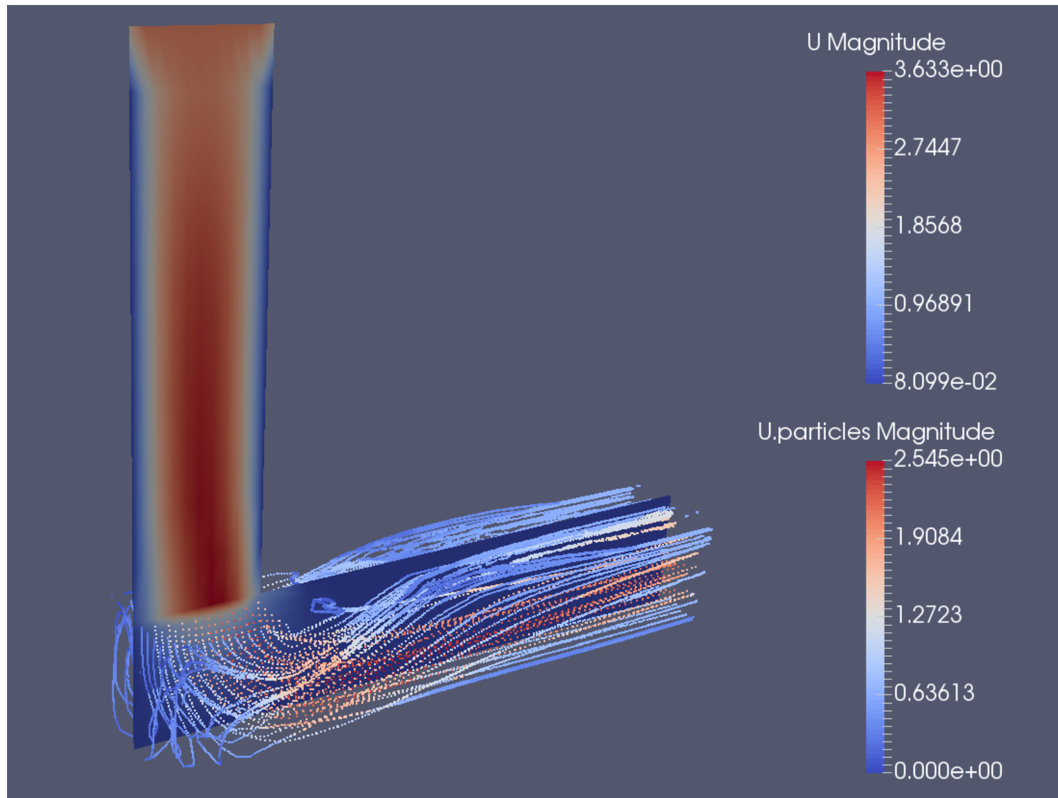


Figure 5.5: Hybrid model showing 2D slice of 2nd Eulerian phase velocity contours, and Lagrangian particles in second region: both coloured by velocity magnitude.

Figure 5.5 shows the results of the hybrid model at the final time step. The fluid domain has been cut into a 2D slice, while the Lagrangian particles have been left whole. The transition can be seen at the bottom of the vertical section of pipe, where the solver transitions from EE to EL and particles are introduced (in place of the second Eulerian phase). The velocity of the Eulerian particle phase can be seen to stop at the baffle, since there was no Eulerian particle phase in the bottom region. This image can be compared with a similar 90 degree pipe bend in the paper by Mansouri [33]. Mansouri's pipe was circular in cross section, and the velocities were much higher than the velocities simulated here.

As well as checking execution times, the particle impact locations were compared between the EL model and the Hybrid model. A file called 'PatchPostProcessing.C', a cloud function, was edited, recompiled, and then used to give the data in Figures 5.6 and 5.7. This function records the position of the particles each time they strike the bottom surface of the pipe. The

Z direction (width of the pipe) is shown on the vertical axis, and the X direction shown on the horizontal axis, both in metres.

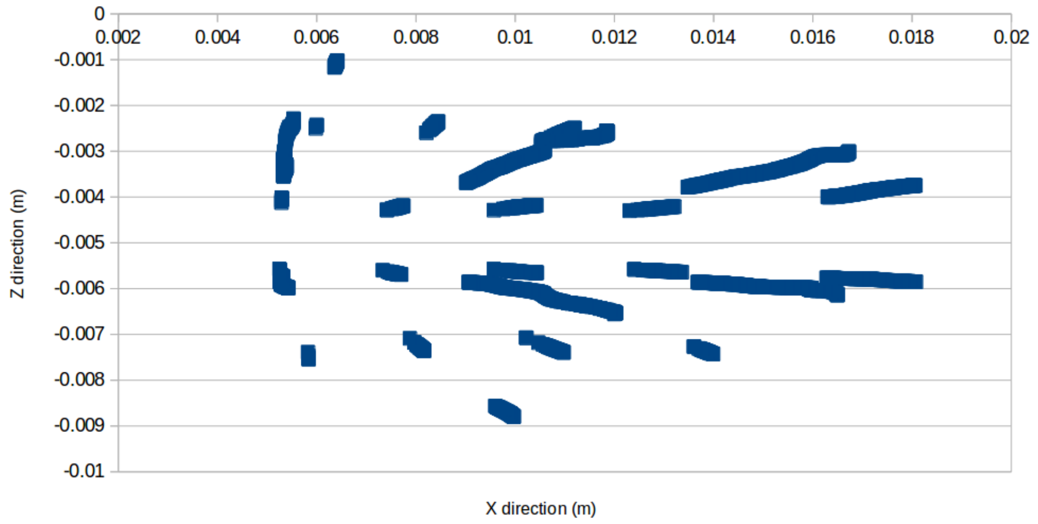


Figure 5.6: EL particle impacts.

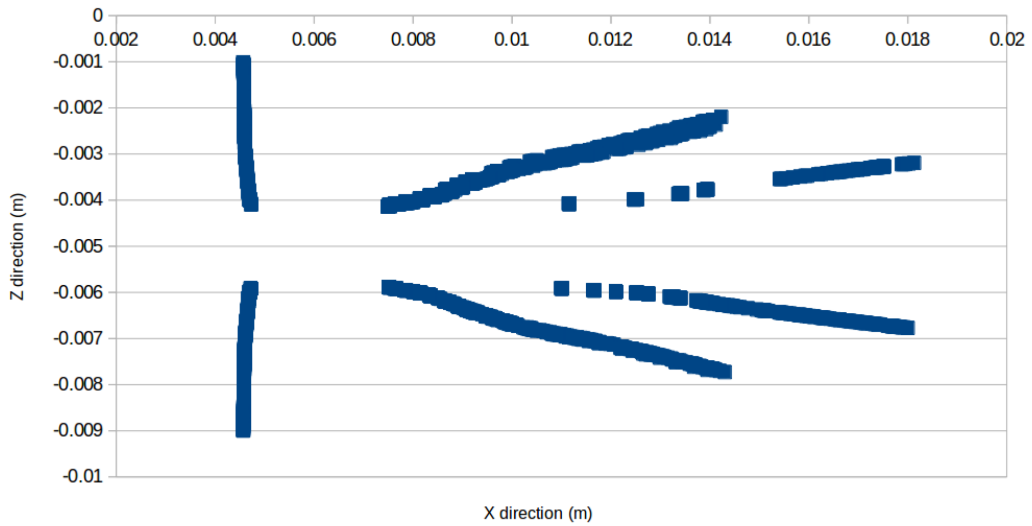


Figure 5.7: Hybrid model particle impacts.

The general trend of particle impacts is captured by the hybrid model, however there is much less scatter of the data. This is mainly due to two reasons: firstly, the injection site is much closer to the sampling surface in the Hybrid model than the EL, which gives the particles less time to disperse. Secondly, the mesh used was very coarse which meant that there were

only 100 injection sites for the particles in the Hybrid model. The Lagrangian particles were injected from the inlet of the pipe and had time to scatter before they reached the baffle surface. So rather than passing through the baffle at 100 specific, predetermined locations, they could pass through randomly. This lead to a much wider scatter as shown in Figure 5.6.

The impact angles of the particles were not recorded, as an appropriate method for obtaining the impact angles was not developed at this stage.

5.3 First phase comparison with CFD: Hybrid model (Jet Impingement Test)

After the Hybrid model was developed to the stage which it is at present in this thesis, initial experiments were carried out to compare the results of the fluid from the CFD with experimental data of the fluid. The rig was used to compare three different sample geometries: the cylinder, cone and hemisphere. The frame straddling technique was used (see Figure 4.3 for more detail) on 250 images per experiment with $\Delta T = 67\mu s$, using seeding particles to capture the fluid flow. A similar ($< 5\%$ difference) jet velocity of around 2 m/s was used for both experimental and computational simulations which gave Reynolds numbers $\approx 10^5$, which allowed comparisons to be made between results. The velocities were all normalised by the centreline jet exit velocity for all graphs and images. Y+ values were kept within limits suitable for the wall model.

The boundary conditions for the Hybrid model were:

Parameter	Value
alpha.particles	0.0039 (1% mass concentration)
alpha.water	0.996
epsilon.water	51
k.water	0.36
nut.particles	0 (no turbulent viscosity)
nut.water	0
p	1e5 Pa
Theta.particles	1e-7 (granular temperature)
T.particles	300 Celsius
T.water	300 Celsius
U.particles	2 m/s
U.water	2 m/s

Half symmetric 3D CFD meshes were made on cfMesh [83] using the Cartesian meshing tool. The nozzle and sample regions were refined with smaller cells to better capture velocity gradients. Having more cells on the baffle face also meant that there were more particle injection sites than if there was a less dense mesh. There were on average 3500 cells on the baffle, which gave 3500 injection sites. Figure 5.8 below shows Region 1 in blue, and Region 2 in green, with the baffle region in-between. The baffle/transition between the solvers was placed 7mm below the nozzle exit. The distance between the tip of the cone and hemisphere and the nozzle exit was 10mm, and the distance from the top of the cylinder to the nozzle exit was 20mm.

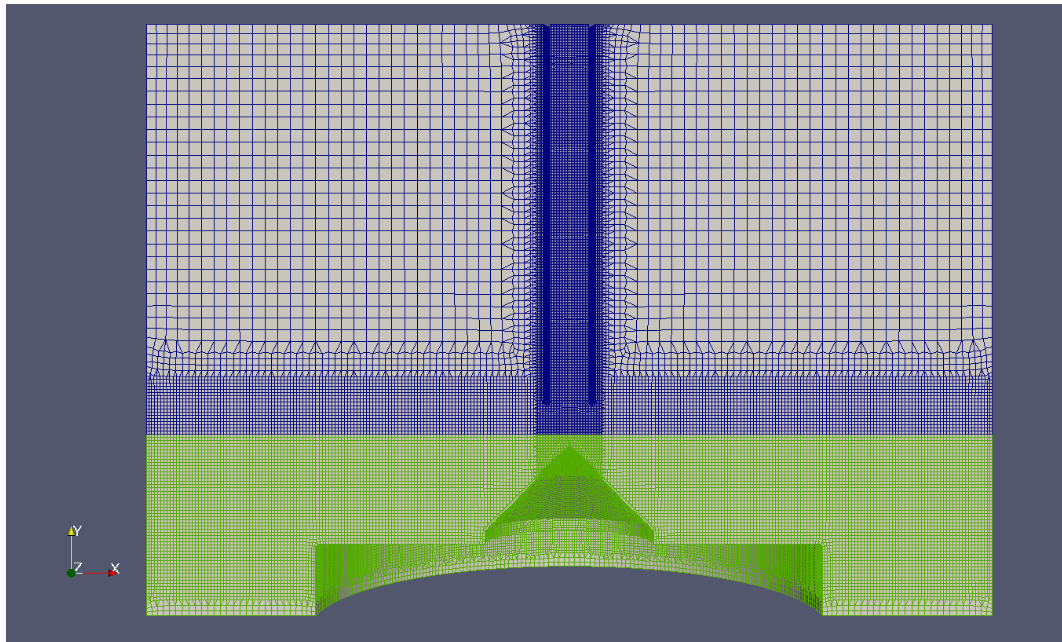


Figure 5.8: Cartesian mesh of cone, showing regions one and two in blue and green respectively.

The transition was placed in this challenging position to test the transition in an area where high pressure and velocity gradients were present (the exit of the pipe would be a more suitable position in practise, however due to time constraints, this was not done). The baffle also had to be in a position where particles were not going to bounce back into the EE region, as the solver cannot convert Lagrangian particles back to Eulerian. This is something to be considered for future work. This Section 5.5 below demonstrates this not to be the case. A mass flow inlet boundary condition was used for all three geometries (around 2m/s), with the K-Omega SST

model used for turbulence. Figure 5.9 shows the velocity magnitude contours of the converged solutions for each geometry.

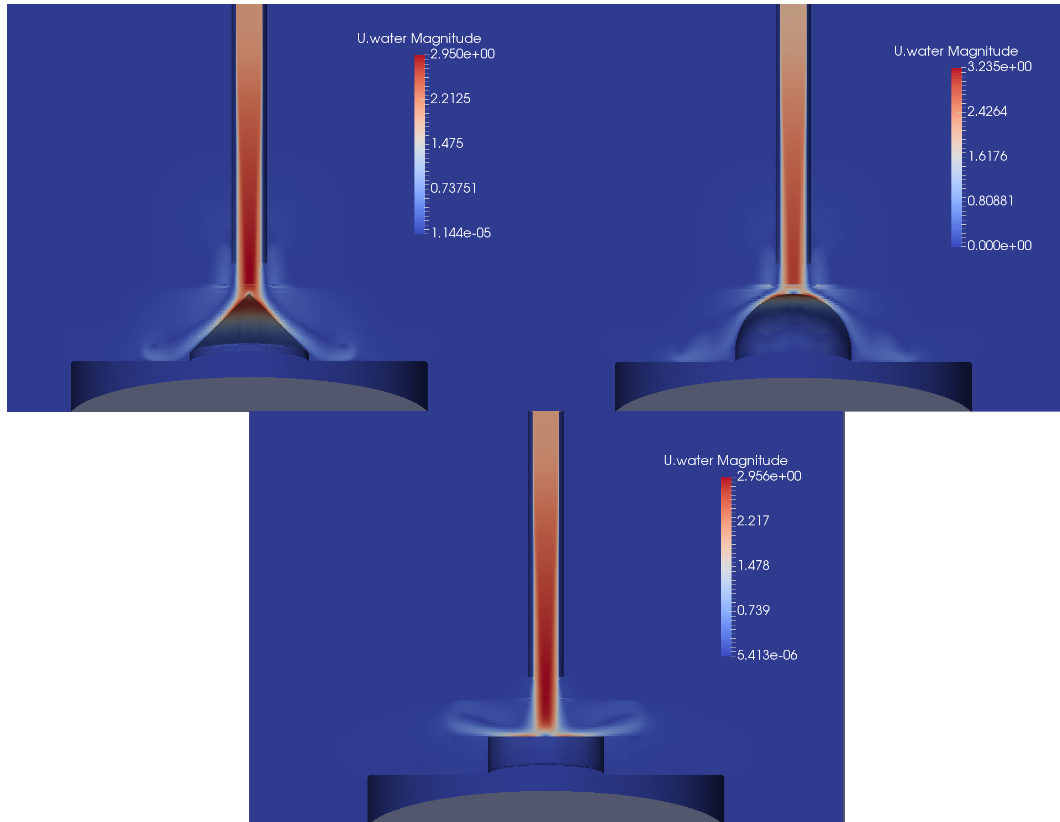


Figure 5.9: Velocity (m/s) contours of all three geometries (slight discontinuity shows baffle location).

Results were compared in three ways for each geometry: a line was placed above and below the interpolation region, and a velocity contour comparison plot was created. The locations of the sample lines on the hemisphere can be seen in Figure 5.10, where the top line is 5mm from the nozzle, and the bottom line is 9.5mm from the nozzle (top of cone and hemisphere are 10mm from nozzle). As the cylinder is flat, the lower sample line is taken 5mm above its surface. The lower lines were kept off the surfaces of the samples, as results can be erroneous here due to the lack of light from the laser. The lines were positioned above and below the baffle to ascertain how much the baffle affected the results.

The results for each geometry are shown in each subsection below. The layout and format for each geometry subsection were the same:

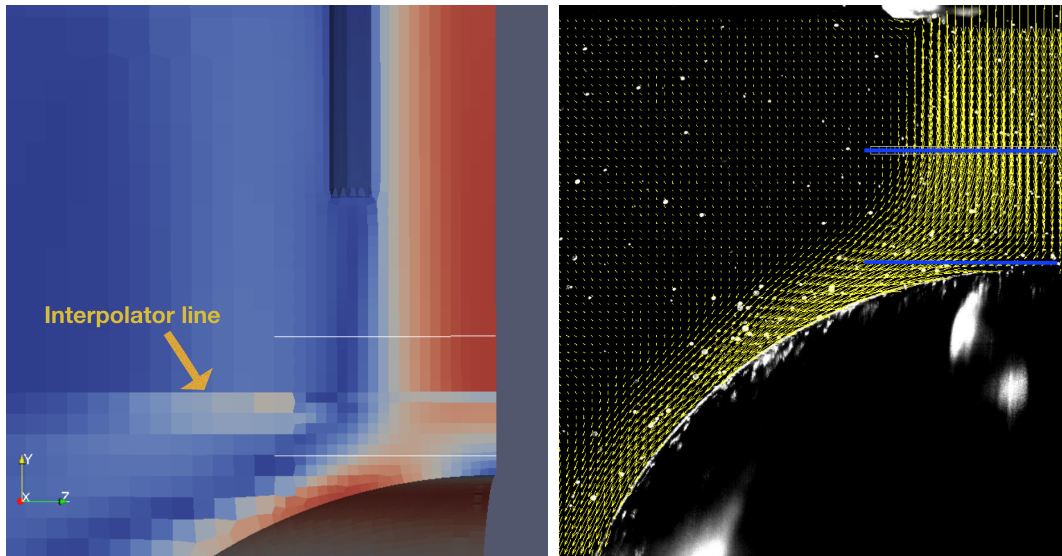


Figure 5.10: Left image shows baffle location and two white sample lines on CFD velocity magnitude contours. Right image shows same sample lines in blue on the velocity vector results from PIV data.

1. Velocity contour images of the vertical and horizontal components are shown, with experimental results shown on the left and CFD results shown on the right. The experimental results are in the region of the dashed square (dashes only shown in the cone image). They were interpolated onto the CFD mesh for accurate comparison with the PIV data. All results are normalised by the maximum vertical velocity component of the jet (since CFD and experimental velocities were slightly different (less than 5%). See Figures 5.11, 5.14, 5.17.

2. Graphs show the sample line results: with the top line shown in the first graph and the bottom line shown in the second. Again, results are normalised by the maximum vertical centreline velocity. Green depicts the 'U' (horizontal) component of velocity, red depicts the 'V' (vertical) component of velocity; the crosses depict the PIV data, and the filled shapes depict the CFD data. All graphs extend from right to left, from the centreline at 0 to 7mm radius. The PIV results extend past the centreline to show that there is some asymmetry in the results. See Figures 5.12, 5.13, 5.15, 5.16, 5.18, 5.19.

CFD simulations using the EE model were also run without the baffle, to determine whether it was having a significant effect on the results or not. The EE results were so similar to the

Hybrid model's results that no discernible difference could be seen on the graphs, and therefore the EE results were left off the graphs to reduce clutter.

Dynamic studio was used for all of the PIV analysis. The images were first grouped into double frames, then masked (to select area of interest), then adaptive PIV was carried out on them. Adaptive PIV iteratively optimises the size and shape of each interrogation area in order to adapt to local flow gradients and seeding densities. The mean velocity vectors were then calculated for the whole set of vector maps, to obtain an averaged result: which are shown below.

5.3.1 Cone results

The contour comparison for the vertical velocity component shows clear correlation with experimental results, however the CFD under predicts the horizontal component by 100% in some locations. The graphs shown in Figures 5.12 and 5.13 also show close agreement with experimental results, with discrepancies of less than 10%. CFD matches closer to the experimental result nearer the nozzle exit as expected, and the asymmetry of the jet can be seen in the red 'V PIV' from Figure 5.12. This could be due to a number of different factors (misalignment with the laser, geometry of pipe etc.), however the discrepancy is less than 10% and is therefore not significant enough to adversely affect the results.

5.3.2 Hemisphere results

The hemisphere was the most challenging shape for the CFD to model, as is confirmed by the results. Again, the vertical velocity component is more accurate than the horizontal component seen in Figure 5.14. The solver 'U' velocity component from the CFD simulation was not being interpolated to the top region correctly. Although the baffle location is not indicated (location can be seen in Figure 5.10), it can clearly be seen by the abrupt end to the lower contour plot on the right side. Other than this interpolation issue, both contour plots seem quite similar.

The hemisphere graph comparisons give the worst results of the geometries tested. The sample line nearest the nozzle shows differences in the horizontal velocity component of almost

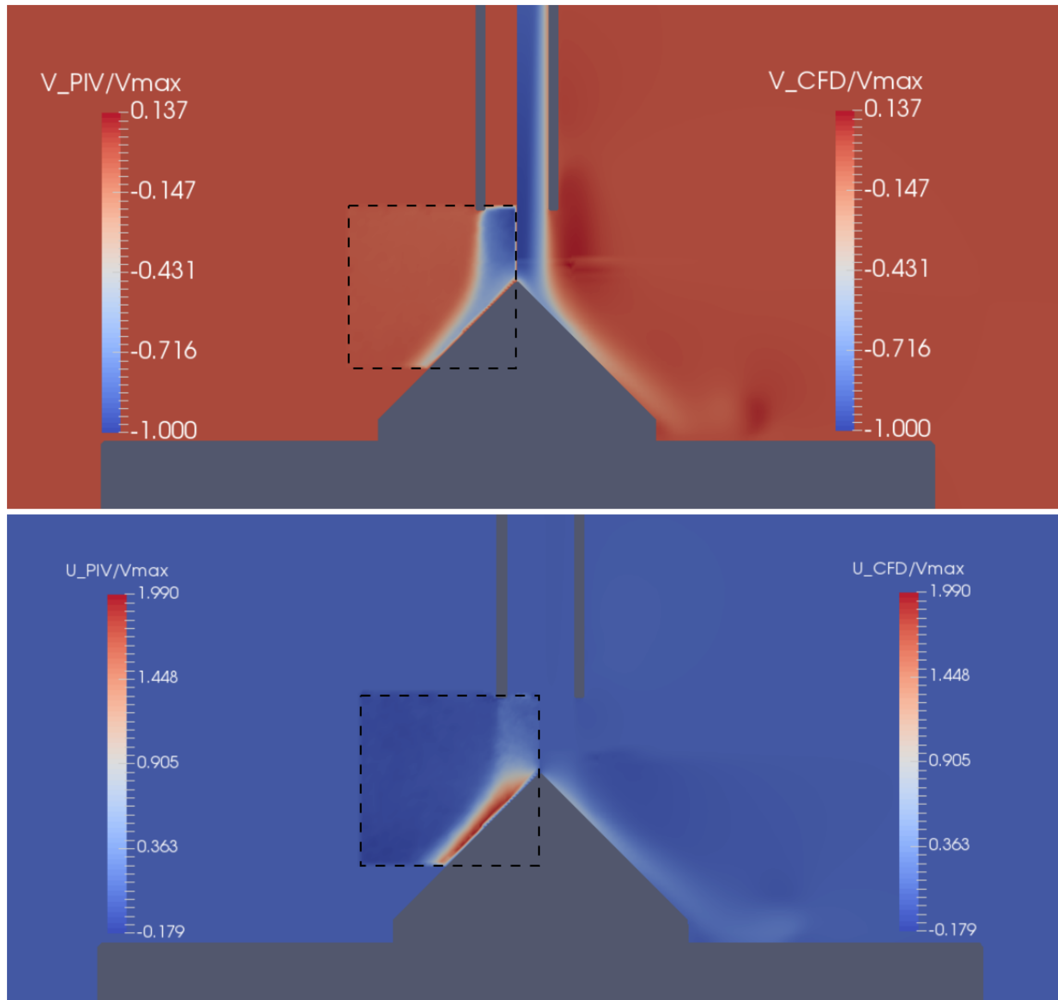


Figure 5.11: Comparison of experimental and computational results displaying contours of vertical (top image) and horizontal (bottom image) velocity components (experimental results shown inside dashed squares). Data is non-dimensional.

20%, however the general shape is captured well. This discrepancy is most likely caused by the presence of the baffle (as the top line is above the baffle). Figure 5.16 shows the results taken from the upper line, and the reason there is such a large difference is most likely because the sample line is too close to the surface of the hemisphere. Differences of this magnitude (up to 40%) are not expected in CFD, and therefore it is probably due to an experimental or data analysis error.

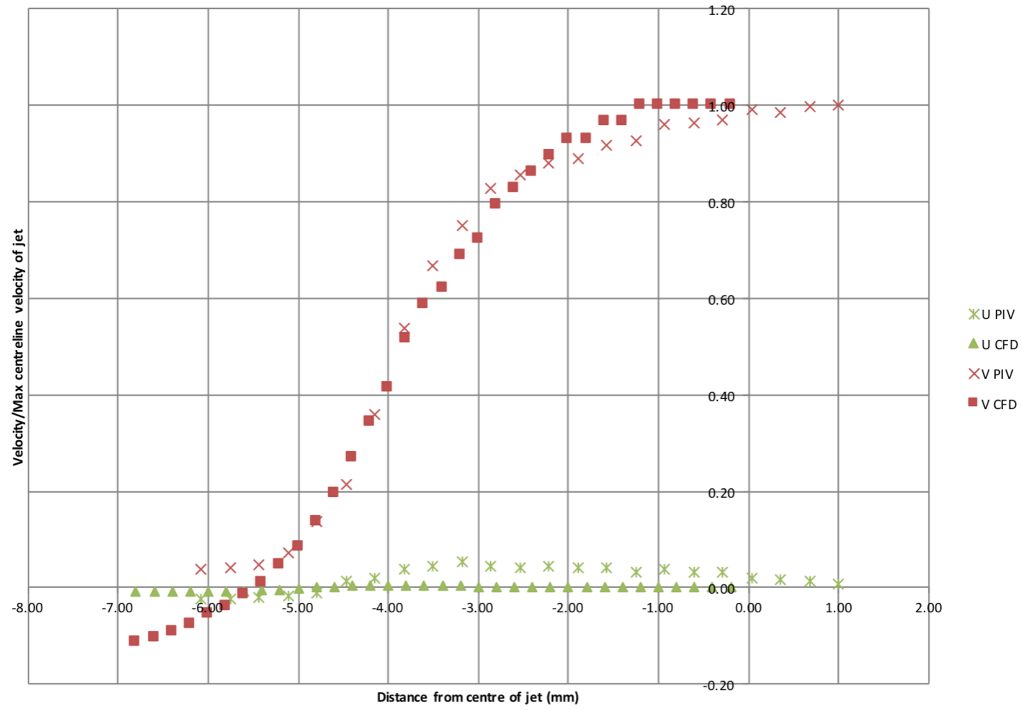


Figure 5.12: Cone results: 5mm below nozzle exit velocity profiles.

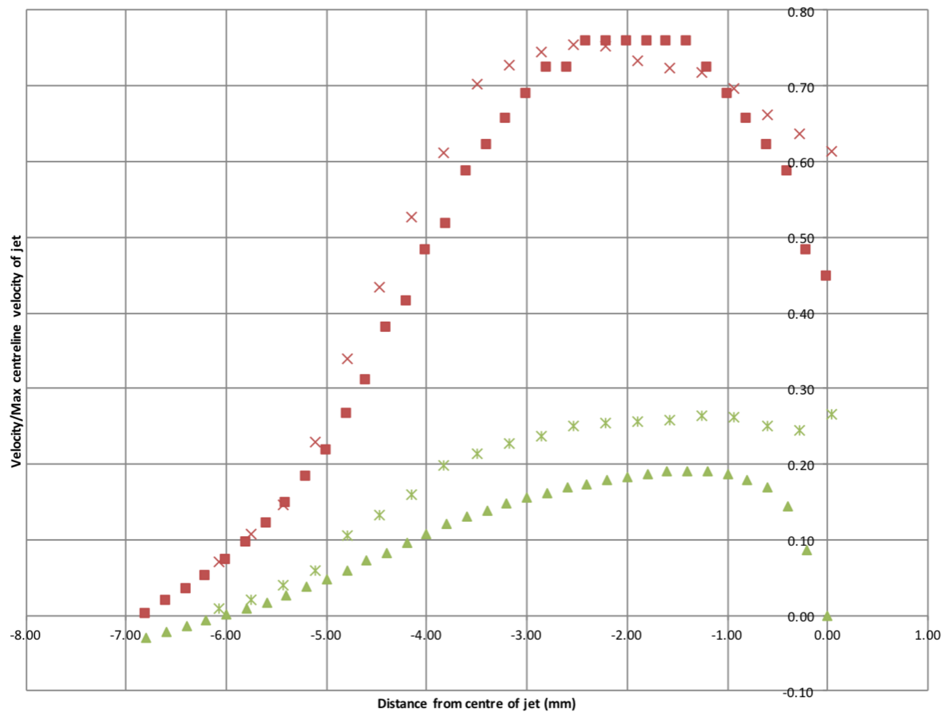


Figure 5.13: Cone results: 9.5mm below nozzle exit velocity profiles.

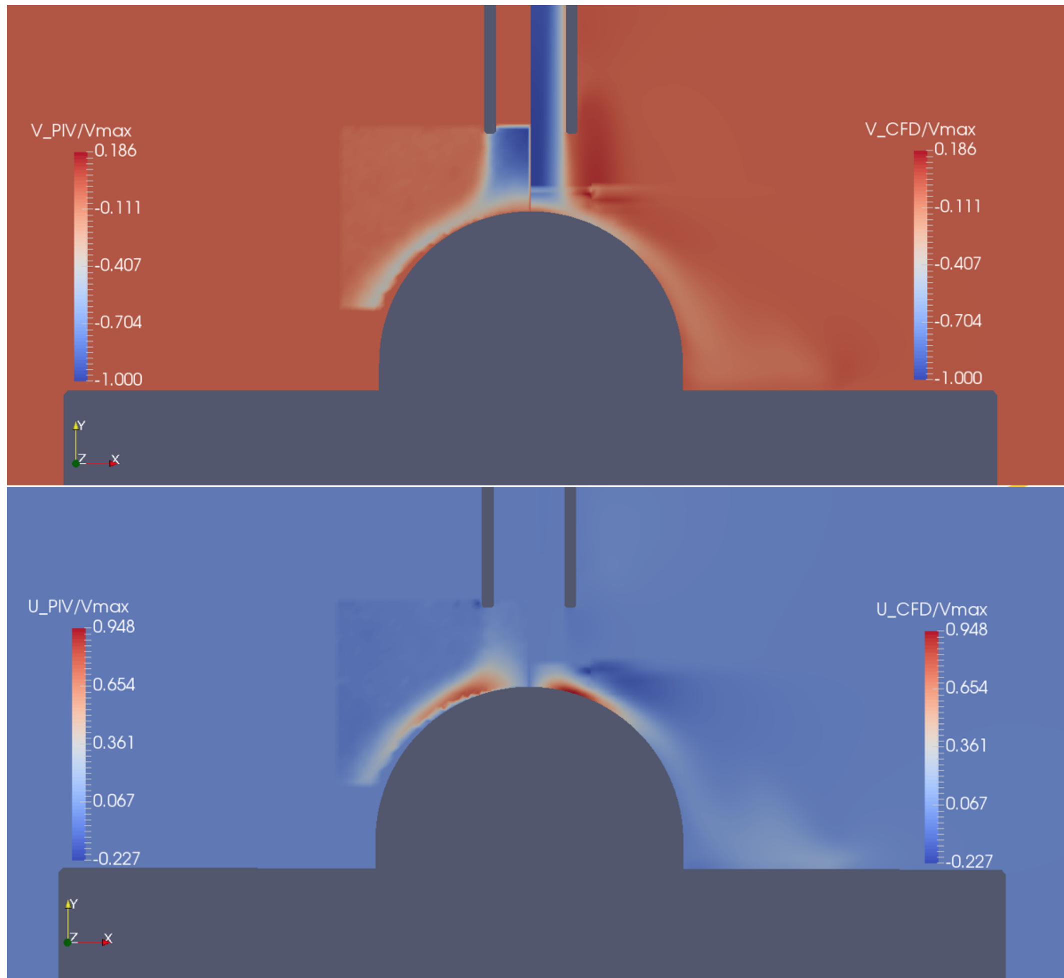


Figure 5.14: Comparison of experimental (left) and computational (right) results displaying contours of vertical (top image) and horizontal (bottom image) velocity components: Vmax is the maximum centreline vertical velocity component. Data is non-dimensional.

5.3.3 Cylinder results

The cylinder is the most widely used sample shape for erosion/PIV analysis, and as seen in Figure 5.17 the contour plots are very similar. The CFD plot has a slightly smaller stagnation region than the experimental, but overall there is good agreement. The sample line graphs also show good agreement, usually less than 10% difference, with only the horizontal component for the top line diverging more than this at 5mm radius (see Figure 5.18. This again seems to be caused by the interpolation region not working well for horizontal (parallel to baffle) velocities. The slight asymmetry in the jet can be seen with the extended experimental results on the x-axis, showing the jet was not perfectly axisymmetric.

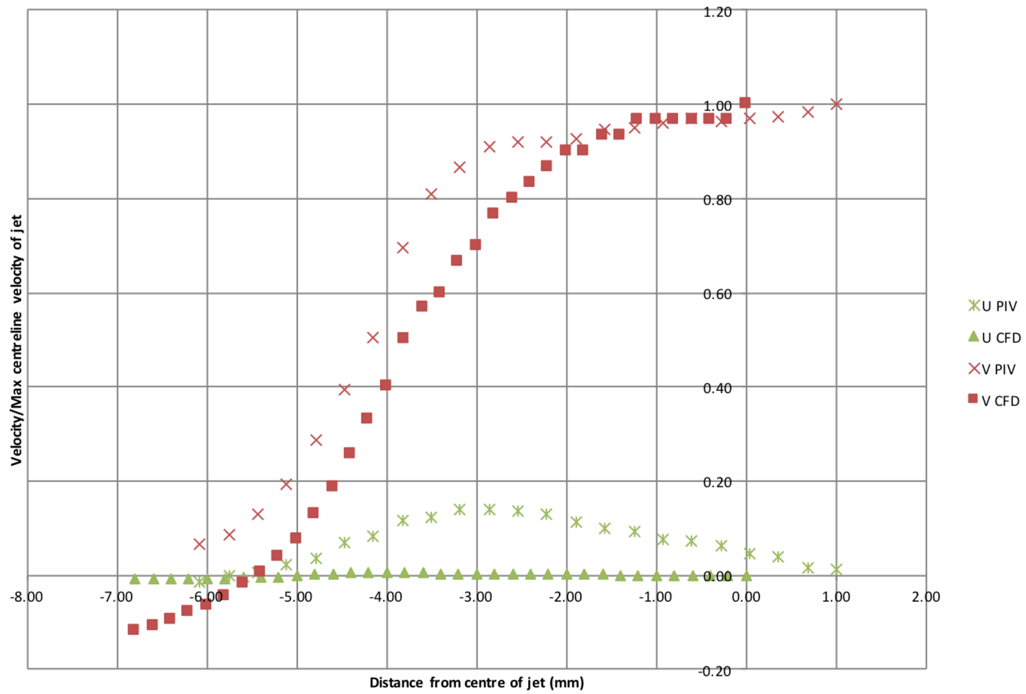


Figure 5.15: Hemisphere results: 5mm below nozzle exit velocity profiles.

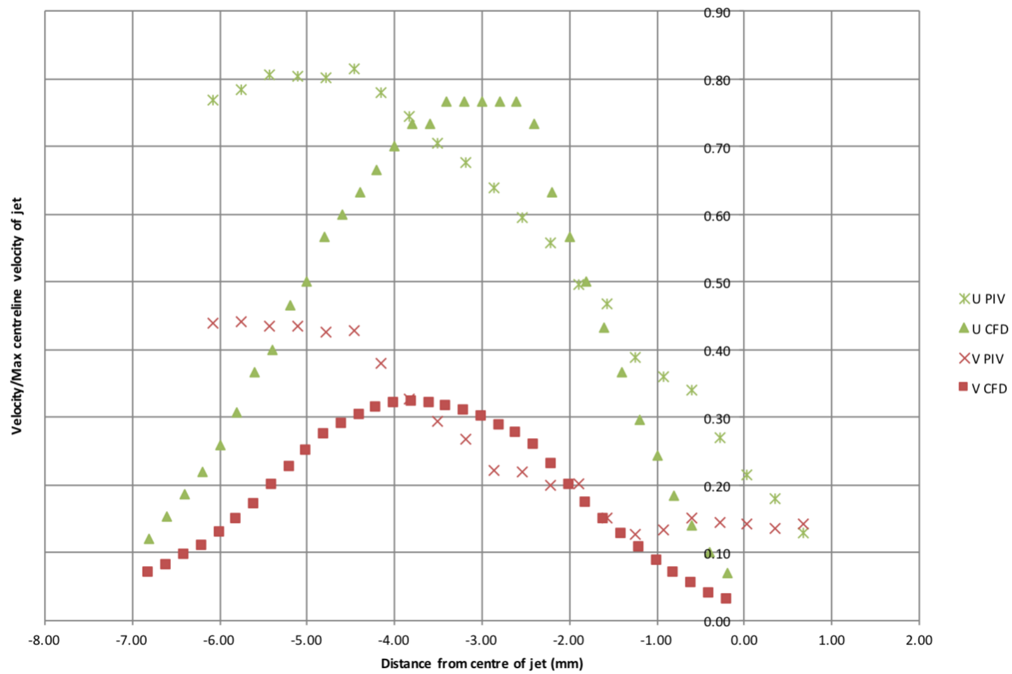


Figure 5.16: Hemisphere results: 9.5mm below nozzle exit velocity profiles.

The baffle and interpolating code does not work as well when the fluid is traversing through the baffle with a low perpendicular component of velocity, compared to when there is a high per-

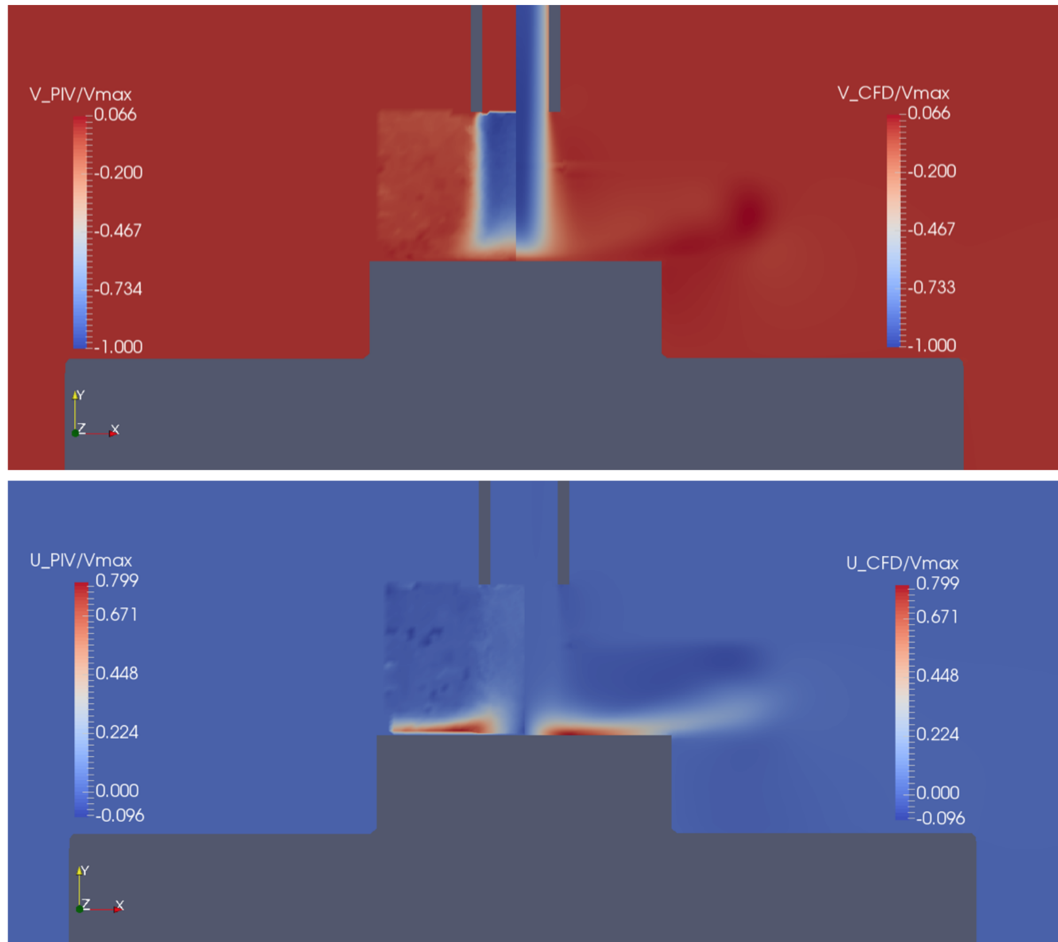


Figure 5.17: Comparison of experimental (left) and computational (right) results displaying contours of vertical (top image) and horizontal (bottom image) velocity components: Vmax is the maximum centreline vertical velocity component. Data is non-dimensional.

pendicular component. The rational of interpolating one value to an adjacent cell downstream is stronger when the fluid is flowing in the direction of interpolation. This is a limitation of the Hybrid Model, and other solutions could be thought of in the future for reducing this effect. Currently the best solution for reducing errors, is to position the baffle where the majority of the fluid is flowing through the baffle, and not parallel with it.

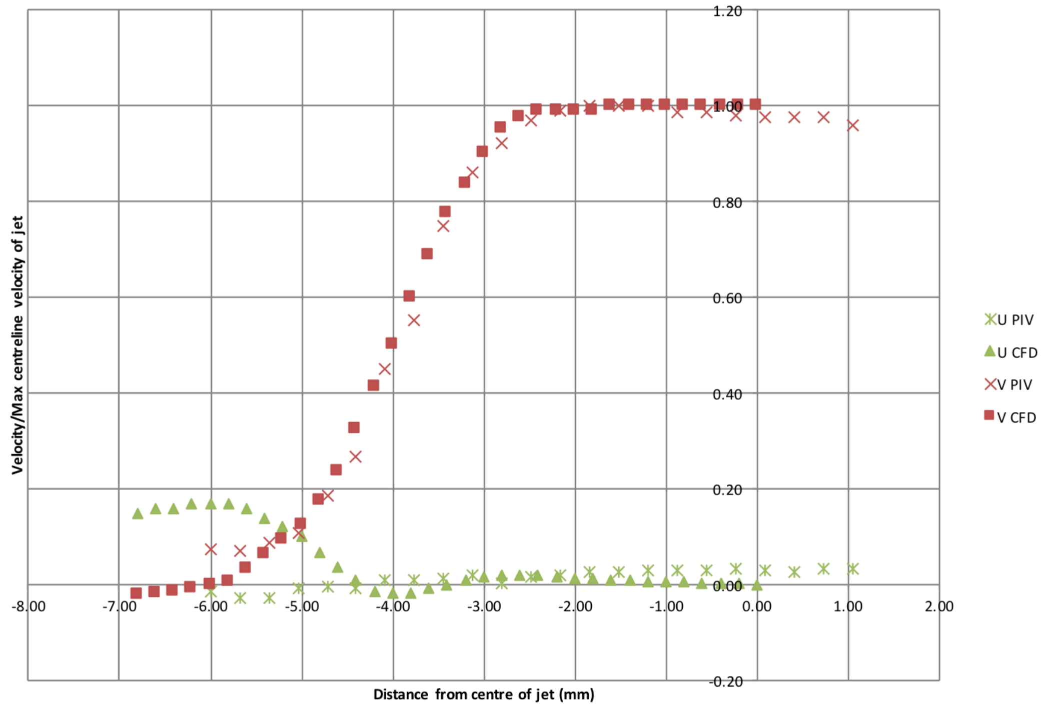


Figure 5.18: Cylinder results: 5mm below nozzle exit velocity profiles.

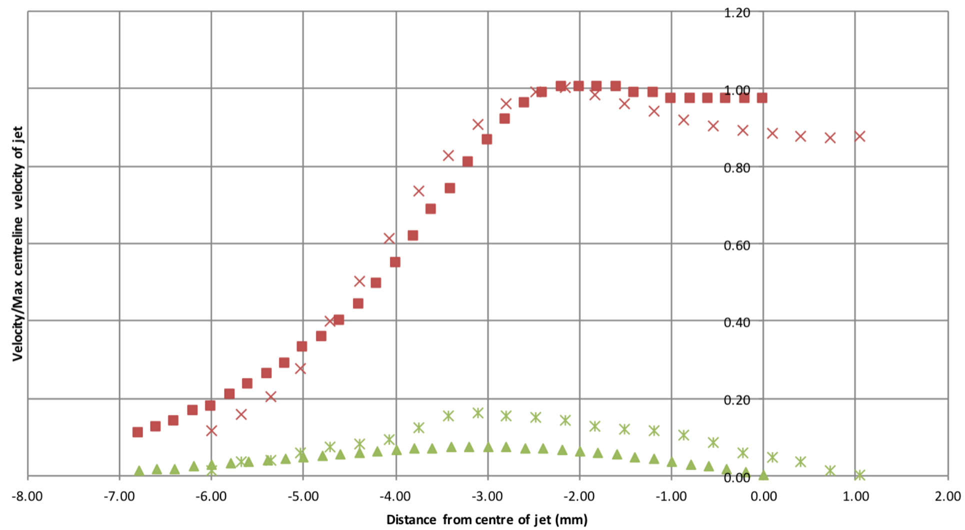


Figure 5.19: Cylinder results: 5mm above sample velocity profiles.

5.4 Second phase comparison with CFD- Hybrid model (Jet Impingement Test)

Up until now the particulate phase has been ignored. From this point onwards, particles are included in the simulations and results.

5.4.1 Time to solve comparison

The solver was tested to compare compute times on the different geometries previously mentioned. Table 5.2 shows the execution times on the square pipe bend from the Tutorial (see Figure 5.5) and Table 5.3 shows the times from a simulation on the cone geometry.

Model	**Execution Time (s)**		**No. of particles injected**
	0-0.03 (no particles)	0.03-0.06 (with particles)	
Hybrid model	210	1976	34300
Euler-Lagrange	210	2179	30000

Table 5.2: Comparison of execution times on the pipe bend.

Model	**Execution Time (s)**		**No. of particles injected**
	0-0.084 (no particles)	0.084-0.09 (with particles)	
Hybrid model	263000	4160	185000
Euler-Lagrange	251000	4570	180000

Table 5.3: Comparison of execution times on the cone shaped sample.

Results for the execution times with no particles added are similar for both the EL and the Hybrid model, however the Hybrid model is faster than the EL when particles are added. The Hybrid model takes 90% of the time that the EL model takes to model the same amount of time and similar numbers of particles in both tests. The time-step was the same for each simulation, with the maximum Courant number kept below 1 and particle concentration kept below 10%.

Reducing the compute time by 10% may not seem very large, however there are a few factors that should be highlighted. Firstly, the solver wasn't completed very long before the thesis was written, and therefore little work was carried out to improve the efficiency of the code. This

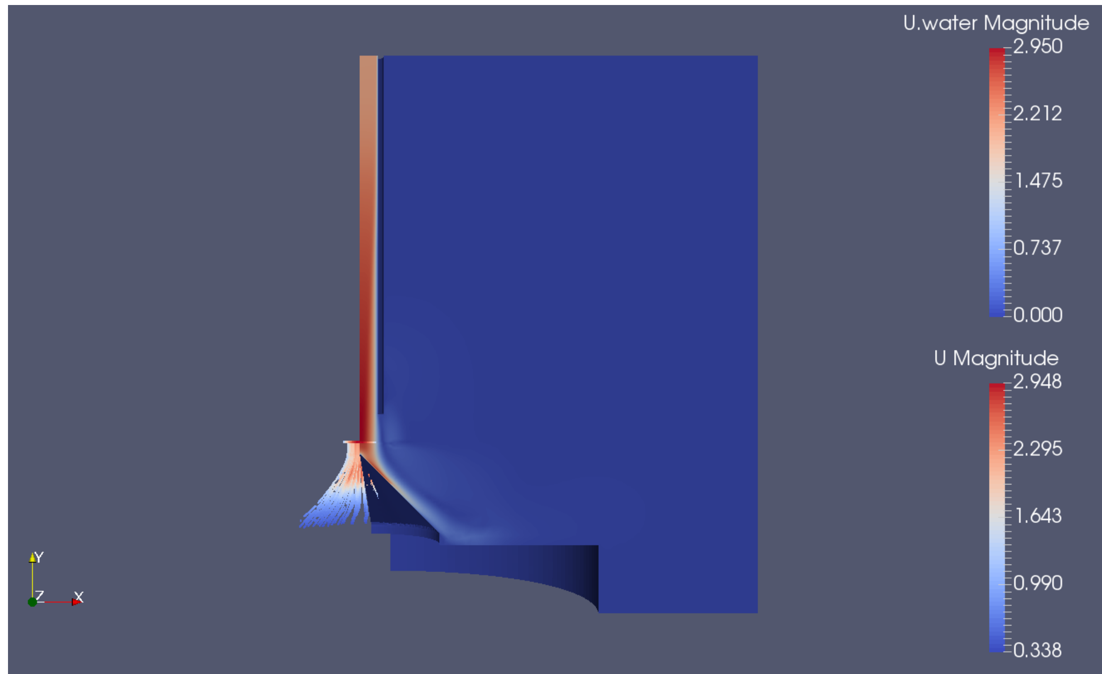
could be revised, and would enable faster execution times. Secondly, the saving on execution times will grow with a greater number of particles, as the Hybrid model will have to model less numbers. There is also a 'ceiling' on the number of particles a certain computer can handle, and the Hybrid model will allow more erosion modelling to take place for the same total amount of particles. Thirdly, the number of injection sites depends on the number of cells adjacent to the baffle, which will alter the demand on the code. Each injection site needs another column in the lookupTable, and it is reading and writing from this table which uses a lot of computational resource. Fourthly, an important point is that the Hybrid model starts modelling erosion before the EL model does if started at the same time. The particles have less distance to travel before they meet the surface of the sample, as they start from the baffle and not the inlet of the pipe. This can be seen in Figure 5.20 where 5.20a (Hybrid model) and 5.20b (EL model) show the particle positions 0.028 seconds after injection time. Having the injection location nearer the sample (or wear surface in a more complex geometry) allows the erosion modelling process to start earlier than if the EL model was used.

These points should be contrasted with the most similar published paper to this thesis, by Messa [4]. Messa managed to reduce the computational simulation time by 30%, however the particles were injected as a post-process step, the particles did not affect the flow field of the fluid, and the solver of Messa's cannot cope with high density slurry flows.

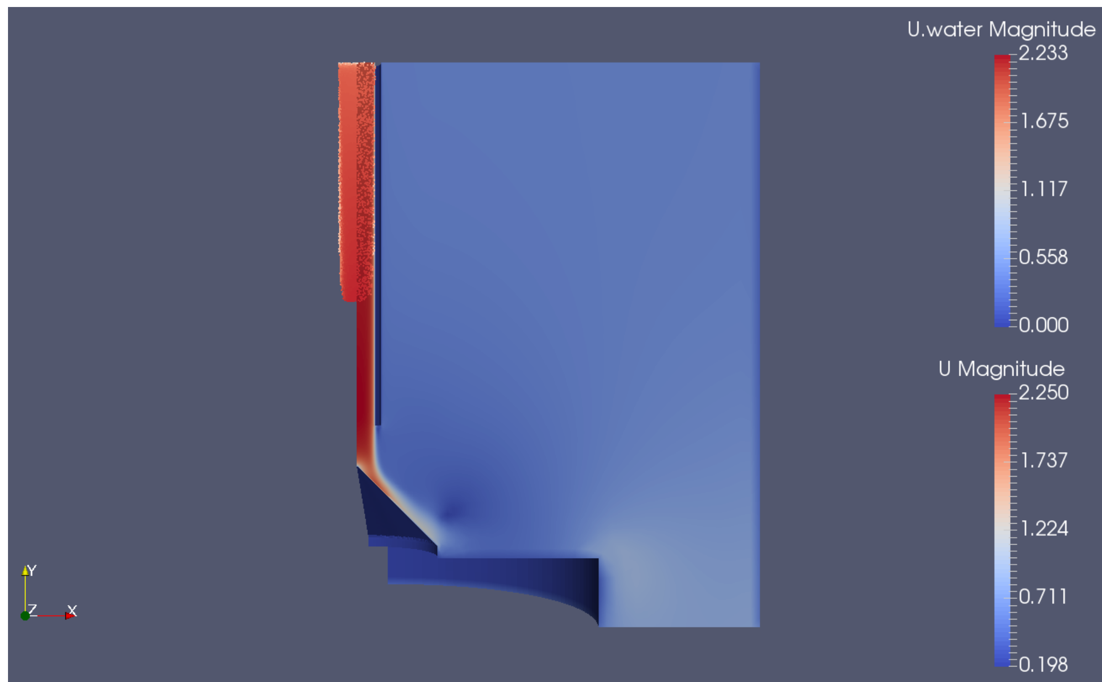
5.4.2 Particle impact comparison

A method used by Dr Alejandro Lopez in a journal paper [11] was used to compare impact velocities on the cylinder sample. The file `PatchPostProcessing.C` was modified to enable velocities to be recorded each time a particle hit the impingement surface (file in appendix). Simulations were run and 4000 impacts were recorded for each solver.

After the raw data were obtained, an excel sheet was created and the data grouped into bins according to the radius location of impact. These bins are shown in Figure 5.21 on the cylindrical sample. The velocity data was again normalised by the maximum velocities, since both simulations used slightly different velocities. The results are shown in the graph in Figure



(a) Hybrid model



(b) EL model

Figure 5.20: Comparison of particle positions 0.028 seconds after injection. Particles coloured by velocity magnitude, with quarter model shown of the fluid domains coloured by fluid velocity magnitude.

5.22 with the Hybrid model represented by blue squares and the EL model by orange diamonds.

As seen there is close agreement between both models in the first 10mm of the sample radius, thereafter the hybrid model predicts lower impact velocities. The reason for this is unknown,

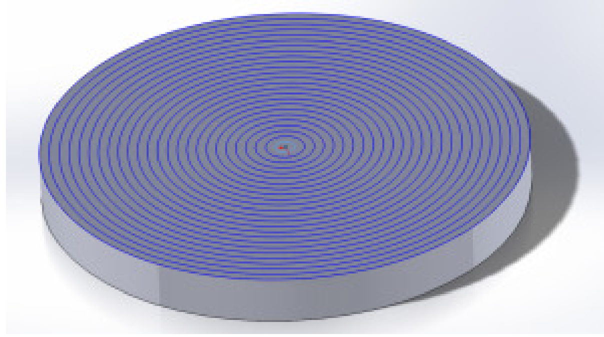


Figure 5.21: Cylinder impingement surface divided into 1mm bins [11].

but could be caused by turbulence model differences or particle drag models. These were both setup to be as similar as possible, however they are not identical since the Hybrid model still uses the EE model as it's foundation. The location and value of the peak velocity were both captured well, and this is what is most crucial for erosion modelling. These results also show a promising outlook for the Hybrid model.

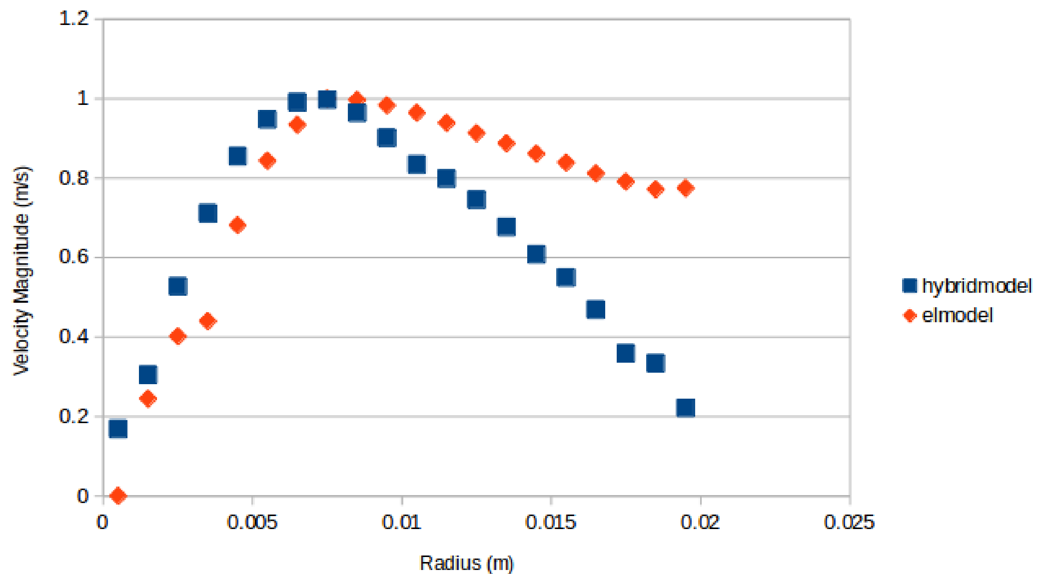


Figure 5.22: Particle impact velocities on cylinder sample.

5.4.3 Visual comparison

Visual comparisons of the Hybrid model and EL model are shown to allow the reader to see the similarities between results on the different geometries.

As is the same for all simulations, to model the same amount of erosion work (particles striking the impingement surface), the Hybrid model uses much fewer particles, which saves computational resource (particularly computational memory). This is exemplified by Figure 5.23 where the EL model is shown on the right and the Hybrid on the left. Although the fluid or geometry is not shown, the inlet of the pipe can be seen as it is where the EL particles are injected from. The Hybrid model does not model Lagrangian particles until the baffle location, which is much closer to the impingement surface, thus saving a lot of unnecessary modelling where particle data are not important.

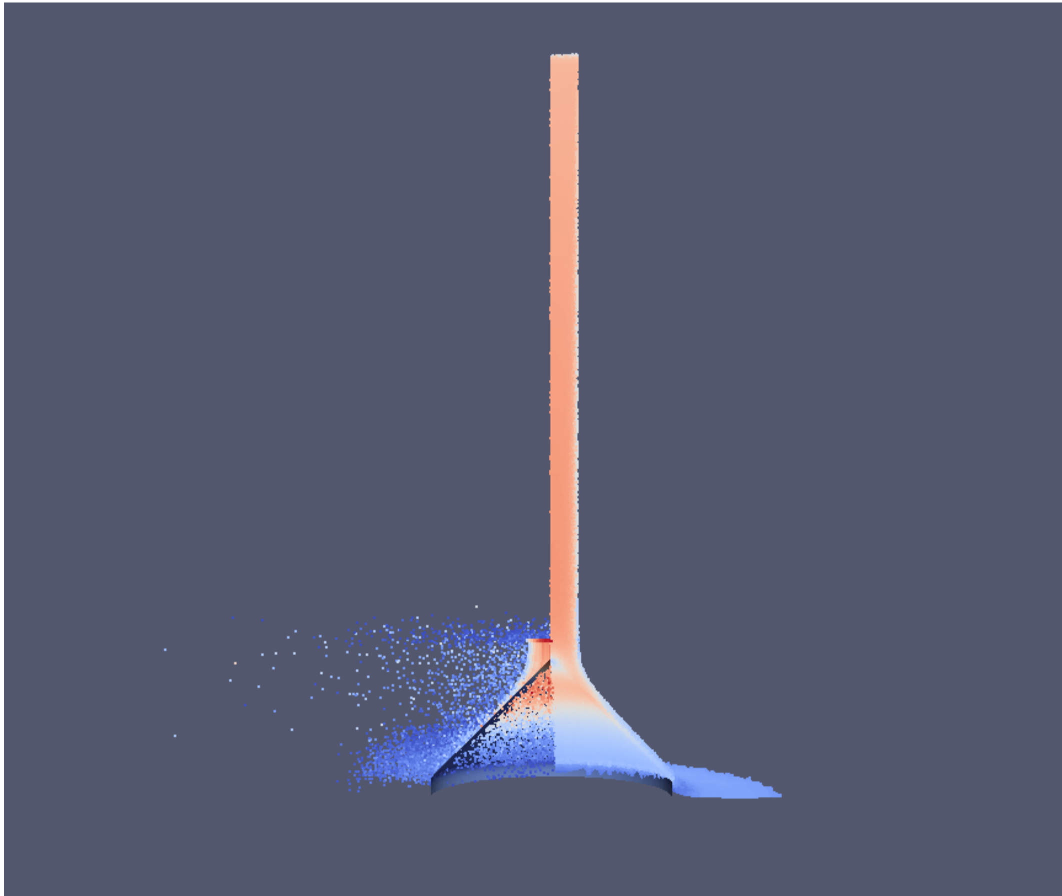


Figure 5.23: Particles simulation on cone sample, showing impingement surface and omitting fluid: Hybrid model on left, EL model on right. (EL particles do not seem to bounce as much as the Hybrid model particles, however the same boundary conditions were used in both cases.)

Figure 5.24 is the same comparison as in Figure 5.23 but focused on the sample location, and Figure 5.25 is the same but on the cylinder sample. The particles showing through the sample

is a rendering error by Paraview, as the particles do not pass through the impingement surface. Again, the baffle location can be seen by the start of the Hybrid model particle injections. Particle spread looks similar in both instances, with the Hybrid model's particles dispersing more freely after impact than the EL model's particles. The images look similar with some key differences.

Firstly, each dot is actually a *parcel*, not a *particle*. The injection method used for the EL model (Patch Injection) only allows a fixed number of particles per parcel and so it was set to two. The Hybrid model works by calculating the number of particles which should be injected per parcel, and so in areas where the mass concentration of particles is high (centre of the pipe), there can be as many as five particles in a single parcel. It is for this reason that there seem to be more particles in the EL simulation than the Hybrid model, even though this is not the case.

Secondly, the parcels are more dispersed in the EL simulation than they are in the Hybrid. This is because they have much more time to scatter and randomise than the Hybrid model, due to its injection nature and the proximity of the injection site to the impingement surface. By the time the particles have reached the surface of the samples they are more dispersed as can be seen. This phenomenon is discussed more in Section 3.4.2 and can be seen in Figure 3.6. The results presented here show the ability of the model to work when two-way coupled, unlike the Messa [4] paper cited in an earlier chapter.

5.5 Comparison of CFD and experimental work

Directly comparing experimental particulate data against CFD particle data is very challenging, as is exemplified by the lack of evidence in the literature.

One method of comparing experimental data to the numerical results is to visually compare particle track plots. PIV experiments were carried out to capture particle tracks to compare with CFD data. Frame straddling as used in previous experiments was not useful as it captures sets of two images back to back, and then a time break, and then another set of two images.

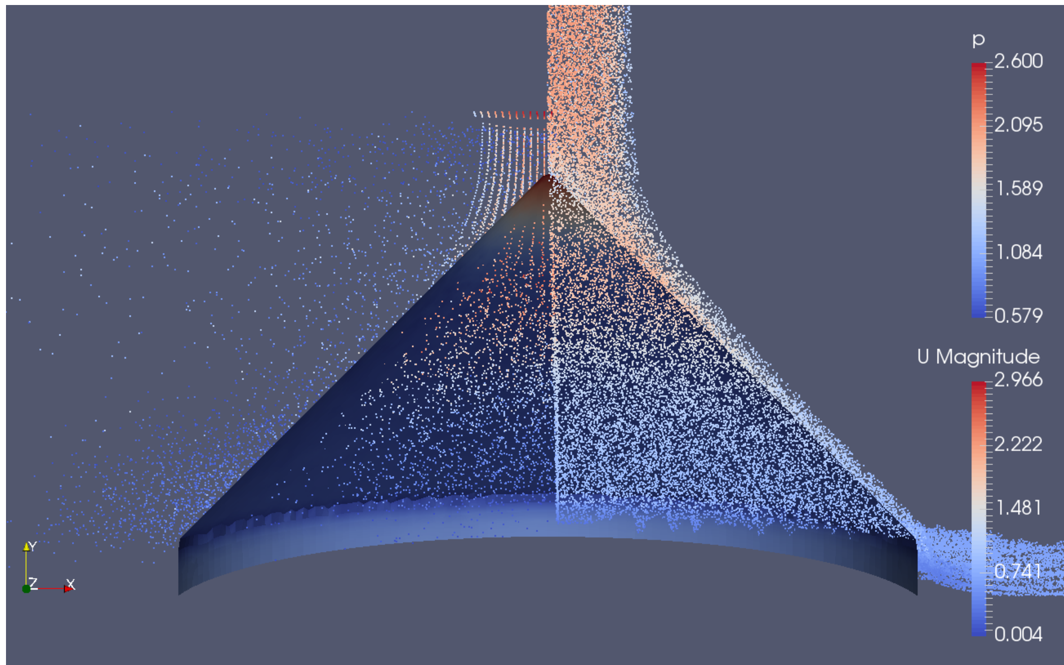


Figure 5.24: Particles simulation on cone sample, showing impingement surface coloured by pressure : Hybrid model on left, EL model on right.

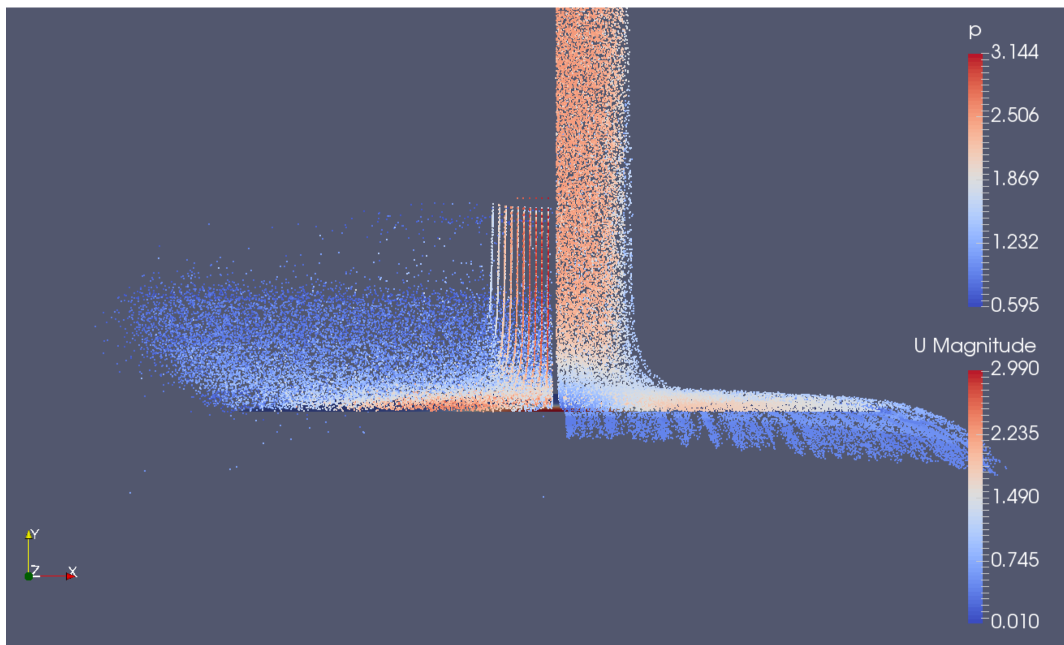


Figure 5.25: Particles simulation on cylinder sample, showing impingement surface coloured by pressure : Hybrid model on left, EL model on right.

This results in a partial, as opposed to continuous, particle track.

Frac sand ($500\mu m$ diameter) was injected into the main pump for analysis, resulting in a

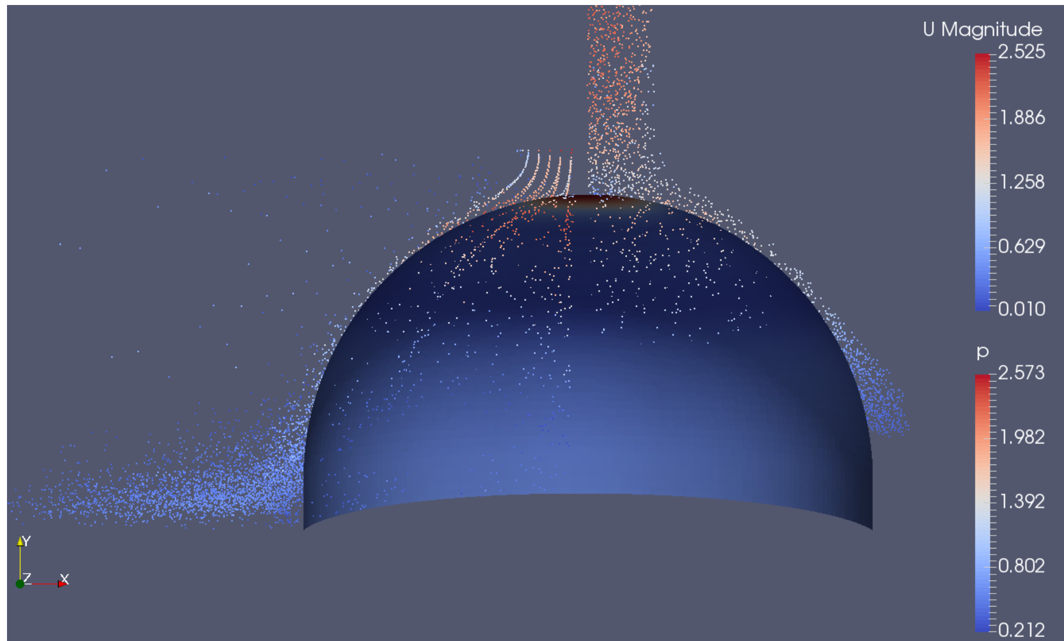


Figure 5.26: Particles simulation on hemisphere sample, showing impingement surface coloured by pressure : Hybrid model on left, EL model on right.

slurry with a mass concentration of around 2 %. If the concentration was higher than this, the images became difficult to analyse due to their brightness and the masking of other particles. The particles on the left side of the image (closest to the laser) cast a shadow to the right, and thus obstruct the other particles from being analysed.

The camera was set to 500fps and set to take a continuous burst of 500 images, thus capturing one second of real time. The 500 images were almost back to back, with only the time for the optics to refresh separating the images from one another.

One of the images from a set of 500 is shown in Figure 5.27. The laser was set to approximately 5kHz (on an analogue dial) and was verified by counting the number of times the particle could be seen in the image which lasts 1/500th of a second (at 500fps). In Figure 5.27 there are 11 images of the same particle, which gives a laser frequency of 5.5 kHz ($\frac{11}{1/500}$). The smaller particles which can be seen are debris in the tank or remaining seeding particles.

The velocity of the particle was found by measuring the distance travelled in the 1/500th of a second image. A scale image of a ruler was always taken before each set of experiments was carried out, and by using this, the distance travelled could easily be measured. In the

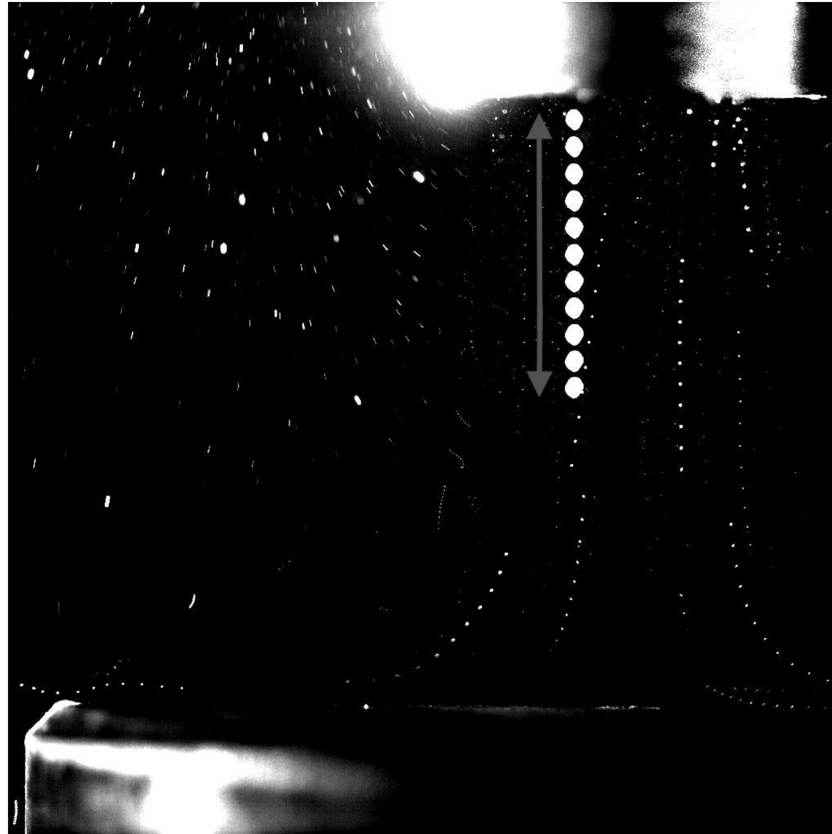


Figure 5.27: Single image from set of 500 showing one particle track over 1/500th of a second, with arrow showing distance travelled.

above image the distance travelled was 4mm, and since there was regular spacing between the images of the particle, the particle would have been travelling at a speed of 2 m/s throughout the frame. Clearly if the particle images were closer together then the particle would have been travelling slower, and faster if the images were further apart. Particles changing speed can be seen in further images, especially when there is a direction change involved or when the particle comes close to the surface of the sample.

Within each set of 500 images, there were on average 10-15 particles recorded which stayed in the same plane as the laser (statistically only 1/180 of the particles will do this if the scatter is random, since there are 180 degrees in a half circle). This is illustrated in Figure 5.28 where only the particles which stay within the green dashed line are analysed.

These particles were then stitched together into one image, using Adobe Photoshop's *crop* tool combined with the *layer* tool. This process was repeated for each geometry, with the

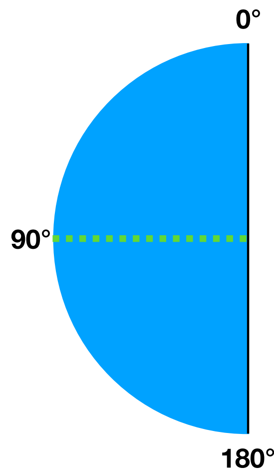


Figure 5.28: Plan view of sample coloured in blue, with green dashed line illustrating area where particles will be analysed by laser.

results shown below. Additional stitched images can be seen in the appendix, as the images would have been too cluttered if all of the particle tracks were joined together into one image.

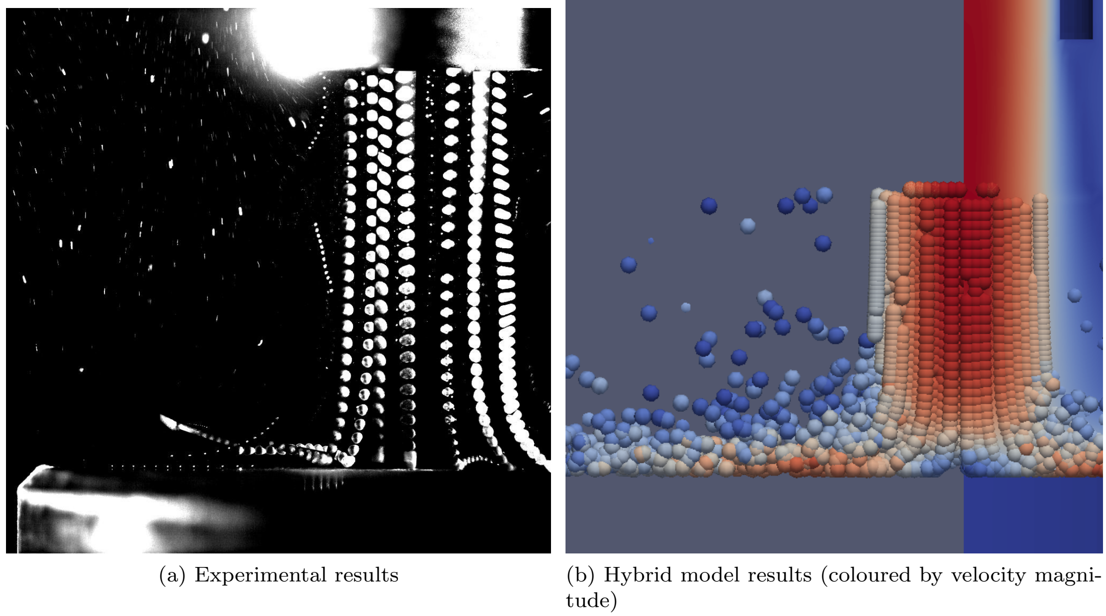
Figures 5.29 to 5.31 show the comparisons between the experimental results shown on the left (a) and the CFD results shown on the right (b). The Hybrid model results show the particle spread, cut to display the particles travelling in the same plane as the experimental results. The right side of the fluid domain is also displayed to show the geometry location, coloured by velocity magnitude of the fluid. The baffles for particle injections were kept in the same locations throughout the thesis and can be clearly seen in the images. Legends have been omitted from the CFD images as there are no velocity data for the experimental images to compare with. CFD results show good match up with the experimental results, however it is difficult to quantify the accuracy of such images from a visual comparison. However, it can be stated that the Hybrid model here proves it's ability to model a real world experiment with reasonable accuracy.

These results can be compared with the particle track image shown in Figures 4 and 5 of the paper by Gnanavelu [53]. The velocities used in this paper are much higher though, so comparisons should be for illustrative purposes only. Figure 1 of the paper by Mansouri [34] also shows particle impacts, however the particle velocities are also much higher than in this

paper. There are few papers which have such low impacts velocities on the jet impingement test, since the papers which use this test are usually interested in erosion modelling: which requires high impact velocities. For example, this paper by Nguyen [55] has impact velocities ranging from 15-30 m/s, an order of magnitude higher than the results shown here. The lack of papers with similar velocities makes comparisons difficult.

It should be noted that as explained by Figure 4.11 only the left side of the cone and hemisphere could be analysed due to the shadow effect. This means there is only half the data obtained in the same amount of time for these samples compared to the cylinder. This should be considered for future work, and samples which cast shadows should be experimented on for double the time for the same statistical scatter as the cylinder.

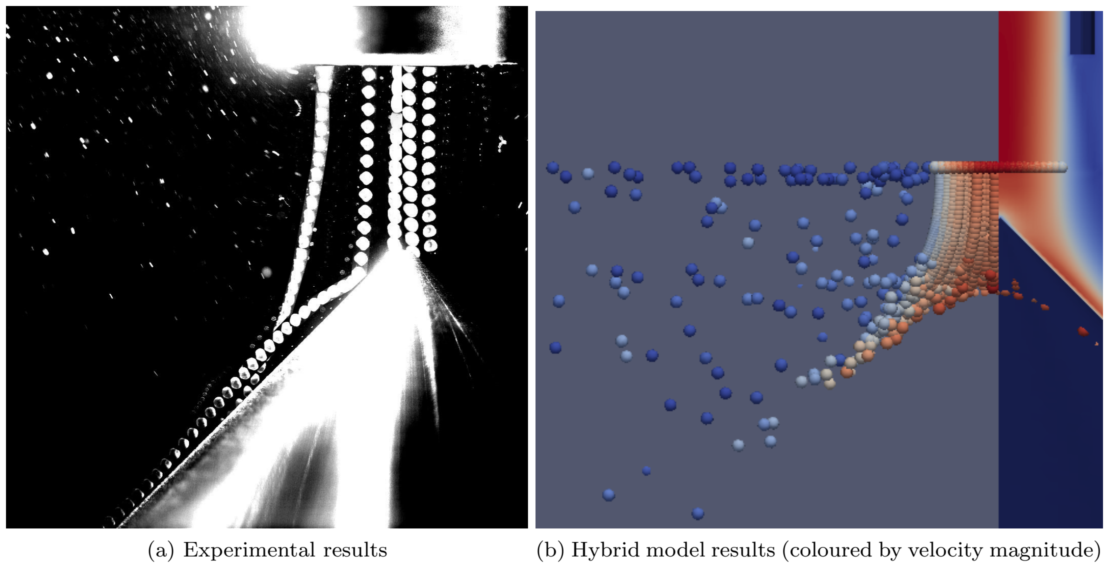
Another thing the experimental results show is that there is no 'bounce back' through the baffle. A particle travelling from the bottom region to the top would cause modelling errors at present as the solver is not capable of converting a Lagrangian particle into the Eulerian frame, and so it would not be able to travel into the top region. It should be noted that the particles which are proximal to the baffle in some of the images below have almost zero velocity, and are almost 'floating in the flow'. These particles will not contribute to the erosion process. These results therefore show that the baffle location is suitable for the present work.



(a) Experimental results

(b) Hybrid model results (coloured by velocity magnitude)

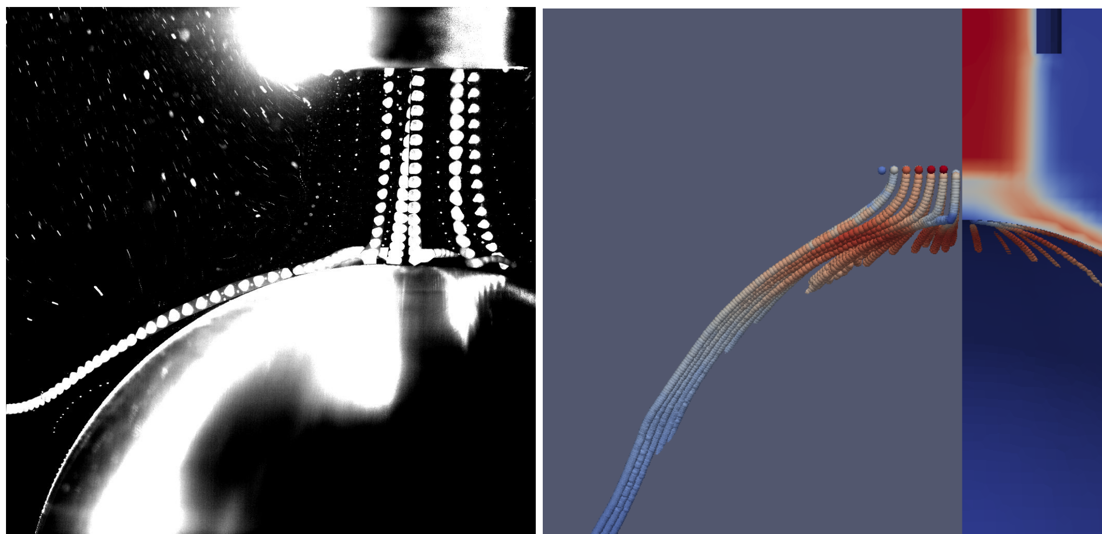
Figure 5.29: Comparison of particle tracks on cylindrical shaped sample.



(a) Experimental results

(b) Hybrid model results (coloured by velocity magnitude)

Figure 5.30: Comparison of particle tracks on conical shaped sample.



(a) Experimental results

(b) Hybrid model results (coloured by velocity magnitude)

Figure 5.31: Comparison of particle tracks on hemisphere shaped sample.

Chapter 6

Conclusions and further work

> The shortest distance between two points is a straight line.
>
> Archimedes

This thesis has presented the development and results of a new Hybrid CFD slurry model. The Hybrid model has been shown to provide shorter compute times for erosion modelling purposes. In addition, an experimental rig was designed, constructed, and used for gaining insight to the particle dynamics of the submerged jet impingement test and to validate the accuracy of the solver. This is necessary as highlighted in the paper by Lopez [11], which shows discrepancies between two of the largest CFD software packages.

The first two chapters of the thesis introduced the problem with current modelling methods and the reasons erosion is a problem for manufacturers in the mining and oil and gas industries. An overview of the current literature regarding CFD slurry modelling for erosion purposes was then presented. The reasons why slurry modelling is inherently difficult were shown, and important phenomena were highlighted. A review of other hybrid models in different fields (bubbles/spray) was carried out, and the methods adopted were examined. Experimental methods for validating CFD slurry solvers were also investigated, showing that the submerged jet impingement test is one of the most suitable.

Next, the development of the Hybrid solver was explained. The chapter gives reasons why each solver was chosen to be combined for creating the Hybrid model. The Tutorial by the

author already published at Chalmers University, Gothenburg [60] was the basis for the first part of the chapter. The issues that existed at the completion of the tutorial were addressed and the solutions to them were shown. The main changes to the code and why the changes were made were also discussed.

Following this, there is a chapter on the experimental equipment (Figure 6.1). The philosophy of the design of the rig is explained and a brief explanation of how particle image velocimetry works is included, along with the software and hardware used. Finally, the completed experimental setup is shown, along with how the experiments were carried out.

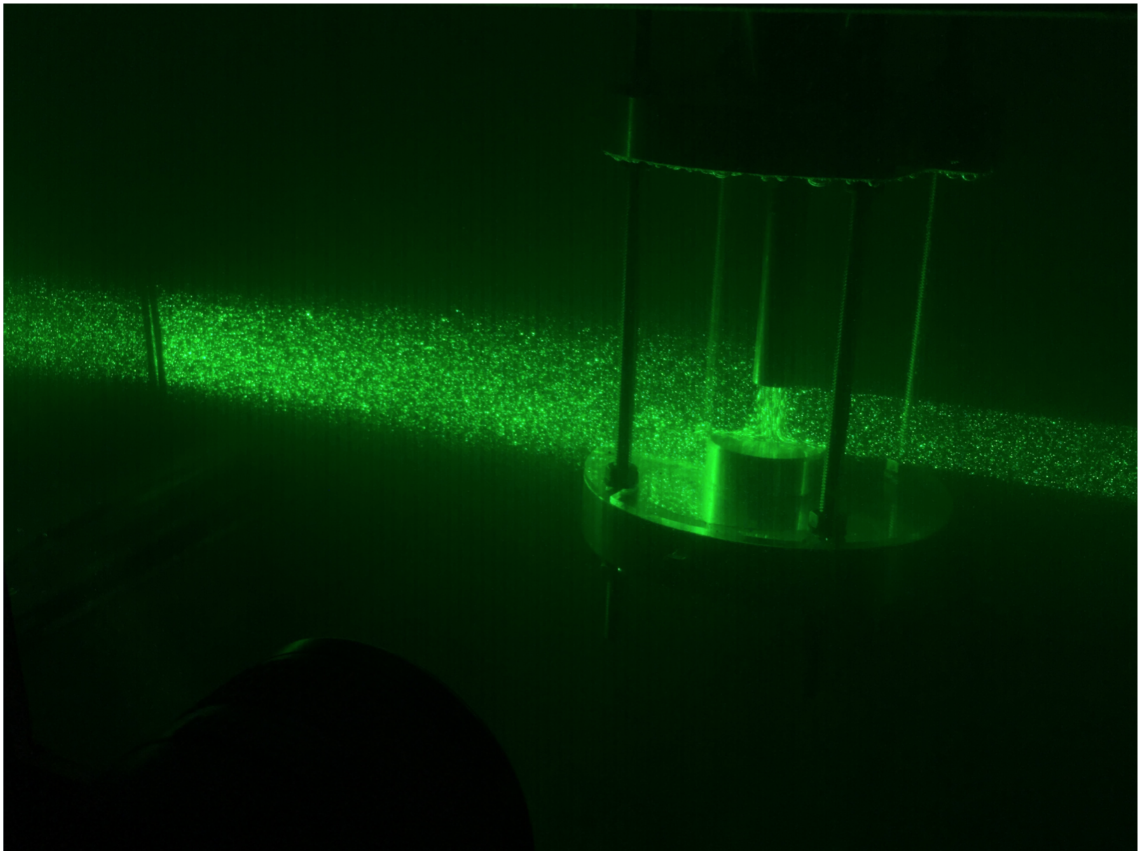


Figure 6.1: Green laser light shining through tank onto cylinder sample

In the last chapter the results gathered over the course of the PhD are presented. Firstly, there is a comparison of how three different CFD packages (ANSYS Fluent, Star-CCM+ and OpenFOAM) model the fluid phase of the submerged jet impingement test, combined with a comparison of the k-omega SST and k-epsilon turbulence models. Results from the Tutorial

are shown, along with results of the first phase comparison of the Hybrid model against the standard OpenFOAM models. The results were tested on three different geometries, to show that the solver is not geometry dependent. The second (particulate) phase of the Hybrid model is compared with the standard EL OpenFOAM model, DPMFoam, and lastly compared with experimental data from the test rig.

This literature review has shown that despite the extensive research into erosion and slurry modelling, the field is still in relative infancy. This thesis introduced some new insights into the field and showed promising results. Significant improvements to the accuracy of CFD models need to be made before there is a fully geometry independent erosion model [31]. A Hybrid model combining two solvers, a Euler-Euler and a Euler-Lagrange, was developed and validated against existing solvers and against experimental data. The model has been shown to decrease compute times, without significantly affecting the accuracy of results. Through the development of the PIV slurry test rig, data of particle tracks and of the fluid over different shaped samples have been obtained. These data can be used in the future for further validation work.

Although there is more work to be done, this has shown the potential for the hybrid model approach to be a solution for high concentration slurry erosion modelling. The benefits of only having one mesh, and the ability of being able to model high particle concentrations, along with the ability to change injection parameters in real-time are factors that make it ideal for combining with a mesh deforming solver. This gives a promising outlook for the future: for a geometry dependant erosion model.

Further work

Based on what has been done in this thesis, there are several extensions which could be done to continue progress:

1. The solver was not long completed before the end of the PhD, and so the code could be tidied up and debugged. This would improve the efficiency of it and enable shorter compute times compared to the standard EL model. Moving the lookupTable to memory

instead of a text file would also save computational effort, as the solver would not have to write and read a text file every timestep. The significance of this change is proportional to the number of injection sites, as each site has one row allocated in the text file lookupTable.

2. The solver's capabilities could be further developed by enabling transition from Lagrangian particles back to the Eulerian reference. This would allow the baffle/transition to be closer to the impingement surface, as the baffle could deal with 'bounce back' particles. This would be particularly helpful in more complex geometries where the location of particle tracks is unknown before the simulation is run, as it would allow the engineer to place the baffle at a convenient position close to the impact surface, without worrying about whether or not the particles would pass back through the surface.

3. The injection locations for the solver are currently in the cell centres, which leads to an unrealistic 'tidying' of the often chaotic particle positions. A random function could be added to select different locations over the cell face to create more of a scatter which would be more realistic. However, the problem of going from less to more information will always exist since an averaged model (Eulerian) is transferring data to a particulate (Lagrangian). See Section 3.4.2 for more details.

4. It was difficult to compare computational and theoretical particle track data, due to the nature of the data. A technique for comparing these data better would have enabled a more accurate comparison between what happens in reality and what the computer model simulates the particles motion to be.

5. Further simulations on more complex geometries would be beneficial, as this would test the 'geometry independence' aspect of the erosion model more.

References

[1] Love Hakansson Mikael Mortensen Rahman Sudiyo Berend van Wachem Bengt Andersson, Ronnie Andersson. *Computational Fluid Dynamics for Engineers.* Cambridge University Press, 1 edition, 2011.

[2] T. Frosell, M. Fripp, and E. Gutmark. Investigation of slurry concentration effects on solid particle erosion rate for an impinging jet. *Wear*, 342-343:33–43, 2015.

[3] Wenshan Peng and Xuewen Cao. Numerical simulation of solid particle erosion in pipe bends for liquid-solid flow. *Powder Technology*, 294:266–279, 2016.

[4] Gianandrea Vittorio Messa, Giacomo Ferrarese, and Stefano Malavasi. A mixed Euler-Euler/Euler-Lagrange approach to erosion prediction. *Wear*, 342-343:138–153, 2015.

[5] Amir Mansouri, Hadi Arabnejad Khanouki, Siamack A Shirazi, and Brenton S Mclaury. PARTICLE TRACKING VELOCIMETRY (PTV) MEASUREMENT OF ABRASIVE MICROPARTICLE IMPACT SPEED AND ANGLE IN BOTH AIR-SAND AND SLURRY EROSION TESTERS. *Proceedings of the ASME 2016 Fluids Engineering Division Summer Meeting*, pages 1–9, 2016.

[6] Lehrstuhl fur Verfahrenstechnik des industriellen Umweltschutzes. Particle image velocimetry. http://vtiu.unileoben.ac.at/en/labor-laserlabor-piv/, Accessed 2017.

[7] TSI Incorporated. How frame straddling works. http://www.tsi.com/uploadedFiles/_Site_Root/Products/Literature/Technical_Notes/HowFrame-StraddlingWorks.pdf, Accessed 2017.

[8] Alejandro Lopez. *Computational Fluid Dynamics study of Erosion Processes*. 2017.

[9] Velocimetry.net. Piv setup from velocimetry website: edited. http://velocimetry.net/images/center_txt_img.jpg, accessed 2017.

[10] F. Brownlie, T. Hodgkiess, A. Pearson, and A.m. Galloway. Effect of nitriding on the corrosive wear performance of a single and double layer stellite 6 weld cladding. *Wear*, 376-377:1279–1285, 2017.

[11] Alejandro López, William Nicholls, Matthew T. Stickland, and William M. Dempster. CFD study of Jet Impingement Test erosion using Ansys Fluent® and OpenFOAM®. *Computer Physics Communications*, 197:88–95, 2015.

[12] Weir group website. https://www.global.weir, accessed 2017.

[13] Weir Minerals. Animation showcasing weir minerals products in processing circuit. https://www.youtube.com/watch?v=kACwALB-xtA, accessed 2017.

[14] Australian government: Department of industry innovation and science. Australian energy update 2016. https://industry.gov.au/Office-of-the-Chief-Economist/Publications/Documents/aes/2016-australian-energy-statistics.pdf, accessed 2017.

[15] Clayton T. Crowe, John D Schwarzkopf, Martin Sommerfeld, and Yutaka Tsuji. *Multiphase flows: with droplets and particles*. CRC Press, 2012.

[16] Alessandro Manni. An introduction to twophaseeulerfoam with addition of an heat exchanger model. http://www.tfd.chalmers.se/~hani/kurser/OS_CFD_2014/Alessandro%20Manni/ReportAM.pdf, March 2017. PhD course in CFD with OpenSource software, Chalmers University, Sweden.

[17] Process Industries. COMPARISON OF A TWO-FLUID MODEL AND AN EULER-LAGRANGE MODEL FOR SIMULATION OF DENSE GAS-FLUIDIZED BEDS. 0(December):1–6, 2015.

[18] Andreas Hölzer and Martin Sommerfeld. Lattice Boltzmann simulations to determine drag, lift and torque acting on non-spherical particles. *Computers and Fluids*, 38(3):572–589, 2009.

[19] The OpenFOAM Foundation. Openfoam website. www.openfoam.org, accessed 2017.

[20] Resolved Analytics. Industry survey graph, showing openfoam to be the most popular open source cfd program. https://www.resolvedanalytics.com/theflux/comparing-popular-cfd-software-packages, accessed 2018.

[21] I. Finnie. Erosion of surfaces by solid particles. *Wear*, 3:87–103, 1960.

[22] I. Finnie. Some observations on the erosion of ductile metals. *Wear*, 19(1):81–90, jan 1972.

[23] Alan V. Levy and Pauline Chik. The effects of erodent composition and shape on the erosion of steel. *Wear*, 89(2):151–162, 1983.

[24] P Griffith. Erosion of Metallic Plate by Solid Particles Entrained in a Liquid Jet. 105(August 1983):215–222, 1983.

[25] Jae Hyung Kim, Hyung Goun Joo, and Kang Yong Lee. Simulation of solid particle erosion in WC-Ni coated wall using CFD. *Journal of Materials Processing Technology*, 2015.

[26] A. Gnanavelu, N. Kapur, A. Neville, J.F. Flores, and N. Ghorbani. A numerical investigation of a geometry independent integrated method to predict erosion rates in slurry erosion. *Wear*, 271(5-6):712–719, jun 2011.

[27] H. McI. Clark and L. C. Burmeister. The influence of the squeexe film on particle impact velocities in erosion. *Vol, Impatt Enqnq and Britain, Great and Lid, Pergamon Press and Engineering, Mechanical*, 12(3):415–426, 1992.

[28] H.McI Clark. The influence of the squeeze film in slurry erosion. *Wear*, 256(9-10):918–926, may 2004.

[29] J.a.C Humphrey. Fundamentals of fluid motion in erosion by solid particle impact. *International Journal of Heat and Fluid Flow*, 11(3):170–195, 1990.

[30] Y. Zhang, E.P. Reuterfors, B.S. McLaury, S.a. Shirazi, and E.F. Rybicki. Comparison of computed and measured particle velocities and erosion in water and air flows. *Wear*, 263(1-6):330–338, sep 2007.

[31] H.C. Meng and K.C. Ludema. Wear models and predictive equations: their form and content. *Wear*, 181-183:443–457, 1995.

[32] Amir Mansouri, Marzieh Mahdavi, Siamack A Shirazi, and Brenton S Mclaury. Paper No. 6130. (6130):1–10.

[33] Amir Mansouri, Hadi Arabnejad, Soroor Karimi, Siamack A. Shirazi, and Brenton S. McLaury. Improved CFD modeling and validation of erosion damage due to fine sand particles. *Wear*, 338-339:339–350, 2015.

[34] A. Mansouri, H Arabnejad, S.A. Shirazi, and B S McLaury. A combined CFD/experimental methodology for erosion prediction. *Wear*, 332-333:1090–1097, 2015.

[35] Alasdair MacKenzie, A Lopez, K Ritos, M T Stickland, and W M Dempster. A comparison of cfd software packages' ability to model a submerged jet. http://www.cfd.com.au/cfd_conf15/PDFs/152MAC.pdf, 2015.

[36] Mazdak Parsi, Kamyar Najmi, Fardis Najafifard, Shokrollah Hassani, Brenton S. McLaury, and Siamack a. Shirazi. A comprehensive review of solid particle erosion modeling for oil and gas wells and pipelines applications. *Journal of Natural Gas Science and Engineering*, 21:850–873, 2014.

[37] Christopher B. Solnordal, Chong Y. Wong, and Joan Boulanger. An experimental and numerical analysis of erosion caused by sand pneumatically conveyed through a standard pipe elbow. *Wear*, 336-337:43–57, 2015.

[38] Halima Hadžiahmetović, Nedim Hodžić, Damir Kahrimanović, and Ejub Džaferović. COMPUTATIONAL FLUID DYNAMICS (CFD) BASED EROSION PREDICTION MODEL IN ELBOWS. 3651:275–282, 1848.

[39] Chong Y Wong, Christopher B Solnordal, and Henri Morand. Flexible pipe erosion modelling. (December):1–7, 2015.

[40] Halima Hadžiahmetović, Nedim Hodžić, Damir Kahrimanović, and Ejub Džaferović. COMPUTATIONAL FLUID DYNAMICS (CFD) BASED EROSION PREDICTION MODEL IN ELBOWS. 3651:275–282, 2014.

[41] Jukai Chen, Yueshe Wang, Xiufeng Li, Renyang He, Shuang Han, and Yanlin Chen. Erosion prediction of liquid-particle two-phase flow in pipeline elbows via CFD DEM coupling method. *Powder Technology*, 275:182–187, 2015.

[42] G.J. Brown. Erosion prediction in slurry pipeline tee-junctions. *Applied Mathematical Modelling*, 26(2):155–170, feb 2002.

[43] Agrawal Madhusuden Lu, Yaojun. A Computational-Fluid-Dynamics-Based Eulerian-Granular Approach for Characterization of Sand Erosion in Multiphase-Flow Systems. (August), 2014.

[44] V.B. Nguyen, Q.B. Nguyen, Z.G. Liu, S. Wan, C.Y.H. Lim, and Y.W. Zhang. A combined numerical-experimental study on the effect of surface evolution on the water-sand multiphase flow characteristics and the material erosion behavior. *Wear*, 319(1-2):96–109, nov 2014.

[45] Milan Vujanovi, Zvonimir Petranovi??, Wilfried Edelbauer, and Neven Dui. Modelling spray and combustion processes in diesel engine by using the coupled Eulerian-Eulerian and Eulerian-Lagrangian method. *Energy Conversion and Management*, 125:15–25, 2016.

[46] Aurélia Vallier. *Simulations of cavitation - from the large vapour structures to the small bubble dynamics*. Number June. 2013.

[47] Mahdi Saeedipour, Stefan Pirker, Salar Bozorgi, and Simon Schneiderbauer. An eulerian-lagrangian hybrid model for the coarse-grid simulation of turbulent liquid jet breakup. *International Journal of Multiphase Flow*, 82:17 – 26, 2016.

[48] Shankar Subramaniam. Lagrangian-eulerian methods for multiphase flows. *Progress in Energy and Combustion Science*, 39(2):215 – 245, 2013.

[49] M. Herrmann. A parallel eulerian interface tracking/lagrangian point particle multi-scale coupling procedure. *Journal of Computational Physics*, 229(3):745 – 759, 2010.

[50] H Yu, L Goldsworthy, M Ghiji, P A Brandner, and V Garaniya. A parallel Volume of Fluid-Lagrangian Parcel Tracking coupling procedure for diesel spray modelling. 150:46–65, 2017.

[51] Kent E. Wardle and Henry G. Weller. Hybrid multiphase CFD solver for coupled dispersed/segregated flows in liquid-liquid extraction. *International Journal of Chemical Engineering*, 2013(1), 2013.

[52] Risa Okita, Yongli Zhang, Brenton S. McLaury, and Siamack a. Shirazi. Experimental and Computational Investigations to Evaluate the Effects of Fluid Viscosity and Particle Size on Erosion Damage. *Journal of Fluids Engineering*, 134(June 2012):061301, 2012.

[53] A. Gnanavelu, N. Kapur, A. Neville, and J.F. Flores. An integrated methodology for predicting material wear rates due to erosion. *Wear*, 267(11):1935–1944, oct 2009.

[54] Jae Hyung Kim, Hyung Goun Joo, and Kang Yong Lee. Simulation of solid particle erosion in WC-Ni coated wall using CFD. *Journal of Materials Processing Technology*, 2015.

[55] Q. B. Nguyen, C. Y H Lim, V. B. Nguyen, Y. M. Wan, B. Nai, Y. W. Zhang, and M. Gupta. Slurry erosion characteristics and erosion mechanisms of stainless steel. *Tribology International*, 79:1–7, 2014.

[56] Mehdi Azimian and Hans-Jörg Bart. Erosion investigations by means of a centrifugal accelerator erosion tester. *Wear*, 328-329:249–256, 2015.

[57] R.J.K. Wood, T.F. Jones, J. Ganeshalingam, and N.J. Miles. Comparison of predicted and experimental erosion estimates in slurry ducts. *Wear*, 256(9-10):937–947, may 2004.

[58] Christopher B. Solnordal, Chong Y. Wong, and Joan Boulanger. An experimental and numerical analysis of erosion caused by sand pneumatically conveyed through a standard pipe elbow. *Wear*, 336-337:43–57, 2015.

[59] T. Deng, M.S. Bingley, M.S.A. Bradley, and S.R. De Silva. A comparison of the gas-blast and centrifugal-accelerator erosion testers: The influence of particle dynamics. *Wear*, 265(7):945 – 955, 2008.

[60] Alasdair MacKenzie. A hybrid slurry cfd model: Euler-euler to euler-lagrange tutorial. http://www.tfd.chalmers.se/~hani/kurser/OS_CFD_2016/AlasdairMackenzie/tutorial1.pdf, March 2017. PhD course in CFD with OpenSource software, Chalmers University, Sweden.

[61] The OpenFOAM Foundation. Extended code guide: reactingtwophaseeulerfoam. http://www.openfoam.com/documentation/cpp-guide/html/reactingTwoPhaseEulerFoam_8C.html, accessed 2017.

[62] Phanindra.P.Thummala. Description of reactingtwophaseeulerfoam solver with a focus on mass transfer modeling terms. http://www.tfd.chalmers.se/~hani/kurser/OS_CFD_2016/PhanindraPrasadThummala/reactingTwoPhaseEulerFoam.pdf, March 2017. PhD course in CFD with OpenSource software, Chalmers University, Sweden.

[63] V.h. Bhusare, M.k. Dhiman, D.v. Kalaga, S. Roy, and J.b. Joshi. Cfd simulations of a bubble column with and without internals by using openfoam. *Chemical Engineering Journal*, 317:157–174, 2017.

[64] The OpenFOAM Foundation. Discrete particle modelling. https://openfoam.org/release/2-3-0/dpm/, accessed 2017.

[65] Franziska Greifzu, Christoph Kratzsch, Thomas Forgber, Friederike Lindner, and Rudiger Schwarze. Assessment of particle tracking models for dispersed particle laden flows implemented in openfoam and ansys fluent. *Engineering Applications of Computational Fluid Mechanics*, 10(1):30–43, 2015.

[66] Linmin Li, Baokuan Li, and Zhongqiu Liu. Modeling of spout-fluidized beds and investigation of drag closures using OpenFOAM. *Powder Technology*, 305:364–376, 2017.

[67] Alasdair Mackenzie. OpenFOAM. *Openfoam 11 workshop....*, page 3, 2017.

[68] A Lopez. LPT for erosion modeling in OpenFOAM. http://www.tfd.chalmers.se/~hani/kurser/OS_CFD_2013/AlejandroLopez/LPT_for_erosionModelling_report.pdf, 2014.

[69] OpenFOAM. Openfoam c++ source code guide. https://cpp.openfoam.org/v3/a03228_source.html, March Accessed 2019.

[70] OpenFOAM wiki. Object registry. https://openfoamwiki.net/index.php/OpenFOAM_guide/objectRegistry, accessed 2017.

[71] Kai Willasden. Meld is a visual diff and merge tool targeted at developers. http://meldmerge.org/, accessed 2017.

[72] M Stickland, T Scanlon, T Vidinha, W Dempster, and R Jaryczewski. The development and application of time resolved piv at the university of strathclyde. *Optical and Laser Diagnostics Institute of Physics Conference Series*, pages 199–207, 2003.

[73] Markus Raffel. *Particle image velocimetry : a practical guide*. Springer, Heidelberg New York, 2007.

[74] Hydromar. Waterjet cutting - how does it work? http://www.hydromar.co.uk/waterjet-cutting-how-does-it-work.aspx, accessed 2017.

[75] Phontron LTD. User's manualfastcam ultima1024. http://www.highspeedimaging.com/media/photron_manuals/FASTCAM-u1024_HW_manual.pdf, accessed 2017.

[76] Lee laser inc. Frequency-doubled,diode-pumped nd:yag laser. http://leelaser.com/wp-content/uploads/2016/01/LDP-100MQG_multimode_R41.pdf, accessed 2017.

[77] Dantec Dynamics. Dynamicstudio-the most user-friendly and comprehensive software platform for scientific imaging. https://www.dantecdynamics.com/dynamicstudio, accessed 2017.

[78] Alasdair MacKenzie, Vanja Skuric, MT Stickland, and WM Dempster. A new hybrid slurry cfd model compared with experimental results. http://openfoamworkshop.org, 2017.

[79] Alasdair MacKenzie. A hybrid slurry cfd model: Euler-euler to euler-lagrange (in development). http://adhesion.ucd.ie/5th_OpenFOAM_User_Meeting/Home.html, 2017.

[80] Alasdair MacKenzie, Alejandro Lopez, Matthew Stickland, and William Dempster. A combined euler-euler euler-lagrange slurry model. http://sourceforge.net/projects/openfoam-extend/files/OpenFOAM_Workshops/OFW11_2016_Guimaraes/files/slides/OFWP008.pdf, 2016.

[81] A Mansouri, S A Shirazi, and B S Mclaury. Experimental and numerical investigation of the effect of viscosity and particle size on erosion damage caused by solid particles. *ASME*, pages 1–10, 2015.

[82] Van Bo Nguyen, Hee Joo Poh, and Yong-Wei Zhang. Predicting shot peening coverage using multiphase computational fluid dynamics simulations. *Powder Technology*, 256:100–112, apr 2014.

[83] Dr Juretic. cfmesh: An advanced library of meshing algorithms and workflows. https://cfmesh.com/, accessed 2017.

Appendices

Appendix A

Tutorial on the development of the Hybrid Model, published at Chalmers University

Cite as: Mackenzie, A.: A hybrid slurry CFD model. In Proceedings of CFD with OpenSource Software, 2016, Edited by Nilsson. H., http://www.tfd.chalmers.se/~hani/kurser/OS_CFD_2016

CFD with OpenSource software

A course at Chalmers University of Technology
Taught by Håkan Nilsson

A hybrid slurry CFD model: Euler-Euler to Euler-Lagrange

Developed for OpenFOAM-3.0.x
Requires: PyFoam

Author:
Alasdair Mackenzie
alasdair.mackenzie.100@strath.ac.uk

Peer reviewed by:
Shengnan Liu
Håkan Nilsson

Licensed under CC-BY-NC-SA, https://creativecommons.org/licenses/

Disclaimer: This is a student project work, done as part of a course where OpenFOAM and some other OpenSource software are introduced to the students. Any reader should be aware that it might not be free of errors. Still, it might be useful for someone who would like learn some details similar to the ones presented in the report and in the accompanying files. The material has gone through a review process. The role of the reviewer is to go through the tutorial and make sure that it works, that it is possible to follow, and to some extent correct the writing. The reviewer has no responsibility for the contents.

January 9, 2017

Contents

1 The problem with erosion modelling **5**

2 Choosing the two solvers **6**

2.1 Euler-Lagrange Model 6

2.1.1 Modifying DPMFoam 8

2.1.2 The 0 directory 8

2.1.3 The constant directory 9

2.1.4 The system directory 10

2.1.5 Running the solver 10

2.2 Euler-Euler Model 12

2.2.1 Modifying reactingTwoPhaseEulerFoam 12

2.2.2 The 0 directory 12

2.2.3 The constant directory 14

2.2.4 The system directory 14

2.2.5 Running the solver 14

3 Results from test case **16**

4 Start new solver **19**

4.1 Creating baffles 19

4.2 Creating regions 21

4.3 Create own solver 22

4.4 Preparation for interpolation 22

4.5 Add patchToPatchInterpolation 25

5 Addition of particles to solver **28**

5.1 Editing injection method 28

5.1.1 Editing the KinematicLookupTableInjection folder 29

5.1.2 Editing the InjectionModel folder 31

5.2 Writing lookupTable in solver 31

5.3 Turning off interpolator 33

5.4 Adding DPMFoam 34

6 Hybrid test case **36**

6.1 Results 37

7 Conclusions and further work **42**

8 Study questions **43**

Learning outcomes

The reader will learn:

How to use it:

- how to use an Euler-Euler dispersed phase slurry model: `reactingTwoPhaseEulerFoam`
- how to use an Euler-Lagrange slurry model: `DPMFoam`
- how to use `createBaffles`
- how to use `splitMeshRegions` and run regions in sequence
- how to use `patchToPatchInterpolation`

The theory of it:

- the theory of `pyFoamPlotWatcher`
- an outline of the theory of the other utilities

How it is implemented:

- how to implement a new solver from two existing ones

How to modify it:

- how to modify files for solver to read as boundary conditions
- how to edit and add code to a solver

Acknowledgements

I would like to acknowledge the Institute of Mechanical Engineers for their financial support for this course, which otherwise may not have been possible.

Thanks also needs to be given to Professor Nilsson for his time and effort put into this course, it is very much appreciated.

Chapter 1

The problem with erosion modelling

Erosion caused by dense slurry flows is a common problem and computational modelling of the process is very difficult. The aim of this work is to find a more efficient way of modelling the fluid and particles, so that the erosion modelling efficiency can also be improved.

Modelling erosion of slurries using computational fluid dynamics (CFD) can be done in two ways: Euler-Euler (EE) or Euler-Lagrange (EL). The Euler-Euler technique models both the particles and fluid as continuous phases. The technique is volume averaged, and therefore not computationally expensive. Euler-Lagrange however, models particles as separate objects having their own velocity vectors. This means the particles have physical vector values near walls, which is substantially better for erosion modelling. The downside of Euler-Lagrange is that it is more computationally demanding, with varying degrees depending on one, two or four way coupling.

The tutorial will go through the two solvers chosen and show how they work, and then show how to combine them. This will create a hybrid model which should enable shorter computational times, but crucially give similar results.

Chapter 2

Choosing the two solvers

Before a hybrid model was developed, the two solvers were set up and run on a simple case, to ensure they both worked individually. A simple square-section pipe with a 90° bend was chosen, which had an inlet, outlet, and walls for boundary conditions. The geometry as seen in Figure 2.1 was taken from the tutorial by Alejandro Lopez [1], on lagrangian particle tracking. The blockMeshDict for this geometry can be found in the attached files. The pipe is 10mm square section and 100mm long, and with only 3000 cells the solution time is low. Accuracy would be improved with a finer mesh, however it is more desirable for shorter run times at this development stage. The purpose of this tutorial is not to get accurate results, but rather to develop a new model.

OpenFOAM contains various multiphase solvers and tutorials, and the idea is to find a tutorial that best matches your needs, and modify it to suit your case. Tutorials can be found by typing in the terminal window:

```
cd $FOAM_TUTORIALS
```

Or by using the alias:

```
tut
```

The standard OpenFOAM tutorials do not contain liquid/particle flow as is required; therefore the standard tutorials will need modified.

2.1 Euler-Lagrange Model

Lagrangian particle tracking uses Newton's equations of motion to determine the particle trajectories. There are three different coupling possibilities:

- one way coupled: fluid affects particles
- two way coupled: as above, but particles also affect fluid
- four way coupled: as above, but particle also affects its neighbour, and neighbour affects the particle. (i.e A affects B, and B affects A)

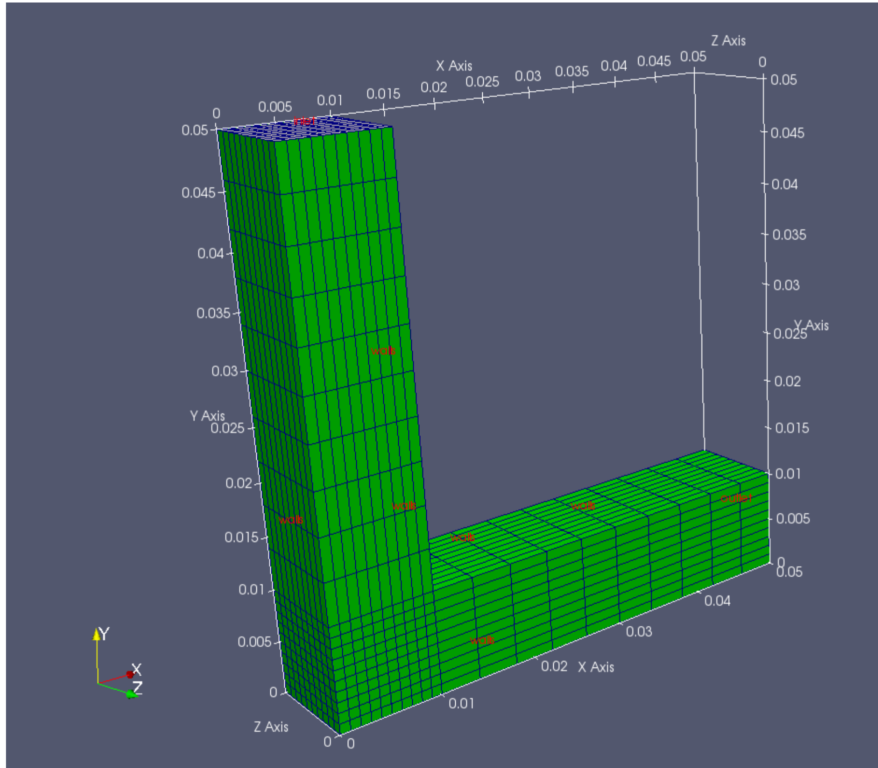


Figure 2.1: Geometry for test case

Two way coupling is used here since the Euler-Euler is also two way coupled. The PIMPLE solver is used for the fluid flow; the name coming from PISO-SIMPLE. PISO stands for Pressure Implicit Splitting of Operators, and SIMPLE stands for Semi-Implicit Method for Pressure Linked Equations and is used for steady state problems. This PIMPLE solver is used in combination with the kinematic cloud, to form a two way coupled algorithm. In the fvSolution dictionary, it is possible to set the number of PIMPLE loops per kinematic cloud loop by changing the variables in the PIMPLE section:

```
PIMPLE
{
    nOuterCorrectors 3; //or 2 for two loops;
    nCorrectors      4;
    nNonOrthogonalCorrectors 4;
    pRefCell         0;
    pRefValue        0;
}
```

The number nOuterCorrectors being set to 3, means the PIMPLE iterates three times for every kinematic cloud loop. More information on the PIMPLE solver can be found in the tutorial by *Olle Penttinen* [2], or on the online storyboard on CFD-online [3].

2.1.1 Modifying DPMFoam

The solver DPMFoam was chosen for the Euler-Lagrange simulation, as it is the most appropriate Lagrangian solver from the tutorials. Lopez [1] has shown that it can be used to model slurry flows, and more details can be found in his tutorial. The DPMFoam tutorial should be copied into the 'run' directory as follows:

```
tut
cd lagrangian
cp -r DPMFoam $FOAM_RUN
run
cd DPMFoam
```

These commands will copy the DPMFoam tutorial from the tutorials directory to your 'run' directory, then open the DPMFoam folder. Typing `pwd` will yield:

```
/home/username/OpenFOAM/username-3.0.x/run/DPMFoam
```

Once in this directory, the new model can be set up as shown.

2.1.2 The 0 directory

The following files should be created in the 0 directory:

```
└─ 0/
   ├─ epsilon.water
   ├─ k.water
   ├─ nuTilda
   ├─ nut.water
   ├─ p
   └─ U.water
```

All files in the 0 directory need to have boundary conditions assigned for each patch: in this case inlet, walls and outlet. The epsilon file is given as an example below, and all other files can be created with the same format (by copying from other tutorials), using boundary conditions that suit.

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  3.0.x                                 |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       volScalarField;
```

```
    location    "0";
    object      epsilon;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dimensions      [0 2 -3 0 0 0 0];

internalField   uniform 51;

boundaryField
{
    inlet
    {
        type            fixedValue;
        value           uniform 51;
    }
    wall
    {
        type            epsilonWallFunction;
        value           uniform 0;
    }
    outlet
    {
        type            zeroGradient;
    }
}
// ************************************************************************* //
```

The value of epsilon has to be based on the inlet velocity and the geometry. The parameters used in this tutorial are as follows:

Parameter	Value
epsilon.water	51
k.water	0.36
nut.water	0 (no turbulent viscosity)
p	0
U.water	5

Care must be taken to ensure boundary conditions are in the correct direction, so for example ‘U’ must be `(0 -5 0)`, i.e the negative y-direction.

2.1.3 The constant directory

The following files should be created in the constant directory, if not there already:

```
constant
    ├── g
    ├── kinematicCloudProperties
    ├── polyMesh
    │   └── blockMeshDict
    ├── RASProperties
    ├── transportProperties
    └── turbulenceProperties.water
```

Note: `blockMeshDict` has moved from OpenFOAM-3.0- onwards to the system directory, however this tutorial is using some cases from OpenFOAM-2.3.x.

The 'g' file contains the magnitude of gravity and its direction.'kinematicCloudProperties' defines parameters for the lagrangian particles. Things such as velocity, density, coupling, and calculation frequency are in this dictionary. 'RASproperties' defines which turbulence model is used (k-epsilon in this case). The 'turbulenceProperties' defines the RAS turbulence model, again the k-epsilon.

2.1.4 The system directory

The following files should be created in the system directory:

```
system
  ├── controlDict
  ├── fvSchemes
  └── fvSolution
```

The controlDict can be used from the DPMFoam tutorial. fvSchemes tells OpenFOAM what numerical schemes to use for each parameter, and fvSolution gives the user control over each solver for each parameter.

2.1.5 Running the solver

Once all files in their correct place, the solver can be run using the following commands:

```
blockMesh
DPMFoam >& log&
```

This will first use blockMesh to mesh the domain, then start running the solver DPMFoam. The command `>& log&` puts error messages to the file named 'log', and also puts the job to the background of the terminal window. The file can be named whatever the user wishes. Typing `fg` will bring the job to the foreground, so that it can be changed. Typing `ctr+c` will cancel the job, and `ctrl+z` will pause the job, which can be resumed in the background by typing `bg`. If pyFoam is installed, the application 'pyFoamPlotWatcher' can be used, by typing:

```
pyFoamPlotWatcher.py log
```

Where `log` is the log filename. If `pyFoamPlotWatcher.py --help` is typed, the available options for the application will be shown. The short description says:

```
Usage
=====
  pyFoamPlotWatcher.py [options] <logfile>

Gets the name of a logfile which is assumed to be the output of a OpenFOAM-
solver. Parses the logfile for information about the convergence of the solver
and generates gnuplot-graphs. Watches the file until interrupted.
```

This is a great tool to monitor residuals, and to ensure convergence criteria are met. Figure 2.2 shows the resulting gnuplot graph from the DPMFoam solver. One of the reasons this residual plot

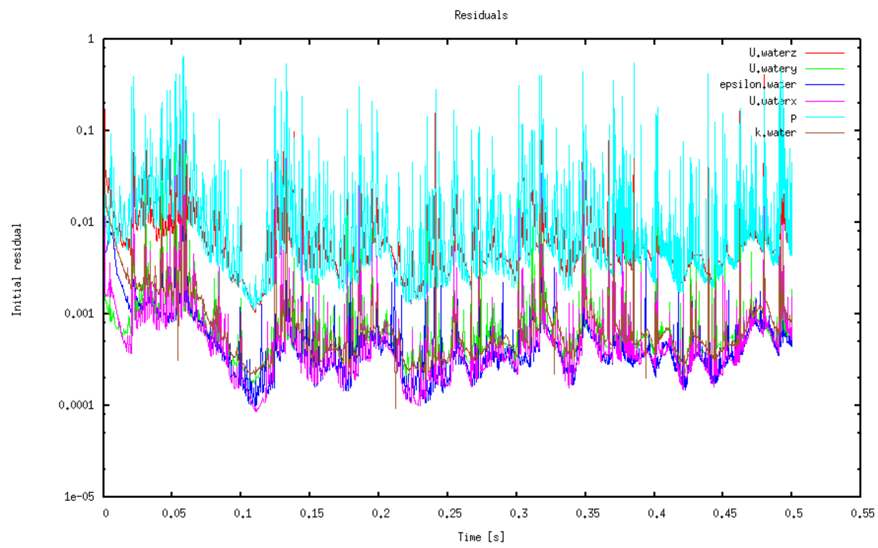


Figure 2.2: Residuals from the Euler-Lagrange model

is so unstable is because of the PIMPLE loop set up. Each loop the solver begins to converge but when a particle cloud is inserted the residuals start to increase again. More PIMPLE loops would bring the residuals down further, but only until the next particle package was introduced.

2.2 Euler-Euler Model

The solver reactingTwoPhaseEulerFoam was used as the Euler-Euler model. It is a solver for two compressible fluid phases with a common pressure, but otherwise separate properties. Although the two 'fluids' are not compressible, this can be ignored here as other models were tried and tested, but none gave results as close to the Lagrangian simulation as reactingTwoPhaseEulerFoam. twoPhaseEulerFoam was considered however the energy equation cannot be turned off in this (without editing the source code), which causes convergence problems.

To compare the two chosen models both were run as transient cases for 0.5 seconds real-time, with the same set-up parameters. The EE case took 684 seconds, whereas the EL took 1667 seconds; so the EE is almost 2.5 times faster than the EL for this simple case. This justifies the plan to use a hybrid model, especially when considering the time difference gets larger with more complex problems.

Note: There is a bug in myReactingTwoPhaseEulerFoam. Two files need replaced, and can be downloaded from `www.openfoam.org/mantisbt/view.php?id=1902`. The files which need replacing are 'IsothermalModel.C/H', and should be replaced in the folder:

```
$WM_PROJECT_DIR/applications/solvers/multiphase/reactingEulerFoam/phaseSystems/
phaseModel/IsothermalPhaseModel
```

After replacing these files, the models need to be recompiled by moving back into the 'phaseSystems' directory (since the 'Make' folder is here) and typing `wclean; wmake` .

2.2.1 Modifying reactingTwoPhaseEulerFoam

The fluidisedBed tutorial is to be copied in the same manner as DPMFoam was copied over to the run directory. It is found in:

```
.../OpenFOAM-3.0.x/tutorials/multiphase/reactingTwoPhaseEulerFoam/RAS/fluidisedBed
```

The folder before fluidisedBed is called RAS, indicating this is a Reynolds Averaged Simulation. Laminar or LES can also be used if required.

2.2.2 The 0 directory

The following files should be created in the 0 directory:

```
└─ 0/
   ├─ alpha.particles
   ├─ alpha.water
   ├─ epsilon.water
   ├─ k.water
   ├─ nut.particles
   ├─ nut.water
   ├─ p
   ├─ Theta.particles
   ├─ T.particles
   └─ T.water
```

```
│
├── U.particles
└── U.water
```

These should be filled in the same way as for the Euler-Lagrange simulation. For example, alpha.particles is as follows:

```
/*-------------------------------*- C++ -*---------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  3.0.x                                 |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       volScalarField;
    object      alpha.particles;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dimensions      [0 0 0 0 0 0 0];

internalField   uniform 0;

boundaryField
{
    inlet
    {
        type            fixedValue;
        value           0.039;
    }

    outlet
    {
        type            zeroGradient;
    }

    walls
    {
        type            zeroGradient;
    }
}

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

The parameters used for the Euler-Euler case should as similar as possible to the Euler-Lagrange. Below is a table with the parameters used.

Parameter	Value
alpha.particles	0.039 (10% mass concentration)
alpha.water	0.961
epsilon.water	51
k.water	0.36
nut.particles	0 (no turbulent viscosity)
nut.water	0
p	1e5
Theta.particles	1e-7 (granular temperature)
T.particles	300
T.water	300
U.particles	5
U.water	5

These parameters should be inserted into the appropriate dictionaries, with the boundary conditions required.

2.2.3 The constant directory

The following files should be created in the constant directory:

```
constant
  ├─ g
  ├─ phaseProperties
  ├─ polyMesh
  ├─ thermoPhysicalProperties.particles
  ├─ thermoPhysicalProperties.water
  ├─ turbulenceProperties.particles
  └─ turbulenceProperties.water
```

The constant directory is similar to the DPMFoam one, but kinematicCloudProperties has been replaced with phaseProperties, and there is also thermoPhysicalProperties since there is a possibility for heat exchange to be taken into account. blockMeshDict has also moved to the 'system' directory.

2.2.4 The system directory

The following files should be created in the system directory:

```
system
  ├─ blockMeshDict
  ├─ controlDict
  ├─ fvSchemes
  └─ fvSolution
```

2.2.5 Running the solver

Now that all of the files are in place, we can run the solver. The blockMeshDict is the same as before, so this can be copied from the Euler-Lagrange simulation. The following commands will run

the solver:

```
blockMesh
reactingTwoPhaseEulerFoam >& log&
```

Chapter 3

Results from test case

The test cases were both ran, and made sure to be similar in appearance. Again it must be remembered that this tutorial is not about results, but rather the application of the solvers in a new way.

A comparison of the water's velocity magnitude is shown below, in Figure 3.1. This is a 2D slice, normal to the Z-plane, and shows the Euler-Lagrange on the left, and Euler-Euler on the right. It is quite clear that the velocities are similar enough to proceed. Results were taken from the last time step in all figures.

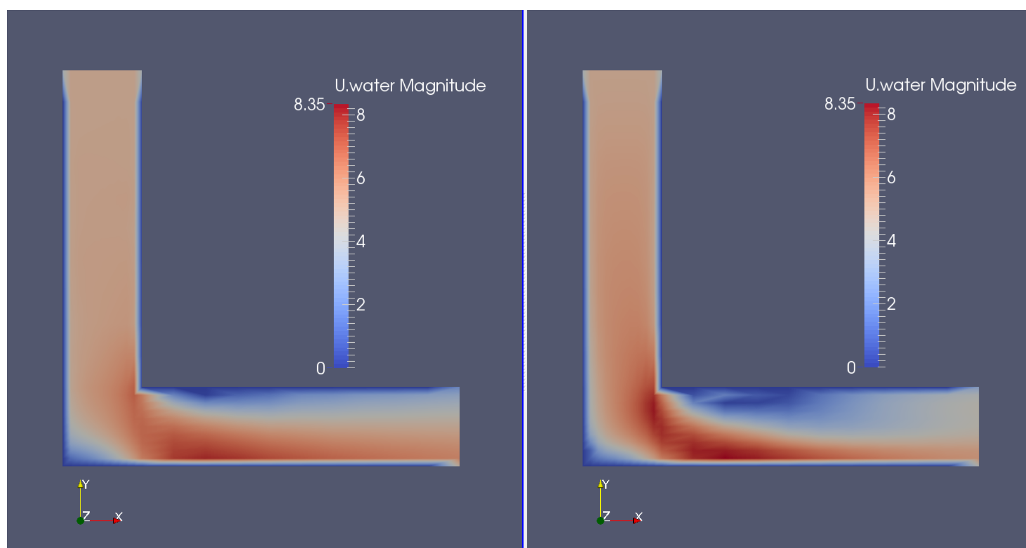


Figure 3.1: Velocity magnitude of water: EL and EE

Getting this comparison in Paraview is quite straight forward. If in the EL case folder, we need to create a case.OpenFOAM file (so that we can open it from inside Paraview). This is done by typing

```
touch casename.OpenFOAM
```

whilst in the Euler-Lagrange directory. With this file made, we can then change directory to the

Euler-Euler folder, and type

```
paraFoam
```

This will open Paraview and automatically create a temporary file called 'casename.OpenFOAM' (note: casename is the name of the case you have named). We can then select split screen view, and simply open the .OpenFOAM file which we created in the Euler-Lagrange folder.

Below in Figure 3.2 a comparison of the velocity magnitudes of the second (dispersed) phase is shown. This is slightly more difficult to compare, as one is Eulerian and the other Lagrangian, or one is continua the other particles. It can be seen that the Lagrangian particles are travelling faster than the Eulerian phase, since the Eulerian contour plot does not go as dark red as the legend does. This is because the legend has been rescaled, as the maximum velocity for the Eulerian phase was 7m/s. This difference of 2-3m/s could be down to its averaged nature, as there may only be one particle in the Lagrangian simulation that has the velocity of 9.64m/s.

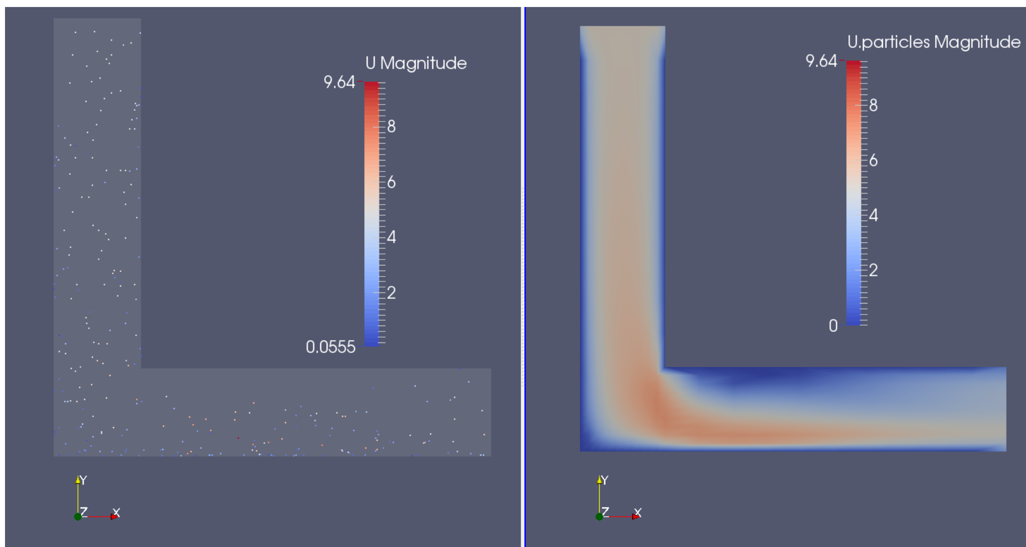


Figure 3.2: Velocity magnitude of second phase: EL and EE

Figure 3.3 below shows the phase fractions, alpha water, and alpha particles. Alpha denotes the phase fraction of the phase name, and in every cell in the domain alpha.particles + alpha.water = 1.

With the results of the individual cases being similar, the task of combining the models can now be started.

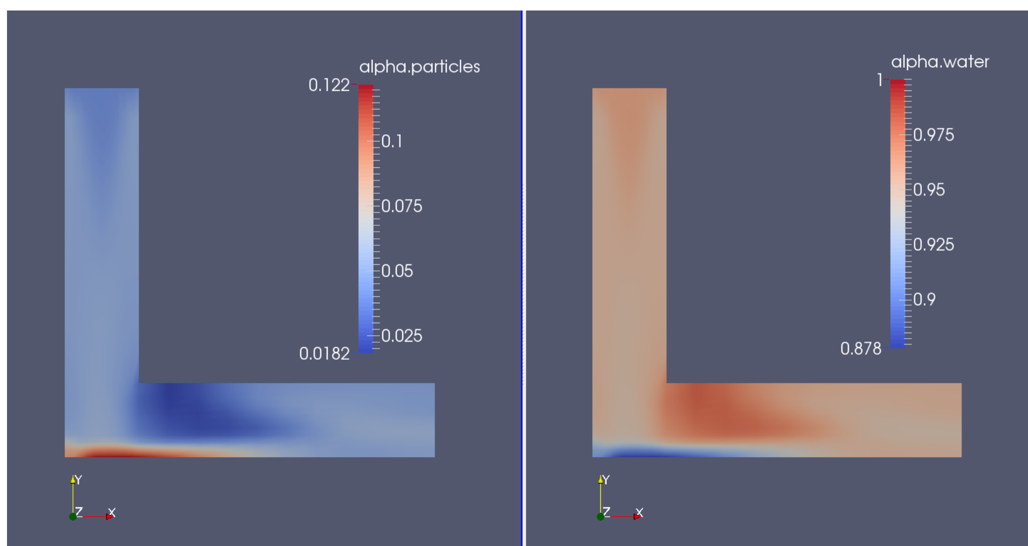


Figure 3.3: Alpha concentration of second phase: EL and EE

Chapter 4

Start new solver

In the new hybrid model, sections of code from DPMFoam were added to reactingTwoPhaseEulerFoam. The test domain was split into two regions, with the continuous phase running through both regions unaffected. The second (particle) phase was run as a fluid through the Euler-Euler region, and then injected as particles in the Lagrangian region, based on the Eulerian result at the interface. This should require less computational effort than if the whole domain had Lagrangian particles throughout. The steps to make this work are summarised:

- Create baffles: Enable boundary conditions to be set
- Create regions: Enable solving regions of domain in series
- Add interpolator: Enable communication between regions
- Add DPMFoam code: Particles were added to the code

To do these things, the pipe bend case from before should be copied into a new folder, so that it can be edited to suit the new hybrid model. The new folder should be called 'hybrid'.

4.1 Creating baffles

The square pipe was to be split into two regions, one for EE, and one for EL. There was also the requirement for having boundary conditions where the transition takes place, so that the particles injected will have the same values as the EE simulation. With these things in mind, creating baffles and then splitting the mesh into regions seemed the best option.

To create baffles, a utility file called 'createBafflesDict' needs to be added to the 'system' folder. These are already included in some tutorials; they can be found by typing:

```
foam
find -iname createBafflesDict
```

This gives a list of locations where these files of this name can be found (note: the 'i' in 'iname' makes the search case insensitive).

After finding a file, it can be copied into the system directory by copying the file. The following commands will copy the file into the system directory, with the case folder being called 'hybrid':

```
run
cd hybrid
cp ../../../OpenFOAM-3.0.x/tutorials/heatTransfer/chtMultiRegionSimpleFoam/
heatExchanger/system/air/createBafflesDict ./system/
```

The dictionary can be edited as required, or copied from the attached files with this report. The dictionary gives directions as to the type of baffle, and what it should be based on. In this case, making an .stl file and using it is a baffle surface is the method shown. In the dictionary, we should define the name of our .stl file next to the 'name' parameter. Other values can be checked in the attached case.

The .stl file now has to be made, in the constant/triSurface directory. The file in this tutorial is called 'baffle1D.stl', and the contents of the file are as follows:

```
solid ascii
 facet normal 0 1 0
  outer loop
   vertex 0 0.01 0
   vertex 0 0.01 -0.01
   vertex 0.01 0.01 0
  endloop
 endfacet
 facet normal 0 1 0
  outer loop
   vertex 0.01 0.01 -0.01
   vertex 0.01 0.01 0
   vertex 0 0.01 -0.01
  endloop
 endfacet
endsolid
```

This is a simple surface, however more complex surfaces could be drawn by using any CAD software, like FreeCAD[4] for example.

Now that the dictionary and .stl file is in place, the commands can be carried out to create the mesh and baffles, when in the 'hybrid' directory.

```
blockMesh
createBaffles -overwrite
```

This will create the blockMesh and also create the baffles. There will be a master and a slave patch created in each file of the 0/ directory. These patches will allow boundary conditions to be set on them, allowing a transition from Euler to Lagrangian for the second phase. The 'overwrite' flag ensures any previous baffles are overwritten. More information on the createBaffles utility can be found by looking at the .C file, using the `find` command as previously described.

```
paraFoam
```

Can be entered to visualise the new baffles. Figure 4.1 shows the location of both baffles, with master on top of the slave patch (top being the positive y-direction).

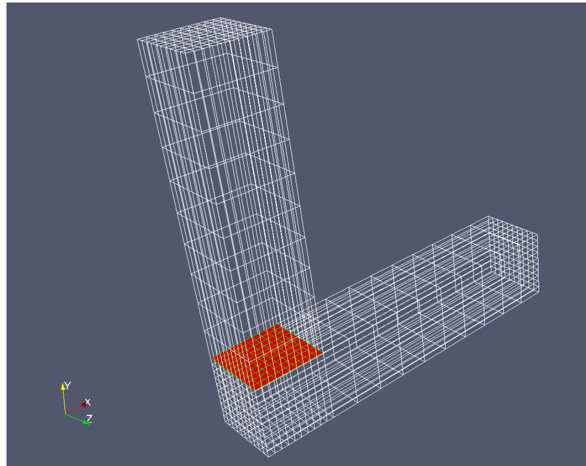


Figure 4.1: View of mesh and baffles in red

4.2 Creating regions

Now that the baffles are in place for injecting, a method of solving the regions in series is required. This is so the EE region can be solved before the EL region in the iterative loop. The utility called 'splitMeshRegions' can be used to split the domain into two regions along the patch line. More can be found on the utility by looking at the .C file, using the technique already provided. The following command is carried out in the working directory:

```
splitMeshRegions -cellZones -overwrite >log.splitMesh
```

`-cellZones` and `-overwrite` split the cellZones into separate regions and overwrites the existing mesh respectively. After the command is executed, a log file will be created called 'log.splitMesh'. This should explain that two regions have been created, giving a cell count for each region, and also what each region is called. Figure 4.1 shows that the bottom region is larger than the top, and therefore is the one with more cells. If in doubt, paraview can be opened and checked which domain is which. From the log file created, *domain0* has 2000 cells, whereas *domain1* has 1000, therefore *domain0* is on the bottom, *domain1* on top. *fvSchemes* and *fvSolution* have also been created in the system/domain(0/1) folders, enabling different schemes to be used in each region. A file called *regionProperties* is made in the constant/ folder, and this is where the region type is defined e.g.

```
regions
(
    fluid       (domain1)
    solid       (domain0)
);
```

In the 0 directory, folders called *domain0* and *domain1* are created, which contain replicas of the files in the 0 directory, however they only include the patches that are contained within the domain. *domain0* (the bottom) contains the patches; `outlet, walls,` and `baffleFace1D_slave`, whereas *domain1* contains the patches; `inlet, walls`, and `baffleFace1D_master`.

4.3 Create own solver

So far, the modifications have been made to the working case directory using utilities. There is now the requirement to edit how the solver works, and therefore a new solver should be created. When creating your own solver, rather than editing the source code, a copy of an existing solver which does a similar thing to your own should be used. The method shown here is the one taught by Håkan Nilsson in Chalmers University (Sweden), and can be seen in Lopez's tutorial[1] page 17.

Copies of directories from OpenFOAM need to be created in the user directory like so:

```
cd $WM_PROJECT_USER_DIR
mkdir -p applications/solvers/multiphase/
cp -r $FOAM_SOLVERS/multiphase/reactingEulerFoam applications/solvers/multiphase/
```

These commands will make a copy of reactingEulerFoam to your own working directory. reactingTwoPhaseEulerFoam needs to be called something different to save confusion, and some files will also need editing.

```
cd applications/solvers/multiphase/reactingEulerFoam
mv reactingTwoPhaseEulerFoam/ hybridModel
cd hybridModel
g Make/*
```

These commands will change the name of the solver to 'hybridModel', and open the files in the 'Make' folder in *gedit*. Note: In this tutorial an alias has been made in the .bashrc file which saves the user typing `gedit` every time they want to edit files (`alias g="gedit"`), this can be added if desired.

Inside Make/files, the parts called 'reactingTwoPhaseEulerFoam' should be changed to 'hybridModel', and the second line should be changed to the the user's application bin, not the OpenFOAM one. This can be done manually, or by the following *sed* commands (from the 'hybridModel' folder):

```
sed -i s/reactingTwoPhaseEulerFoam/hybridModel/g Make/files
sed -i s/FOAM_APPBIN/FOAM_USER_APPBIN/g Make/files
mv reactingTwoPhaseEulerFoam.C hybridModel.C
```

These changes can be visually checked by opening Make/files. The .C file also is changed to reflect the new name of the model.

After these edits, the new solver needs to be compiled. This is done by doing `wclean;wmake` in the hybridModel folder. The compilation can be checked by typing:

```
which hybridModel
```

This should produce the directory where the solver is located.

```
...User-3.0.x/platforms/linux64GccDPInt64Opt/bin/hybridModel
```

This solver can be tested by typing `hybridModel` in the run/hybridModel folder. At this stage it should run, but only in one region and not correctly, as there are more modifications to be made.

4.4 Preparation for interpolation

The solver is now created, but it does the same job as it did before, just with a different name and location. The next step is to add interpolation between region0 and region1 using patchToPatchInterpolation. The description given by OpenFOAM explains that it is an 'Interpolation class dealing with transfer of data between two primitivePatches'. The thread on CFD Online referenced was a great help, and thanks is required to user **bigphil** [5].

Figure 4.2 shows the loop necessary for the interpolators to work within the solver. For the regions to be solved one after the other, the solver has to be edited and the regions defined. This next section will describe these steps, in adding the interpolation header files, and adding the files associated with the regions.

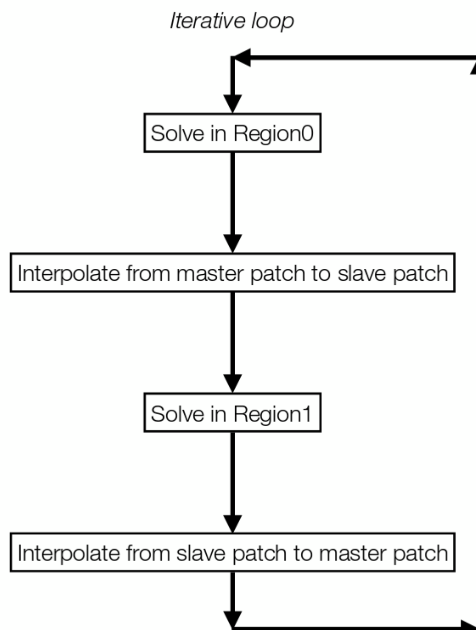


Figure 4.2: Depiction of iterative loop

In the top section of our solver, two lines should be added, which represent two new files added to the solver. The new section should look like:

```
#include "fvCFD.H"
#include "twoPhaseSystem.H"
#include "phaseCompressibleTurbulenceModel.H"
#include "fixedFluxPressureFvPatchScalarField.H"
#include "pimpleControl.H"
#include "localEulerDdtScheme.H"
#include "fvcSmooth.H"
#include "patchToPatchInterpolation.H" // for patchToPatch
#include "regionProperties.H" // for regions/domains
```

Now that the files are referenced, the solver needs to know what is in them, and they need to be in a place where the solver can find them. patchToPatchInterpolation.H is copied into the model directory, by typing:

```
cp $WM_PROJECT_DIR/src/OpenFOAM/interpolations/patchToPatchInterpolation
/patchToPatchInterpolation.H .
```

The regionProperties.H file is sourced in the Make/options file. It can be found by doing a 'find -iname' in the OpenFOAM main folder, and then referenced accordingly in the options folder. Below the 'sampling' line, the following lines should be added,

```
-I$(LIB_SRC)/OpenFOAM/lnInclude \
-I$(LIB_SRC)/regionModels/regionModel/lnInclude
```

and below the '-lsampling', the following,

```
-lOpenFOAM \
-lregionModels
```

remembering to add the backslash to the above line, and always exclude the backslash from the last line.

The solver can now be recompiled (`wclean;wmake`) without any errors. If the files were not referenced properly, the compilation would have printed the error. Currently the solver does not solve each region separately; to change this we can use examples from chtMultiRegionSimpleFoam. This solver does what is required here, and so taking lines from its '.C' file, and other '.H' files is a good way to make the hybridModel do the same.

The .C files of the two solvers can be compared, and it should be noticed that in chtMultiRegionSimpleFoam.C, there are lines which create fluid and solid meshes (lines 52 and 53). These lines can be added to hybridModel.C, under 'createMesh.H':

```
#include "createFluidMeshes.H" //from chtMultiRegionSimpleFoam
#include "createSolidMeshes.H" //from chtMultiRegionSimpleFoam
```

A distinction needs to be made between each mesh for when it comes to solving them individually later on. These files will now be searched for by the solver, and so should be copied into the hybridModel folder. The two files can be found within the chtMultiRegionSimpleFoam folder, inside fluid/ and solid/. The solver can be found and the files copied over in the usual way. If desired, these file names could be changed to region0 and region1, but since there is only two regions they are left as is in this tutorial. Because of this, the header files do not need editing. It should be also noted that 'fluid' and 'solid' were defined in section 4.2, in the regionProperties file, and this would need editing if file names were changed.

In the .C file of chtMultiRegionSimpleFoam, it can be seen that inside the runtime loop, there are 'forAll' statements for the two regions. This code has to be copied to the hybrid model, and inserted within the PIMPLE loop, for fluid and for solid regions. Below `while (pimple.loop())`, the forAll statement should be added:

```
forAll(fluidRegions, i)
{
                    Info<< "\nSolving for first region "
                        << fluidRegions[i].name() << endl;

                    INSERT PIMPLE LOOP HERE
}
```

along with the Info statement. The PIMPLE loop should be included inside of these brackets, with the same being done for the solid/second region. This will mean that all of of the fluid regions will be solved first, then all of the solid regions. Under 'createTime.H', there is another line which references the regionProperties, and this also needs to be added.

```
regionProperties rp(runTime);
```

Once these lines are in place, the solver can be compiled and checked to see if it runs in the hybridModel tutorial. The log file should show the info statements, informing of what region is being solved and what order. Now that the solver is solving each region in sequence, the interpolators can be added. Right now, the region inlet values are all based on what the user defines at the start of the iterations, however the aim is for the second region's inlet boundary conditions to be based on the first region's outlet conditions.

4.5 Add patchToPatchInterpolation

The solver is now ready for adding the lines of code required for interpolating. Labels for the master and slave patch need to be created, so that they can be referenced easily in the interpolator code. Below the `Info<< "\nStarting time loop....` line, the following two lines should be added:

```
label master = mesh.boundaryMesh().findPatchID("baffleFace1D_master");
label slave = mesh.boundaryMesh().findPatchID("baffleFace1D_slave");
```

When 'master' or 'slave' is used later in the code, this will tell the solver what the definition of the label is.

After the first forAll loop, the first set of interpolators should be added (master to slave patch), and after the second forAll loop, the second set should be added (slave to master). The first include should look like this:

```
#include "interpolateMasterSlave.H"
```

Rather than including all the lines of code required in the .C file, the interpolaters are saved as header files in the hybridModel folder, and called with the 'include' statement. 'interpolateMasterSlave.H' is for master to slave interpolation, and 'interpolateSlaveMaster.H' is for slave to master interpolation.

The method here shown is to take every variable and pass them on from patch to patch. The file interpolateMasterSlave.H will be described here. The slave to master is of the same format, just with the orders reversed, therefore it will not be explained. At the top of the file, there are some variables listed eg.

```
volScalarField& k = const_cast<volScalarField&>
(mesh.lookupObject<volScalarField>("k.water"));
```

They are the variables which are not normally editable, and therefore need to be defined and referenced. The listed variables are usually constants, and need overwritten with the `const_cast` command. This is not usually recommended in C++, however there are no immediate adverse affects of using it in this case. An example of an interpolator is shown, U1 in this case:

```
//U1 interpolator    note:U1=U.particles                (master to slave)
                       patchToPatchInterpolation interpolatorU1
                        (
                                mesh.boundaryMesh()[master], // from patch
                                mesh.boundaryMesh()[slave], // to patch
                                intersection::HALF_RAY,
                                intersection::VECTOR //
                        );

           // interpolate from outlet to inlet
                        vectorField interpolatedInletU1 =
                        interpolatorU1.faceInterpolate <vector>

                            (
                                U1.boundaryField()[master]
                            );

                        if(U1.boundaryField()[slave].type() !=
```

```
                        fixedValueFvPatchVectorField::typeName)

           FatalError << "inlet patch should be fixedValue!" << exit(FatalError);

                        U1.boundaryField()[slave] == interpolatedInletU1;

           Info<< "inletU1InterpolDomain1 = " << interpolatedInletU1 << endl;
```

Explaining from the top, the comment states that U1=U.particles, so it is the solid phase dealt with here (U1 is defined in createFields.H). The next line states that patchToPatchInterpolation is used, and the name of this particular interpolator is called 'interpolatorU1': this is user defined.

Inside the first set of brackets, we first have the patch to be interpolated from, and then the patch interpolated to, and then definitions of the intersection methods. Master and slave will reference to the lines already defined at the top of the .C file. More information can be found in the intersection.H file, which can be found in the source code, or online on the OpenFOAM C++ Documentation [6]. For direction, there can either be `HALF_RAY`, which is the normal direction, `FULL_RAY`, which is both directions, and `VISABLE` which is in the normal direction to the visable part of the surface. Distance is calculated be fitting spheres between the surfaces, `CONTACT_SPHERE` or by a normal vector, `VECTOR`.

The next section explains that it is a vectorField which is to be interpolated, and that the name of the new variable is 'interpolatedInletU1'. There then is an error message warning, which prints and causes a fatal error if the patch is not set to 'fixedValue'. The penultimate line forces the values of the slave patch to become the values of 'interpolatedInletU1', meaning the outlet conditions (master patch) become the inlet conditions (slave patch). The final line is an 'info' statement that will print to the log file every time the solver passes the line, giving the values of interpolatedInletU1. If tested on the hybrid test case, the log file will print one hundred vector values (x,y,z), which is one for each cell centre: since there are 100 cells in the cross section.

Every other variable should be interpolated like this: `U1, U2, P, p_rgh, alpha1, alpha2, K, epsilon, nutWater, nutParticles and thetaParticles`. A slave to master file should also be created, ensuring to swap around the patch order, and renaming the variables; an example of U1b is shown below:

```
//U1b interpolator    note:U1=U.particles                (slave to master)
                        patchToPatchInterpolation interpolatorU1b
                        (
                                mesh.boundaryMesh()[slave], // from patch
                                mesh.boundaryMesh()[master], // to patch
                                intersection::HALF_RAY,
                                intersection::VECTOR
                        );

                        // interpolate from outlet to inlet
                        vectorField interpolatedInletU1b =
                        interpolatorU1b.faceInterpolate <vector>

                        (
                                U1.boundaryField()[slave]
                        );

                        U1.boundaryField()[master] == interpolatedInletU1b;
```

The files attached to this document can be viewed for more detail if required. The solver can now be tested in the tutorial case. With the following controlDict settings, the solver should run quite quickly, and give similar contour results as if there were no patches present, which means the interpolation is not affecting the results.

```
controlDict settings:
startTime                  0;
endTime                         0.04;
deltaT                        0.04;
writeInterval        0.002;
```

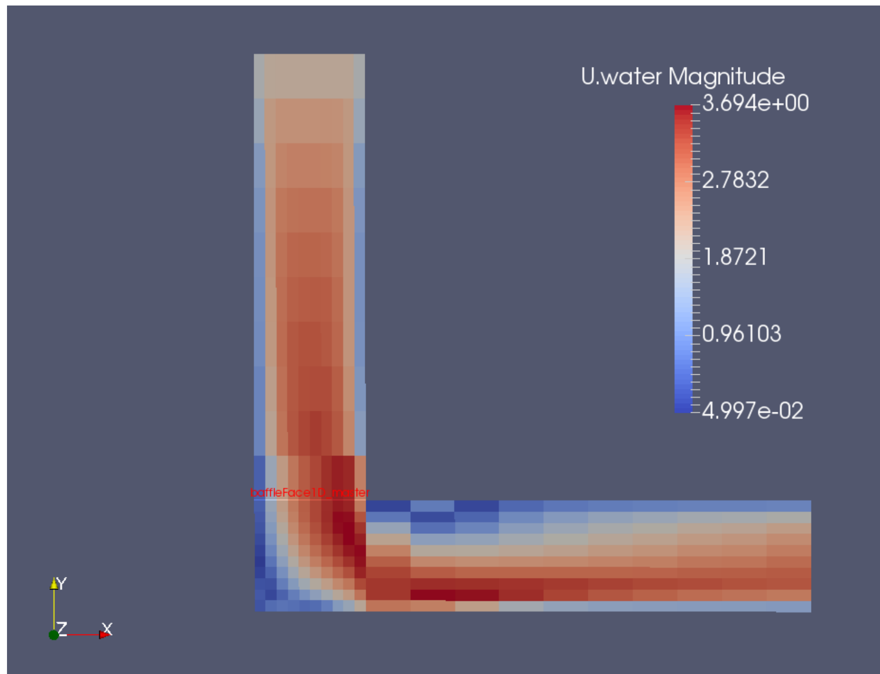


Figure 4.3: Velocity of water contours: cell values

Figure 4.3 shows cell values of the water velocity magnitude, with inlet velocity set to 2m/s. The baffle also has its patch name on it, showing the location of the interpolation. From the way the interpolator works, it can be seen that the cells on either side of the baffle have the same value. This is a drawback with the method however the error can be minimised by making the cells close to the baffle smaller in size, and placing the baffle away from large velocity/pressure gradients. The cells in this test case are far too big to give realistic results, however as said previously, the mesh is for development purposes.

Chapter 5

Addition of particles to solver

The hybridModel solver now runs smoothly with the interpolation, and so is ready for adding particles. The second phase of the Euler-Euler simulation will be 'turned off' as it gets to the baffle, and the Euler-Lagrange will take over, ensuring mass continuity. Adding the parts of DPMFoam and editing the solver to allow for this is explained in this chapter.

5.1 Editing injection method

DPMFoam has several injection methods available: kinematicLookupTableInjection is used in this tutorial, as it allows parameters to be written to a file. The information in the header file of KinematicLookupTableInjection explains what parameters are used and the order they should be in. The following directory contains the different injection models:

`OpenFOAM-3.0.x/src/lagrangian/intermediate/submodels/Kinematic/InjectionModel`

Below is the description given in the header file of KinematicLookupTableInjection.H.

```
Description
    Particle injection sources read from look-up table. Each row corresponds to
    an injection site.

    (
        (x y z) (u v w) d rho mDot   // injector 1
        (x y z) (u v w) d rho mDot   // injector 2
        ...
        (x y z) (u v w) d rho mDot  // injector N
    );

    where:
        x, y, z = global cartesian co-ordinates [m]
        u, v, w = global cartesian velocity components [m/s]
        d       = diameter [m]
        rho     = density [kg/m3]
        mDot    = mass flow rate [kg/m3]

SourceFiles
    KinematicLookupTableInjection.C
```

The user has a lot of control over the injector, however the lookup table was not designed to be edited as the solver runs. There is therefore no control over how many particles are injected from each location, other than adding more lines to the lookup table. The parameter 'mDot' is used to calculate the volume of particles to inject only, and doesn't actually control the mass flow rate.

The method used here to get around this problem is to define how many particles should be injected from the cell centre location, based on the alpha distribution from the Eulerian phase. This means that there are the same number of rows in the lookup table (one row per cell), and also that a new parameter needs to be created as a new column in the table.

5.1.1 Editing the KinematicLookupTableInjection folder

A new parameter which tells the injector how many parcels to inject from the cell was created, and called `numParticles`. The following method for editing the source code of OpenFOAM is not the safest, as any errors could follow through into the whole program. Copies of folders to be edited should be made in case this happens. As well as copies of folders, before editing any file in the following sections, a copy of the file was created with 'org' at the end of the name; so kinematicParcelInjectionData.H is copied as kinematicParcelInjectionDataOrg.H . A safer way would be to copy over the required code to the user directory, as done with the 'applications/solvers' folder.

The first file to be edited is kinematicParcelInjectionData.H. The variable is added as a scalar with name 'numParticles', and should be added below the other variables after lines 87, 129 and 150 as follows:

```
//- No of Particles
scalar numParticles_;

//- Return const access to the no of particles
inline scalar numParticles() const;

//- Return access to the no of particles
inline scalar& numParticles();
```

These lines entered correctly will create a new data variable, with the last two allowing the new variable to be accessed. The format of these lines are copied from other variables in the header file.

kinematicParcelInjectionData.C should be edited to include 'numParticles' too, copying the same style as the other variables. After mDot, numParticles should be added with the same format, after the lines 43 and 57:

```
numParticles_(0.0)

numParticles_(readScalar(dict.lookup("numParticles")))
```

Note: a comma should be added after mDot in both lists.

kinematicParcelInjectionDataIO.C should be edited in the same manner, adding the following lines after the mDot variable (after line 45 and 83).

```
is.check("reading numParticles"); //added by me
is >> numParticles_;
```

```
is.check("reading numParticles");  //added by me
is >> data.numParticles_;
```

The following `<< data.numParticles_ ;` should also be added in the list of operators on line 62. This is the file which tells the injector where to look for the variables defined. When numParticles is added at the end of the data list on line 62, the solver will expect to find it in this place.

numParticles should be added in the same manner to kineticParcelInjectionDataI.H. The lines from mDot can be copied an edited, as both variables are scalars. Two sections are added in this file too, after line 57, and after line 92. kinematicLookupTableInjection.H/C should also be edited to suit the new variable. parcelsToInject is already a variable which exists (line 133 of the .C file), and this can be edited to be based upon the new variable 'numParticles'. The following is the new edited section of parcelsToInject:

```
template<class CloudType>
Foam::label Foam::KinematicLookupTableInjection<CloudType>::parcelsToInject
(
    const scalar time0,
    const scalar time1
)
{

    scalar parcelsToInject = 0;
    if ((time0 >= 0.0) && (time0 < duration_))
    {

       // return floor(injectorCells_.size()*(time1 - time0)*parcelsPerSecond_);
        forAll(injectors_, i)
        {
        parcelsToInject += injectors_[i].numParticles();
        }

    Info<< "Time1 = " << time1 << nl << endl;
    Info<< "Time0 = " << time0 << nl << endl;
    Info<< "parcelsToInject = " << parcelsToInject << nl << endl;
    }

    return parcelsToInject;

}
```

The old definition of the variable is commented out, with the new one below. The code was copied from the variable below it (volumeToInject), and directs the injector to look for the injector variable 'numParticles', which has already been defined in the data dictionaries. The += sign means parcelsToInject= parcelsToInject + numParticles, and is a standard C++ operator. Info statements are added for clarification later on, however these can be commented out if not required. There is nothing to change in the header file.

5.1.2 Editing the InjectionModel folder

No files have changed in this folder, however the solver does not work quite like it should yet. This will be explained in further sections.

After the changes have been made in the two folders, the sections should be cleaned and compiled by navigating back to the src/lagrangian/intermediate folder and typing `wclean; wmake`.

5.2 Writing lookupTable in solver

Now that the injection method has been changed and a new variable added, the lookupTable code can be added to the solver. This will be written every timestep, and be based on the alpha distribution from the second Euler-Euler phase. Since there will be file writing taking place, and particles being injected, the following two files need to be added to the top list of header files:

```
#include "OFstream.H" // for writing files etc.
#include "basicKinematicCloud.H" // from DPMFoam
```

As the comments explain, OFstream is for reading/writing files, and the kinematicCloud file is taken from DPMFoam for inserting particles. After adding to the hybridModel.C file, the following lines also need to be added to the Make/options file (not forgetting to add a backslash to the line above):

```
-I$(LIB_SRC)/lagrangian/basic/lnInclude \
-I$(LIB_USER_SRC)/lagrangian/intermediate/lnInclude \
-I$(LIB_SRC)/lagrangian/distributionModels/lnInclude \
-I$(LIB_SRC)/regionModels/surfaceFilmModels/lnInclude
```

```
-llagrangian \
-llagrangianIntermediate \
-llagrangianTurbulence \
-lsolidParticle \
-lsurfaceFilmModels
```

Cleaning and re-compiling can be carried out after these lines have been added.

The lookupTable should be written after the first region has been solved and after the master to slave interpolation has taken place. This will mean the values written will be the most up to date. Before it is written, there are some new variables which need to be defined and added to the hybridModel.C solver.

In this example, the flow is mainly flowing perpendicularly to the cell face, however this is not always going to be the case. To account for such different situations, the normal vector should be used to calculate the mass flow rate through the cell. Firstly, the definition of the normal vector is given by:

```
vectorField normalSlaveVector = mesh.Sf().boundaryField()[slave];
```

This defines the normal vector to the cells on the slave patch as ‘normalSlaveVector’. This is then

used to get the dot product, by multiplying it by the velocity vector (in C++, the `&` operator can be used to give the dot product of two vectors).

```
scalarField dotProduct = U1.boundaryField()[master] & normalSlaveVector;
```

dotProduct is now divided by the magnitude of the direction vector, to negate the size of the cell.

```
scalarField uNormal = dotProduct/mag(normalSlaveVector);
```

Giving uNormal as the velocity vector normal to the surface of the master/slave patch. The final line to be added defines the x,y,z coordinates of the cell centres of the slave patch:

```
vectorField centres = mesh.C().boundaryField()[slave];
```

After these lines are added, the lookupTable code can be inserted by using the OFstream application which allows output to file streams. The first line describes where the file is to be written, in the constant folder in this case, and also the name of the file. The code is inserted below:

```
const fileName& fName(mesh.time().constant()/"kinematicLookupTableInjection");
OFstream os(fName);
os << "/*---------------------------------*- C++ -*-----------------------------------* " << endl;
os <<" | =========                 |                                                 |\n";
os <<" | \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |\n";
os <<" |  \\    /   O peration     | Version:  3.0.x                                 |\n";
os <<" |   \\  /    A nd           | Web:      www.OpenFOAM.org                      |\n";
os <<" |    \\/     M anipulation  |                                                 |\n";
os << "*-----------------------------------------------------------------------------*/ //" << endl;
os <<"FoamFile\n{\n";
os <<"    version     2.0;\n";
os <<"    format      ascii;\n";
os <<"    class       dictionary;\n";
os <<"    location    " "constant" ";\n";
os <<"    object      kinematicLookupTableInjection;\n}\n";
os <<"// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //\n\n";
os <<"/* (x y z) (u v w) d rho mDot numParcels  \n";
os <<"     where: \n";
os <<"   x, y, z = global cartesian co-ordinates [m] \n";
os <<"   u, v, w = global cartesian velocity components [m/s] \n";
os <<"   d       = diameter [m] \n";
os <<"   rho     = density [kg/m3] \n";
os <<"   mDot    = mass flow rate [kg/m3] \n";
os <<"   numParcels = number of Parcels  \n";
os <<"   Dictionary for the KinematicLookupTableInjection */ \n";
os << "( " << endl;
forAll(interpolatedInletU2, i)
{
os << centres[i] << " " << U1.boundaryField()[slave][i] << " "
<< 55e-6 << " " << 2750 << " " << 0.005 << " "
<< floor ((alpha1.boundaryField()[slave][i]*(mag(normalSlaveVector[i]))*uNormal[i])/
((8.71e-14)*3*(-1)*5000)) << endl;

}
os << "//The end"<< endl;
os << ");"<< endl;
```

The first few lines of the file are not really necessary, but are included to make the file look the same as all the other OpenFOAM files. Each line starts with an `os <<`, and finishes with `<< endl;`. The characters in-between the quotation marks are the characters that are printed to file.

After all the comments, the important code starts with the forAll statement. This states that for all of the interpolatedInletU2 (the liquid phase) values, do the following. The next line defines what will be printed in each column. Since there are 100 values of interpolatedInletU2 (since there are 100 cells on the plane), there will be 100 rows printed. The data printed in the columns are the same as the line at the top of the comments, each column is explained below.

- `centres[i]` gives cell centre coordinates in (x,y,z) [m]
- `U1.boundaryField()[slave][i]` gives velocity components in (u,v,w) [m/s]
- `55e-6` diameter in [m]
- `2750` density in [kg/m3]
- `0.005` mass flow rate [kg/m3]
- `floor ((alpha1.boundaryField()[slave][i]*(mag(normalSlaveVector[i]))*uNormal[i])/ ((8.71e-14)*3*(-1)*5000))` gives number of parcels to inject

The last line starts with the word `floor`, which forces the output of the equation to be an integer. This makes sense, as there cannot be 'half' of a parcel which contains 3 particles (so 1.5 particles). The equation in written form is: Number of parcels = (alpha of 2nd Eulerian phase * Area of cell * normal velocity component) / (Volume of one particle * number of particles/parcel * number of timesteps/second). This equation is using the volume flowrate to calculate how many parcels should be injected. This could be cleaned up by making the solver search for the variables like diameter etc. in the working directory, rather than the user having to edit the source code everytime they want to make a change to particle parameters. This method was tried, but it was found to be difficult to reference externally whilst inside the 'forAll' loop.

Again, the model should be recompiled.

5.3 Turning off interpolator

Now that there is a working lookupTable, the interpolator for the second/solid Eulerian phase can be turned off. If this was not turned off, there would be mass continuity errors in the solver, as there would be double the amount of second phase in region two as there should be; Lagrangian particles and Eulerian 2nd phase.

Inside the 'solidRegion' loop, above the Info statement, the following line should be added:

```
alpha1.boundaryField()[slave] == 0;
```

This forces alpha1 (second phase) to become zero on the slave patch. Inside the two interpolate header files, some sections should be commented out, to prevent the values from being interpolated. In interpolateMasterSlave.H, interpolatorU1 should be commented out, and in interpolateSlaveMaster.H, interpolatorU1b and interpolatoralpha1b should also be commented out.

Note: commenting large sections can be done quickly by using the following technique:

```
/* Opens comment
TEXT, TEXT TEXT, TEXT TEXT, TEXT TEXT, TEXT
*/ Closes comment
```

The solver will no longer interpolate the alpha or velocity values of the 2nd Eulerian phase from the Master-Slave or vice versa. The solver can be recompiled again.

5.4 Adding DPMFoam

Sections from DPMFoam should now be added to the solver, which will introduce particles. The first piece of code should be added at the very top of ‘solidRegions’ and is as follows:

```
argList::addOption // from ALopez
(
        "cloudName",
        "name",
        "specify alternative cloud name. default is 'kinematicCloud'"
);
```

This is taken from Lopez[1], and gives the user the option to specify an alternative cloud name. After interpolating from Slave to Master, but within the loop, add the following:

```
Info<< "Evolving " << kinematicCloud.name() << endl; // from ALopez
kinematicCloud.evolve();
```

This command will print an info statement, evolve the kinematicCloud, and insert particles as required: more information can be found in Lopez’ tutorial.

This section of code references ‘kinematicCloud’ which needs to be defined in createFields.H. The following lines should be added at the bottom of the file:

```
Info<< "\nReading transportProperties\n" << endl;

IOdictionary transportProperties
    (
        IOobject
        (
        "transportProperties",
        runTime.constant(),
        mesh,
        IOobject::MUST_READ_IF_MODIFIED,
        IOobject::NO_WRITE
        )
    );

dimensionedScalar rhoInfValue
    (
        transportProperties.lookup("rhoInf")
    );

volScalarField rhoInf
    (
```

```
        IOobject
        (
        "rho",
        runTime.timeName(),
        mesh,
        IOobject::NO_READ,
        IOobject::AUTO_WRITE
        ),
    mesh,
    rhoInfValue
    );

dimensionedScalar nu
    (
        transportProperties.lookup("nu")
    );

volScalarField mu
    (
        IOobject
        (
        "mu",
        runTime.timeName(),
        mesh,
        IOobject::NO_READ,
        IOobject::AUTO_WRITE
        ),
    mesh,
    nu*rhoInfValue
    );

word kinematicCloudName("kinematicCloud");
    args.optionReadIfPresent("cloudName", kinematicCloudName);
    Info<< "Constructing kinematicCloud " << kinematicCloudName << endl;
    basicKinematicCloud kinematicCloud
    (
        kinematicCloudName,
        rhoInf,
        U2,         // this used to be U1
        mu,
        g
    );
```

This large chunk of code defines the variables, and tells the solver where to find them. It starts off by saying to read the transportProperties file, which contains rhoInf and nu. The bottom gives arguments to kinematicCloud which are the fields to use. Note that U2 is entered here, which means the particle paths are based upon the fluid phase, not the particulate phase.

After a `wclean; wmake` the solver will be complete and ready to be tested.

Chapter 6

Hybrid test case

The model can now be tried out on a test case, to ensure that it is functioning properly. The files that are required are in the attached folder, however some sections will be explained here. The 0 and constant directories are shown below.

```
├── 0/
│   ├── alpha.particles
│   ├── alphat.particles
│   ├── alpha.water
│   ├── alphat.water
│   ├── domain0/
│   ├── domain1/
│   ├── epsilon.water
│   ├── k.water
│   ├── lagrangian/ .3 nut.particles
│   ├── nut.water
│   ├── p
│   ├── prgh
│   ├── Theta.particles
│   ├── T.particles
│   ├── T.water
│   ├── U.particles
│   └── U.water
└── constant/
    ├── domain0/
    ├── domain1/
    ├── kinematicCloudProperties
    ├── kinematicLookupTableInjection
    ├── particleProperties
    ├── phaseProperties
    ├── regionProperties
    ├── thermophysicalProperties.particles
    ├── thermophysicalProperties.water
    ├── transportProperties
    ├── triSurface/
    ├── turbulenceProperties.particles
    └── turbulenceProperties.water
```

The directories should look familiar, apart from the domain0 and domain1 sections added. It also must be noted that before the simulation takes place, patches referencing the baffles must also be added. This can be seen by opening 0/p as after the ‘walls’, there are two new entries:

```
baffleFace1D_master
    {
        type            fixedValue;
        value           uniform 1e5;
    }
baffleFace1D_slave
    {
        type            zeroGradient;
    }
```

For the users own case, the utility ‘changeDictionary’ can be used. The type should be set to fixed value for the master patches, but the value can be anything, as it will be overwritten by the interpolators. kinematicCloudProperties contains the definition of the injectionModel. The section below is explained with comments.

```
model1
{
  type              kinematicLookupTableInjection; //tells solver what type of injection
  patchName         baffleFace1D_slave; //injection patch name
  massTotal         3; //this isn't used
  parcelBasisType   fixed; //this isn't used
  nParticle         3; //how many particles per parcel - update .C file if changed
  SOI               0.02; //start of injection
  inputFile         "kinematicLookupTableInjection"; //input file
  duration          1; //duration of injection
  parcelsPerSecond 50; // doesn't do anything
  randomise         false;
}
```

kinematicLookupTableInjection will be written as the solver is run, from the OFStream code in the hybridModel.C file. phaseProperties contains the properties of each phase in the Euler Euler simulation. regionProperties names each region: fluid and solid in this case. The thermophysicalProperties files contain information about the heat transfer, however this should be negligible as both phases are set to the same temperature. transportProperties defines which phase is the continuous one: water in this case.

6.1 Results

The case should be run with the following commands:

```
blockMesh
createBaffles -overwrite
splitMeshRegions -cellZones -overwrite
hybridModel
```

The results shown here are from 0-3.9 seconds, with the start of injection (SOI) at 0.29 seconds. All results are shown at the last time step unless otherwise stated.

As well as the hybrid model, benchmark EE and EL cases were also run. They were set up with the same boundary conditions and parameters as far as possible. The EL injection site was moved to the inlet, as this is a more realistic comparison than injecting from the same location as the hybrid model. Table 6.1 below shows a comparison of the execution times of the 3 models, with two different mass concentrations. As expected the pure EE model is the fastest. The hybrid model is twice as fast as the EL: a marked improvement. If the transition layer was closer to the wall the hybrid model would most likely be faster than this. With the hybrid model taking half the time to solve, this is a promising sign that it is a valid solution to reduce long execution times.

Model	**Execution Time (s)**	
	1% MC	2% MC
Hybrid model	225	298
Euler-Lagrange	420	585
Euler-Euler	102	105

Table 6.1: Comparison of execution times from 0-0.39 seconds with different mass concentrations (MC)

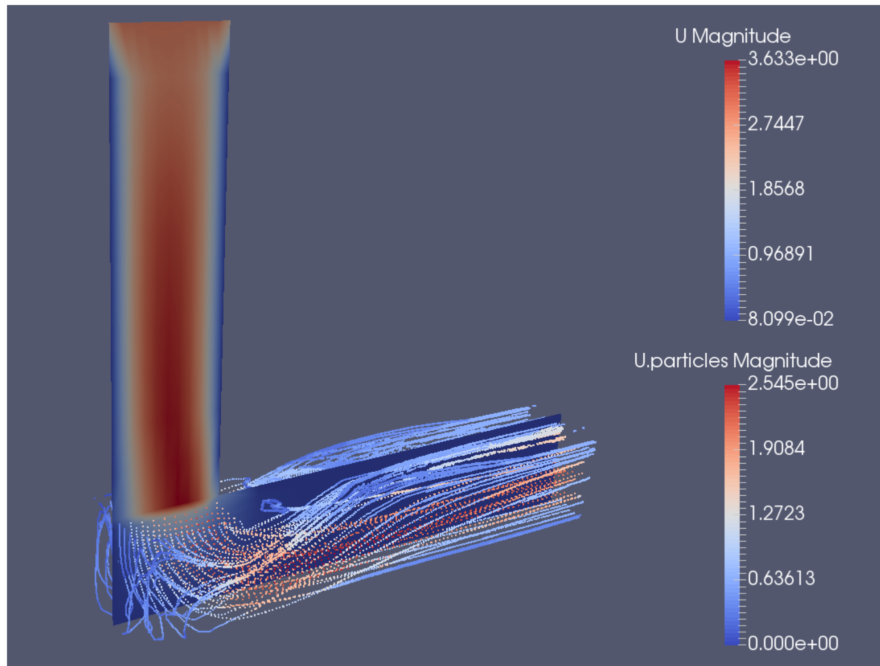


Figure 6.1: Hybrid model showing 2D slice of 2nd Eulerian phase velocity contours, and lagrangian particles in second region also coloured by velocity magnitude.

Figure 6.1 shows a 2D slice though the Z-normal direction of the 2nd Eulerian phase, and also the lagrangian particles coloured by their velocities. The particle injections are based on the values of the 2nd Eulerian phase as explained earlier in the report.

Figure 6.2 shows a pure EL model, which was run as a comparison or a reference case. The number of particles per cell injected are based on the same lookupTable values as used in the hybrid model, and the velocities are all set to 2m/s. The contour plots for 'U Magnitude' are set to the

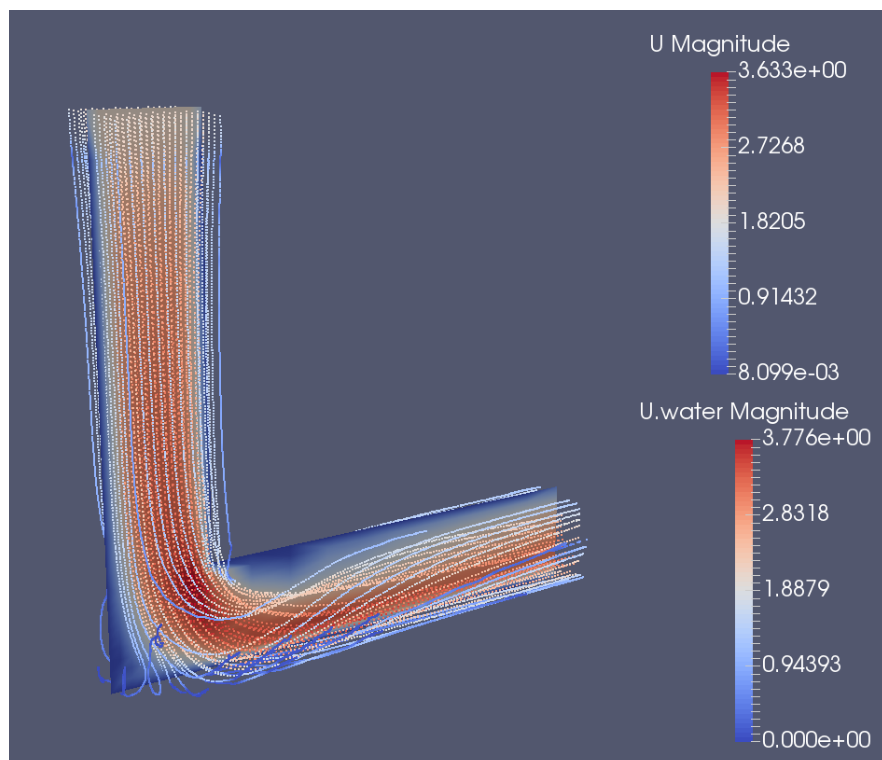


Figure 6.2: EL model showing 2D slice of velocity magnitude of the first phase, with Lagrangian particles also coloured by velocity magnitude.

same scale, so that particle tracks can be directly compared. It can be seen from the particle tracks that there is agreement between the models. To compare in more detail, the particle impacts on the bottom surface are compared in the following two figures. To get this data, the file called 'PatchPost-Processing.C' which is a cloud function had to be edited and recompiled. The 'patchPostProcessing' section then had to be added under the 'cloudFunctions' part of the 'kinematicCloudProperties' file, so that the function would be used. This file can be compared to the standard one in the attached files. The data is then outputted in the 'postProcessing' folder which is created when a time directory is written. The The z-direction (m) is shown on the vertical axis, and the x-direction (m) on the x-axis.

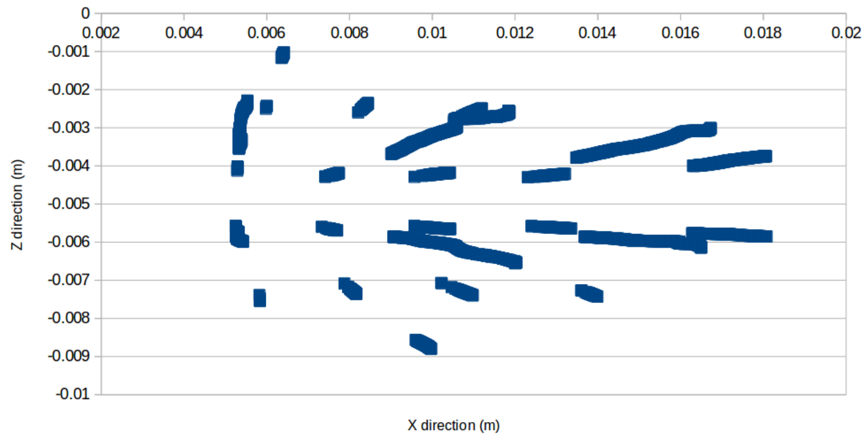


Figure 6.3: EL particle impacts

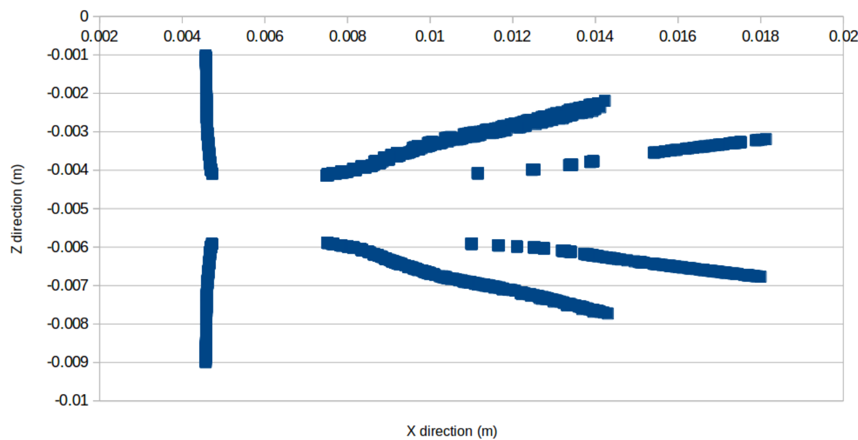


Figure 6.4: Hybrid model particle impacts

The particle impacts are more grouped together in the hybrid model graph, and this can be explained by the fact that the injection site is closer to the wall than in the EL model. This means the particles have less spread over the baffle as they are only injected from 100 sites, whereas the

particles in the EL model can spread out over the baffle to any position since they are injected from the inlet. There is a good general agreement in both particle impact graphs though, especially considering such a coarse mesh was used.

Chapter 7

Conclusions and further work

The hybrid model developed here shows a promising future as a faster way to model slurries, however there are still developments to be made. The volume fraction of the Lagrangian particles in the second region currently has no effect on the first Eulerian phase. This means there is 'too much' of the first Eulerian phase in the second region. To rectify this, code from DPMFoam should be added to the hybridModel solver which includes the volume of the particles.

Another issue is the fact that the lookupTable is written every time step but is only read when the solution is initialised. This effectively decouples the solver, and turns it into a steady state problem, since the injection values cannot change so further work needs to be done here.

The solver also needs further testing and development, however it is hoped that this tutorial has served a useful purpose by explaining some of OpenFOAM's utilities and the development of a new model.

Chapter 8

Study questions

1. What is the bug in reactingTwoPhaseEulerFoam and what files need to be replaced to solve the problem?
2. What command can be used to create an empty file in a case directory for comparing paraview plots?
3. Name all of the tutorials which use ‘createBafflesDict’.
4. How do you label a patch to your own name for use in the solver?
5. What 3 directions can be used for intersecting in the patchToPatchInterpolation?
6. How can we get the dot product of two vectors in OpenFOAM?
7. What does SOI mean in kinamaticCloudProperties?

Bibliography

[1] A Lopez. LPT for erosion modeling in OpenFOAM. http://www.tfd.chalmers.se/~hani/kurser/OS_CFD_2013/AlejandroLopez/LPT_for_erosionModelling_report.pdf, 2014.

[2] O Penttinen. A pimpleFoam tutorial for channel flow, with respect to different LES models. http://www.tfd.chalmers.se/~hani/kurser/OS_CFD_2011/OlofPenttinen/projectReport.pdf, 2011.

[3] Tobi. All about the PIMPLE algorithm. http://www.cfd-online.com/Forums/blogs/tobi/2489-all-about-pimple-algorithm-part-i.html, 2014.

[4] FreeCAD. Freecad. http://www.freecadweb.org/, 2016.

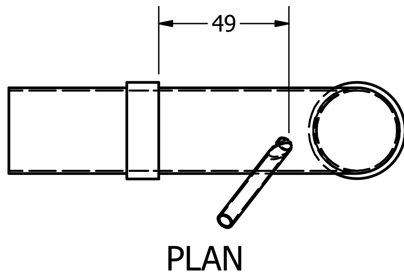
[5] bigphil PatchToPatchInterpolationfaceInterpolate http://www.cfd-online.com/Forums/openfoam-solving/59868-patchtopatchinterpolationfaceinterpolate.html

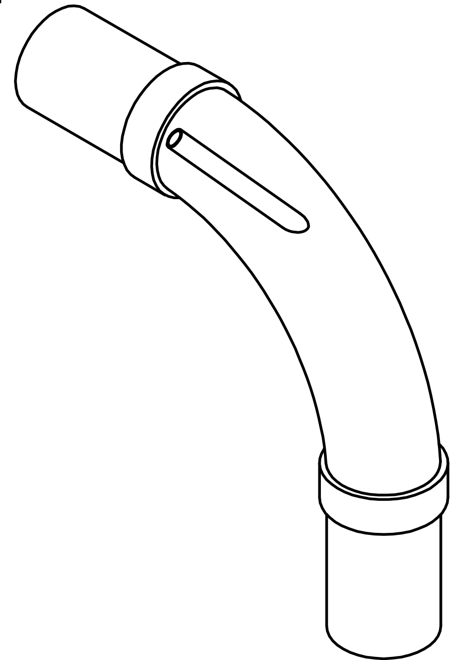
[6] OpenFOAM OpenFOAM C++ Documentation - intersection.H http://www.openfoam.com/documentation/cpp-guide/html/a09257_source.html

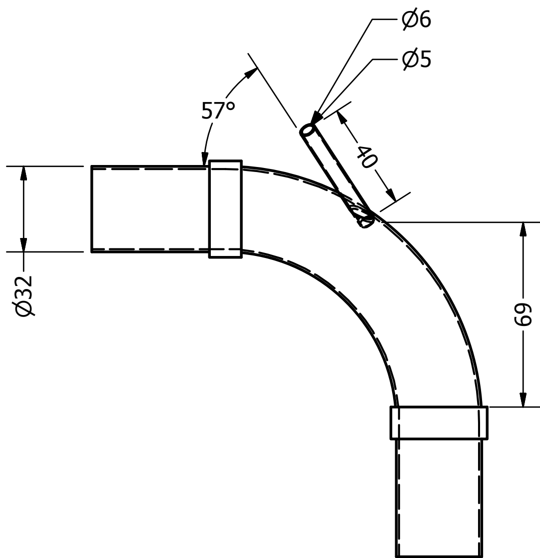
Appendix B

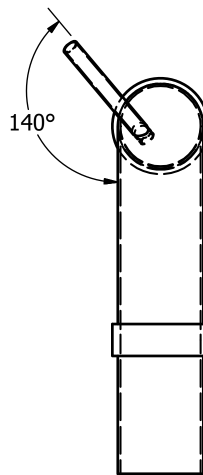
Figures of experimental rig components

Stainless steel pipe into PVC pipe







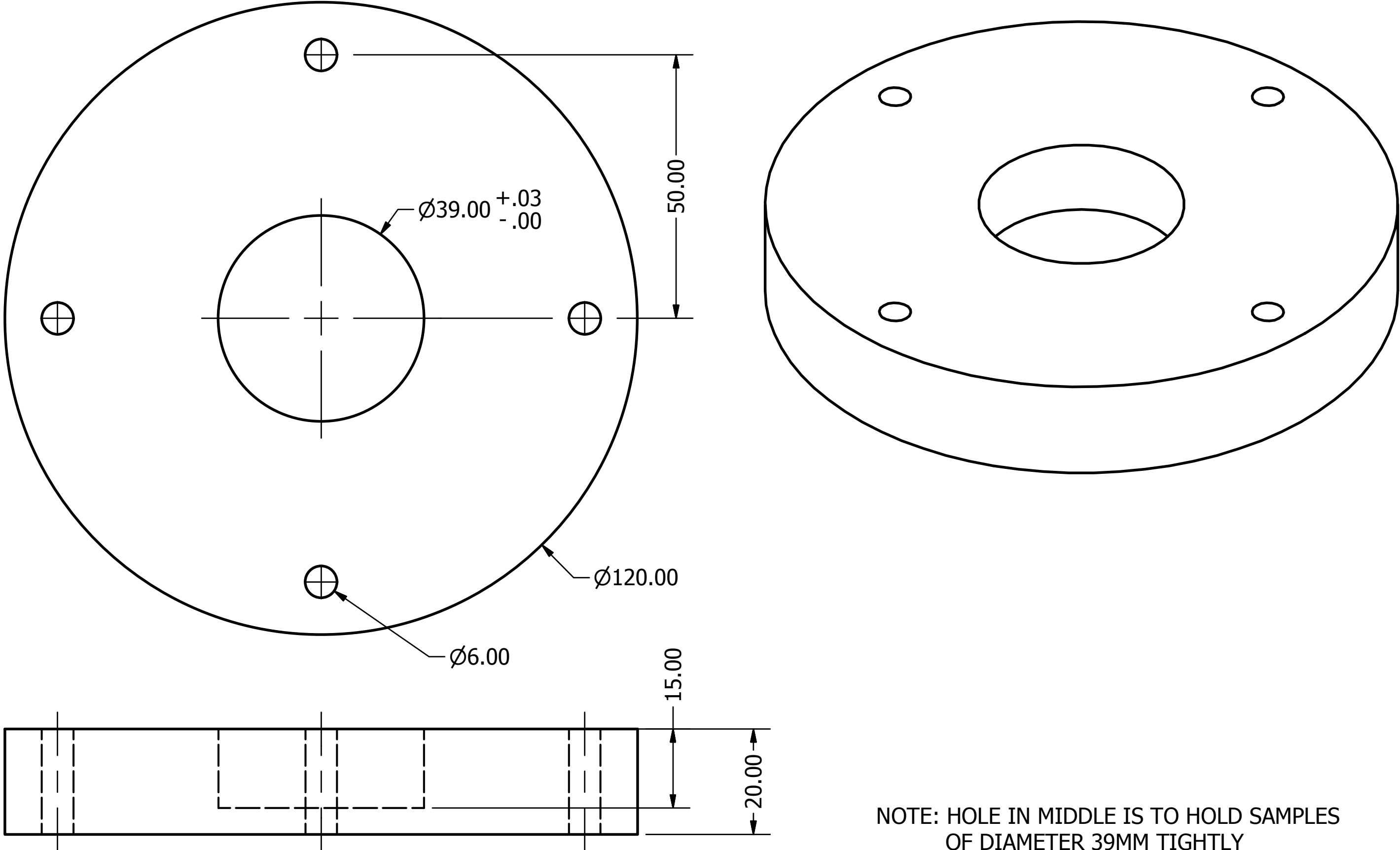


Stainless steel pipe inserted into PVC pipe on rig. Orientation as shown, and angles are rough guides. SS pipe is 6mm O/D, with 0.5mm wall thickness. Pipe should be attached so that it's watertight. SS pipe should also only extend into PVC pipe as far as necessary.

Alasdair Mackenzie 21/11/16

SAMPLE HOLDER

MADE FROM ALUMINIUM

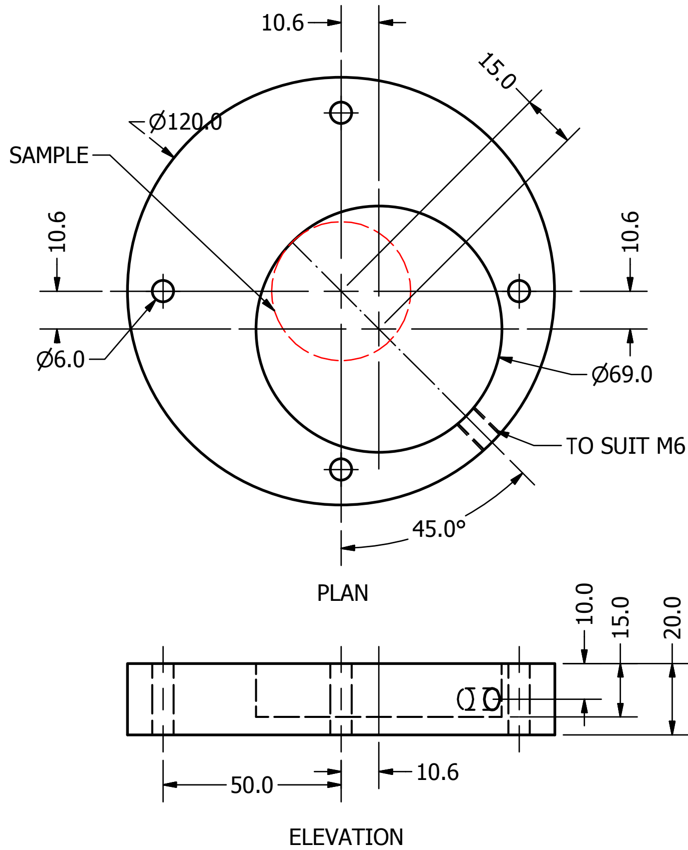


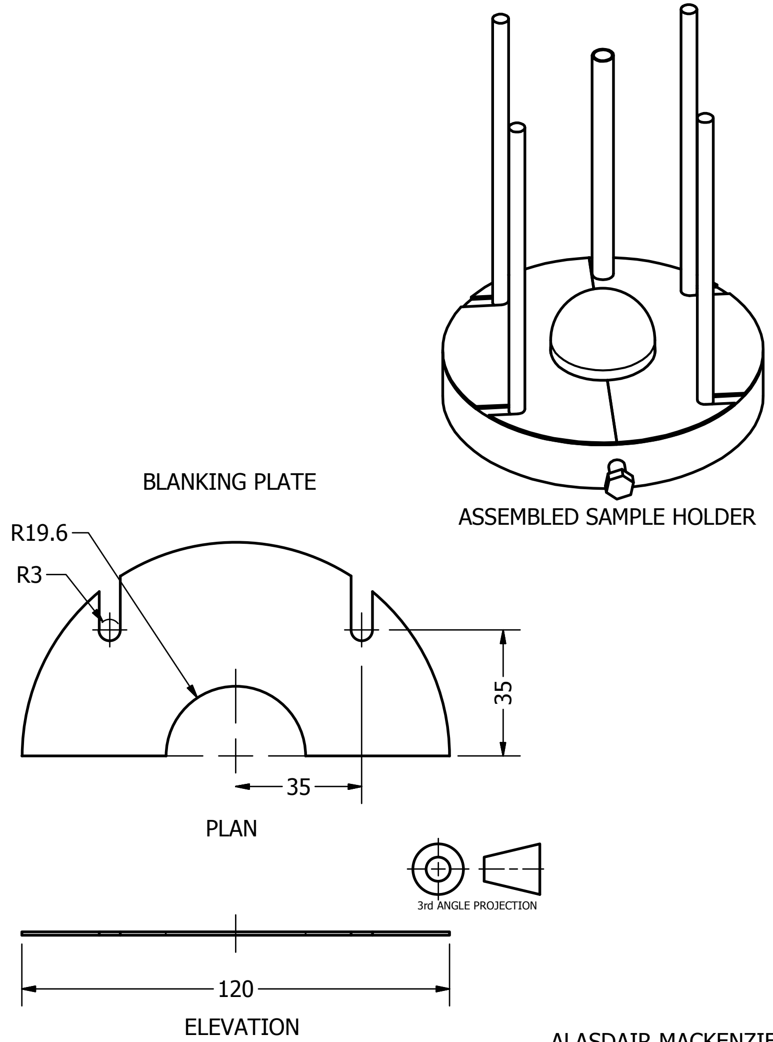
NOTE: HOLE IN MIDDLE IS TO HOLD SAMPLES OF DIAMETER 39MM TIGHTLY

ALASDAIR MACKENZIE
13/12/16

SAMPLE HOLDER

HOLDER MADE FROM ALUMINIUM
BLANKING PLATE MADE FROM 1MM THK STAINLESS STEEL
1 X HOLDER
2 X PLATES
HOLDER TO SUIT 39MM DIA SAMPLE

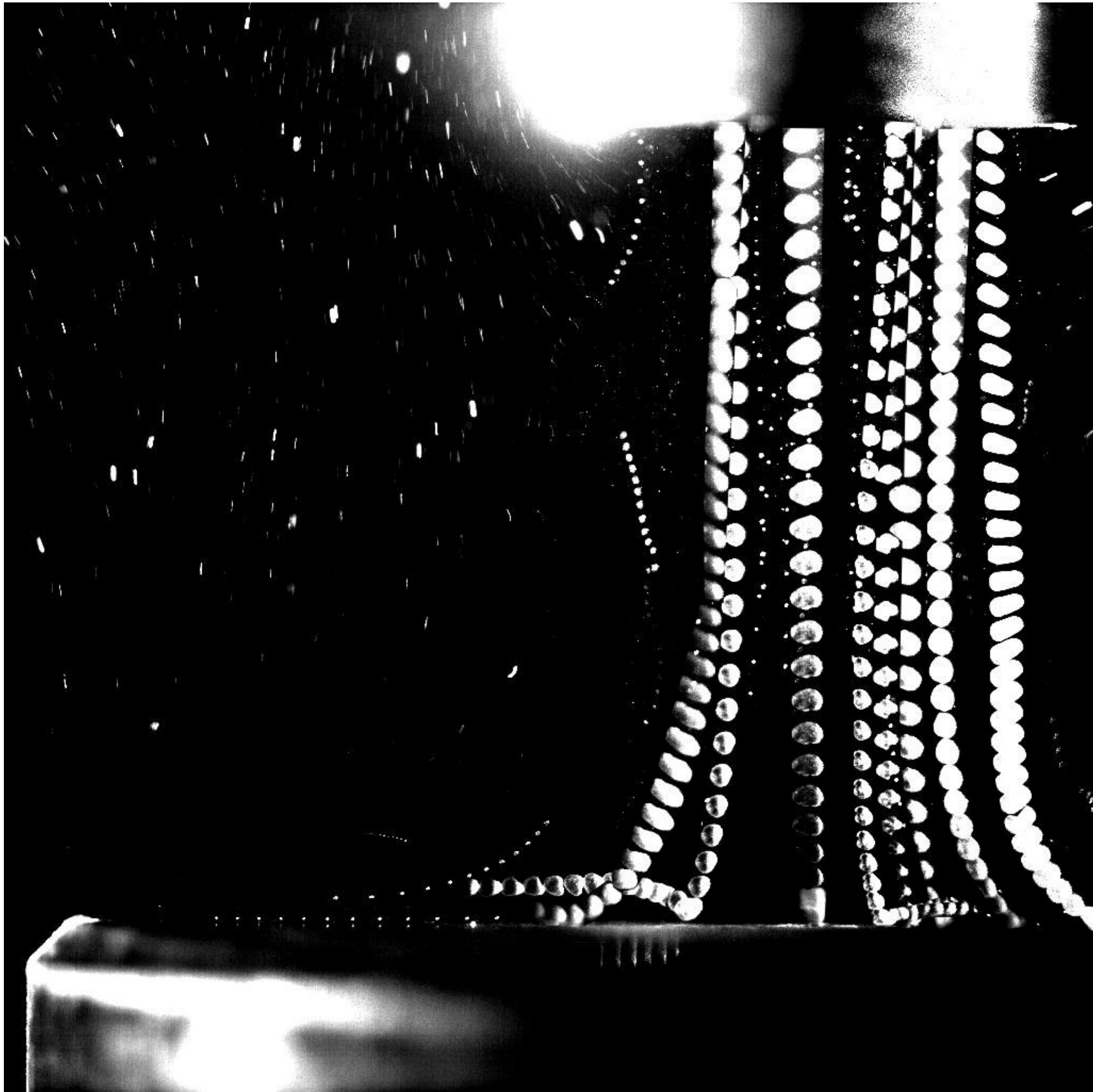


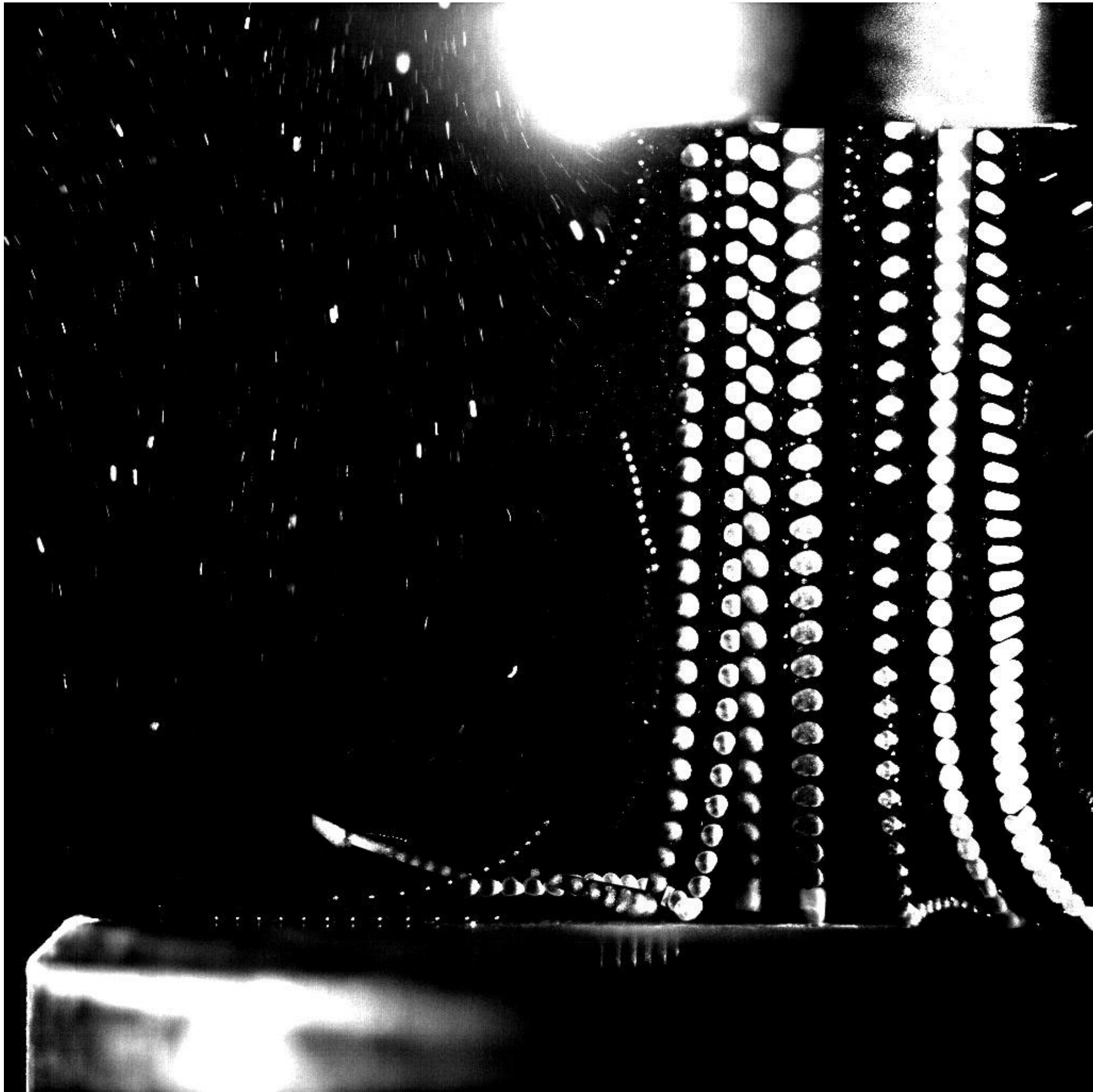


ALASDAIR MACKENZIE
14/12/16

Appendix C

Stitched particle images





Appendix D

Modified files from solver

The files which were edited or modified from the solver are included below. The solver is called ‘mySolverZagrebOmega’, but of course can be called whatever the user wishes. Files are presented in alphabetical order.

createFields.H

```
#include "readGravitationalAcceleration.H"
#include "readhRef.H"

Info<< "Creating phaseSystem\n" << endl;

autoPtr<twoPhaseSystem> fluidPtr
(
    twoPhaseSystem::New(mesh)
);
twoPhaseSystem& fluid = fluidPtr();

phaseModel& phase1 = fluid.phase1();
phaseModel& phase2 = fluid.phase2();
```

```
volScalarField& alpha1 = phase1;

volScalarField& alpha2 = phase2;


//next section added by me for volume fraction update//////////////////////////////






volVectorField& U1 = phase1.U();

surfaceScalarField& phi1 = phase1.phi();

surfaceScalarField& alphaPhi1 = phase1.alphaPhi();

surfaceScalarField& alphaRhoPhi1 = phase1.alphaRhoPhi();


volVectorField& U2 = phase2.U();

surfaceScalarField& phi2 = phase2.phi();

surfaceScalarField& alphaPhi2 = phase2.alphaPhi();

surfaceScalarField& alphaRhoPhi2 = phase2.alphaRhoPhi();


surfaceScalarField& phi = fluid.phi();




//next section added by from Alejandro's stuff ///////////////////////////////////////


Info<< "\nReading transportProperties\n" << endl;


IOdictionary transportProperties

    (
```

```
        IOobject
        (
        "transportProperties",
        runTime.constant(),
        mesh,
        IOobject::MUST_READ_IF_MODIFIED,
        IOobject::NO_WRITE
        )
    );




// This modificaation didn't make a difference.
//    Info<< "\nReading kinematicLookupTableInjection\n" << endl;


/*IOdictionary numParticles
        (
            IOobject
            (
            "kinematicLookupTableInjection",
            runTime.constant(), // /"kinematicLookupTableInjection",
            mesh,
            IOobject::MUST_READ_IF_MODIFIED,
            IOobject::NO_WRITE
            )
        );
*/
dimensionedScalar rhoInfValue
```

```
    (
        transportProperties.lookup("rhoInf")
    );

volScalarField rhoInf
    (
        IOobject
        (
        "rho",
        runTime.timeName(),
        mesh,
        IOobject::NO_READ,
        IOobject::AUTO_WRITE
        ),
    mesh,
    rhoInfValue
    );

dimensionedScalar nu
    (
        transportProperties.lookup("nu")
    );


volScalarField mu
    (
        IOobject
        (
```

```
        "mu",
        runTime.timeName(),
        mesh,
        IOobject::NO_READ,
        IOobject::AUTO_WRITE
        ),
    mesh,
    nu*rhoInfValue
    );

    word kinematicCloudName("kinematicCloud");
    args.optionReadIfPresent("cloudName", kinematicCloudName);
   Info<< "Constructing kinematicCloud " << kinematicCloudName << endl;
    basicKinematicCloud kinematicCloud
    (
        kinematicCloudName,
        rhoInf,
        U2,         // this used to be U1
        mu,
        g
    );

// Particle fraction upper limit
scalar alpha2Min
(
    1.0
  - readScalar
    (
```

```
        kinematicCloud.particleProperties().subDict("constantProperties")
       .lookup("alphaMax")
    )
);
scalar SOI
(
    readScalar
    (
        kinematicCloud.particleProperties().subDict("subModels")
        .subDict("injectionModels").subDict("model1")
       .lookup("SOI")
    )
);
scalar timestepsPerSecond
(
    readScalar
    (
        kinematicCloud.particleProperties().subDict("subModels")
        .subDict("injectionModels").subDict("model1")
       .lookup("timestepsPerSecond")
    )
);
scalar nParticle
(
    readScalar
    (
        kinematicCloud.particleProperties().subDict("subModels")
        .subDict("injectionModels").subDict("model1")
```

```
        .lookup("nParticle")
    )
);

// Update alphac from the particle locations
alpha2 = max(1.0 - kinematicCloud.theta(), alpha2Min);
alpha2.correctBoundaryConditions();

surfaceScalarField alpha2f("alpha2f", fvc::interpolate(alpha2));
//surfaceScalarField alphaPhi1("alphaPhi2", alpha2f*phi2); why is this commented out?
/*
//below added 20/10/16
Info<< "Creating turbulence model\n" << endl;
singlePhaseTransportModel continuousPhaseTransport(U2, phi2);



autoPtr<PhaseIncompressibleTurbulenceModel<singlePhaseTransportModel> >
continuousPhaseTurbulence
(
    PhaseIncompressibleTurbulenceModel<singlePhaseTransportModel>::New
    (
        alpha2,
        U2,
        alphaPhi2,
        phi2,
        continuousPhaseTransport
    )
```

```
*/



/// New section


/*

IOdictionary kinematicCloudProperties

    (

        IOobject

        (

        "kinematicCloudProperties",

        runTime.constant(),

        mesh,

        IOobject::MUST_READ_IF_MODIFIED,

        IOobject::NO_WRITE

        )

    );


scalarField rho0

    (

        kinematicCloudProperties.lookup("rho0")

    );

*/


// end of sectcion added by me ////////////////////////////////////////


dimensionedScalar pMin

(
```

```
    "pMin",
    dimPressure,
    fluid
);

#include "gh.H"

rhoThermo& thermo1 = phase1.thermo();
rhoThermo& thermo2 = phase2.thermo();

volScalarField& p = thermo1.p();

volScalarField& rho1 = thermo1.rho();
const volScalarField& psi1 = thermo1.psi();

volScalarField& rho2 = thermo2.rho();
const volScalarField& psi2 = thermo2.psi();

Info<< "Reading field p_rgh\n" << endl;
volScalarField p_rgh
(
    IOobject
    (
        "p_rgh",
        runTime.timeName(),
        mesh,
        IOobject::MUST_READ,
        IOobject::AUTO_WRITE
```

```
    ),
    mesh
);

label pRefCell = 0;
scalar pRefValue = 0.0;
setRefCell
(
    p,
    p_rgh,
    pimple.dict(),
    pRefCell,
    pRefValue
);
mesh.setFluxRequired(p_rgh.name());

const IOMRFZoneList& MRF = fluid.MRF();
fv::IOoptionList& fvOptions = fluid.fvOptions();
```

interpolateMasterSlave.H

```
// Beginning of interpolators from master to slave, or region1 to region0.

volScalarField& k = const_cast<volScalarField&>(mesh.lookupObject<volScalarField>("k.water")
//volScalarField& epsilon = const_cast<volScalarField&>(mesh.lookupObject<volScalarField>("e
volScalarField& omega = const_cast<volScalarField&>(mesh.lookupObject<volScalarField>("omega
volScalarField& nutWater = const_cast<volScalarField&>(mesh.lookupObject<volScalarField>("nu
volScalarField& nutParticles = const_cast<volScalarField&>(mesh.lookupObject<volScalarField>
```

```
volScalarField& thetaParticles = const_cast<volScalarField&>(mesh.lookupObject<volScalarFiel

/*

                if (runTime.timeOutputValue() < SOI)

                        {

                            patchToPatchInterpolation interpolatorU1

                        (

                            mesh.boundaryMesh()[master], // from patch

                                                                mesh.boundaryMesh()[slave], // t

                                                                intersection::HALF_RAY,

                                                                intersection::VECTOR //was CONTA

                        );


                        // interpolate from outlet to inlet

                        vectorField interpolatedInletU1 = interpolatorU1.faceInterpolate <ve


                                                                                    (

                                                                U1.boundaryField()[master]

                                                                                    );


                        if(U1.boundaryField()[slave].type() != fixedValueFvPatchVectorField:

                        FatalError << "inlet patch should be fixedValue!" << exit(FatalError


                        U1.boundaryField()[slave] == interpolatedInletU1;

                        }

 */


//                      //U1 interpolator    note:U1=U.particles

patchToPatchInterpolation interpolatorU1
```

```
(
        mesh.boundaryMesh()[master],                            // from patch
        mesh.boundaryMesh()[slave],                             // to patch
        intersection::HALF_RAY,
        intersection::VECTOR                          //was CONTACT_SPHERE doesn't seem to ma
);

// interpolate from outlet to inlet
vectorField interpolatedInletU1 = interpolatorU1.faceInterpolate <vector>

                                  (
        U1.boundaryField()[master]
                                  );

if(U1.boundaryField()[slave].type() != fixedValueFvPatchVectorField::typeName)
        FatalError << "inlet patch should be fixedValue!" << exit(FatalError);

U1.boundaryField()[slave] == interpolatedInletU1;

//Info<< "U1.boundaryFieldMasterDomain1 = " << U1.boundaryField()[master] << endl;
//Info<< "U1.boundaryFieldSlaveDomain1 = " << U1.boundaryField()[slave] << endl;
//Info<< "inletU1InterpolDomain1 = " << interpolatedInletU1 << endl;

//U2 interpolator    note:U2=U.water
patchToPatchInterpolation interpolatorU2
(
        mesh.boundaryMesh()[master],                            // from patch
        mesh.boundaryMesh()[slave],                             // to patch
```

```
        intersection::HALF_RAY,

        intersection::VECTOR

);


vectorField interpolatedInletU2 = interpolatorU2.faceInterpolate <vector>


                                  (

        U2.boundaryField()[master]

                                  );


if(U2.boundaryField()[slave].type() != fixedValueFvPatchVectorField::typeName)

        FatalError << "inlet patch should be fixedValue!" << exit(FatalError);


U2.boundaryField()[slave] == interpolatedInletU2;


//Info<< "inletU2InterpolDomain1 = " << interpolatedInletU2 << endl;


//p interpolator    note: needed to change scalar for vector

patchToPatchInterpolation interpolatorP

(

        mesh.boundaryMesh()[master],                         // from patch

        mesh.boundaryMesh()[slave],                         // to patch

        intersection::HALF_RAY,

        intersection::VECTOR

);


scalarField interpolatedInletP = interpolatorP.faceInterpolate <scalar>
```

```
                                  (
        p.boundaryField()[master]
                                  );


//if(p.boundaryField()[slave].type() != fixedValueFvPatchScalarField::typeName)
//FatalError << "inlet patch should be fixedValue!" << exit(FatalError);


p.boundaryField()[slave] == interpolatedInletP;


// Info<< "inletPInterpolDomain1 = " << interpolatedInletP << endl;




//p_rgh interpolator    note: needed to change scalar for vector
patchToPatchInterpolation interpolatorp_rgh
(
        mesh.boundaryMesh()[master],                         // from patch
        mesh.boundaryMesh()[slave],                          // to patch
        intersection::HALF_RAY,
        intersection::VECTOR
);


scalarField interpolatedInletp_rgh = interpolatorp_rgh.faceInterpolate <scalar>


                                      (
        p_rgh.boundaryField()[master]
                                      );


// if(p_rgh.boundaryField()[slave].type() != fixedValueFvPatchScalarField::typeName)
```

```
// FatalError << "inlet patch should be fixedValue!" << exit(FatalError);


p_rgh.boundaryField()[slave] == interpolatedInletp_rgh;


// Info<< "inletp_rghInterpolDomain1 = " << interpolatedInletp_rgh << endl;


//alpha1 interpolator    note: alpha1 is alpha.particles


patchToPatchInterpolation interpolatoralpha1
(
        mesh.boundaryMesh()[master],                         // from patch
        mesh.boundaryMesh()[slave],                         // to patch
        intersection::HALF_RAY,
        intersection::VECTOR
);


scalarField interpolatedInletalpha1 = interpolatoralpha1.faceInterpolate <scalar>


                                      (
        alpha1.boundaryField()[master]
                                      );


// if(p_rgh.boundaryField()[slave].type() != fixedValueFvPatchScalarField::typeName)
// FatalError << "inlet patch should be fixedValue!" << exit(FatalError);


alpha1.boundaryField()[slave] == interpolatedInletalpha1;


        //Info<< "inleterpolatoralpha1 = " << interpolatedInletalpha1 << endl;
```

```
//alpha2 interpolator    note: alpha2 is alpha.water
patchToPatchInterpolation interpolatoralpha2
(
        mesh.boundaryMesh()[master],                        // from patch
        mesh.boundaryMesh()[slave],                        // to patch
        intersection::HALF_RAY,
        intersection::VECTOR
);

scalarField interpolatedInletalpha2 = interpolatoralpha2.faceInterpolate <scalar>

                                        (
        alpha2.boundaryField()[master]
                                        );

// if(p_rgh.boundaryField()[slave].type() != fixedValueFvPatchScalarField::typeName)
// FatalError << "inlet patch should be fixedValue!" << exit(FatalError);

alpha2.boundaryField()[slave] == interpolatedInletalpha2;

//            Info<< "alpha2, water = " << interpolatoralpha2 << endl;


//k interpolator    note:
patchToPatchInterpolation interpolatorK
(
        mesh.boundaryMesh()[master],                        // from patch
```

```
        mesh.boundaryMesh()[slave],                         // to patch
        intersection::HALF_RAY,
        intersection::VECTOR
);

scalarField interpolatedInletK = interpolatorK.faceInterpolate <scalar>

                                (
        k.boundaryField()[master]
                                );

// if(p_rgh.boundaryField()[slave].type() != fixedValueFvPatchScalarField::typeName)
// FatalError << "inlet patch should be fixedValue!" << exit(FatalError);

k.boundaryField()[slave] == interpolatedInletK;

//Info<< "inletKInterpolDomain1 = " << interpolatedInletK << endl;

//omega interpolator    note:
patchToPatchInterpolation interpolatorOmega
(
        mesh.boundaryMesh()[master],                         // from patch
        mesh.boundaryMesh()[slave],                         // to patch
        intersection::HALF_RAY,
        intersection::VECTOR
);

scalarField interpolatedInletOmega = interpolatorOmega.faceInterpolate <scalar>
```

```
                                    (
        omega.boundaryField()[master]
                                    );

// if(p_rgh.boundaryField()[slave].type() != fixedValueFvPatchScalarField::typeName)
// FatalError << "inlet patch should be fixedValue!" << exit(FatalError);

//omega.boundaryField()[slave] == interpolatedInletOmega;

//Info<< "inletOmegaInterpolDomain1 = " << interpolatedInletOmega << endl;

//nut.water interpolator    note:
patchToPatchInterpolation interpolatorNutWater
(
        mesh.boundaryMesh()[master],                         // from patch
        mesh.boundaryMesh()[slave],                          // to patch
        intersection::HALF_RAY,
        intersection::VECTOR
);

scalarField interpolatedInletNutWater = interpolatorNutWater.faceInterpolate <scalar>

                                     (
        nutWater.boundaryField()[master]
                                     );

// if(p_rgh.boundaryField()[slave].type() != fixedValueFvPatchScalarField::typeName)
```

```
// FatalError << "inlet patch should be fixedValue!" << exit(FatalError);


nutWater.boundaryField()[slave] == interpolatedInletNutWater;


//Info<< "inletNutWInterpolDomain1 = " << interpolatedInletNutWater << endl;


//nut.particles interpolator    note:
patchToPatchInterpolation interpolatorNutParticles
(
        mesh.boundaryMesh()[master],                         // from patch
        mesh.boundaryMesh()[slave],                          // to patch
        intersection::HALF_RAY,
        intersection::VECTOR
);


scalarField interpolatedInletNutParticles = interpolatorNutParticles.faceInterpolate <scalar


                                            (
        nutParticles.boundaryField()[master]
                                            );


// if(p_rgh.boundaryField()[slave].type() != fixedValueFvPatchScalarField::typeName)
// FatalError << "inlet patch should be fixedValue!" << exit(FatalError);


nutParticles.boundaryField()[slave] == interpolatedInletNutParticles;


//Info<< "inletNutPInterpolDomain1 = " << interpolatedInletNutParticles << endl;
```

```
//theta.particles interpolator    note:

patchToPatchInterpolation interpolatorThetaParticles

(

        mesh.boundaryMesh()[master],                         // from patch

        mesh.boundaryMesh()[slave],                          // to patch

        intersection::HALF_RAY,

        intersection::VECTOR

);


scalarField interpolatedInletThetaParticles = interpolatorThetaParticles.faceInterpolate <sc


                                              (

        thetaParticles.boundaryField()[master]

                                              );


thetaParticles.boundaryField()[slave] == interpolatedInletThetaParticles;


// Info<< "inletThetaParticlesInterpolDomain1 = " << interpolatedInletThetaParticles << endl


//Info<< "alpha1slave = " << alpha1.boundaryField()[slave] << endl;

//Info<< "alpha1master = " << alpha1.boundaryField()[master] << endl;

//Info<< "U1slave = " << U1.boundaryField()[slave] << endl;




//End of interpolators
```

interpolateSlaveMaster.H

```
// This is to update the 1st region, after the 2nd region has solved.
/*      This code actually slowed down the solver by double.
                if (runTime.timeOutputValue() < SOI)
                        {
                            patchToPatchInterpolation interpolatorU1b
                                                (
                                                        mesh.boundaryMesh()[slave], // from p
                                                        mesh.boundaryMesh()[master], // to pa
                                                        intersection::HALF_RAY,
                                                        intersection::VECTOR //was CONTACT_SP
                                                );

                                                // interpolate from outlet to inlet
                                                vectorField interpolatedInletU1b = interpolat

                                                                                    (
                                                        U1.boundaryField()[slave]
                                                                                    );

                                                U1.boundaryField()[master] == interpolatedIn

                        }
*/

                        //U1 interpolator    note:U1=U.particles
/*                        patchToPatchInterpolation interpolatorU1b
                        (
                                mesh.boundaryMesh()[slave], // from patch
```

```
                mesh.boundaryMesh()[master], // to patch

                intersection::HALF_RAY,

                intersection::VECTOR //was CONTACT_SPHERE doesn't seem to ma

        );


        // interpolate from outlet to inlet

        vectorField interpolatedInletU1b = interpolatorU1b.faceInterpolate <


                                                (

                U1.boundaryField()[slave]

                                                );


        U1.boundaryField()[master] == interpolatedInletU1b;

*/

        //U2 interpolator    note:U1=U.particles

        patchToPatchInterpolation interpolatorU2b

        (

                mesh.boundaryMesh()[slave], // from patch

                mesh.boundaryMesh()[master], // to patch

                intersection::HALF_RAY,

                intersection::VECTOR //was CONTACT_SPHERE doesn't seem to ma

        );


        // interpolate from outlet to inlet

        vectorField interpolatedInletU2b = interpolatorU2b.faceInterpolate <


                                                (

                U2.boundaryField()[slave]
```

```
                                        );

    U2.boundaryField()[master] == interpolatedInletU2b;

    //p interpolator    note: needed to change scalar for vector
    patchToPatchInterpolation interpolatorP1
    (
            mesh.boundaryMesh()[slave], // from patch
            mesh.boundaryMesh()[master], // to patch
            intersection::HALF_RAY,
            intersection::VECTOR
    );

    scalarField interpolatedInletP1 = interpolatorP1.faceInterpolate <sc

                                        (
            p.boundaryField()[slave]
                                        );

    //if(p.boundaryField()[master].type() != fixedValueFvPatchScalarFiel
    //FatalError << "inlet patch should be fixedValue!" << exit(FatalErr

    p.boundaryField()[master] == interpolatedInletP1;

//    Info<< "inletP1InterpolDomain0 = " << interpolatedInletP1 << endl;

    //p_rgh interpolator    note: needed to change scalar for vector
    patchToPatchInterpolation interpolatorP_rgh1
```

```
                (

                        mesh.boundaryMesh()[slave], // from patch

                        mesh.boundaryMesh()[master], // to patch

                        intersection::HALF_RAY,

                        intersection::VECTOR

                );


                scalarField interpolatedInletP_rgh1 = interpolatorP_rgh1.faceInterpo


                                                        (

                        p_rgh.boundaryField()[slave]

                                                        );


                //if(p.boundaryField()[master].type() != fixedValueFvPatchScalarFiel

                //FatalError << "inlet patch should be fixedValue!" << exit(FatalErr


                p_rgh.boundaryField()[master] == interpolatedInletP_rgh1;


                //Info<< "inletP_rgh1InterpolDomain0 = " << interpolatedInletP_rgh1
/*
        if (runTime.timeOutputValue() < SOI)

                {

                    patchToPatchInterpolation interpolatoralpha1b

                    (

                            mesh.boundaryMesh()[slave], // from patch

                            mesh.boundaryMesh()[master], // to patch

                            intersection::HALF_RAY,

                            intersection::VECTOR
```

```
        );

        scalarField interpolatedInletalpha1b = interpolatoralpha1b.faceI

                                               (
                alpha1.boundaryField()[slave]
                                               );

        // if(p_rgh.boundaryField()[slave].type() != fixedValueFvPatchSc
        // FatalError << "inlet patch should be fixedValue!" << exit(Fat

        alpha1.boundaryField()[master] == interpolatedInletalpha1b;
    }
*/

/*                        //alpha1 interpolator    note: alpha1 is alpha.particles
    patchToPatchInterpolation interpolatoralpha1b
    (
            mesh.boundaryMesh()[slave], // from patch
            mesh.boundaryMesh()[master], // to patch
            intersection::HALF_RAY,
            intersection::VECTOR
    );

    scalarField interpolatedInletalpha1b = interpolatoralpha1b.faceInter

                                           (
            alpha1.boundaryField()[slave]
```

```
                                              );


                        // if(p_rgh.boundaryField()[slave].type() != fixedValueFvPatchScalar

                        // FatalError << "inlet patch should be fixedValue!" << exit(FatalEr


                        alpha1.boundaryField()[master] == interpolatedInletalpha1b;

*/
                        //alpha2 interpolator    note: alpha2 is alpha.water

                        patchToPatchInterpolation interpolatoralpha2b

                        (

                                mesh.boundaryMesh()[slave], // from patch

                                mesh.boundaryMesh()[master], // to patch

                                intersection::HALF_RAY,

                                intersection::VECTOR

                        );


                        scalarField interpolatedInletalpha2b = interpolatoralpha2b.faceInter


                                              (

                                alpha2.boundaryField()[slave]

                                              );


                        alpha2.boundaryField()[master] == interpolatedInletalpha2b;


//              Info<< "alpha2before = " << alpha2 << endl;


                        //k interpolator    note:

                        patchToPatchInterpolation interpolatorK1
```

```
(
        mesh.boundaryMesh()[slave], // from patch
        mesh.boundaryMesh()[master], // to patch
        intersection::HALF_RAY,
        intersection::VECTOR
);

scalarField interpolatedInletK1 = interpolatorK1.faceInterpolate <sc

                                  (
        k.boundaryField()[slave]
                                  );

//if(p.boundaryField()[master].type() != fixedValueFvPatchScalarFiel
//FatalError << "inlet patch should be fixedValue!" << exit(FatalErr

k.boundaryField()[master] == interpolatedInletK1;

//Info<< "inletK1InterpolDomain0 = " << interpolatedInletK1 << endl;

//omega interpolator    note:
patchToPatchInterpolation interpolatorOmega1
(
        mesh.boundaryMesh()[slave], // from patch
        mesh.boundaryMesh()[master], // to patch
        intersection::HALF_RAY,
        intersection::VECTOR
);
```

```
scalarField interpolatedInletOmega1 = interpolatorOmega1.faceInterpo

                                        (
        omega.boundaryField()[slave]
                                        );

//if(p.boundaryField()[master].type() != fixedValueFvPatchScalarFiel
//FatalError << "inlet patch should be fixedValue!" << exit(FatalErr

omega.boundaryField()[master] == interpolatedInletOmega1;

//thetaParticles1 interpolator    note:
patchToPatchInterpolation interpolatorThetaParticles1
(
        mesh.boundaryMesh()[slave], // from patch
        mesh.boundaryMesh()[master], // to patch
        intersection::HALF_RAY,
        intersection::VECTOR
);

scalarField interpolatedInletThetaParticles1 = interpolatorThetaPart

                                        (
        thetaParticles.boundaryField()[slave]
                                        );

//if(p.boundaryField()[master].type() != fixedValueFvPatchScalarFiel
```

```
                        //FatalError << "inlet patch should be fixedValue!" << exit(FatalErro

                        thetaParticles.boundaryField()[master] == interpolatedInletThetaPart
//                        Info<< "inletThetaParticlesInterpolDomain0 = " << interpolatedInle

                        patchToPatchInterpolation interpolatorNutWater1
                        (
                                mesh.boundaryMesh()[master],                        // from
                                mesh.boundaryMesh()[slave],                         // to pa
                                intersection::HALF_RAY,
                                intersection::VECTOR
                        );

                        scalarField interpolatedInletNutWater1 = interpolatorNutWater1.faceI

                                                                (
                                nutWater.boundaryField()[master]
                                                                );

                        // if(p_rgh.boundaryField()[slave].type() != fixedValueFvPatchScalar
                        // FatalError << "inlet patch should be fixedValue!" << exit(FatalEr

                        nutWater.boundaryField()[slave] == interpolatedInletNutWater1;
//                        Info<< "inletnutWaterInterpolDomain0 = " << interpolatedInletNutWa

                        patchToPatchInterpolation interpolatorNutParticles1
                        (
                                mesh.boundaryMesh()[master],                        // from
```

```
                mesh.boundaryMesh()[slave],                           // to pa
                intersection::HALF_RAY,
                intersection::VECTOR
        );

        scalarField interpolatedInletNutParticles1 = interpolatorNutParticle

                                                    (
                nutParticles.boundaryField()[master]
                                                    );

        // if(p_rgh.boundaryField()[slave].type() != fixedValueFvPatchScalar
        // FatalError << "inlet patch should be fixedValue!" << exit(FatalEr

        nutParticles.boundaryField()[slave] == interpolatedInletNutParticles
//                Info<< "inletnutParticlesInterpolDomain0 = " << interpolatedInletN
        //Info<< "inletEpsilon1InterpolDomain0 = " << interpolatedInletEpsil



        //Info<< "inletU1InterpolDomain0 = " << interpolatedInletU1 << endl;
        //Info<< "inletU2InterpolDomain0 = " << interpolatedInletU2 << endl;
        //Info<< "inletPInterpolDomain0 = " << interpolatedInletP << endl;
        //Info<< "inletp_rghInterpolDomain0 = " << interpolatedInletp_rgh <<
        //Info<< "inletalpha1InterpolDomain0 = " << interpolatedInletalpha1
        //Info<< "inletalpha2InterpolDomain0 = " << interpolatedInletalpha2
```

mySolverZagrebOmega.C

```
/*---------------------------------------------------------------------------*\
  =========                 |
\\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
\\    /   O peration     |
\\  /    A nd           | Copyright (C) 2011-2015 OpenFOAM Foundation
\\\\\\\\\\\\/     M anipulation  |
  -------------------------------------------------------------------------------
  License
   This file is part of OpenFOAM.

   OpenFOAM is free software: you can redistribute it and/or modify it
   under the terms of the GNU General Public License as published by
   the Free Software Foundation, either version 3 of the License, or
   (at your option) any later version.

   OpenFOAM is distributed in the hope that it will be useful, but WITHOUT
   ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
   FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
   for more details.

   You should have received a copy of the GNU General Public License
   along with OpenFOAM.  If not, see <http://www.gnu.org/licenses/>.

  Application
   reactingTwoPhaseEulerFoam

  Description
   Solver for a system of 2 compressible fluid phases with a common pressure,
```

```
    but otherwise separate properties. The type of phase model is run time
    selectable and can optionally represent multiple species and in-phase
    reactions. The phase system is also run time selectable and can optionally
    represent different types of momentun, heat and mass transfer.

\*---------------------------------------------------------------------------*/

#include "fvCFD.H"
#include "twoPhaseSystem.H"
#include "phaseCompressibleTurbulenceModel.H"
#include "fixedFluxPressureFvPatchScalarField.H"
#include "pimpleControl.H"
#include "localEulerDdtScheme.H"
#include "fvcSmooth.H"
#include "patchToPatchInterpolation.H" // for patchToPatch
#include "regionProperties.H" // for regions/domains
#include "kEpsilon.H"
#include "turbulentTransportModel.H"
#include "OFstream.H" // for writing files etc.
#include "basicKinematicCloud.H" // from ALopez
#include "kinematicParcelInjectionData.H"
#include "kinematicParcelInjectionDataIOList.H"
#include "KinematicLookupTableInjection.H"


//#include "TimeDataEntry.H"
//#include "kinematicLookupTableInjection.H"
```

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



int main(int argc, char *argv[])

{

    #include "setRootCase.H"

    #include "createTime.H"


        regionProperties rp(runTime); // from chtMultiRegionSimpleFoam


    #include "createMesh.H"

    #include "createFluidMeshes.H" // This is the first region mesh

    #include "createSolidMeshes.H" // This is the second region mesh


        pimpleControl pimple(mesh);


    #include "createTimeControls.H"

    #include "createRDeltaT.H"

    #include "createFields.H"


        if (!LTS)

        {

        #include "CourantNo.H"

        #include "setInitialDeltaT.H"

        }


        Switch faceMomentum

        (
```

```
            pimple.dict().lookupOrDefault<Switch>("faceMomentum", false)
    );

    Switch implicitPhasePressure
    (
            mesh.solverDict(alpha1.name()).lookupOrDefault<Switch>
            (
                    "implicitPhasePressure", false
            )
    );

#include "pUf/createDDtU.H"

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

    Info<< "\nStarting time loop\n" << endl;

    label master = mesh.boundaryMesh().findPatchID("baffleFace1D_master"); //added by me
    label slave = mesh.boundaryMesh().findPatchID("baffleFace1D_slave"); //added by me

    while (runTime.run())
    {

    #include "readTimeControls.H"

            if (LTS)
            {
        #include "setRDeltaT.H"
```

```
    }
    else
    {
#include "CourantNos.H"
#include "setDeltaT.H"
    }

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

    // --- Pressure-velocity PIMPLE corrector loop
    while (pimple.loop())
    {
            forAll(fluidRegions, i)
            {
                    Info<< "\nSolving for first region "
                        << fluidRegions[i].name() << endl;

                    fluid.solve();
                    fluid.correct();

    #include "YEqns.H"
                    if (faceMomentum)
                    {
    #include "pUf/UEqns.H"
    #include "EEqns.H"
    #include "pUf/pEqn.H"
    #include "pUf/DDtU.H"
```

```
                    }


                    else

                    {

        #include "pU/UEqns.H"

        #include "EEqns.H"

        #include "pU/pEqn.H"

                    }


                    fluid.correctKinematics();


                    if (pimple.turbCorr())

                    {

                            fluid.correctTurbulence();

                    }


                }

                Info<< "\nbefore interpolate" << endl;

        #include "interpolateMasterSlave.H"

                Info<< "\nafter interpolate" << endl;

        if ((runTime.timeOutputValue()+0.002) > SOI)

        {


// The above equation is: Number of particles = (Alpha Particles * Volume of Master cells)/(


                vectorField normalSlaveVector = mesh.Sf().boundaryField()[slave]; //

                //Info << "NormalSlaveVector " << normalSlaveVector; // Unormal give
```

```
                        //The magnitude of the vector is the area of the face. This is used 
                        scalarField dotProduct = U1.boundaryField()[master] & normalSlaveVec
                        scalarField uNormal = dotProduct/mag(normalSlaveVector);
                        //Info << "uNormal " << uNormal;

//The above equation is: Number of particles = alpha particles * Area of cell * velocity (sh
                        //number of particles/parcel * number of timesteps
                        vectorField centres = mesh.Cf().boundaryField()[slave] - (0.00005); 
                        // 'centres' is used in the below file.
                        //I'll try Cf as this is the face centeroid. This mod didn't seem to



                        // Maybe put an 'if' statement here too????
// Next section was added by Philip in Ireland


                        //kinematicParcelInjectionDataIOList injectorsTest_;
                        //Info<< "mesh objects" << mesh.objectRegistry::names() << endl;
                        //Info<< "time objects" << runTime.objectRegistry::names() << endl;
//                        const volScalarField& inject = mesh.lookupObject<kinematicLookupTa

//                        const kinematicLookupTableInjection& inject = mesh.lookupObject<ki
                        //const kinematicParcelInjectionDataIOList& injectorsTest_ = mesh.lo
                        //Info << "injectorList " << injectorsTest_ << endl;
                        //const scalar nu = injectors_.lookup("nu");
                        //injectors_.set("nu", 0.1);


                        ////////////////////////////////////// KinematicLookupTableInjection
```

```
// Comment this out if want to make static dictionary




                        kinematicParcelInjectionDataIOList& injectors =
                            const_cast<kinematicParcelInjectionDataIOList&>
                            (
                                mesh.lookupObject<kinematicParcelInjectionDataIOList>("kinem
                            );

                        forAll(injectors, i)
                        {
                            injectors[i].x() = centres[i]; //forgot to add this when in Croa
                            injectors[i].U() = U1.boundaryField()[slave][i];
                            injectors[i].numParticles() = abs((alpha1.boundaryField()[master
                            *uNormal[i])/(((((pi)*pow3(injectors[i].d()))/6))*1*(-1)*timeste
                            // i removed nParticle from here, as it is no longer correct.
                        }
                                injectors.write();

                    }



                    //I've changed the 8.71... to calculate the volume (based on the diamete
                                        //if (injectors[i].numParticles() > 0)
//                      {
//                              Info << "alpha1i " << alpha1[i];
//                      }
```

```
// Note: On the file output, I have hard written d, rho, and mDot. This can be changed in th
// This can be done in createFields.H
// The 'floor (blah blah)' function was also used to give integer values as outputs.
//These are rounded down, as per the floor command. The negative 1 is to make the output pos
//changed floor to abs, as floor was rounding down, and never rounding up.

//////////////////////////////////////////////////////







                        forAll(solidRegions, i)
                        {

                                argList::addOption // from ALopez gives option to change clo
                                (
                                        "cloudName",
                                        "name",
                                        "specify alternative cloud name. default is 'kinemat
                                );

                                alpha1.boundaryField()[slave] == 0; // This stops any 2nd ph

/*                          The code below actually slows down the solver. With the if state
                            Probably because there is double the area to solve in.
                                if (runTime.timeOutputValue() > SOI)
                            {
```

```
                                alpha1.boundaryField()[slave] == 0; // added by me

                                //alpha1.boundaryField()[master] = 0; // added by me

                            }

*/

                                Info<< "\nSolving for second region "

                                    << solidRegions[i].name() << endl;


                                fluid.solve();

                                fluid.correct();


            #include "YEqns.H"

                                if (faceMomentum)

                                {

                #include "pUf/UEqns.H"

                #include "EEqns.H"

                #include "pUf/pEqn.H"

                #include "pUf/DDtU.H"

                                }


                                else

                                {

                #include "pU/UEqns.H"

                #include "EEqns.H"

                #include "pU/pEqn.H"

                                }


                                fluid.correctKinematics();
```

```
                if (pimple.turbCorr())
                {
                        fluid.correctTurbulence();
                }

        }

        Info << "time_t= " << runTime.timeOutputValue() << endl;
        Info << "SOI= " << SOI << endl;

        #include "interpolateSlaveMaster.H"

        //Below added from DPMFoam
//                        continuousPhaseTransport.correct();
//                        mu2 = rho2*continuousPhaseTransport.nu();



        //newParcels = 0;


        //Info<< "mDot = " << injectors_[i].mDot() << endl;

// following is from DPMFoam
//
        Info<< "Evolving " << kinematicCloud.name() << endl; // from ALopez

        kinematicCloud.evolve();
```

```
kinematicCloud.updateMesh();
//injectors_.updateMesh();

// Update continuous phase volume fraction field
alpha2 = max(1.0 - kinematicCloud.theta() - alpha1, alpha2Min);
alpha2.correctBoundaryConditions();
alpha2f = fvc::interpolate(alpha2);
alphaPhi2 = alpha2f*phi1;

fvVectorMatrix cloudSU(kinematicCloud.SU(U2));
volVectorField cloudVolSUSu
(
        IOobject
        (
                "cloudVolSUSu",
                runTime.timeName(),
                mesh
        ),
        mesh,
        dimensionedVector
        (
                "0",
                cloudSU.dimensions()/dimVolume,
                vector::zero
        ),
        zeroGradientFvPatchVectorField::typeName
);
cloudVolSUSu.internalField() = -cloudSU.source()/mesh.V();
```

```
                        cloudVolSUSu.correctBoundaryConditions();

                        cloudSU.source() = vector::zero;

                        //

//              Info<< "alpha2after = " << alpha2 << endl;

                }


                runTime.write();


                /*Info<< "ExecutionTime = "

                    << runTime.elapsedCpuTime()

                    << " s\n\n" << endl;*/

                    Info<< "ExecutionTime = " << runTime.elapsedCpuTime() << " s"

                        << "  ClockTime = " << runTime.elapsedClockTime() << " s"

                        << nl << endl;


        }


        Info<< "End\n" << endl;


        return 0;

}




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

Appendix E

PatchPostProcessing.C

```
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | Copyright (C) 2011-2014 OpenFOAM Foundation
     \\/     M anipulation  |
-------------------------------------------------------------------------------
License
    This file is part of OpenFOAM.

    OpenFOAM is free software: you can redistribute it and/or modify it
    under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    OpenFOAM is distributed in the hope that it will be useful, but WITHOUT
    ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
```

```
    FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
    for more details.

    You should have received a copy of the GNU General Public License
    along with OpenFOAM.  If not, see <http://www.gnu.org/licenses/>.

\*---------------------------------------------------------------------------*/

#include "PatchPostProcessing.H"
#include "Pstream.H"
#include "stringListOps.H"
#include "ListOps.H"
#include "ListListOps.H"

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

template<class CloudType>
Foam::label Foam::PatchPostProcessing<CloudType>::applyToPatch
(
    const label globalPatchI
) const
{
    forAll(patchIDs_, i)
    {
        if (patchIDs_[i] == globalPatchI)
        {
            return i;
        }
```

```
    }

    return -1;
}



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

template<class CloudType>
void Foam::PatchPostProcessing<CloudType>::write()
{
    forAll(patchData_, i)
    {
        List<List<scalar> > procTimes(Pstream::nProcs());
        procTimes[Pstream::myProcNo()] = times_[i];
        Pstream::gatherList(procTimes);

        List<List<string> > procData(Pstream::nProcs());
        procData[Pstream::myProcNo()] = patchData_[i];
        Pstream::gatherList(procData);

        if (Pstream::master())
        {
            const fvMesh& mesh = this->owner().mesh();

            // Create directory if it doesn't exist
            mkDir(this->outputTimeDir());
```

```
const word& patchName = mesh.boundaryMesh()[patchIDs_[i]].name();

OFstream patchOutFile
(
    this->outputTimeDir()/patchName + ".post",
    IOstream::ASCII,
    IOstream::currentVersion,
    mesh.time().writeCompression()
);

List<string> globalData;
globalData = ListListOps::combine<List<string> >
(
    procData,
    accessOp<List<string> >()
);

List<scalar> globalTimes;
globalTimes = ListListOps::combine<List<scalar> >
(
    procTimes,
    accessOp<List<scalar> >()
);

labelList indices;
sortedOrder(globalTimes, indices);

string header("# Time currentProc " + parcelType::propertyList_);
```

```
            patchOutFile<< header.c_str() << nl;


            forAll(globalTimes, i)
            {
                label dataI = indices[i];


                patchOutFile
                    << globalTimes[dataI] << ' '
                    << globalData[dataI].c_str()
                    << nl;
            }
        }


        patchData_[i].clearStorage();
        times_[i].clearStorage();
    }
}



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


template<class CloudType>
Foam::PatchPostProcessing<CloudType>::PatchPostProcessing
(
    const dictionary& dict,
    CloudType& owner,
    const word& modelName
)
```

```
:
    CloudFunctionObject<CloudType>(dict, owner, modelName, typeName),
    maxStoredParcels_(readScalar(this->coeffDict().lookup("maxStoredParcels"))),
    patchIDs_(),
    times_(),
    patchData_()
{
    const wordList allPatchNames = owner.mesh().boundaryMesh().names();
    wordList patchName(this->coeffDict().lookup("patches"));

    labelHashSet uniquePatchIDs;
    forAllReverse(patchName, i)
    {
        labelList patchIDs = findStrings(patchName[i], allPatchNames);

        if (patchIDs.empty())
        {
            WarningIn
            (
                "Foam::PatchPostProcessing<CloudType>::PatchPostProcessing"
                "("
                    "const dictionary&, "
                    "CloudType& "
                ")"
            )   << "Cannot find any patch names matching " << patchName[i]
                << endl;
        }
```

```
        uniquePatchIDs.insert(patchIDs);
    }

    patchIDs_ = uniquePatchIDs.toc();

    if (debug)
    {
        forAll(patchIDs_, i)
        {
            const label patchI = patchIDs_[i];
            const word& patchName = owner.mesh().boundaryMesh()[patchI].name();
            Info<< "Post-process patch " << patchName << endl;
        }
    }

    patchData_.setSize(patchIDs_.size());
    times_.setSize(patchIDs_.size());
}


template<class CloudType>
Foam::PatchPostProcessing<CloudType>::PatchPostProcessing
(
    const PatchPostProcessing<CloudType>& ppm
)
:
    CloudFunctionObject<CloudType>(ppm),
    maxStoredParcels_(ppm.maxStoredParcels_),
```

```
    patchIDs_(ppm.patchIDs_),
    times_(ppm.times_),
    patchData_(ppm.patchData_)
{}


// * * * * * * * * * * * * * * * * Destructor  * * * * * * * * * * * * * * * //

template<class CloudType>
Foam::PatchPostProcessing<CloudType>::~PatchPostProcessing()
{}


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

template<class CloudType>
void Foam::PatchPostProcessing<CloudType>::postPatch
(
    const parcelType& p,
    const polyPatch& pp,
    const scalar trackFraction,
    const tetIndices& tetIs,
    bool&
)
{
    const label patchI = pp.index();
    const label localPatchI = applyToPatch(patchI);
```

```
    if (localPatchI != -1 && patchData_[localPatchI].size() < maxStoredParcels_)
    {
        vector nw;
        vector Up;

        // patch-normal direction
        this->owner().patchData(p, pp, trackFraction, tetIs, nw, Up);

        // particle velocity reletive to patch
        const vector& U = p.U() - Up;

        // quick reject if particle travelling away from the patch
        if ((nw & U) < 0)
        {
            return;
        }

        const scalar magU = mag(U);
        const vector Udir = U/magU;

        // determine impact angle, alpha
        const scalar alpha = mathematical::pi/2.0 - acos(nw & Udir);  //alpha mag(U) p.posit

const label patchFaceI = pp.whichFace(p.face());

        times_[localPatchI].append(this->owner().time().value());
```

```
        OStringStream data;

data<< Pstream::myProcNo() << ' ' << p <<" Alpha= " << alpha << " mag(U)= " << mag(U) << " F

        patchData_[localPatchI].append(data.str());

    }

}

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