University of Strathclyde

Department of Mechanical & Aerospace Engineering

Computational Fluid Dynamics Study of Erosion Processes

Alejandro López García

A thesis presented in fulfilment of the requirements
for the degree of Doctor of Philosophy

2017

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.

Alejandro López García

October 3, 2017

Abstract

A very large number of papers have been published on erosion for over 50 years. For the first time, a series of geometry independent tools and methodologies have been developed to calculate erosion and its associated mesh deformation in any three dimensional domain. The software used for development is verified first against the literature and Ansys Fluent. An error was found in the implementation of the Lagrangian phase in Fluent which was corrected in a later version. It is suspected that the error affects a large number of CFD publications in which the Discrete Phase Model was used. An experimental methodology and test-rig that is able to erode samples at mass concentrations ranging from 1% to 7% were developed and repeatability confirmed through testing. Despite not being able to use the test-rig due to technical issues, erosion associated deformation was validated from the literature, confirming the appearance of a new stagnation area as the wear scar deepens in the Jet Impingement Test. The algorithm was also applied to centrifugal slurry pumps in combination with state of the art erosion modeling and its results validated through visual inspection of eroded models.

Acknowledgements

First of all, I would like to thank my PhD supervisor Dr. Matthew Stickland for his support and help from the start to the end and for his unmeasurable patience with me. I would also like to thank my second supervisor Dr. William Dempster for his valuable input and help throughout the PhD. I wouldn't have had this opportunity if it wasn't for them.

I would also like to thank Prof. Donald Mackenzie and David Cunningham, who sadly passed away, for their support and help, not only to me but to all the PhD students in the Weir Advanced Research Centre.

I would also like to express my deep gratitude to The Weir Group for the funding that has made this project possible and in particular to Dr. Luis Moscoso Lavagna for his help as a Project Champion and Alan Bickley and Anthony Kinsella for their support.

My special thanks are extended to all the colleagues in WARC and especially to Athanasios, Georgios, Frazer and Aleksandar. Dr. Aldo Ianetti was especially helpful and his advice was of great assistance to me during the project as well as making my stay in Glasgow much more enjoyable.

I would like to offer my special thanks to my friend Dr. David Garcia for many enjoyable discussions about our projects and lots of help throughout the years.

I would also like to thank my dear friends Arrate and Julio for their kindness and help during my stay in Glasgow.

I would like o express my deep gratitude to my friends from Leeds for their help and support and, in particular, Dr. Yahia Fouda for his advice and excellent suggestions.

I don't think I will ever be able to thank my wife Nicole, my parents and family as well as my friends back home enough for their support and help throughout my life.

Publication list

Journal Papers

- CFD study of jet impingement test erosion using Ansys Fluent ®and OpenFoam ®. Lopez, A., Nicholls, W., Stickland, M. and Dempster, W. Dec 2015, Computer Physics Communications. 197, p. 88-95 8 p.
- Modeling erosion in a centrifugal pump in an Eulerian-Lagrangian frame using OpenFOAM ®. Lopez, A., Stickland, M. and Dempster, W. 24 Jul 2015, Open Engineering. 5, 1, p. 274-279 6 p.
- Computational study of fluid flow changes with erosion. Lopez, A., Nicholls, W., Stickland, M. and Dempster, W. Computer Physics Communications. (under review)

Conference Proceedings

- Comparative study different erosion models in an Eulerian-Lagrangian frame using Open Source software. Lopez, A., Stickland, M. and Dempster, W. 6 Oct 2014 12 p.
- Modelling erosion in a centrifugal pump in an Eulerian-Lagrangian frame using OpenFOAM ®. Lopez, A., Stickland, M. and Dempster, W. 20 Oct 2014 9 p.
- A comparison of CFD software packages' ability to model a submerged jet MacKenzie, A., Lopez, A., Ritos, K., Stickland, M. T. and Dempster, W. M. 7 Dec 2015 The 11th International Conference on CFD in the Minerals and Process Industries. Dickson, Australia, p. 1-4.

- A combined Euler-Euler Euler-Lagrange slurry model. MacKenzie, A., Lopez, A., Stickland, M. and Dempster, W. 27 Jun 2016 15 p.

Workshops

- Erosion modeling in OpenFOAM, Training Course. OpenFOAM Workshop 2016, University of Minho, Guimaraes, Portugal. `http://openfoam-extend.sourceforge.net/OpenFOAM_Workshops/OFW11_2016_Guimaraes/courses.html`
- Simulations with particles, using the Lagrangian method, Training Course. OpenFOAM Workshop 2016, University of Minho, Guimaraes, Portugal. `http://openfoam-extend.sourceforge.net/OpenFOAM_Workshops/OFW11_2016_Guimaraes/courses.html`
- Predicting surface evolution in erosion processes with OpenFOAM ®A. Lopez, OpenFOAM Workshop 2015, Ann Arbor, Michigan (USA) `http://openfoam-extend.sourceforge.net/OpenFOAM_Workshops/OFW10_2015_AnnArbor/?page_id=146`
- Comparison of Jet Impingement Test CFD modeling using Ansys Fluent®and OpenFOAM®Alejandro Lopez*, Matthew Stickland, William Dempster, William Nicholls `http://openfoam-extend.sourceforge.net/OpenFOAM_Workshops/OFW9_2014_Zagreb/download.html`

Contents

Abstract **ii**

Acknowledgements **iii**

Publication List **iv**

Contents **vi**

List of figures **xii**

List of tables **xx**

Nomenclature **xxi**

1 Introduction **1**

2 Literature Review **4**

2.1 Factors affecting erosion 4

2.1.1 Particles 5

2.1.2 Characteristics of the surfaces involved 19

2.1.3 Carrier fluid 23

2.2 Erosion prediction with CFD 27

2.3 Aims and objectives 29

3 Computational Fluid Dynamics in OpenFOAM **30**

3.1 CFD in OpenFOAM 30

3.2 Verification of OpenFOAM 31

3.2.1 Introduction 31
3.2.2 The Euler-Lagrange Approach 32
3.2.3 Simulation parameters 34
3.2.4 Eulerian phase steady-state 37
3.2.5 Discrete phase modeling 40
3.2.6 Impingement conditions 42
3.2.7 Erosion modeling in OpenFOAM 51
3.2.8 Conclusions 52

4 CFD of Erosion Processes **54**

4.1 Introduction 54
4.2 Erosion field calculation in OpenFOAM 54
4.3 Mesh deformation in OpenFOAM 61
4.3.1 Introduction 61
4.3.2 erodedBoundaryCellList.C 62
4.3.3 erosion.H 66
4.4 Implementation of an additional erosion model in OpenFOAM 79
4.4.1 Introduction 79
4.4.2 Implementation of an additional erosion model in OpenFOAM . 79
4.5 Patch interaction models 84
4.5.1 Introduction 84
4.5.2 Patch interaction models 84
4.6 Patch post processing 86
4.7 Turbulence model 89
4.7.1 Dispersed phase transient simulation 89
4.8 Statistics of target impacts 90
4.8.1 General case 90
4.9 Implementation of Euler-Lagrange and dynamic mesh solver 94

5 Experimental work **96**

5.1 Introduction 96
5.2 Test rig design 96

5.2.1 Experimental configuration 99
5.2.2 Simulation parameters 101
5.3 Three Dimensional Scanning 102
5.3.1 Alicona Infinite Focus IFM G4 102
5.4 Particle Image Velocimetry 103
5.4.1 Principles of PIV 103
5.4.2 Post-processing the PIV data 105

6 Results discussion 112
6.1 Introduction 112
6.2 Experimental results 112
6.2.1 1.15% Concentration tests 113
6.2.2 7% Concentration tests 115
6.3 Test rig preliminary CFD simulations 115
6.4 Validation fluid flow changes due to wear scar 122
6.4.1 Introduction 122
6.4.2 Methodology 122
6.4.3 Influence of the rebound model 125
6.4.4 Validation of the 3-dimensional wear scar 125
6.4.5 Time-scaling 138
6.5 Erosion calculation with a dynamic mesh solver 139
6.6 Three dimensional implementation of the Wear Map Method 142
6.6.1 Equation fitting 143
6.6.2 Equation fit with 120 points 146
6.6.3 Equation fit with 24 points 151
6.6.4 Discussion 155
6.7 Application to centrifugal pumps 156
6.7.1 86 AH slurry pump volute 156
6.7.2 150 WBH slurry pump impeller 161
6.8 Application to other cases 168

7 Conclusions 169

Appendices **171**

A Facewise Average and Standard Deviation calculation **172**

A.1 Procedure 172

A.1.1 FacewiseStandardDeviation.C 172

A.1.2 FacewiseStandardDeviation.H 183

B Implementation of additional rebound models **190**

B.1 Procedure 190

B.1.1 StandardWallInteraction.C 190

B.1.2 StandardWallInteraction.H 201

B.1.3 PatchInteractionModel.C 206

B.1.4 PatchInteractionModel.H 213

B.1.5 LocalInteraction.C 220

C Matlab Script for Scar comparison **235**

C.1 Procedure 235

C.2 Matlab Script for Scar comparison 235

D LPT for Erosion Modelling in OpenFOAM **239**

D.1 Introduction 239

D.2 Report 241

D.2.1 Theoretical Background 242

D.2.2 Introduction 242

D.2.3 Lagrangian Particle Tracking 242

D.2.4 Erosion 244

D.2.5 Implementation of LPT in OpenFOAM 245

D.2.6 Introduction 245

D.2.7 SolidParticle Class 245

D.2.8 The intermediate library 246

D.2.9 KinematicParcel Class 247

D.2.10 KinematicCloudProperties dictionary 248

D.2.11 Submodels 251

D.2.12 Erosion modeling 258
D.2.13 Implementation of Erosion Modelling in OpenFOAM 258
D.2.14 Templating in OpenFOAM 262
D.2.15 Function Templates [104] [98] [62] 262
D.2.16 Class Templates [104] [98] [62] 264
D.2.17 Coupling of the kinematicCloud class and an incompressible solver 265
D.2.18 Uncoupled Lagrangian Particle Tracking 265
D.2.19 Coupled Lagrangian Particle Tracking 275
D.2.20 Preprocessing 280
D.2.21 Geometry definition 280
D.2.22 The 0/ directory 281
D.2.23 The constant/ directory 282
D.2.24 The system/ directory 283
D.2.25 Running the case 284
D.2.26 Postprocessing 284
D.2.27 Lagrangian Particles in Paraview 284
D.2.28 Results of Coupled and Uncoupled Simulations 285
D.2.29 Post-processing erosion in Paraview 3.12.0 285
D.2.30 Report Appendix 1 286
D.2.31 kinematicCloudProperties Dictionary 286
D.2.32 Report Appendix 2 295
D.2.33 blockMeshDict 295

E Implementation of E-L solver with Dynamic meshing 299

E.1 introduction 299
E.2 Implementation of an Euler-lagrange solver with Dynamic meshing in OpenFOAM 2.2.x 300
E.3 Implementation of an Euler-lagrange solver with Dynamic meshing in OpenFOAM 2.3.x 301
E.3.1 DPMErosionFOAM.C 301
E.3.2 erosion.H 308

F Implementation of Gnanavelu's methodology **312**

F.1 Introduction 312

F.2 gnanaveluErosion.C 312

F.3 erosion.H 315

G Average velocity field calculation **319**

G.1 Introduction 319

G.2 avgVelocity.C 319

H Average truncated vorticity field calculation **324**

H.1 Introduction 324

H.2 avgTruncVorticity.C 324

I Erosion calculation in a pump **333**

I.1 Introduction 333

I.2 pumpErosion.C 333

J Mesh deformation with dynamic meshing **343**

J.1 Introduction 343

J.2 erodedBoundaryAdaptive.C 343

J.3 erosion.H 347

K Application to count the minimum number of impacts **350**

K.1 Introduction 350

K.2 ecountParticles.C 350

References **355**

List of figures

2.1 Velocity exponent n plotted against the ratio particle to target hardness ($\frac{H_p}{H_t}$): (a) for the glass targets, (b) for the ceramic targets [19] 7

2.2 Influence of rotation on weight loss-angle relation. The assumed distribution for the dimensionless parameter $a = \phi_0 r/U$ is also shown, where ϕ_0 is the rotational velocity, U is the particle velocity and r is the particle radius. [13] 8

2.3 Illustration of the contact between spinning spherical particles and a target wall [20] 8

2.4 Example of a particles with different shapes in a DEM simulation [29] . 12

2.5 Effect of an increase in particle concentration on different alloys [34] . . 13

2.6 Experimental data gathered in [41] and interpolated plots where τ is the incubation time of fracture 16

2.7 Erosion rate Vs. particle impingement angle for various sample temperatures (particle size, 451.5 μm; particle velocity, 183 $\frac{m}{s}$): $\circ, 577°C; \triangle, 493°C; \bigtriangledown, 371°C$. (Left) and Erosion rate us. sample temperature for various impingement angles (particle size, 451.5 μm; particle velocity, 183 $\frac{m}{s}$ (Right) [44]. 21

2.8 Velocity exponents for erosion data: normal incidence [17] 21

2.9 Effect of turbulence intensity on the wear of a ductile metal by 5 μm particles ($h = 0.41$)in an air jet with $Re_j = 20000$, $\frac{H}{d} = 12$ and $\frac{\rho_p}{\rho_f} = 1709$. Finnie's [12] model was used to calculate the erosion [56] 24

2.10 Variation in erosion rate R_t ($x10^6 \frac{g}{m^2*min}$) and collision efficiency of cylindrical steel targets 4.76 mm in diameter as a function of viscosity for 3 wt.% 75-106 μm Al_2O_3 suspensions in a slurry pot tester at 18.7 $\frac{m}{s}$ [59] 27
2.11 Example of Wear Map obtained by Gnanavelu et al in [9] 28
2.12 CFD of target after 30 mins erosion showing appearance of new stagnation point [60] 28

3.1 Overview of OpenFOAM structure [62] 31
3.2 3D geometry used showing the location of the boundary conditions . . . 34
3.3 5mm JIT domain with the inlet boundary highlighted in red and dimensions in mm 35
3.4 25 mm JIT domain with the inlet boundary highlighted in red and dimensions in mm 36
3.5 5 mm separation Steady state velocity contours comparison. OpenFOAM(right) and Ansys Fluent 38
3.6 25 mm separation Steady state velocity contours comparison. Units in m/s 38
3.7 Contours of the absolute differences for the 5 mm separation case 39
3.8 Contours of the absolute differences for the 25 mm separation case. Units in m/s 39
3.9 25 mm diameter cillindrical target subdivided into 1mm regions 43
3.10 Average particle velocities ($\frac{m}{s}$) at impingement versus distance from the centre of the target (m) in the two Fluent versions 44
3.11 Average impact angles at impingement ($degrees$) versus distance from the centre of the target (m) in the two Fluent versions 45
3.12 Impact locations for the 5mm nozzle distance case obtained with OpenFoam on the central part of the target 46
3.13 Impact locations for the 25mm nozzle distance case obtained with OpenFoam on the whole target 46
3.14 Fluent 15.0 Vs OpenFOAM 2.2.x particle velocities at impingement ($\frac{m}{s}$) for the 5 mm distance case 48

3.15 Fluent 15.0 Vs OpenFOAM 2.2.x particle velocities at impingement ($\frac{m}{s}$) for the 25 mm distance case . . . 49
3.16 Fluent 15.0 Vs OpenFOAM 2.2.x angles at impingement (*degrees*) for the 5 mm distance case . . . 50
3.17 Fluent 15.0 Vs OpenFOAM 2.2.x angles at impingement (*degrees*) for the 25 mm distance case . . . 51

4.1 Different views of cube with manually set boundary values for erosion (1a and 1b, non-zero values coloured in red), same values interpolated with IDW (2a and 2b, red being the highest values; blue the lowest) and deformation proportional to the interpolated values (3a and 3b. . . . 76
4.2 Erosion contours in a flat cylindrical probe and the resulting deformed geometry . . . 78
4.3 Erosion contours in a pipe bend and the resulting deformed geometry . 78
4.4 Representation of a circular domain divided in four faces, where N1, N2, N3 and N4 are the sample sizes to be obtained for each of the faces . . . 91
4.5 Face-wise impact velocity average ($\frac{m}{s}$) after 10 seconds with an escape condition at the target's boundary . . . 92
4.6 Face-wise impact velocity average ($\frac{m}{s}$) after 10 seconds with Forder's [16] rebound model at the target's boundary . . . 92
4.7 Face-wise impact angle average (degrees) after 10 seconds with an escape condition at the target's boundary . . . 93
4.8 Face-wise impact angle average ($\frac{m}{s}$) after 10 seconds with Forder's [16] rebound model at the target's boundary . . . 93
4.9 Face-wise impact number after 10 seconds with an escape condition at the target's boundary . . . 93
4.10 Face-wise impact number after 10 seconds with Forder's [16] rebound model at the target's boundary . . . 94

5.1 3D printed plastic venturi for particle injection . . . 98
5.2 Schematic of the venturi for particle injection . . . 99
5.3 Design of the contraction before the nozzle (all dimensions in mm) . . . 100

5.4 Velocity contours of the Jet Pump configuration obtained with Start CCM+ . . . 100
5.5 Velocity vectors at the pipe inside the Jet Pump . . . 101
5.6 Manufactured jet pump configuration . . . 102
5.7 Modification of venturi to be adapted to a jet pump configuration (all dimensions in mm) . . . 103
5.8 Jet pump design . . . 104
5.9 Nozzle configuration with sacrificial pump . . . 106
5.10 Test rig with sacrificial pump, scale and stirrer . . . 107
5.11 Schematic of test rig . . . 107
5.12 Cumulative siize distribution of the Frac Sand used for the experimental work . . . 108
5.13 Profilometry for three different samples eroded under the same conditions (1.15% sand concentration) for 15 minutes. Depth in μm and horizontal axis in mm . . . 108
5.14 Profilometry for three different samples eroded under the same conditions (7.2% sand concentration) for 15 minutes. Depth in μm and horizontal axis in mm . . . 109
5.15 Wear scar obtained in a 15 minutes experiment with 7.2% sand concentration . . . 109
5.16 Alicona Infinite Focus IFM G4 . . . 110
5.17 PIV setup schematics for a wind tunnel[80] . . . 110
5.18 Contours of velocity magnitude ($\frac{m}{s}$) obtained with PIVlab [83] . . . 111

6.1 Contours of erosion and wear scar profile after 30 mins of experiment at 1.15% sand concentration. Depth in μm and horizontal axis in mm . . . 113
6.2 Contours of erosion and wear scar profile after 45 mins of experiment at 1.15% sand concentration. Depth in μm and horizontal axis in mm . . . 114
6.3 Contours of erosion and wear scar profile after 1 hour of experiment at 1.15% sand concentration. Depth in μm and horizontal axis in mm . . . 114
6.4 Contours of erosion and wear scar profile after 2 hours of experiment at 1.15% sand concentration. Depth in μm and horizontal axis in mm . . . 115

6.5 Contours of erosion and wear scar profile after 1 hour of experiment at 7% sand concentration. Depth in μm and horizontal axis in mm 115
6.6 Test rig steady state velocity contours($\frac{m}{s}$) 116
6.7 Test rig steady state velocity contours($\frac{m}{s}$) around the particle injection . 117
6.8 Test rig steady state velocity contours($\frac{m}{s}$) around the target 117
6.9 Test rig steady state pressure contours(Pa) 118
6.10 Test rig steady state pressure contours(Pa) around the target 118
6.11 Mean particle impact velocity ($\frac{m}{s}$) on test rig's target 119
6.12 Mean particle impact angle (degrees) on test rig's target 119
6.13 Lagrangian particles coloured by velocity magnitude in the test rig domain and target coloured in blue 120
6.14 Erosion contours at the targets surface showing the symmetry axes . . . 121
6.15 Erosion contours at the target's surface showing the symmetry axes and the mesh faces 121
6.16 Contours of erosion per unit mass of impacting particles at 4 different simulation times. From left to right and from top to bottom: 1, 4, 8 and 10 seconds 123
6.17 Contours of erosion per unit mass of impacting particles at 0.5 and 10 seconds of simulation 124
6.18 Comparison of the normalised erosion ratio over the radius of the probe (mm) 124
6.19 Contours of erosion per unit mass of impacting particles for the same erosion model [84] and different rebound models. Forder et al [16] (left) and OpenFOAM's default rebound model (right) 125
6.20 Wear scar profile depth comparison (μm) along the radius (mm) for different scaling factors 126
6.21 Wear scar profile comparison with the experimental scars measured by Nguyen et al in [60] 127
6.22 Velocity contours of the uneroded geometry ($\frac{m}{s}$) 127
6.23 Static pressure contours of the uneroded geometry(Pa) 128
6.24 Velocity contours of the eroded geometry ($\frac{m}{s}$) for a scaling factor of 0.00349128

6.25 Static pressure contours of the eroded geometry(Pa) for a scaling factor of 0.00349 . . . 129
6.26 Velocity ($\frac{m}{s}$, left) and pressure contours (Pa, right) for scaling factor = 0.001976 . . . 130
6.27 Velocity ($\frac{m}{s}$, left) and pressure contours (Pa, right) for scaling factor = 0.0027 . . . 131
6.28 Velocity ($\frac{m}{s}$, left) and pressure contours (Pa, right) for scaling factor = 0.00349 . . . 132
6.29 Velocity ($\frac{m}{s}$, left) and pressure contours (Pa, right) foscaling factor = 0.00428 . . . 133
6.30 Velocity ($\frac{m}{s}$, left) and pressure contours (Pa, right) for scaling factor = 0.006585 . . . 134
6.31 Velocity ($\frac{m}{s}$, left) and pressure contours (Pa, right) for all the scaling factor = 0.02634 . . . 135
6.32 Surfaces obtained for all the scaling factors. From top to bottom and left to right: 0.001976, 0.0027, 0.00349, 0.00428, 0.006585 and 0.02634 . 136
6.33 Pressure contours (Pa) at the surfaces for all the scaling factors. From top to bottom and left to right: 0.001976, 0.0027, 0.00349, 0.00428, 0.006585 and 0.02634 . . . 137
6.34 Progressive mesh deformation and result of the dynamic meshing . . . 140
6.35 Progressive mesh deformation results for the velocity contours . . . 141
6.36 Progressive surface deformation with high damaging particles . . . 142
6.37 Impact velocity average across the radius of the test sample in $\frac{m}{s}$ . . . 143
6.38 Impact angle average across the radius of the test sample in degrees . . 144
6.39 Velocity at impingement average in $\frac{m}{s}$ . . . 144
6.40 Impact angle average in degrees . . . 145
6.41 Wear scars after 5, 15 and 30 minutes of test [60] . . . 145
6.42 Surface fitting for the wear scar and CFD case in [60] . . . 147
6.43 Residuals after fitting . . . 147
6.44 Wear map for the wear scar and CFD case in [60] . . . 148
6.45 Velocity contours in $\frac{m}{s}$ . . . 148

6.46 Static pressure contours in Pa . . . 149
6.47 Static pressure contours in Pa showing the formation of a possible stagnation point . . . 149
6.48 Velocity contours in $\frac{m}{s}$ . . . 150
6.49 Velocity contours in $\frac{m}{s}$ for the edge of the wear scar . . . 150
6.50 Static pressure contours in Pa at the target's surface . . . 151
6.51 Surface fitting for the wear scar and CFD case in [60] with 24 points . . 152
6.52 Residuals after fitting using 24 points . . . 153
6.53 Wear map for the wear scar and CFD case in [60] fitted with 24 points . 153
6.54 Velocity contours and deformed surface in $\frac{m}{s}$ . . . 154
6.55 Static pressure contours in Pa and deformed surface showing no new stagnation point . . . 154
6.56 Vorticity contours in s^{-1} at the surface of the eroded geometry truncated to a value of 100000 . . . 155
6.57 Picture of 86AH centrifugal slurry pump's mesh . . . 156
6.58 Picture of 86AH centrifugal slurry pump's steady state coloured by velocity magnitude $\frac{m}{s}$ . . . 157
6.59 Picture of 86AH centrifugal slurry pump's volute . . . 157
6.60 Picture of 86AH centrifugal slurry pump's volute coloured by erosion ratio 158
6.61 Picture of 86AH centrifugal slurry pump's volute deformed according to erosion . . . 158
6.62 Picture of 86AH centrifugal slurry pump's volute deformed according to erosion field with contours of erosion . . . 159
6.63 Top: 86AH centrifugal slurry pump's volute operating with Impeller WRT1, Middle:Picture of 86AH centrifugal slurry pump's volute obtained from worn unit operating with F6145WRT1 impeller and F6083 throatbush, Bottom: Computational volute wall deformed according to erosion showing similar erosion pattern . . . 160
6.64 Cross section of 150WBH pump showing the different parts . . . 162
6.65 Picture of 150WBH centrifugal slurry pump showing uneroded impeller 162
6.66 Picture of 150WBH centrifugal slurry pump showing volute and wear disc 163

6.67 Picture of 150WBH centrifugal slurry pump front vanes scan before (right) and after erosion (left) . . . 163
6.68 Picture of 150WBH centrifugal slurry pump impeller scan before and after erosion . . . 164
6.69 Picture of 150WBH centrifugal slurry pump front vanes truncated vorticity field . . . 164
6.70 Picture of 150WBH centrifugal slurry pump impeller truncated vorticity field . . . 165
6.71 Picture of 150WBH centrifugal slurry pump impeller truncated vorticity field . . . 165
6.72 Picture of 150WBH centrifugal slurry pump impeller before erosion deformation . . . 166
6.73 Picture of 150WBH centrifugal slurry pump eroded impeller according to calculated erosion magnitude . . . 166

D.1 Predicted variation of volume removal with angle of impingement for a single abrasive grain. Experimental points for erosion by many grains (△ copper, □ SAE I020 steel, ∘ aluminium) are plotted so that the maximum erosion is the same in all cases. . . . 245
D.2 Normal Distribution for Particle diameters between 150μm and 350μm . 256
D.3 Rosin-Rammlerl Distribution for Particle diameters between 150μm and 350μm . . . 257
D.4 Geometry of the pipe used for the tutorial case. . . . 280
D.5 Check "Skip Zero Time" box . . . 285
D.6 Check "kinematicCloud-lagrangian" and any of the available lagrangian fields . . . 286
D.7 Visualization of the particles in paraview . . . 287
D.8 Erosion contours in paraview . . . 288

List of tables

2.1 Factors affecting erosion [10] 4

2.2 Velocity exponents for erosion data: normal incidence [17] 6

2.3 Review of particle diameter and velocity exponents as shown in [18] . . 10

2.4 Particle density data [40] 15

2.5 Parameters selected in erosion wear models [1] 19

3.1 Transient simulation features 42

6.1 Results of the fit for 120 points 146

6.2 Results of the fit for 24 points 152

Nomenclature

$\mathbf{V}$	velocity ($\frac{m}{s}$)
Re_p	Particle Reynolds Number (-)
d_p	Particle diameter (m)
u_p	Particle velocity ($\frac{m}{s}$)
u	Fluid velocity ($\frac{m}{s}$)
H_p	Particle Hardness (HB, HV, HRC...)
H_t	Target material Hardness (HB, HV, HRC...)
C_D	Drag coefficient (-)
ρ_p	Particle density ($\frac{kg}{m^3}$)
τ_p	Particle characteristic time (s)
μ_f	Fluid's dynamic visosity($\frac{N*s}{m^2}$)
$\hat{\boldsymbol{u}}$	Unitary surface normal vector (-)
$\boldsymbol{u}$	Surface normal vector (m^2)
$\lVert\boldsymbol{u}\rVert$	Modulus of surface normal vector (m^2)
$E(P_j)$	unknown values of the erosion field in location P_j (m)
$E(C_i)$	values of the erosion field at the known locations(m)
λ_i	weighting factors (-)
l_i	inverse distance ($\frac{1}{m}$)
l_it	sum of inverse distances ($\frac{1}{m}$)
d_i	distance from each face centre to each point (m)
D_j	distance to be added to each of the boundary points (m)
P_i^1	location of the new boundary points (m)
P_i^0	initial location of the boundary points (m)

Acronyms

ARCHIE-WeST	Academic and Research Computer Hosting Industry and Enterprise in the West of Scotland
CFD	computational fluid dynamics
JIT	Jet Impingement Test

Chapter 1

Introduction

Erosion is responsible, amongst others, for destroying a wide variety of equipment and causing vast losses in all kinds of industries. For over 50 years, engineers have been trying to understand the process and, as a result, a large number of scientific papers have been published on this subject. Most of these authors have captured their very own and specific ideas about the way erosion mechanisms work as well as the equations to predict wear in a number of different geometries. Meng and Ludema [1] carried out a broad literature review of more than 5000 papers dating from 1957 to 1992. In their article they identified 28 separate erosion models out of the almost 2000 existing empirical models, a fact which exemplifies the poor agreement between authors on this subject. One of the few theories on which there seems to be some kind of agreement describes two mechanisms acting together to produce the wear scar: cutting and deformation wear. When particles hit the surface and they tear material away with them in a cutting action, it is called cutting wear. This mechanism is the predominant one for ductile materials and particles impinging at low angles of attack with respect to the surface being eroded. Alternatively, several particles might impact on the same place transferring some of their kinetic energy to the surface in the form of hardening work [2]. According to this theory, in a given collision with the target material, as soon as the particle contacts the surface, stress concentrations appear as a result of the elastic deformation that takes place. If these stresses are not over the elastic limit of the target material, and also leaving aside fatigue damage effects, they should cause no deformation. However, if the elastic limit is reached, plastic

deformation will occur at the location of the maximum stress. The repeated impacts then create a plastically deformed layer that will deform further upon repetition of the particle collisions. This deformation causes hardening and increases the elastic limit in that region turning the material harder and more brittle until it reaches a point where it can no longer be plastically deformed. Eventually, upon further load, pieces of the material's surface separate from the target and are carried away by the fluid. This hypothesis was studied by Davies in [3], and Van Riemsdijk and Bitter in [4] and then adopted by several authors [2, 5–8]. This mechanism is called deformation wear and it predominates for high angles of impingement and in brittle materials.

However, the final forms of the equations differ tremendously from each other. This is due to the fact that the number of factors to be taken into account is very large. An assessment of some of the variables mentioned in the literature yields more than 20 different factors, of which many are susceptible of further subdivision. One of the factors that has not yet been properly addressed is the effect on the fluid flow that the erosion-modified surface has. The aim of this work is to develop a three dimensional mesh deformation algorithm which, combined with a suitable mechanism for subdividing the enlarged cells at the eroded boundaries, will enable computing how the fluid flow changes with progressing erosion. The test chosen for both modelling and validation is the Jet Impingement Test. Numerous works have been carried out on erosion modelling of the jet impingement, giving as a result a number of different formulae for calculating erosion induced by solid particles. These formulae serve as an optimum starting point for calculating erosion contours which will be the precursor of the deformation algorithm.

Numerical verification of the software used for erosion modelling (OpenFOAM ®) will be carried out in a comparison of the averages of the particle impact variables as previously outlined by Gnanavelu et al in [9]. A test rig will be developed and implemented after this with the aim of validating the CFD results through Particle Image Velocimetry. This technique allows comparing both particle trajectories for the particles eroding the geometry as well as the contours of the fluid flow before and after erosion.

Finally, additional cases of real eroded slurry pumps will be set up to test if the

deformation algorithm is able to capture erosion induced deformation in such complex systems.

Chapter 2

Literature Review

2.1 Factors affecting erosion

When considering erosion induced by solid particles, factors affecting erosion may be grouped under three categories [10], corresponding to the particles, the surfaces involved in the process and the carrier fluid as shown in table 2.1.

	For particles		For surfaces		For the carrier fluid
1	Impact and rebound angles	1	Physical properties	1	State of motion (laminar versus turbulent)
2	Impact and rebound speeds	2	Change in shape caused by erosion	2	Velocity
3	Rotation before and after impact	3	Stress level	3	Temperature
4	Shape and size	4	Temperature	4	Chemical composition and physical properties
5	Volume concentration	5	Presence of oxide (or other) coatings		
6	Physical properties (hardness strength and density)	6	Simultaneous occurrence of corrosion		
7	Fragmentation				
8	Interactions (with surfaces, fluid or other particles)				
9	Temperature				
10	Presence of additives				
11	Electrical charge				

Table 2.1: Factors affecting erosion [10]

In this section, some of the factors in table 2.1 and their influence on the process will be discussed. In many cases, individual factors may be considered in different ways by different authors. Sometimes, even the same authors may add some new ideas or factors that they hadn't previously considered in the initial theory. Because each of these authors have their very own approach to erosion, a variety of their opinions about each of the factors will be highlighted.

2.1.1 Particles

Impact and rebound angles

Dependency of the erosion rate on the impact and rebound angles varies significantly between authors. Most of the equations have been developed empirically. From the observations for each material, there seems to be an angle of impingement for which erosion is maximum. In ductile materials, the angle of maximum erosion tends to be small, while values closer to 90 degrees maximise erosion for brittle materials. The difference in the angle with which the particle strikes makes it affect the surface in a different way, either by ploughing and cutting, when the angles are smaller, or by crack formation , fatigue and material extrusion when the angles are closer to normal impingement [3, 6, 7, 11]. In the equations developed by some authors like Finnie [12, 13] and Bitter [6, 7] the angle dependency in the erosion equation is derived from a theoretical approach. Others like Gnanavelu et al [9, 14], develop empirical formulae for the angle of impingement taking into account their experimental wear scar. The rebound angle depends on a number of parameters; the size of the particle, surface roughness, materials involved, density of the particulate phase and the fluid etc. There are several models which attempt to represent the relationship between the normal and tangential components of the velocity before and after impact like the ones developed by Sommerfeld et al [15] and Forder et al. [16]. According to Humphrey [10], the accuracy of mechanical methods in determining impact and rebound angles usually decreases with increasing particle concentration and decreasing particle size.

Impact and rebound speeds

The main expression used in erosion prediction is equation 2.1:

$$wear\ rate \propto V^n \tag{2.1}$$

Where the exponent of the impact velocity may vary depending on each author and also on the methodology used in the experiments from which the equation was derived. In general, the wear rate specifies the rate at which material is removed from the surface or the rate at which dimensions change due to wear of the surface. The

units used in the definition of the wear rate may differ depending on the dimensional analysis of the parameters used in its calculation. A set of velocity exponents was gathered by Wiederhorn et al [17] and is shown in table 2.2. Sheldon and Kanhere also made a survey [18] and found different exponents in the erosion equation for both diameter and particle velocity. These exponents are represented in table 2.2. Bitter [6] also postulated that when very high velocities are taken into account (2000 $\frac{m}{s}$ and dry solid particle erosion) different phenomena appear such as enlarged craters several times bigger than the particle along with large heat effects due to the collision.

Target Material	Erosion particles	Exponent
Soda-lime-silica glass	SiC, 120 gritt	3.0
MgO (96.5%)	SiC, 120 gritt	2.7
Al_2O_3 (99.5%)	SiC, 120 gritt	2.7
Pyrex glass	Al_2O_3 30μm 10μm	2.2 2.7
Hot pressed Si_3N_4	SiC 8μm *to* 940μm	4.0
Silicon	Al_2O_3 23μm *to* 270μm	3.4 to 2.6 depending on particle size
Reaction bonded SiC	Al_2O_3 130μm 270μm	2.3 to 2.0
Hot pressed SiC (96.5%)	Al_2O_3 130μm 270μm	1.8 1.5

Table 2.2: Velocity exponents for erosion data: normal incidence [17]

Shipway and Hutchings [19] investigated the effect of the ratio of particle to target hardness on the velocity exponent and illustrated the results in figure 2.1

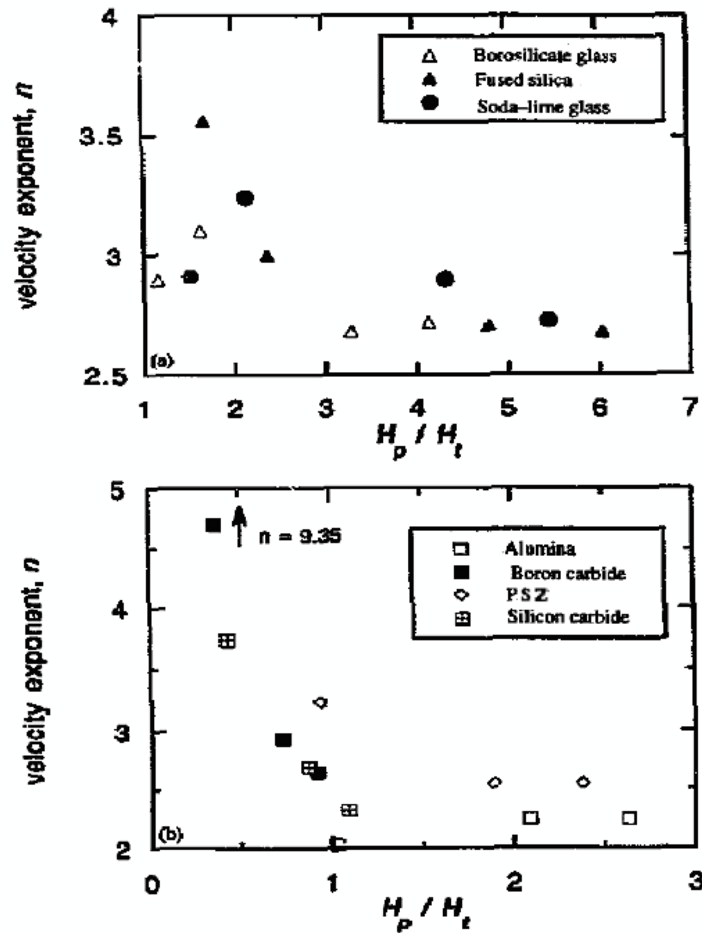


Figure 2.1: Velocity exponent n plotted against the ratio particle to target hardness ($\frac{H_p}{H_t}$): (a) for the glass targets, (b) for the ceramic targets [19]

In the table, values for the exponent ranging from around 2.2 up to 4.7 can be found, exemplifying its variability and its dependence on the properties of the particle and target materials.

Rotation before and after impact

One of the first formulae for calculating wear was proposed by Finnie [12]. However, results are not accurate when angles close to normal incidence are considered. In order to have a better representation of what he found in his experiments, Finnie published a second article a few years later [13] in which he incorporated other factors such as particle rotation at impingement. A rotational component to the particle movement is

incorporated through a hypothetical omega distribution and it's effect is illustrated on figure 2.2, which represents the weight loss against angle of impingement.

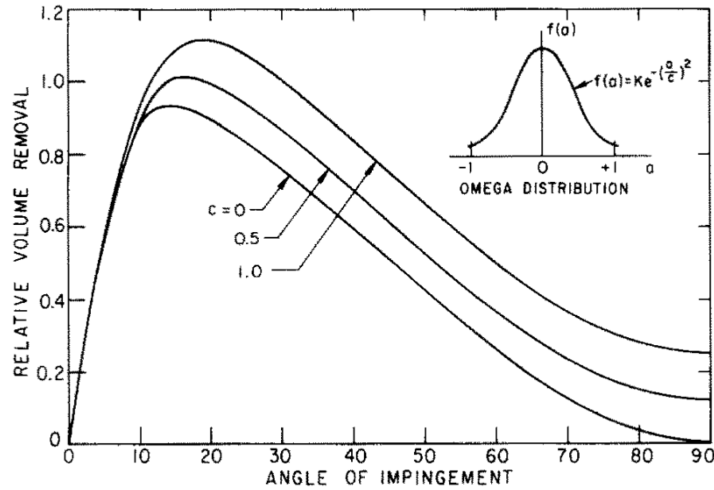


Figure 2.2: Influence of rotation on weight loss-angle relation. The assumed distribution for the dimensionless parameter $a = \phi_0 r/U$ is also shown, where ϕ_0 is the rotational velocity, U is the particle velocity and r is the particle radius. [13]

Bingley et al investigated the effect of particle rotation on the erosion rate of metals in [20]. From their experiments they concluded that higher erosion rates are expected when the impinging particles have back-spin (see figure 2.3) and the difference was more significant at low impact angles, when the cutting wear mechanism is of importance.

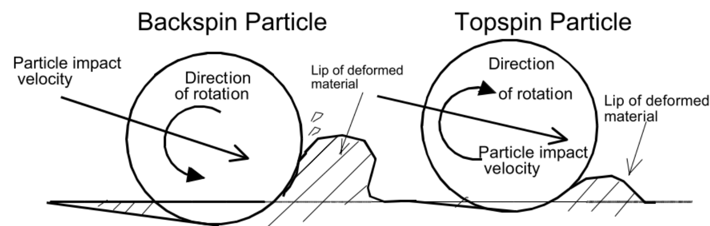


Figure 2.3: Illustration of the contact between spinning spherical particles and a target wall [20]

As Bingley et al stated in [21], there is not much work carried out on the influence of particle rotation on erosion, although it's effect has been repeatedly acknowledged. One of the problems is the difficulty in the measurement of particle rotation when the particles are very small. Thus, most research on particle rotation has been carried out for relatively big sized particles or theoretical modelling [13]. The effect of rotation

becomes more important in rotating flows such as centrifugal erosion testers [21] and centrifugal pumps. Finnie [13] suggested that its effect may be accounted for by introducing particle rotation by means of a statistical distribution. He analysed the effect of this distribution on the theoretical erosion of metals, obtaining higher rates when particle rotation was considered. It should be possible to estimate particle rotation velocities with high speed photography. However, even then, it will still be difficult to determine how this variable influences the erosion rate. One of the tests that has been most commonly used in previous studies is the Jet Impingement Test (JIT). Particle rotation may also be of importance in this test. In fact, as the particles impinge on the target, they might do it with back spin, since, in the region outside the stagnation point, the fluid layers are traveling faster as they get closer to the surface, generating vorticity and, consequently, particle rotation. With the development of Molecular Dynamics theory [22], an in depth study of how rotation affects the erosion rate is now possible. A suitable starting point would be simulating particles with and without rotation in order to obtain a quantifiable difference in the amount of material eroded.

Apart from this, particle rotation may also notoriously affect its trajectory when the Magnus force (the force caused by particle rotation inside a viscous flow that acts orthogonally to the particles movement direction) is strong enough. In the cases treated in this thesis the effect of this force is considered to be included with the incorporation of turbulent dispersion to the particles. It can be said that the variations in the impact location that would be induced by the Magnus force due to the rotation of the particles (random a priori) inside the viscous fluid (a rotation induced by the fluid shear stresses) are successfully represented by the turbulent dispersion term that provides the randomness to the simulation that the real impacts would have.

Shape and size

Regarding the size of the erodent, the exponent of the diameter of the particles also changes significantly with each author and experimental procedure, as shown in table 2.3. Regarding particle trajectories, in a submerged jet, these are going to be strongly dependent on the particle size, as well as the jet velocity at the nozzle exit, as pointed out by Benchaita et al in [23]. Assuming the same density, if the volume is increased,

particles will follow the flow less faithfully. In their article, Benchaita et al refer to a size of the particle ($d_p > 2000\mu m$) for which the buoyancy force will probably be larger than the drag force on the particle. The outcome of this is that the particles will hit the target at approximately the same impact angle as the jet. When the opposite is true, i.e. the particles are smaller than a certain size ($d_p < 200\mu m$ was the critical diameter in Benchaita's study), they will tend to follow the flow. Having, as a consequence, less impacts on the target since some of the particles will escape the area without interacting with it. In the JIT the number of impacts on the target will also depend on the distance between the exit of the jet and the target as outlined by Lopez et al in [24]. Sheldon and Kanhere carried out a literature review in [18] and confirmed this disparity. They assumed the equation used to calculate wear produced by single particles has the form of equation 2.2 and found different exponents for both diameter and velocity depending on materials and type of experimental procedure.

$$wear \quad rate = K * V^a * D^b \tag{2.2}$$

Test type	Impact angle	a	b
Air blast, 60 mesh, SiC grit on steel, Cu, Al	20°	2.36	-
Air blast, 180 mesh, SiC grit on steel	20°	2.36-2.69	-
Air blast, $SiC > 100\mu m$ on steel, Cu, Al	30°	3.0	-
Air blast, 60 mesh, SiC on many metals	20°	2.05-2.44	2
Air blast, 60 mesh, $SiC > 100\mu m$ on Aluminium	20°	-	3.0*
Air blast, 60 mesh, SiC grit on Al, Cu, steel	20°	2.9	3.0
Whirling arm, quartz grit $> 125\mu m$ on 11% Cr-steel aluminium, etc.	90°	2.3	3.0*
Whirling arm, $125 - 150\mu m$ glass shot on Aluminium	90°	2.4	-

*As inferred from the independence of erosion weight loss,
per unit weight of abrasive particles (mg/g) on abrasive particle size.

Table 2.3: Review of particle diameter and velocity exponents as shown in [18]

According to the literature survey made by Zhong and Minemura in [25], wear increases with increasing particle size. However, there seems to be a critical size (represented by the diameter of the particle d_p for which the rate at which the wear changes will depend on the elastic and plastic properties of the material being eroded. Finnie also investigated the effect of particle size in [13]. In this study he located the value of the critical size around $100\mu m$ stating that the erosion rate becomes independent of particle size when this number is reached. He also found erosion to be less and less efficient as the particles become smaller. Some of the explanations he offered as an

attempt to explain this behaviour relate to particle fragmentation when their size is bigger, grain size of the metal being eroded or an oxidized layer of the eroded surface. Regarding the shape of the abrasive, in general, the rate at which the material is being eroded increases with particle sharpness [26]. The typical variable used to describe the sphericity of the abrasive is the particle roundness. A sphere would have a roundness value of 1, while very sharp and angular abrasives have typical roundness values which range from 0.25 to 0.40 [27]. Some authors like Oka et al [28] reported some effect of the angularity of the particles on the erosion rate, not affecting however the impact angle dependence. Jennings et al [27] tested three different abrasives with three different shapes; one spherical and two angular. In their experiments they noted that at low velocities, the difference between the erosion rate produced by the angular and spherical particles was much higher than at high velocities. This difference became much smaller at high velocities, with the erosion rate of the angular particles not increasing significantly. According to them, this meant that the erosion rate for the spherical particles drastically increases when moving from low to high velocities. In CFD, the simplest way of modelling particles with different shapes is by adding a roundness factor to the drag in Lagrangian simulations which has a value between 0 and 1. This factor is usually calculated as a ratio involving the perimeter of the particle and it's projected area ($\frac{P^2}{4\pi A}$, where P is the perimeter and A is the projected area of the particle [11]). DEM (Discrete Element modelling) simulations enable modelling of particles made up of spheres of different sizes which are able to mimic very different shapes, as shown in figure 2.4.

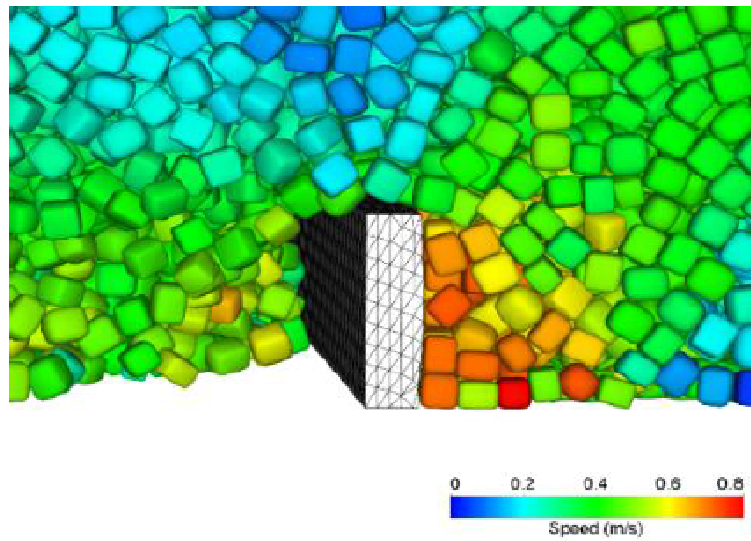


Figure 2.4: Example of a particles with different shapes in a DEM simulation [29]

Volume concentration

The effect of particle concentration has been repeatedly studied [5, 20, 30–34]. Measurements confirm that the erosion rate increases with an increment in the particle concentration. However, differences in the properties of the target materials, the range of concentrations, particle properties, sizes and even the carrier fluid make it difficult to extract definitive conclusions. Grant and Tabakoff's tests [5] on 2024 Aluminium with 80 μm Alumina particles at an angle of impingement of 30 degrees showed almost no variation in the erosion rate with concentration (0.328 to 1.312 $\frac{kg}{m^3}$) when the test velocity was 140.208 $\frac{m}{s}$ and 164.592 $\frac{m}{s}$) using air as the carrier fluid. Similar results were obtained by Sage and Tilly in [35] for sand (15-35 μm and 125-135 μm) eroding a titanium alloy at airspeeds of 243.84 $\frac{m}{s}$ and 335.28 $\frac{m}{s}$, concluding that concentration effects made no sensible difference. When included in empirical formulae, different coefficients for a power-law relationship have been proposed, being between 0.5 and 1.0 [31]. Stack et al studied the erosion-corrosion of pure metals with varying concentration of particles.Their investigation proved higher erosion-corrosion rates with increasing concentration. As the concentration increases so does the frequency of particle interactions. This influences the kinetic energy of the particles when they reach the target, but not the impact frequency [30]. Patil et al [32] measured in their alu-

minium samples a non-linear increase of the erosion rate with concentration of 505 μm sand particles when concentration is increased from 20% to 40%. However, Rajahram et al [33] measured a linear increase in the mass loss when the concentration of sand particles was increased from 1% to 5%. Erosion, synergy (explained in Section 2.1.2) and erosion-corrosion also follow linear trends in the austenitic stainless steel tested. In a recent study, Bart et al [34] showed how erosion rate changed when concentration was increased from 2% to 3.5%, as shown in figure 2.5

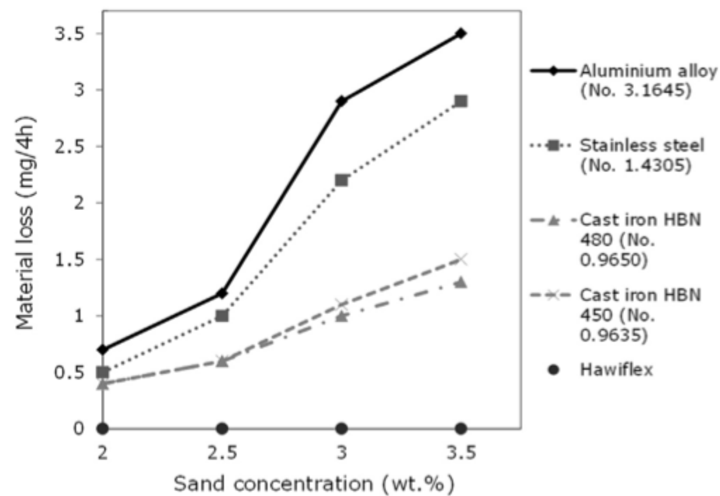


Figure 2.5: Effect of an increase in particle concentration on different alloys [34]

In general, if particle concentration is increased, for the same experimental set up, a higher number of particles will impact the target material thus showing an increase in the erosion rate [34]. It has also been reported [20] that concentration effects on the erosion rate might also depend on particle shape. Bingley et al investigated this effect for low and high particle concentrations. Erosion rate decreases when the angle of impingement is increased in all cases. But, for the same impingement angle, tests with high and low concentrations seem inconclusive. The explanation offered in this case relates to a 'shielding effect' by which the particles rebounding from the target walls protect the target from further impacts. However, the same observation can't be confirmed for spherical particles. For this reason, in their study, Bingley et al attribute the effect to an increase in particle collisions which would reduce the amount of spin the particles may have thus reducing the erosion rate.

Physical properties

The extensive literature review carried out by Meng and Ludema in [1] in which the most important and frequently used equations for erosion prediction are listed, serves as a valuable tool to analyse which physical properties are considered in the different models. The physical properties gathered by Meng and Ludema include the density, hardness, roundness, size (discussed in section 2.1.1)and moment of inertia of the particles. One of the first authors to investigate the effect of the physical properties of the abrasive was Finnie in [12]. Amongst the properties considered were shape (discussed in section 2.1.1), hardness and strength. In general, the harder the abrasive, the higher the erosion rate [26]. However, Head et al [36] reported fluorite particles (Moh's hardness=4.0) being more erosive than alumina particles (Moh's hardness=9.0). Mason et al [37] stated that the angle for which maximum erosion occurs is not dependent upon particle shape. It was also reported that if the surface is softer than the particle, an increase in particle hardness would not impact the erosion rate significantly. Particle hardness is actually the most widely used property to define erosion rates for different materials [38]. Bitter's model [6] assumes the surface is deformed plastically only if the particles are harder than the surface and provided that they don't disintegrate upon impact. Shipway and Hutchings studied the erosion mechanisms of some brittle materials [19] and concluded that fracture induced by indentation was the predominant mechanism when the particles were harder than the target material. If the particles were softer than the target material a small-scale chipping mechanism would dominate erosion. They analysed some of the existing erosion models for brittle materials and stated that deviations occur depending on the ratio of particle hardness to target material hardness. If it was close to unity the velocity exponent increased while the erosion rate decreased. The transition in the erosion rates when this ratio reaches the value of unity had previously been noted by Srinivasan and Scattergood when studying the effect of erodent hardness on the erosion of brittle materials in [39]. Regarding particle density, Clark investigated its effect in [40]. He argued that, under the same test conditions, higher density particles will impact the target with higher velocities and will show higher collision efficiency: effects which are related to the drag force exerted by the surrounding fluid. Table 2.4 shows correspondence between predicted

and experimental velocities for particles with different densities.

Material	Density ($\frac{kg}{m^3}$)	Mean particle diameter (μm)	Mean crater diameter (μm)	Experimental impact velocity ($\frac{m}{s}$) $\nu_{I_{expt.}}$	Predicted impact velocity ($\frac{m}{s}$) ν_I(18.7)
Glass	2420	666	103.5	8.5	11.14
Zirconium Oxide	3810	629	118.2	9.5	13.29
High density	5320	538	114.4	10.6	14.41
zirconium oxide	5320	580	122.7	10.9	14.49
Steel	7830	701	170.8	14.15	15.78

Table 2.4: Particle density data [40]

Fragmentation

Particle fragmentation upon impact with the sample might affect erosion producing what has been named secondary erosion. However, reviewing the literature, in most of the references that consider particle fragmentation the particle velocity is over 100 m/s. In the present case, the velocity magnitude is in the range of a tenth of that value. Thus, fragmentation should not affect the particles at such a low velocity. In [41] fragmentation specific energy and the threshold velocity are analysed for spherical particles at normal incidence. The experimental data gathered allowed Gorbushin and Petrov to conclude that there is a threshold value in the energy below which there is no fracture. According to their experimental data there should be no fragmentation at all when the velocity is kept under 20 m/s and, with particles smaller than 500 μm, fragmentation would definitely not be expected. Figure 2.6 illustrates the data gathered in the experiments in [41] and the interpolated curves obtained.

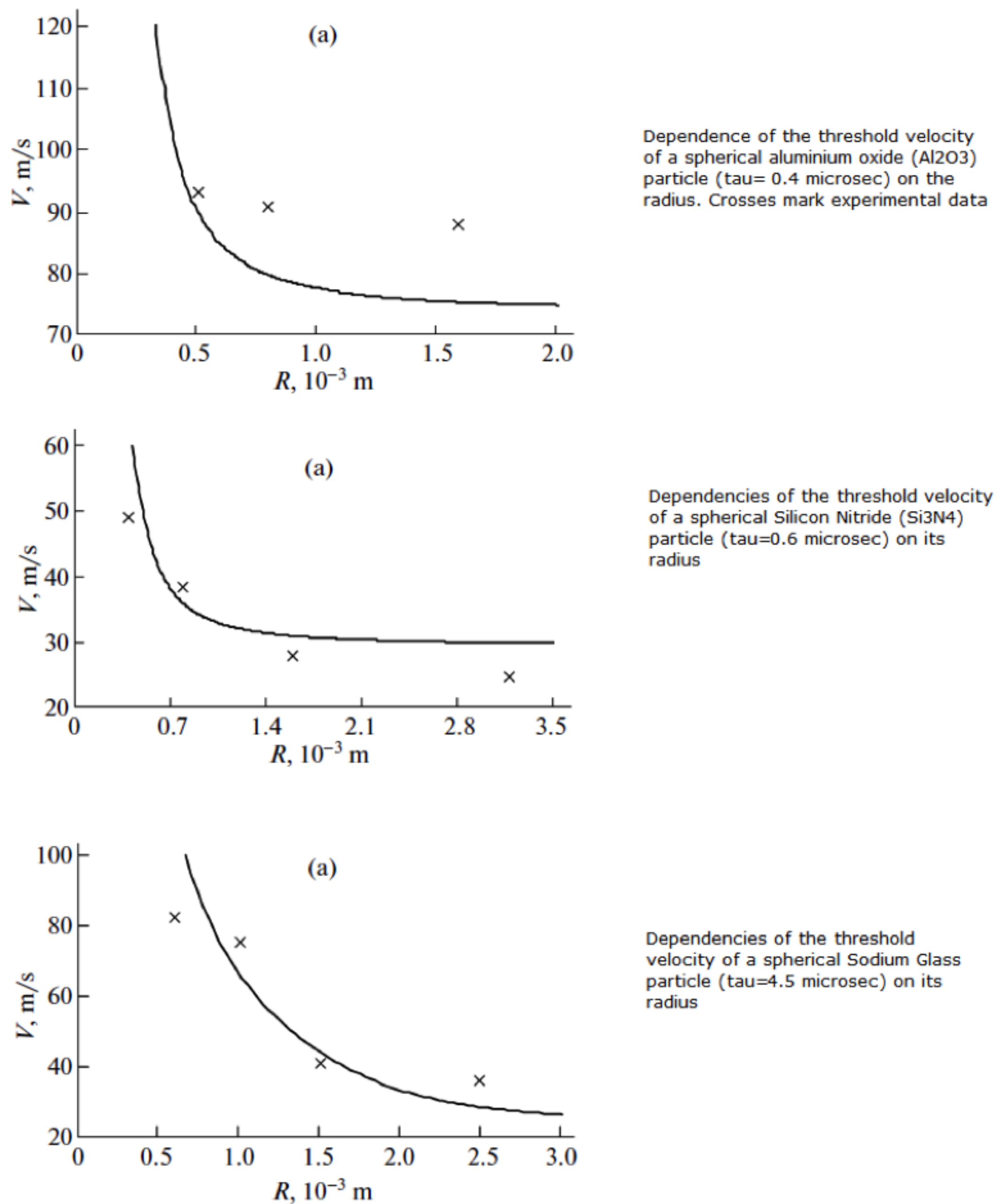


Figure 2.6: Experimental data gathered in [41] and interpolated plots where τ is the incubation time of fracture

Interactions (with surfaces, fluid or other particles)

When dealing with the movement of a group of particles inside a fluid there are basically two different ways to approach the problem. The first method is the Eulerian-Eulerian

approach and it is suitable for medium to large particle concentrations. In this kind of simulation both particle-particle interactions and the influence of the particles on the fluid phase are important. One of the limitations of this methodology is in its application when large particle size distributions need to be simulated. Additional continuity and momentum equations have to be implemented for each of the simulated particle sizes. Inclusion of these additional equations increases the complexity of the simulation considerably [42]. On the other hand, in the Eulerian-Lagrangian approach, the Eulerian continuum equations are solved for the fluid phase, while Newton's equations for motion are solved for the particulate phase in order to determine the trajectories of the particles [43]. There are three different possibilities when constructing the equations that will define the motion of the particles inside the fluid phase. These are outlined by Lopez et al in [43]:

- One way coupling: The influence that the particles exert on the fluid phase is neglected. This approach is suitable for low particle concentrations.
- Two way coupling: This methodology implies that the force the particles exert on the fluid is no longer neglected and a new term is incorporated into the fluid's equations to account for this.
- Four way coupling: In this case also particle-particle interactions are taken into account.

A third approach is the Drift-flux model or algebraic slip. In this methodology, the Eulerian equations are solved for a single fluid phase with variable density depending on the local particle concentration summed across all particle sizes. This makes the fluid density equivalent to the slurry density. The forces and distribution of the particles are obtained by solving a single scalar transport equation for the volume fraction of each particle size. It has an obvious advantage with respect to the computational time compared to the previous two approaches if the particle concentration is high enough and multiple particle sizes are present [42].

Finally, the Discrete Element Method (DEM) treats the particles individually and provides dynamic information on particle interactions with fluid, walls and other particles and is therefore a potential candidate for erosion studies. In this case the fluid flow

would be modeled as a continuous phase using CFD and the particles would be modeled as a discrete phase with the DEM method. This has been called the CFD-DEM approach. In DEM Newton's laws of motion are solved for each particle individually and, as an example, collisions between particles are modeled with the aid of a spring to represent the rebound and a damper to account for deformation upon impact.

Temperature

Although most studies are run under ambient temperature conditions, some authors have studied erosion under higher temperatures ([17], [44]). Two of these studies are briefly commented on in section 2.1.2. In these articles the whole system is subjected to high temperatures. However, if only the abrasive's temperature were to be increased in dry solid particle impingement, more pronounced temperature effects such as melting of the target upon impact should be expected. Changes in the hardness and properties of the abrasive itself along with changes in the material properties after several impacts due to heat transfer phenomena between the abrasive and the target material are also likely to occur. In a submerged jet, if the temperature of the abrasive were higher, most of it would be expected to dissipate before reaching the target, thus making the increase in temperature less noticeable in the target. However, if the heat transfer was high enough as a result of a high concentration of particles for example, it could have an effect on the fluid properties such as viscosity or even on the target material.

Presence of additives

Addition of certain additives to lubricating oils can reduce erosion-corrosion between different parts in contact [45]. In the case of solid particle erosion the use of additives can change the physical properties of the fluid along with its behaviour. This may potentially increase or decrease the wear rates or even change the location of the maximum wear rate. Ilmar and Priit [46] indicated that the content of water in sand-water mixtures can abruptly change the erosive behaviour of the slurry by increasing it in both metals and non-metallic materials. They also mentioned that often, the increase of the wear rate can not be explained solely by the additional effect of corrosion. They presented research studies in [46] confirming that adding a non-corrosive liquid such as

Kerosene will considerably increase the wear rate.

Electrical charge

Cathodic protection is a well known technique which consists of applying an external electric charge in order to reduce corrosion and eliminate its synergystic effect when combined with erosion. Application of an external electric charge can reduce corrosion to almost zero. One way of doing this is connecting the target material with a more active metal. This second metal provides the first one with a constant flow of electrons, thus acting as an anode and the target as the cathode. The target is then protected against corrosion while the anode is sacrificed. This is why this technique is called cathodic protection with sacrificial anode. The second method consists of using an inert anode and applying a direct current between the anode and the target material [47].

2.1.2 Characteristics of the surfaces involved

Physical properties

In [1] Meng and Ludema presented a table citing the variables contained in the 28 models that they separated for special study and applicable for erosion by solid particle impingement. Of these variables, the ones that represent physical properties of the target material are shown in table 2.5.

1	Density
2	Hardness
3	Flow stress
4	Young's modulus
5	Fracture toughness
6	Critical strain
7	Thermal conductivity
8	Melting temperature
9	Enthalpy of melting
10	Heat capacity

Table 2.5: Parameters selected in erosion wear models [1]

Analysing the 28 equations highlighted by Meng and Ludema in [1], material density is only included in a small number of formulae, its exponent ranging between the values of $\frac{1}{3}$ in the equation used by Jennings et al in [27] and 2 in the one by Lhymn and Wapner

in [48]. More complicated expressions for erosion involving the material's density can be found in [49–51]and [52].

Hardness of the target material has been considered largely in the literature and it presents itself as one of the most important factors when calculating erosion by solid particle impingement. Brittle and ductile materials are eroded in different ways as has been discussed in section 2.1. Truscott reviewed the existing literature on erosive wear in hydraulic machinery [26]. Metals, rubbers, plastics and ceramics are the four material types surveyed though not much was found on plastics or ceramics. Truscott found that erosive resistance of some synthetic rubbers, plastics and especially of some ceramics could exceed that of some metals such as some types of steel.

Jennings et al [27] proposed a model which included the melting temperature of the target surface as well as its enthalpy of melting. In his study, Jennings used an electron microscope and concluded that there were signs of melting on the target, adding to what other authors like Smeltzer et al [53] had pointed out before. By examining their samples they argued that several areas appeared to be melted or molten debris had been deposited on them. Sundararajan and Shewmon studied erosion at normal incidence in [52] and developed two different equations: one of them being a simplified form of the other, very complex, one. Both mathematical expressions include the melting temperature of the target material and its heat capacity. Also a number of other properties like the critical plastic strain, densities of both the target material and the eroding particles are included. As with many other erosion related theories the possibility of the particle's kinetic energy being converted into thermal energy, which in turn melts the target's surface, can not be neglected.

Stress level

Finnie investigated this effect in [13] by applying external bending moments to the samples and detected that erosion barely changed, thus concluding that surface stresses have very little effect in erosion by solid particle impingement.

Temperature

Tabakoff and Vittal carried out some tests on the Inconel alloy INCO 600, which was commonly used for turbomachinery blades, and discovered that the erosion rates at elevated temperatures were considerably higher than at ambient temperatures. The test rig used was able to reproduce target temperatures between ambient and 1093°C using air as the carrier fluid. Some of their results are shown in figure 2.7.

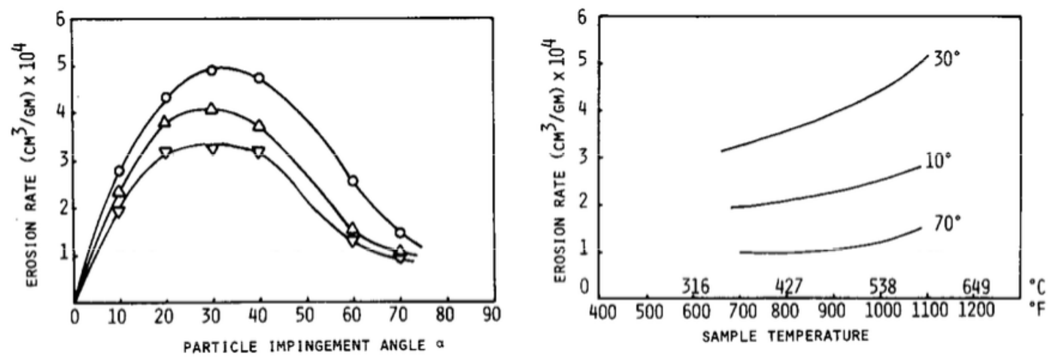


Figure 2.7: Erosion rate Vs. particle impingement angle for various sample temperatures (particle size, 451.5 μm; particle velocity, 183 $\frac{m}{s}$):
$\circ, 577°C; \triangle, 493°C; \nabla, 371°C$. (Left) and Erosion rate us. sample temperature for various impingement angles (particle size, 451.5 μm; particle velocity, 183 $\frac{m}{s}$ (Right) [44].

Wiederhorn and Hockey [17] studied how the velocity exponent changes with temperature for some ceramic materials. Their results are shown in figure 2.8.

Material	25° C	Temperature 500° C	1000° C
Magnesium oxide, polycrystalline	2.2	–	–
Soda-lime–silica glass	2.5 (0.12)*	3.5 (0.20)	–
Vitreous silica	2.9	3.0	–
Sapphire	2.3 (0.10)	2.4 (0.25)	3.3 (0.03)
Sintered aluminium oxide, 30 μm	2.3 (0.003)	2.8 (0.09)	2.7 (0.15)
Hot-pressed aluminium oxide, 3 to 4 μm	2.3 (0.03)	2.1 (0.04)	2.3 (0.11)
Silicon	2.9 (0.03)	3.8	3.4
Hot-pressed silicon carbide	1.8 (0.16)	–	–
Hot-pressed silicon nitride	2.1 (0.08)	2.5 (0.03)	2.4 (0.20)

*The numbers in parentheses give the standard error for the value of the velocity exponent, which was determined by a linear regression analysis of the mean wear values given in Table IB. For exponents that were determined from only two wear values, no standard error is given.

Figure 2.8: Velocity exponents for erosion data: normal incidence [17]

Presence of oxide (or other) coatings

Some metals under certain conditions, develop a film of metal oxide on their outer layer. This oxide is often attributed as having passivating characteristics. Passivity of a metal can be described as the property of a metal, susceptible to corrosion, experiencing a lower corrosion rate than expected for a certain period of time [33]. This ability, of generating an oxide coating that shields the target material from further corrosion, is well known for Aluminium which develops a layer of Alumina (Al_2O_3) when in contact with oxygen. Rajahram et al also encountered this effect in [33] when testing High-Chromium cast irons and stainless steel UNSS31603. However, the passivation layer can also break down through its partial or complete removal. L. L. Shreir, classified the different kinds of breakdown into three main types; electrochemical, chemical and mechanical breakdown [54]. In the present case of solid particle impingement mechanical breakdown of the passivation layer is the main type of breakdown considered. Regarding coatings, other than passivating layers, many have been developed in industry and academia. Applying a layer of coating to the surface affected by erosion would indeed reduce the potential damage on the target. Some pump manufacturers use special coatings for some of the pumps' parts in an attempt to mitigate the effects of erosion and corrosion. However, in the case of coatings, there is always the compromise between the cost of the application of the coating and the potential extension in the pump's life due to that coating.

Simultaneous occurrence of corrosion

When both corrosion and erosion occur at the same time, an additional wear rate is experienced by the target material. This additional wear is called synergy. It's effect can be calculated by subtracting the pure erosion rate and corrosion rates from the combined erosion-corrosion wear rate [33] and it is expressed as in equation 2.3. Where, T is the total wear rate, E and C are the pure erosion and corrosion wear rates respectively and S is the synergistic effect.

$$S = T - (E + C) \tag{2.3}$$

Li et al. made measurements of the influence of the corrosion component on the erosion of Aluminium by aqueous silica slurries in [55]. In most of the practical cases the transported slurries are also corrosive and, even though the corrosion component by itself might be very low, the synergistic effect of both erosion and corrosion usually yields much higher wear rates than the pure erosion ones. In their experiments, Li et al. concluded that, after 1 hour of water jet test with tap water, corrosion on the probe was below the measurable limit, and thus neglected. They also include the chemistry of the corrosion mechanism and concluded that the elimination of the work-hardened layer is not, after comparison of microhardness measurements, the main cause of acceleration of erosion by corrosion. According to them, corrosion leads to propagation of cracks, which accelerates the detachment of the flakes, thus inducing a synergistic effect when combined with erosion.

2.1.3 Carrier fluid

State of motion (laminar versus turbulent)

Sudip and Humphrey studied the influence of turbulence on a particle-laden fluid jet in [56]. In particular, they focused on how the turbulent diffusion affects the particle dispersion by varying the turbulence intensity in a particle-laden jet. They came to the conclusion that turbulence can have a significant effect on erosion by solid particle impingement, as figure 2.9 shows.

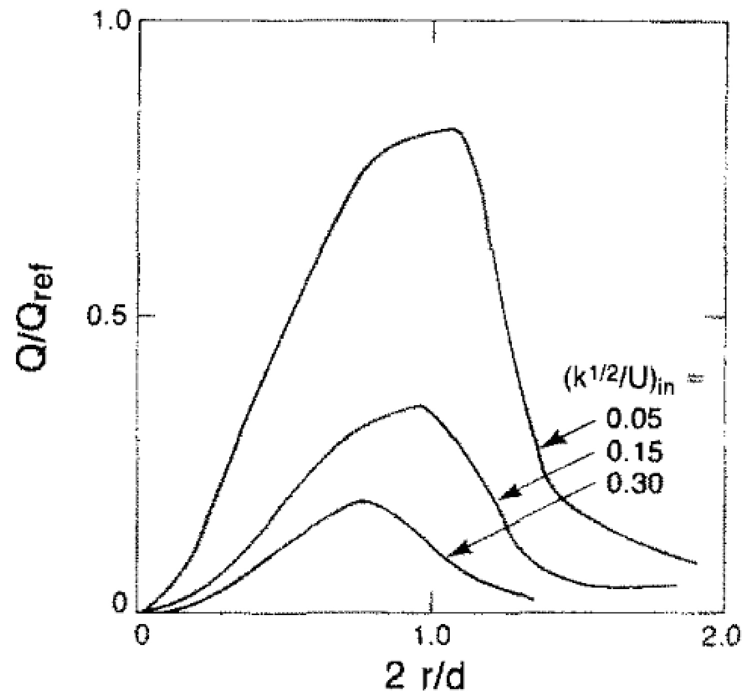


Figure 2.9: Effect of turbulence intensity on the wear of a ductile metal by 5 μm particles (h = 0.41)in an air jet with Re_j = 20000, $\frac{H}{d}$ = 12 and $\frac{\rho_p}{\rho_f}$ = 1709. Finnie's [12] model was used to calculate the erosion [56]

Even if random functions are incorporated to the injection of the particles at the inlet, all the impacts on the target would be fairly organised. In CFD, there are other methods which include terms of the turbulence that are used to add additional randomness to the trajectories of the particles through the computational domain as Lopez et al commented on in [24]. In the Open Source CFD software OpenFOAM, this function is called Stochastic Dispersion , while in Ansys Fluent it is named the Random Walk Model.

Velocity

The main way that the velocity of the fluid phase influences the particles entrained in it is computed through the drag force. The force balance on the particle is shown in equation 2.5. The drag force on spherical particles (which is particularly big in relation to the rest of the forces acting on the particles when dealing with liquids) takes the form of equation 2.6. There are different expressions that can be used for

the drag coefficient. For instance, the drag coefficient (C_D) in OpenFoam is obtained from equation 2.7, while in Fluent the formula developed by Morsi and Alexander in [57] and shown in equation 2.4 is used, where a_1, a_2 and a_3 are constants applicable to smooth and spherical particles given by the authors in [57]. Influence of the velocity of the fluid on erosion is often also studied through the Stokes number. Stokes number is usually calculated as shown in equation 2.8, where τ_p,2.9, is the characteristic time of the particle. U is the average exit velocity and D is the nozzle diameter. Spherical particles will have a higher characteristic time than angular particles such as aluminum oxide resulting in a higher stokes number [58]. Frosell et al reviewed the Stokes number for some articles in the erosion literature in [58] and stated that a high stokes number would imply that the particles have minimal entrainment, a fact which would cause them to impact the surface with almost the same trajectory as the one they had when they left the nozzle.

$$C_D = a_1 + \frac{a_2}{Re} + \frac{a_3}{(Re)^2} \tag{2.4}$$

$$F_p = m_p \frac{du_p}{dt} = F_D \tag{2.5}$$

$$F_D = m_p \frac{18\mu}{\rho_p d_p^2} \frac{C_D Re_p}{24} (u - u_p) \tag{2.6}$$

$$C_D = \begin{cases} \frac{24}{Re_p} & : Re_p < 1 \\ \frac{24}{Re_p}(1 + 0.15 Re_p^{0.687}) & : 1 \leq Re_p \leq 1000 \\ 0.44 & : Re_p > 1000 \end{cases} \tag{2.7}$$

$$S_t = \frac{\tau_p U}{D} \tag{2.8}$$

$$\tau_p = \frac{\rho_p d_p^2}{18\mu_f} \tag{2.9}$$

Equation 2.9 shows how the characteristic time is calculated. In this equation ρ_p corresponds to the particle density, d_p is the particle diameter and μ_f is the fluid

viscosity. Finally Re_p is the particle Reynolds number.

Temperature

Most authors that have carried out erosion studies use ambient temperature in their tests. However, a rise in the temperature of the carrier fluid would change its viscosity and with it, the trajectories of the particles, since a change in the viscosity also affects the drag coefficient. Thus, temperature changes in the carrier fluid could potentially change the shape of the wear scar. Sudip and Humphrey concluded in [56] that some of the erosion parameters vary strongly with temperature. There are some additional studies such as the ones by Tabakoff and Vittal's in [44] or by Wiederhorn et al in [17] which deal with erosion at higher temperatures. These have been briefly commented in 2.1.2.

Chemical composition and physical properties

Chemical composition of the carrier fluid can influence the target's surface by means of corrosion, because some of the chemicals entrained in the fluid may affect the target material. For tap water, Li et al [55] carried out some experimental work commented on 2.1.2 and concluded that the corrosive effect of it after a test of one hour is negligible. Viscosity of the fluid can also change with its chemical composition. Clarke studied the effect of fluid viscosity on the erosion rate of metals [59]. An example of the effect of viscosity on the erosion rate of steel can be observed in figure 2.10 where a significant reduction in erosion rate is obtained with increasing viscosity.

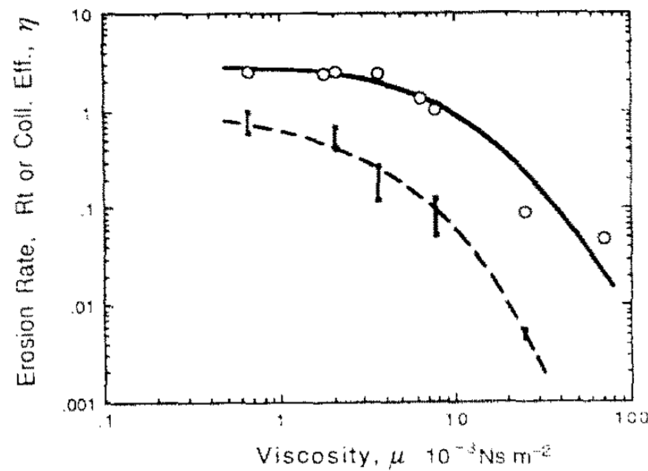


Figure 2.10: Variation in erosion rate R_t ($x10^6 \frac{g}{m^2*min}$) and collision efficiency of cylindrical steel targets 4.76 mm in diameter as a function of viscosity for 3 wt.% 75-106 μm Al_2O_3 suspensions in a slurry pot tester at 18.7 $\frac{m}{s}$ [59]

Change in shape of the target material caused by erosion

The effect of the change in shape of the target material has not yet been studied deeply. However the need for an in depth study of how the fluid flow and erosion are affected by the deformation has been acknowledged repeatedly ([9, 10, 14, 60]). Nguyen et al [60] studied it to some extent in [60]. However, their study seems to be limited to simple targets because it involved 3D scanning of the eroded material, which in most cases is not possible, whether because it is in the field or because the geometry is too complicated to scan it completely.

2.2 Erosion prediction with CFD

Since Finnie's wear model [12] a very large number of papers and wear models have been published in numerous journals and books. Lately, in 2009, a new methodology was published which allows more accurate predictions of material wear rates due to erosion [9]. This method is known as the Wear Map method and it consists of two main stages. In the first stage Jet Impingement Tests are performed in order to obtain the wear rate on a standard disc. The wear map will be drawn for the particular material used for the test coupon. The 90 degrees angle of impingement is chosen

because of its ability to reproduce a wide range of impingement conditions. A CFD model of the same experiment is then used to obtain the local impingement conditions at each point of the surface and combining both tests the wear map can be obtained for the particular material under test. With the wear map it will yield the wear rate as a function of particle impact velocity and particle impact angle [9]. An example of such a map can be found in figure 2.11 .

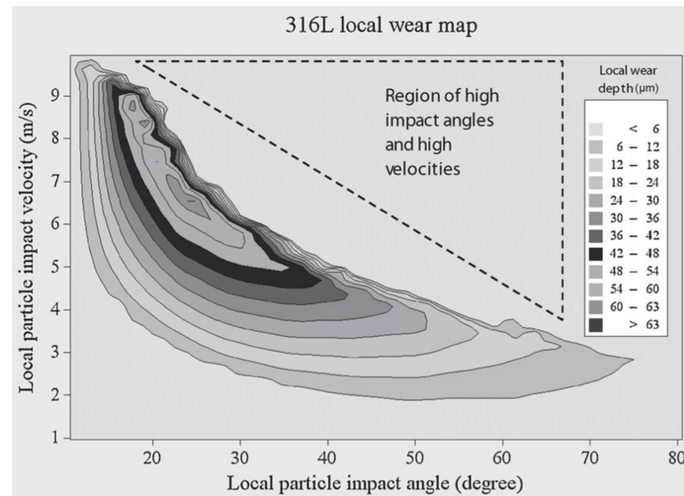


Figure 2.11: Example of Wear Map obtained by Gnanavelu et al in [9]

Nguyen et al tried a novel technique for analysing the flow conditions when erosion has progressed enough so that these have changed. In their article [60] they set up a Jet Impingement Test and 3D scan the target after 30 seconds, 5, 15 and 30 minutes. After that, the 3D scanned target was incorporated into the CFD, substituting the undeformed one and, after 30 minutes they were able to capture the appearance of a new stagnation point around the scar, which is shown in figure 2.12.

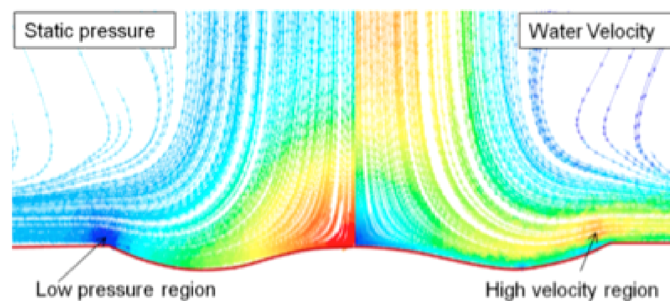


Figure 2.12: CFD of target after 30 mins erosion showing appearance of new stagnation point [60]

Rajahram et al pointed out in [33] how complex erosion-corrosion processes are and stated that, ideally, testing on the materials should be done in the field in order to be able to obtain accurate data on the material's performance under such conditions. Shipway and Hutchings also mentioned in [19] the importance of the erodent particles used in the experiments, having similar properties to those in the real application, in order to have a test which is representative of the reality.

2.3 Aims and objectives

In order to acquire a better understanding of the erosion process, as it has been commented before, a deforming mesh algorithm is of the utmost importance. This algorithm will enable knowing how erosion changes the shape of the domain and how the fluid flow changes and it's effect on the erosive process. To achieve this and, since the work here will be based in the Jet Impingement Test, a methodology needs to be selected amongst the existing ones. Once the method is selected, it will be implemented in OpenFOAM. Given the choice between Ansys Fluent commercial software and OpenFOAM's Open Source CFD softwares, the latter was chosen. The choice of OpenFOAM was due to its ease of manipulation, having no user interface and having the source code completely available. Availability of the code makes implementing new models, creating a deforming mesh algorithm or even new solvers easier. Secondly, a statistical study of the jet impingement will also be carried out since it is very important to be able to optimise the time needed to get an accurate wear scar on the target that can be used for deformation. Finally, once the deforming mesh algorithm and the statistical study are ready, the JIT will be studied experimentally with Particle Image Velocimetry. This technique will allow us to visualisation of the fluid flow and the impinging particles in the vicinity and on the wear scar in order to compare with the deformed geometry and fluid flow in CFD.

Chapter 3

Computational Fluid Dynamics in OpenFOAM

3.1 CFD in OpenFOAM

OpenFOAM (Open source Field Operation and Manipulation) is a free Open Source CFD package developed in C++ with pre and post-processing tools for the solution of fluid flows involving chemical reactions, turbulence and heat transfer, acoustics, solid mechanics or electromagnetics [61]. One of the main reasons that makes OpenFOAM suitable for the purpose of this work is that the code is completely accessible, enabling the user to freely modify the CFD code, which makes programming of new applications, solvers and utilities much easier. OpenFOAM consists of a series of C++ libraries used to create executables. These executables are called applications and these can be subdivided into two categories: solvers and utilities. Each solver is designed to be used for a specific problem in continuum mechanics while the utilities are used to perform tasks involving data manipulation [62]. A general overview of OpenFOAM's structure can be seen in figure 3.1.

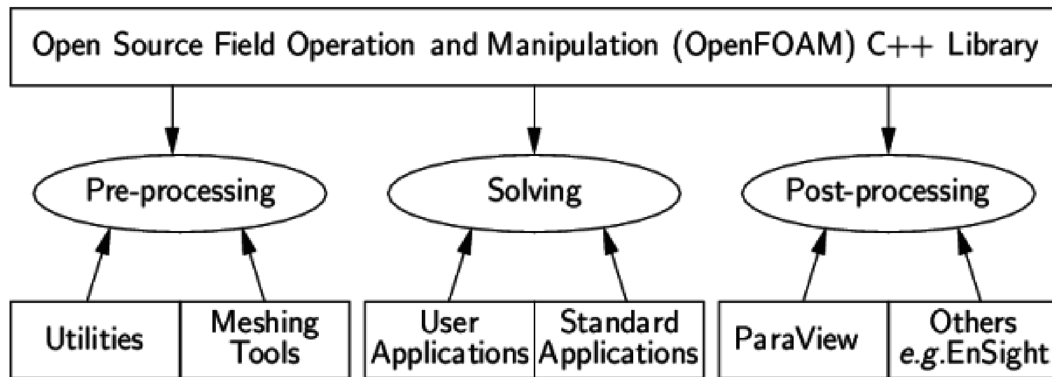


Figure 3.1: Overview of OpenFOAM structure [62]

3.2 Verification of OpenFOAM

3.2.1 Introduction

The initial aim of this section was to compare OpenFoam and Ansys Fluent in order to verify OpenFoam's Lagrangian Library and erosion modelling capabilities. However, it was found that previous versions of Fluent have been providing wrong results for the discrete phase and the differences with the latest version (Ansys Fluent 15) are shown [24]. A Submerged Jet Impingement Test is an effective method for studying erosion created by solid particles entrained in a liquid. When considering low particle concentrations Lagrangian modelling of the particulate phase is a reasonable approach. Proper linkage between OpenFOAM 2.2.x's Lagrangian library and the solver pimpleFoam for incompressible transient flows allows two-phase simulations to be undertaken for comparison with Ansys Fluent.

In this section a thorough comparison of a jet impingement test simulated with Ansys Fluent (Fluent) 15.0 and OpenFOAM 2.2.x. is presented in order to verify that OpenFoam's combination of libraries is able to reproduce correctly the particle behavior in the jet impingement test, focusing on the correlation between particle variables and the radial distance from the centre of the target. Implementation of several erosion models and testing all of them simultaneously is of great interest and OpenFoam proves itself, after comparison, as an asset for erosion modeling. Two different configurations of the JIT were set up and compared; those implemented by Hattori et al. [63] and Gnanavelu

et al. [9, 14]. However, when the same simulation corresponding to the 5 mm Jet Impingement Test was set up in Fluent 14 and OpenFoam 2.2.x, significant differences in both particle velocities and angles were found. OpenFoam's code was thoroughly checked and it's validity confirmed. As the latest Fluent version (15) was available it was verified that, due to a programming error that has been corrected in that latest update, previous versions of the commercial software had been providing wrong values for the particles. These results are discussed here and some of the articles affected by these miscalculation are reviewed. The CFD results confirm that the maximum and minimum velocity magnitude of the impinging particles does not experience large variations when the distance between nozzle exit and target is increased to 25 mm. However, the effect is more visible in its profile variation along the target's radius.

3.2.2 The Euler-Lagrange Approach

When dealing with particles entrained in a fluid flow there are a number of possibilities available for the numerical treatment of the dispersed phase. The number of operations needed for the calculation of the particle trajectories may become numerically unmanageable if high concentrations are taken into account. In these cases, treatment of the dispersed phase as a second Eulerian phase is common practice. As the concentration diminishes solids may be handled in a Lagrangian way, defining the forces acting on the discrete phase and obtaining velocities and positions by means of integration of Newton's equations (3.1, 3.2).

$$m_p \frac{d\vec{u_p}}{dt} = \vec{F_p} \tag{3.1}$$

$$\frac{d\vec{x_p}}{dt} = \vec{u_p} \tag{3.2}$$

Where m_p and $\vec{u_p}$ are the mass and velocity of the particle, $\vec{F_p}$ is the term corresponding to the forces acting on the particle and $\vec{x_p}$ is the position of the particle. The Lagrangian approach may be further classified by the type of coupling between particles and fluid:

- One-way coupling: If the concentration of particles is low enough, as in the case

discussed here, the influence that the particles exert on the fluid phase may is neglected.

- Two-way coupling: In this case the force exerted by the particles on the fluid is no longer neglected and a new term should be included into the fluid's equations of motion in order to account for this.
- Four-way coupling: In this case also particle-particle interactions are taken into account.

Furthermore, the discrete phase can be modeled as computational parcels, which are groups of particles with average quantities for the main variables, or single particles. In this thesis one-way coupling of single particles is selected in both Fluent and OpenFoam due to the very low concentration of solids. This is in the range of 1% by volume and the particles are only affected by the drag force, as other forces like gravity or pressure gradient are considered to be negligible. In this case, the momentum transfer from the particles to the fluid phase can be neglected. This was confirmed by computing the momentum transfer in two-way coupled simulations with the same amount of solids.

Lagrangian and Eulerian phases coupling in OpenFoam

Fluent gives the user the possibility of adding a cloud of particles and tracking their movement at any given point. However, in order to do the same in OpenFoam, some programming was required because of the non-existence of an Eulerian-Lagrangian solver amongst the default solvers in version 2.2.x. To do this, a linkage between an incompressible transient solver and the Lagrangian intermediate library had to be implemented. The incompressible solver chosen was pimpleFoam which, as its name states, has the merged PISO-SIMPLE algorithm implementation. The linkage was established through the inclusion in the top level solver of the header file responsible for the definitions of the particulate phase and a call to the function that uses the fluid flow variables to calculate the forces acting on the solids for integration to obtain their velocities and positions. All the properties of the particles, as well as the templates used to gather the necessary variables at impingement, are defined through a dictionary which is read when the simulation starts. The full linking process between the

intermediate library and the incompressible solver is detailed in Appendix D along with examples of preprocessing, running and postprocessing a simple case of a pipe bend with a squared section with its geometry and mesh defined with blockMesh. Newer versions of OpenFOAM starting with 2.3.x include an Eulerian-Lagrangian solver with the same abilities of the one developed in the tutorial but with improved algorithm efficiency when handling larger cases.

3.2.3 Simulation parameters

Two different configurations of the normal jet impingement test were implemented at distances of 5 mm and 25 mm. Figure 3.2 shows the location of the boundary conditions. Schematics of both configurations are shown in figures 3.3 and 3.4. The velocity magnitude at the inlet of both was chosen to be uniform with a value of 10 $\frac{m}{s}$. For each of the cases one symmetry plane was used so only half of the geometry was simulated and the mesh used was the same for OpenFoam and Fluent. Geometry and mesh sensitivity were obtained by Dr. William Nicholls, as part of an investigation for the Weir Group.

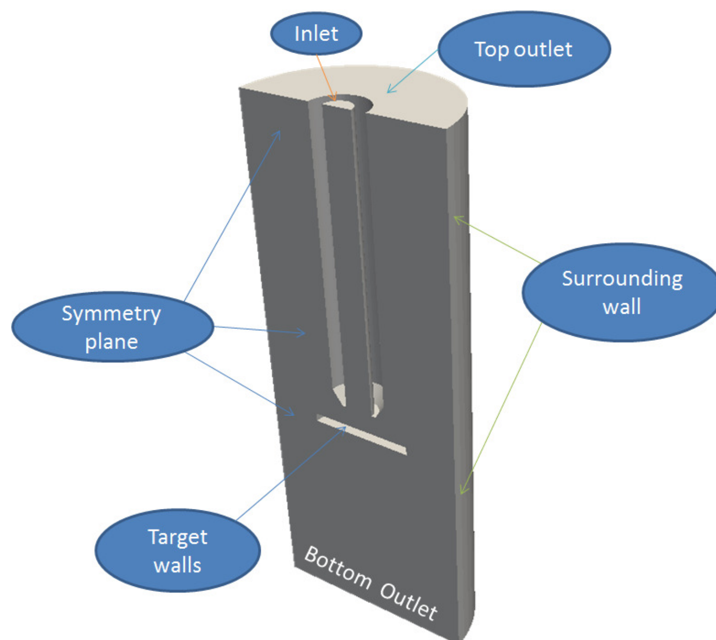


Figure 3.2: 3D geometry used showing the location of the boundary conditions

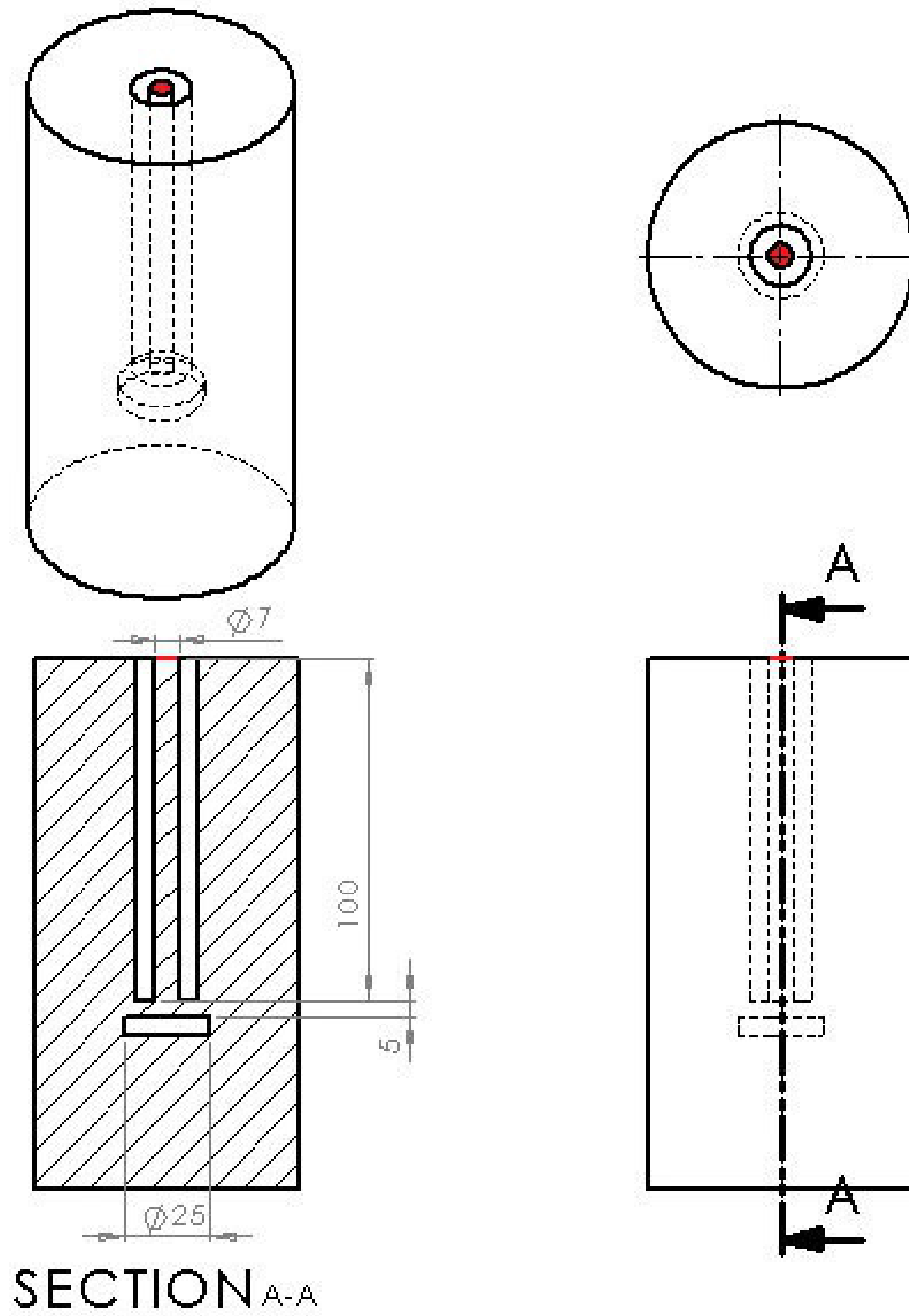


Figure 3.3: 5mm JIT domain with the inlet boundary highlighted in red and dimensions in mm

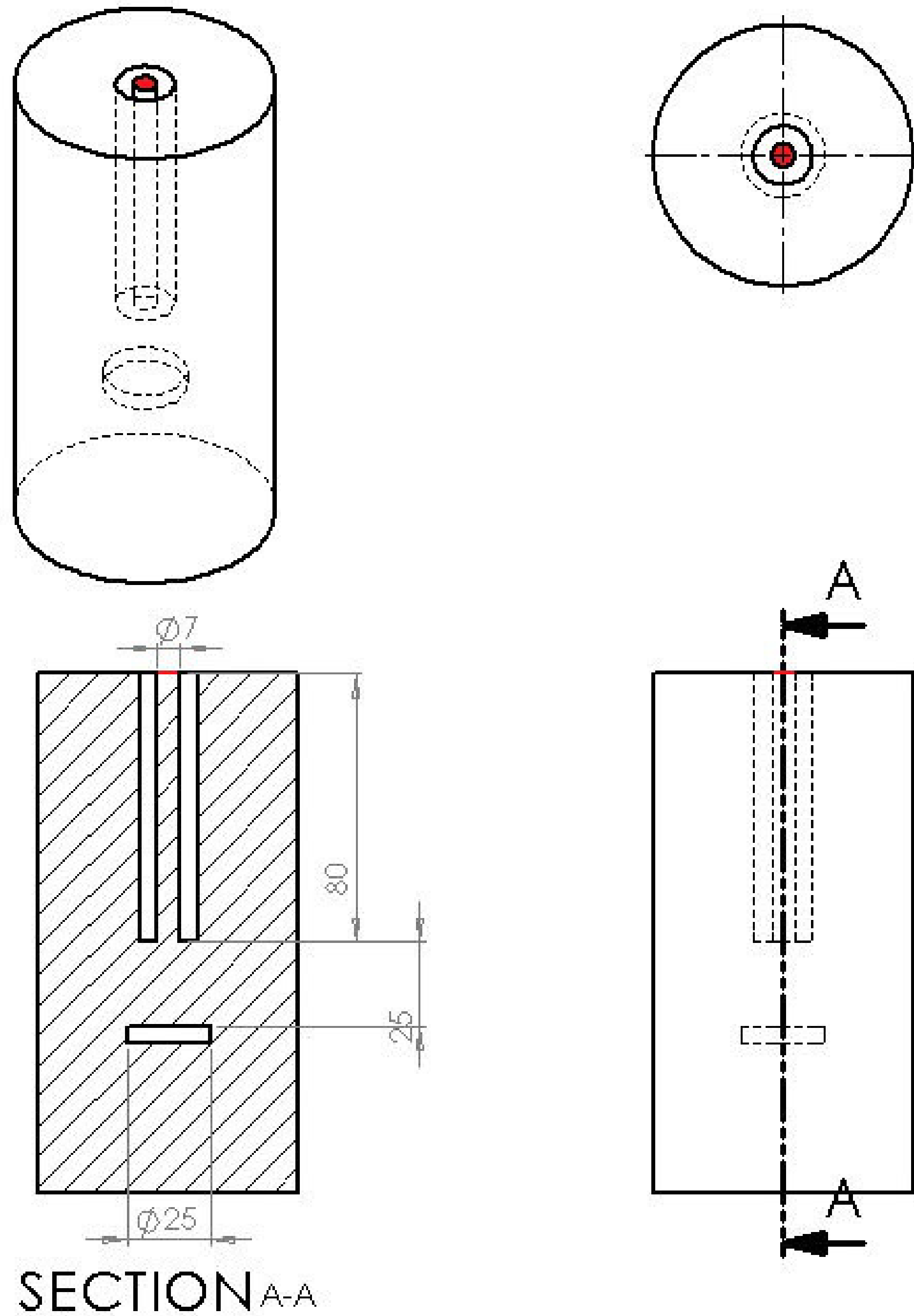


Figure 3.4: 25 mm JIT domain with the inlet boundary highlighted in red and dimensions in mm

Once the steady-state convergence was reached for the continuous phase in both Fluent and OpenFoam, a uniform distribution of 250 μm spherical particles were incor-

porated into the flow and their impacts against the target were individually monitored. The domains shown in Figures 3.3 and 3.4 were discretized into 984960 and 670349 cells respectively, paying special attention to the y^+ around the target which was considered adequate for the K-Epsilon turbulence model used. Two different configurations of the normal jet impingement test were implemented at distances from the pipe outlet and to the target of 5 mm and 25 mm.

3.2.4 Eulerian phase steady-state

Convergence criteria were satisfied for the fluid phase when the residuals fell below 10^{-4} and this was achieved by both packages in a similar number of iterations for both the 5 mm and the 25 mm cases. The SIMPLE algorithm was chosen for the pressure/velocity coupling and the same set of discretization schemes, solvers and boundary conditions were used in both simulations. The inlet was also situated sufficiently far upstream to allow the fluid flow to fully develop inside the pipe before it's outlet and before impinging on the target. As shown in Figures 3.5 and 3.6, minor differences can be observed in the steady state flow fields from both CFD packages. The largest discrepancies between solutions are located at the corners of the jet as shown in figures 3.7 and 3.8, which are contour plots of the absolute difference between the results of each package. In the case of the 5mm distance the highest value of the relative difference is 6%, affecting a very small region of the domain. In the 25mm case the maximum relative difference is 9.7% with a similar region affected. These discrepancies may be attributed to some small differences in the calculation algorithms. The differences between velocities in the region of the domain where the impact velocities and angles of the particles at the moment they reach the test target are considered, and which is most significant for this study, are minor and the fields can be considered to be identical.

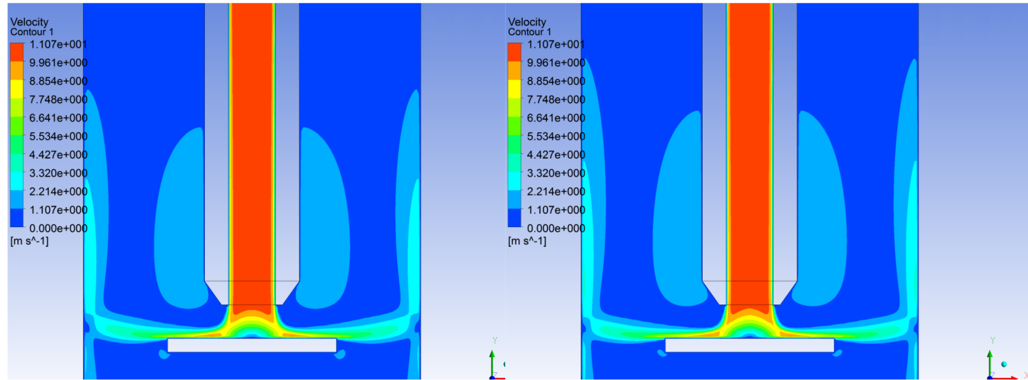


Figure 3.5: 5 mm separation Steady state velocity contours comparison. OpenFOAM(right) and Ansys Fluent

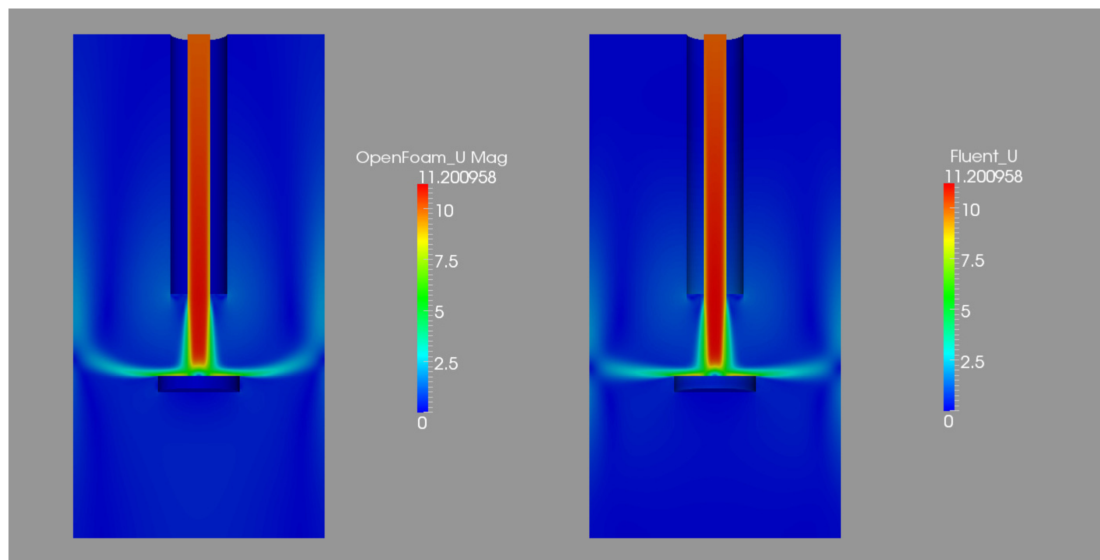


Figure 3.6: 25 mm separation Steady state velocity contours comparison. Units in m/s

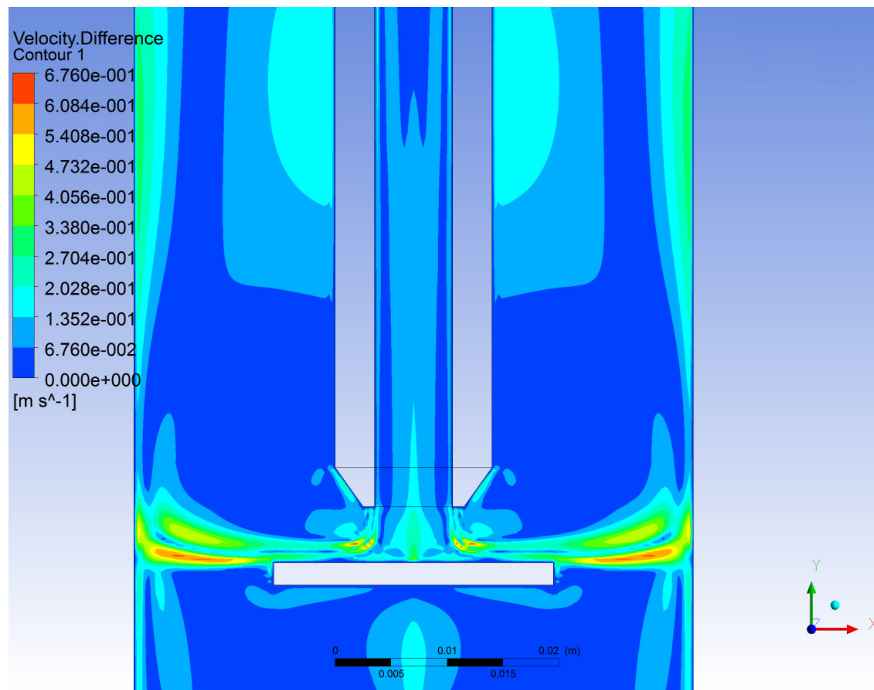


Figure 3.7: Contours of the absolute differences for the 5 mm separation case

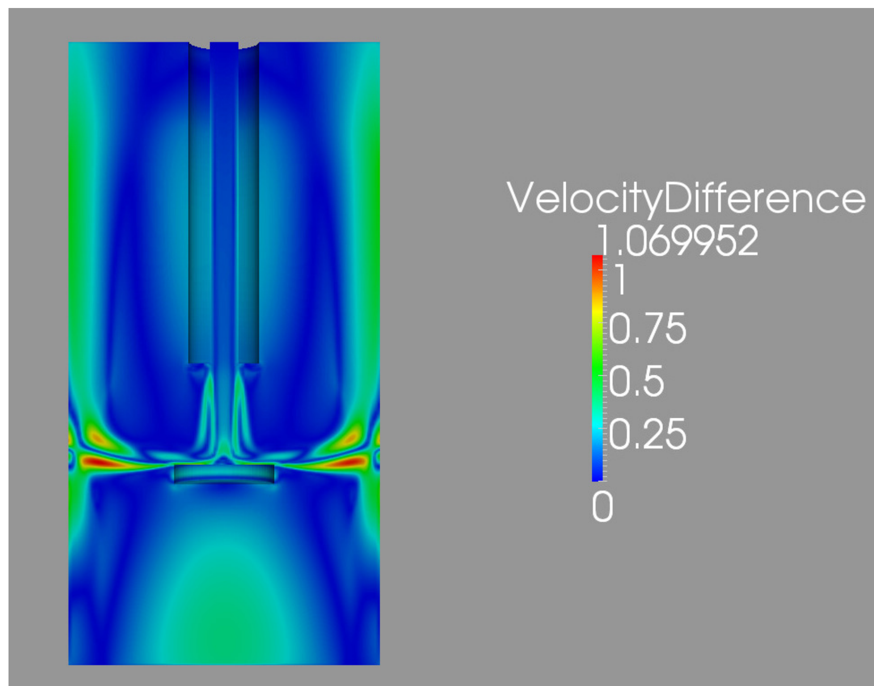


Figure 3.8: Contours of the absolute differences for the 25 mm separation case. Units in m/s

3.2.5 Discrete phase modeling

Once the steady-state values for the main flow field variables were obtained, these were set as the initial conditions for the transient simulations. In fact, the difference between a transient simulation with one or two iterations for the Eulerian phase and another one in which no iterations were considered yielded the same values for the variables of the particles. Particle tracking inside the fluid was carried out in a similar way in both packages. The user needs first to define which are the most relevant forces influencing the particle. In Fluent, the drag force was chosen as the sole force acting on the particles and the same arrangement was made in OpenFoam. The rest of the forces, including gravity, were ignored in this simulation due to their negligible influence on the discrete phase in comparison with the drag force. The force balance according to equation 3.1 is shown in equation 2.5.

The drag force on spherical particles takes the form of equation 2.6 and the drag coefficient in OpenFoam is obtained from equation 2.7, while in Fluent, the formula developed by Morsi and Alexander in [57] is employed and is shown in equation 2.4. The drag formula in [57] was also implemented and tested in OpenFoam and found to have no differences with respect to the impingement conditions.

Particle Injection

Differences arose when taking into consideration the injection models available in both software packages. In order to inject the same mass of particles, surface injection was chosen in OpenFoam and single injections in Fluent. One of the main differences regarding the injection models is that, even though both softwares use the name *surface* injection, they differ considerably from each other. OpenFoam's surface injection calculates the number of particles to be injected per time-step and, using a random function, chooses different locations at the selected surface for the injections. In Fluent however, when surface injection is selected, the number of particles injected per time-step will be equal to the number of faces on the selected surface. For this reason two single injections are chosen in two different locations at the inlet so that the number of particles released matches that in OpenFoam. If surface injection had been used in Fluent the number of particles would have been much greater. Despite OpenFoam's randomization

of the particle injection, if stochastic dispersion is not included, impact locations on the target will lack randomness. Fluent faces the same lack of randomness problem given that all the particles are released from the same point. In this case the so called discrete random walk model was selected in Fluent. In the case of OpenFoam, the stochastic dispersion is included through the *kinematicCloudProperties* dictionary. The particles chosen were spheres with a uniform radius distribution of 250 μm, density of 2206 $\frac{kg}{m^3}$ in the 5 mm case and 2300 $\frac{kg}{m^3}$ in the 25 mm case together with an initial velocity equal to the fluid's inlet velocity (10 $\frac{m}{s}$). A thorough study of the particle trajectories revealed that modification of the particle inlet velocity (setting it to 0 $\frac{m}{s}$ for instance) was not perceived downstream due to the small distance the particles travelled to accelerate close to the fluid's velocity and the comparably long distance to the target's surface.

Particle variables gathering

In Fluent a User Defined Function (UDF) was created which gathered the particle velocity, particle angle of impingement and impact location in terms of radius along the target. In addition to this the UDF also terminates the particle trajectory once they impact the target so that no second impacts of the same particle were considered. The same condition was met in OpenFoam. However, in this case, this was implemented via the kinematicCloudProperties dictionary by defining an *escape* type for the target's boundary. OpenFoam's solution for the output of particle variables was achieved through a cloudFunctionObject called patchPostProcessing. This template allows printing of the desired particle variables into a text file so that radius, velocity and angle of impingement were successfully gathered, calculated and stored as soon as the particles struck the surface.

Transient simulation features

Some of the features corresponding to the transient simulation are shown in Table 3.1:

Variable	Units	Value
Time-step	s	1.8759e-05
Number of time-steps	-	53308
Particle material	-	Carbon
Particle diameter distribution	-	Uniform
Particle diameter value	m	2.5e-04
Coupling between phases	-	One-way
Forces	-	Drag
Drag coefficient	-	Sphere drag
Particle density	$\frac{kg}{m^3}$	2206

Table 3.1: Transient simulation features

When considering a volume fraction of the solids of 1%, the yielded mass flow rate of solid particles was as low as 0.000962 $\frac{kg}{s}$ which, in turn, yielded a rate of 53308 particles per second. This allowed setting up the time-step for the transient simulation as $1.8759 * 10^{-5}$ seconds. During the simulation in OpenFOAM the Courant number was monitored to verify that the particles did not travel too fast through the mesh cells and their trajectory was accurately calculated.

3.2.6 Impingement conditions

Once all the impingement variables of the more than 50000 particles were gathered, they were post-processed in order to obtain their average compared to the distance from the centre of the circular target. In order to do this the 25 mm diameter circle was divided into smaller concentric regions separated by 1 mm from each other as shown in figure 3.9. The impacts located within each of these regions were then averaged and a mean value for both the angle and the velocity was obtained for each of the software packages and compared with each other. A surprising fact was that, for the same parameters, different versions of Ansys Fluent (namely 14.0 and 15.0) yielded very different particle velocity profiles. This may indicate that a programming irregularity was present in previous versions but has been corrected in the latest one, thus affecting a wide range

of publications. This would have been impossible to detect unless a comparison with other software packages like the present one had been carried out. Velocities presented here correspond to Fluent 15.0, while in Gnanavelu et al. [9, 14], a previous version of Fluent (dating from 2011) was used in which the velocity profiles deviated from their correct values due to the irregularity. However, this should not affect the methodology but only the results concerning those simulations and the graphs obtained from them. In fact, the use of data from the corrected version of Fluent should yield results that fit much better to the experimental data than the existing ones. Figures 3.10 and 3.11 show the differences in the particle velocity and angle of impingement profiles between two cases which were exactly the same but solved by the two versions of Fluent when the separation between nozzle outlet and target was 5 mm. These results can be confirmed by comparing them to the profiles obtained by Gnanavelu et al in [9] with Fluent and those obtained by Hattori et al in [63] with the *Star-CD* commercial code.

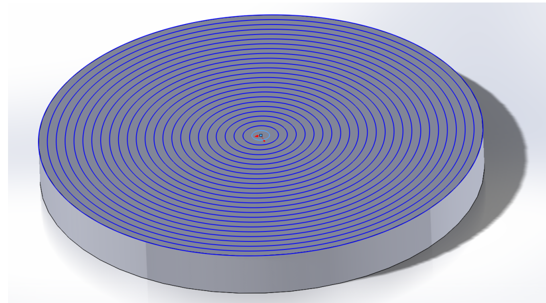


Figure 3.9: 25 mm diameter cillindrical target subdivided into 1mm regions

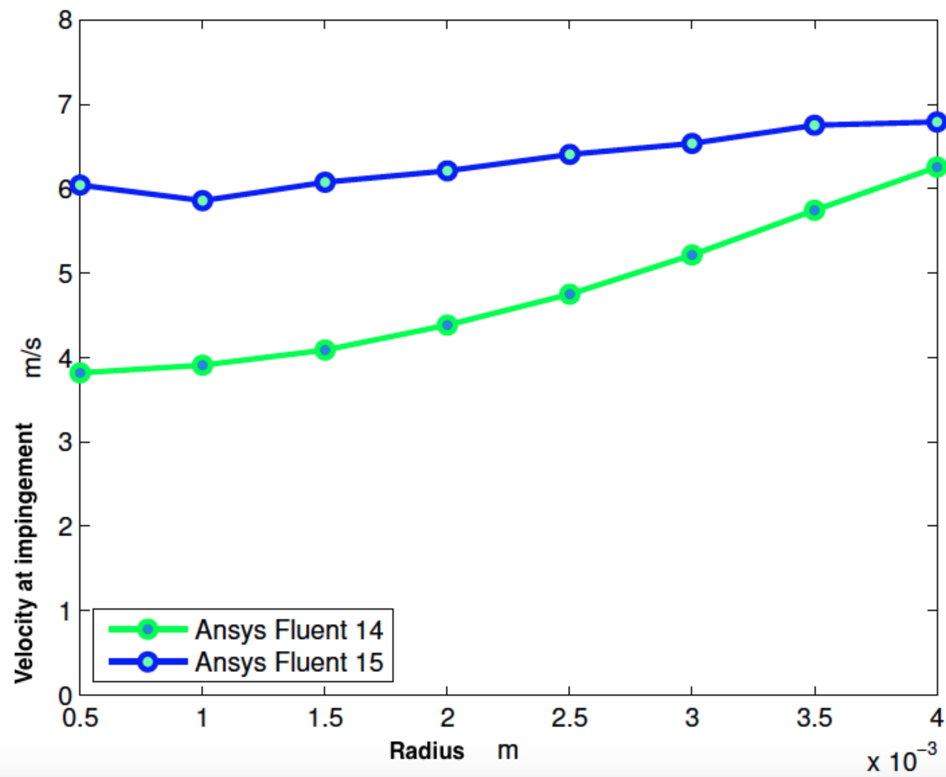


Figure 3.10: Average particle velocities ($\frac{m}{s}$) at impingement versus distance from the centre of the target (m) in the two Fluent versions

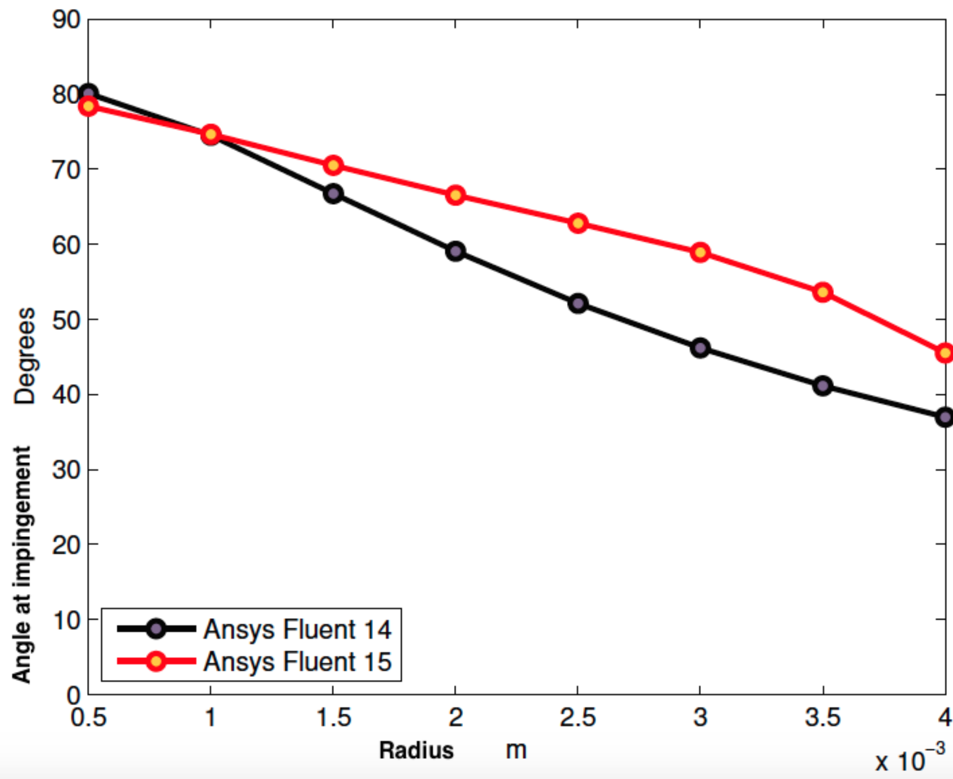


Figure 3.11: Average impact angles at impingement (*degrees*) versus distance from the centre of the target (m) in the two Fluent versions

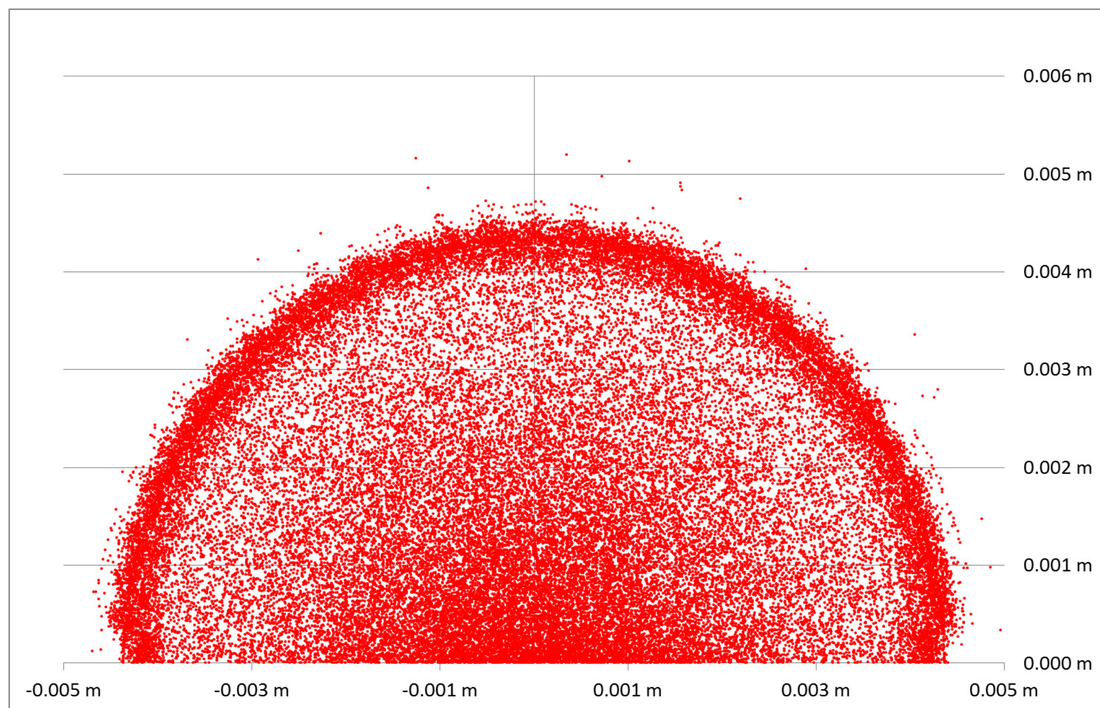


Figure 3.12: Impact locations for the 5mm nozzle distance case obtained with OpenFoam on the central part of the target

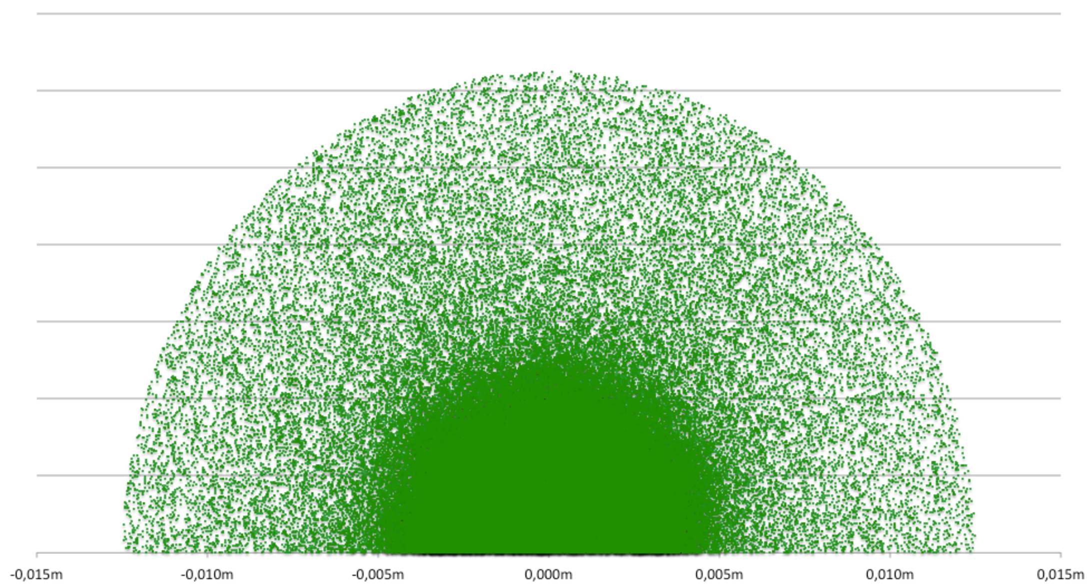


Figure 3.13: Impact locations for the 25mm nozzle distance case obtained with OpenFoam on the whole target

Figures 3.12 and 3.13 illustrate the location of all the particle impacts. Most of these were located within the 0-4 mm region of the target when the separation between

the nozzle and the target was 5 mm. As the number of impacts in the outer region was not high enough, values of the average in those regions were not considered due to the low density of these in comparison with the impact density in the inner (0-4 mm) region.

Velocity at impingement versus radius

As shown in Figures 3.14 and 3.15, minor differences can be observed between the profiles of the particle velocity along the radius for OpenFoam and Fluent 15, with a mean relative difference of 2%, which represents more than a 26% improvement if compared to Fluent 14. It is worth noting that, in the 5mm case, impingement velocities at the centre of the target have lower values, values being higher the further away from the centre. When the separation between nozzle and target is bigger, the contrary is true, the velocities at the centre being slightly higher than further away from it.

This is probably due to the disparity in the accelerations that both fluid and particles experience in the two different gaps left between the target and the nozzle exit.

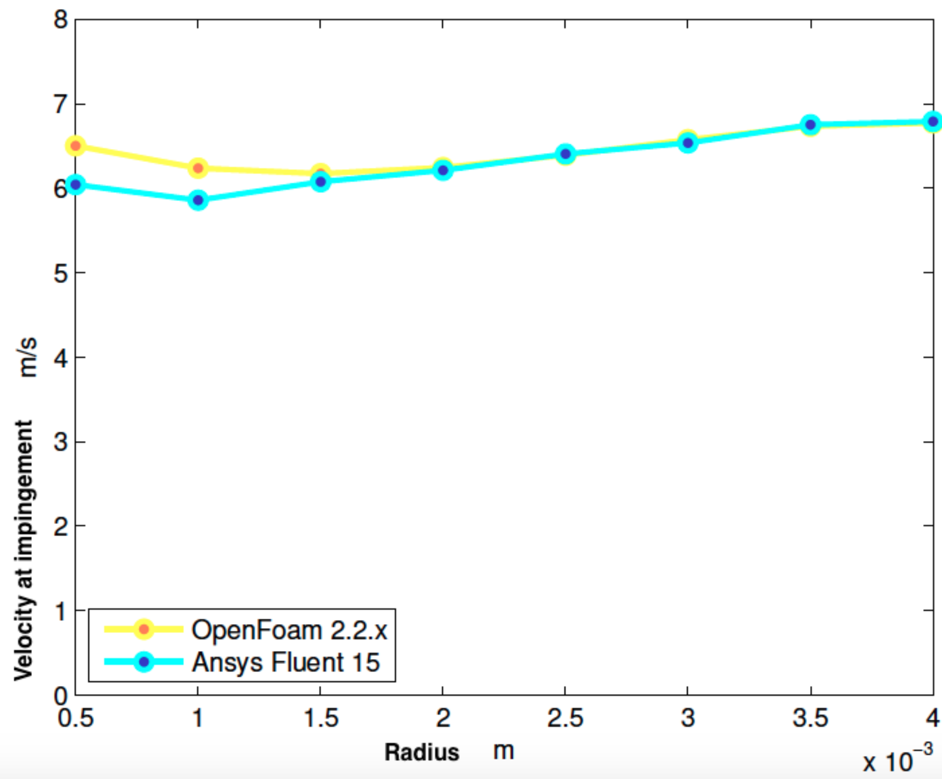


Figure 3.14: Fluent 15.0 Vs OpenFOAM 2.2.x particle velocities at impingement ($\frac{m}{s}$) for the 5 mm distance case

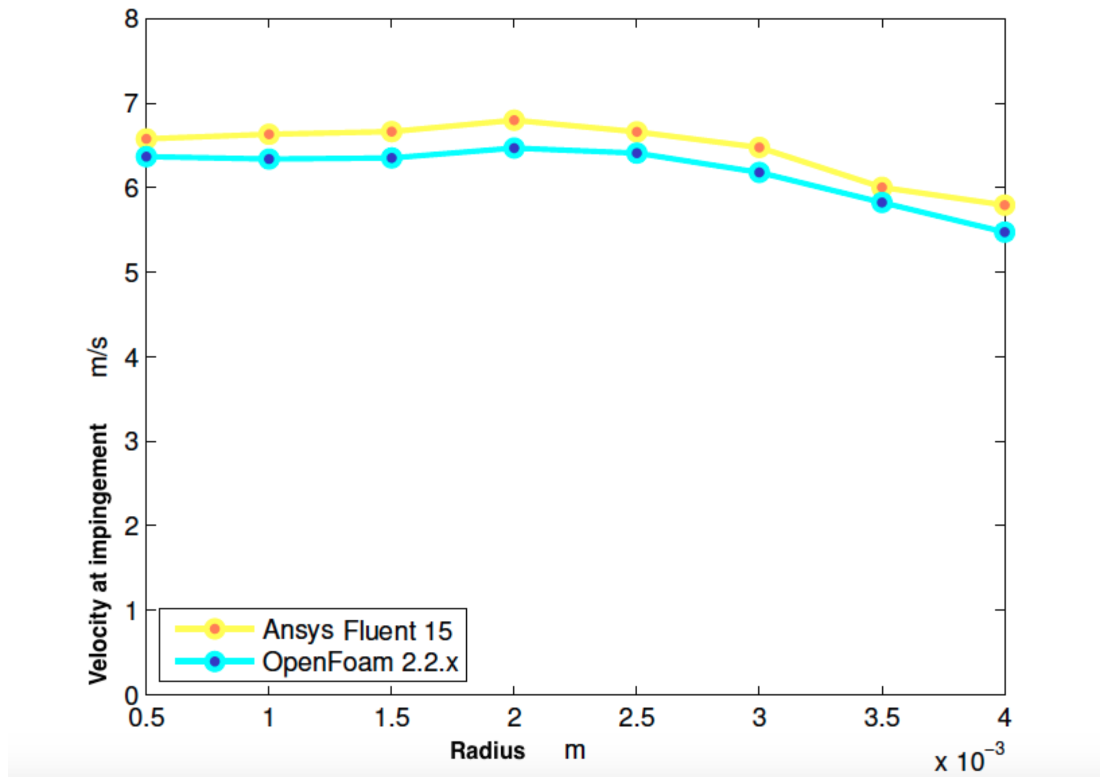


Figure 3.15: Fluent 15.0 Vs OpenFOAM 2.2.x particle velocities at impingement ($\frac{m}{s}$) for the 25 mm distance case

Angle of impingement versus radius

Figures 3.16 and 3.17 represent the trends followed by the angles of impingement for both software packages under the two different configurations of the JIT. Improved agreement between both Fluent and OpenFoam is once more corroborated, the average relative difference being 15.3%, which is a 10% improvement if compared with the Fluent 14 results (25.2%). The small differences could be attributed to the slightly different steady state results, in which differences in the velocity close to the wall would account for the differences in the angles of impingement. However, both softwares capture the same slope for the behavior of the angle of impingement along the target's radius. In this case, the difference in the slope of the lines formed by the averaged angles of impingement is smaller in comparison with that of the velocities. Results indicate that this slope is slightly steeper for the 25 mm distance JIT.

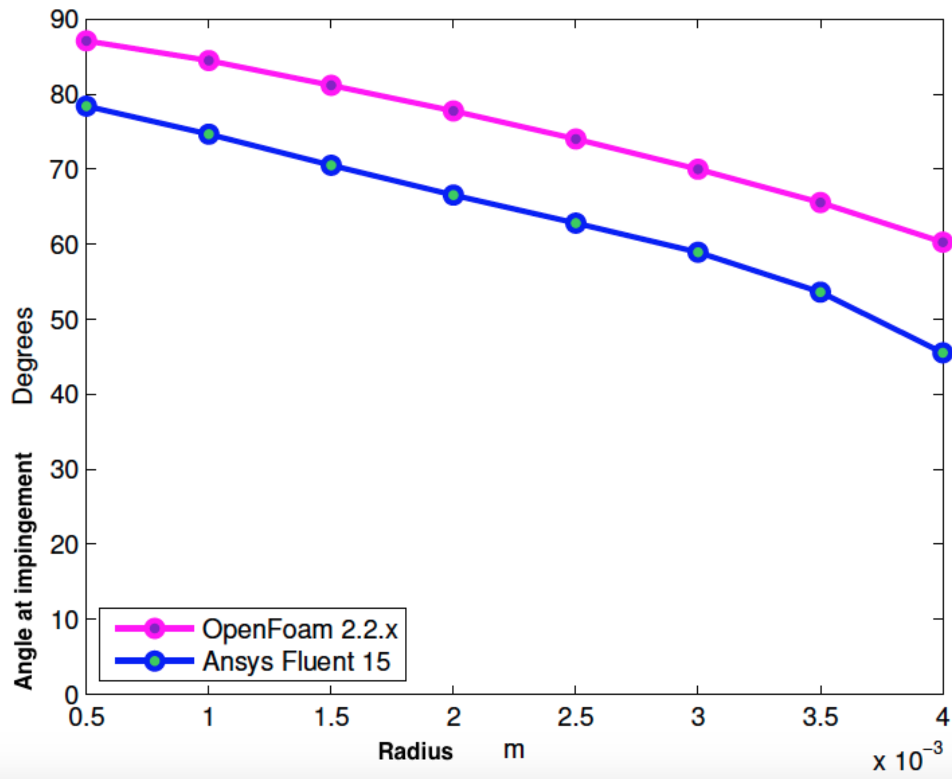


Figure 3.16: Fluent 15.0 Vs OpenFOAM 2.2.x angles at impingement (*degrees*) for the 5 mm distance case

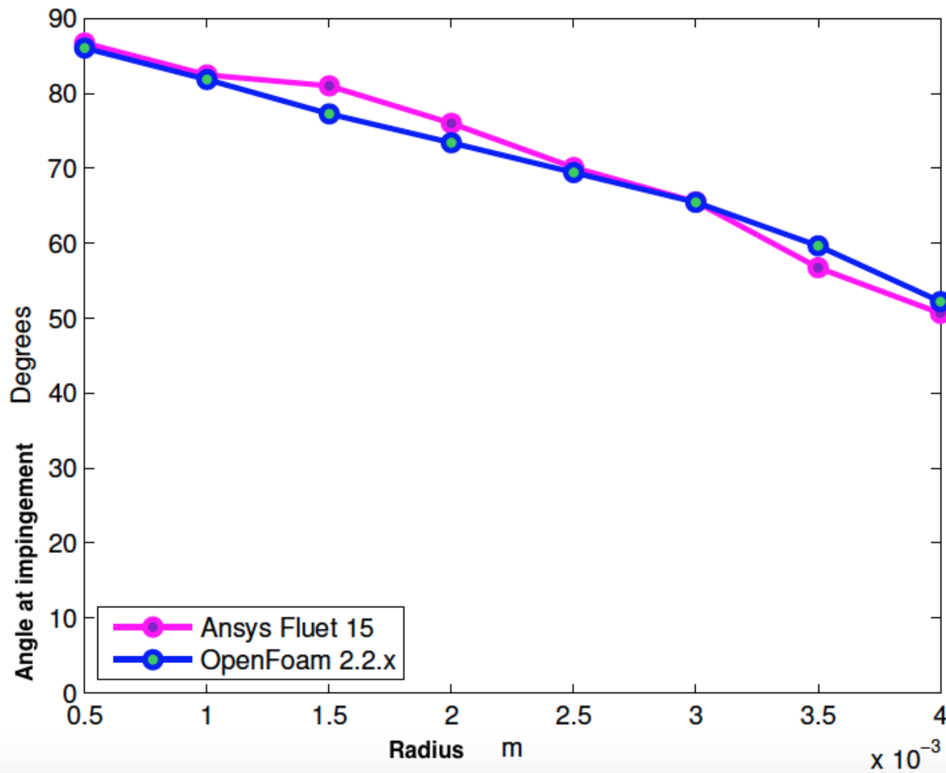


Figure 3.17: Fluent 15.0 Vs OpenFOAM 2.2.x angles at impingement (*degrees*) for the 25 mm distance case

3.2.7 Erosion modeling in OpenFOAM

Erosion is a very complex process, involving a very large number of variables. There is a significant number of papers on erosion, containing many different approaches on how to predict it. Table 2.1 shown in Chapter 2 highlights some of the factors that have been taken into consideration in the literature.

The existence of such a high number of variables and ways to predict erosion results in the following procedure being common practice. Most of the times the performance of different approaches and formulae on the real case are tested and, from the results, the model that compares best to the recorded wear scar is chosen for forecasting similar cases. This is all based on the assumption that the conditions in the CFD simulation are close enough to those of the real problem. The results found in this article show that, CFD can sometimes be misleading if there is a lack of validation data for the given algorithm. The structure of OpenFoam's Lagrangian library is based on C++

templates. This structure allows, in the context of erosion, the definition of as many additional models as required for simultaneous calculation of erosion, making the task of comparing the performance of several approaches for a single problem easier. Each particle impinging the surface generates a value for erosion at the face being hit. All the impacts are summed up and stored in a field of scalars with a value for each face of the selected boundary. By generating several erosion templates and calling them during runtime, the possibility of having erosion calculated by different formulae in the same simulation is enabled.

3.2.8 Conclusions

Minor discrepancies were found between Fluent 15 and OpenFoam. Considering the absolute difference in the steady-state simulations and the portion of the domain affected by these it can be said that both results are basically equivalent. This verified OpenFoam's accuracy and the validity of the package for steady-state calculations of a single incompressible phase as well as for multiphase simulations of a Lagrangian phase dragged by a continuous phase. In fact, versions of Fluent since at least 2011 have been providing miscalculated values of the particle variables due to, most likely, a small programming instability in the discrete phase model that is responsible for integrating the sum of the forces divided by the mass in order to obtain velocities and positions. Unfortunately, this may affect some of the published literature including some of the references mentioned previously [9, 14]. With respect to the discrete phase model of the latest Fluent version minor differences were also found. The level of agreement between trends for the particle velocities and angles of impingement in both software packages is very good. Considering the two different configurations, as figures 3.10 and 3.11 illustrate, for the same values of the inlet particle and fluid velocities, the profiles yielded, especially the one representing the particle velocity, are visibly different. Two of the differentiating factors lie in the location and concentration of the impacts and in the trends for the velocity and the angle of impingement. As the particles have more space to travel between the nozzle outlet and the impact surface, the impacts are more distributed over the whole target. This makes the impact rate lower given that more particles will leave the area without an impact. Steeper slopes are obtained for velocity

and angle of impingement in the 5 mm case due to the greater bend the fluid has to overcome in the smaller gap left for it to flow between the nozzle exit and the target's surface. It is this bend which induces the different slopes in the average velocities between the two cases considered showing an increasing velocity as the distance form the centre is increased in the 5mm case and a decreasing one for the 25mm case.

Chapter 4

CFD of Erosion Processes

4.1 Introduction

In the following section, the procedure used to deform the mesh according to the erosion field is explained in detail. First, the application used to compute the erosion field is described along with the implementation procedure of a further erosion model. Thereafter, the procedure consists of a series of operations with the erosion field produced by the `particleErosion` function object until the desired output is reached. It is after this transformations that the mesh boundary will be deformed by means of an extension of `deformedGeom.C`, which allows moving the mesh boundary according to a field of vectors stored at each time step. After that, the implementation of a Dynamic Mesh solver for erosion modeling will be discussed and finally, a study of the statistics involved in the experiment will be commented along with some utilities developed for that purpose.

4.2 Erosion field calculation in OpenFOAM

One of the many classes OpenFOAM provides the users with is the `CloudFunctionObjects`. This is a templated library that adds additional capabilities to the cloud-based solvers. The available `CloudFunctionObjects` in OpenFOAM 2.2.x are the following ones:

- `facePostProcesing` It records particle face quantities on user-specified face zone. It supports accumulated mass and average mass flux calculations.

- `particleCollector` Function that collects the parcel-mass and mass flow rate over a set of polygons, defined as a list of points.
- `particleTracks` It records all particle variables on each call to postFace.
- `particleTrap` Traps the particles within a given phase fraction for multiphase cases.
- `patchPostProcessing` Standard post-processing. It outputs the desired information at the user-specified patches.
- `voidFraction` Creates the particle void fraction on the carrier phase.
- `particleErosion` This function creates the particle erosion field on the user-specified patches. It outputs a `volScalarField`, or a field of scalars, which, at each face, will be the sum of the volume eroded by all the particle hits.

Focusing on the `particleErosion` function object, an example of an erosion field file in OpenFOAM with only one boundary patch with three boundary faces would be the following one:

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  2.2.x                                 |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       volScalarField;
    location    "1";
    object      kinematicCloudQ;
```

```
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dimensions      [0 3 0 0 0 0 0];

internalField   uniform 0;

boundaryField
{
    boundary-wall
    {
        type            calculated;
        value           nonuniform List<scalar>
3
(
4.42523e-15
1.24376e-14
0
)
;
    }
}


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

The different parameters shown in the file above are described in the following lines:

- **Version** Refers to the version of the code
- **Format** States in which format the file is written, ascii in this case
- **Class** Shows which kind of class is being printed. For this particular func-

tion object it is a field of scalars, which in OpenFOAM is represented as a `volScalarField`

- **Location** Folder where the file is located. `1` means that this file can be found in the folder named with the same number inside the case directory.
- **Object** The name of the object inside the file.
- **Dimensions** Dimensions of the object printed in the file. In this case, as it is Iain Finnie's formula [12] the one being used, the units are cubic meters, corresponding to the volume of material removed.
- **InternalField** Values of the field for the internal cells of the domain. Erosion only takes place at the boundaries, which means the internal field will always be null.
- **BoundaryField** Values of the field for the boundaries of the domain. In this case, as the mesh has only three boundary faces, only three values (one per face) will be written to the file. In any case, numbering of the boundary faces is specified inside the `polyMesh` directory, in the "boundary" file. Information regarding the mesh numbering can be found in [62] and [64].

Impingement information, such as impact speed and impact angle, is gathered as particles hit the walls of the geometry. Taking a look inside the `ParticleErosion.C` file, the constructor is implemented and it requires the names of the patches where it is going to be applied and the plastic flow stress of the material being eroded. Both ratios, depth of contact to length of cut and normal to tangential forces, are also read but in this case, if they are not found, the default ones are used (2 is the default value for both of them).

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

template<class CloudType>
void Foam::ParticleErosion<CloudType>::preEvolve()
{
```

```
    if (QPtr_.valid())
    {
        QPtr_->internalField() = 0.0;
    }
    else
    {
        const fvMesh& mesh = this->owner().mesh();

        QPtr_.reset
        (
            new volScalarField
            (
                IOobject
                (
                    this->owner().name() + "Q",
                    mesh.time().timeName(),
                    mesh,
                    IOobject::READ_IF_PRESENT,
                    IOobject::NO_WRITE
                ),
                mesh,
                dimensionedScalar("zero", dimVolume, 0.0)
            )
        );
    }
}
```

The `preEvolve` member function, as seen in the code, initializes the field. It can be observed that the name the field will be given is going to be the name of the cloud that is being tracked with a Q at the end. In case the `kinematicCloud` is being used, the

erosion field will have the name of `kinematicCloudQ`. The member function in charge of gathering all the necessary information, manipulate it and store it inside the erosion field is called `postPatch`. Here is where one can set a different erosion model from the one that is already implemented, just by reading the necessary particle variables and changing the function into the desired one.

```
template<class CloudType>
void Foam::ParticleErosion<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)

        vector nw;
        vector Up;

        // patch-normal direction
        this->owner().patchData(p, pp, trackFraction, tetIs, nw, Up);

        // particle velocity relative to patch
        const vector& U = p.U() - Up;

        // quick reject if particle traveling away from the patch
```

```
    if ((nw & U) < 0)

        return;


    //Calculate magnitude of the particle velocity at impingement
    const scalar magU = mag(U);
    //Udir is the velocity unitary vector, i.e, the direction of the
    //particle at impingement.
    const vector Udir = U/magU;

    // determine impact angle, alpha
    const scalar alpha = mathematical::pi/2.0 - acos(nw & Udir);

    const scalar coeff=p.nParticle()*p.mass()*sqr(magU)/(p_*psi_*K_);

    const label patchFaceI = pp.whichFace(p.face());
    scalar& Q = QPtr_->boundaryField()[patchI][patchFaceI];
    if (tan(alpha) < K_/6.0)

        Q += coeff*(sin(2.0*alpha) - 6.0/K_*sqr(sin(alpha)));

    else

        Q += coeff*(K_*sqr(cos(alpha))/6.0);




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

If the user-specified-patch is hit, the magnitude of the velocity of the impinging particle at that instant is calculated with the expression `mag(U)` and stored inside `magU`. In order to calculate the angle of impingement, the direction of the particle velocity is determined first and stored in the `Udir` vector. The angle of impingement is the complementary one to the angle between `Udir` and the patch normal direction ($\pi/2 - arccos(angle)$) and, once calculated, it is stored inside `alpha`. The scalar `coeff` is the number of particles inside the parcel times the mass of those particles (i.e., the total mass) multiplied by the velocity and the constant coefficients: plastic flow stress and the two ratios. The field that this `CloudFunctionObject` writes to the case directory is going to be zero everywhere but in the specified patches, where it is going to print a `nonuniform List<scalar>`, which will be the erosion calculated at each face of the boundary patches specified. The equation used for the calculation of the erosion field is exactly Finnie's equation, developed in [12]. The variables used to perform the calculations in the template include members of the class used to define parcels such as:

- `p.nParticle()` Number of particles that the current parcel contains.
- `p.mass()` Mass of the current parcel.
- `p.U()` Velocity of the parcel.
- `p.face()` Face the current parcel is at.

4.3 Mesh deformation in OpenFOAM

4.3.1 Introduction

Once the erosion field has been calculated, the mesh deformation can be accomplished through an already implemented utility called `deformedGeom`. This utility serves as a base for the mesh deformation, providing the function responsible for the point displacement. There is more than one function that could be used to deform the mesh, such as the one used for dynamic meshes which will be discussed later.

4.3.2 erodedBoundaryCellList.C

In this section, an explanation of the code in the utility developed by the author is provided, before the description of the transformations to be applied to the erosion field and their implementation in C++ are described in the following sections. The code inside the file `erodedBoundaryCellList.C`, which is derived from `deformedGeom.C` is represented below:

```
 1 /*---------------------------------------------------------------*\
 2   =========                 |
 3   \\      /  F ield         |OpenFOAM: The Open Source CFD Toolbox
 4    \\    /   O peration     |
 5     \\  /    A nd           |Copyright (C) 2011 OpenFOAM Foundation
 6      \\/     M anipulation  |
 7 -------------------------------------------------------------------
 8 License
 9     This file is part of OpenFOAM.
10
11 OpenFOAM is free software: you can redistribute it and/or modify it
12 under the terms of the GNU General Public License as published by
13 the Free Software Foundation, either version 3 of the License, or
14 (at your option) any later version.
15
16 OpenFOAM is distributed in the hope that it will be useful, but
17 WITHOUT ANY WARRANTY; without even the implied warranty
18 of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
19  See the GNU General Public License for more details.
20
21 You should have received a copy of the GNU General Public License
22 along with OpenFOAM.  If not, see <http://www.gnu.org/licenses/>.
23
24   Application
25       erodedBoundaryCellList
```

```
26   Description
27   Deforms a polyMesh using an erosion displacement field and a
28   scaling factor supplied as an argument. It creates topoSetDict
29   dictionary with the labels of the deformed cells as a cellSet.
30
31 \*---------------------------------------------------------------*/
32 #include "OFstream.H"
33 #include "argList.H"
34 #include "fvMesh.H"
35 #include "pointFields.H"
36 #include "IStringStream.H"
37 #include "volPointInterpolation.H"
38 #include "fvCFD.H"
39
40 using namespace Foam;
41
42 // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
43
44 int main(int argc, char *argv[])
45 {
46     argList::validArgs.append("scaling factor");
47
47 #   include "setRootCase.H"
48
49     const scalar scaleFactor = args.argRead<scalar>(1);
50
51 #   include "createTime.H"
52 #   include "createMesh.H"
53
54     volPointInterpolation pInterp(mesh);
55
```

```
56     // Get times list
57     instantList Times = runTime.times();
58     label lastTime=Times.size()-1;
59     pointField zeroPoints(mesh.points());
60
61  // skip "constant" time
62    if (label timeI = lastTime)
63    {
64        runTime.setTime(Times[timeI], timeI);
65
66 #   include "erosion.H"
67
68        // Check that the erosion field was created successfully
69        if (erosion.headerOk())
70        {
71            Info<< "Reading point displacement" << endl;
72 //Calculate the new point field for the new mesh
73           pointField newPoints
74        (
75                zeroPoints
76 + pointMotionU
77            );
78            mesh.polyMesh::movePoints(newPoints);
79            mesh.write();
80 Info<< "Writing new points in Time = "<< runTime.timeName()<< endl;
81        }
82        else
83        {
84            Info<< "    No erosion Field" << endl;
85        }
86
```

```
87          Info<< endl;
88      }
89
90      Info<< "End\n" << endl;
91
92      return 0;
93  }
94// ************************************************************ //
```

Initially, the user is able to specify a velocity field and, when running in the terminal window `erodedBoundaryCellList 1`, the code will read the scale factor specified (1 in this case) and apply a movement to the points of the mesh equivalent to the `erosion` field specified.

- **Lines 1-31** This part is the default header for OpenFOAM Files. Usually, a brief description of the application or utility is usually included.
- **Lines 32-38** Include statements. These are files that contain definitions or functions that need to be linked to the current file, so that certain variables and functions needed in the utility are available for use.
- **Lines 39-42** Declaration of the namespace where one is working.
- **Line 43** This is where the main program starts and it does it by first declaring that additional information apart from the name of the application has to be provided in order for it to work. This additional information is, in this case, a scaling factor.
- **Lines 44-53** Here the additional arguments are defined. The `scaleFactor` is used to scale the field of scalars by the number entered after the name of the utility in the terminal window. `createTime.H` and create `createMesh.H` are two headers. The first one reads the `controlDict` dictionary and establishes the time parameters, while the latter creates an object to manipulate the mesh.
- **Line 54** `volPointInterpolation` is a class that interpolates volume fields to point fields. The files needed to perform the interpolation can be found inside

`$FOAM_SRC/finiteVolume/interpolation/volPointInterpolation`. This class interpolates from cell centres to points (vertices) using inverse distance weighting. The weight of each of the adjacent data points to the one being interpolated will be inversely proportional to its distance from the latter. First, the algorithm calculates the inverse distances between cell centres and points and it stores them in the weighting factor array. This is done inside the file `volPointInterpolation.C` in two steps. First the weighting factors corresponding to the internal cells are calculated and then the boundary ones. After this, both internal and boundary weights are normalised. For the interpolation, the volume field is multiplied by the weighting factor matrix, after which a field of points is created. Some references about Inverse distance interpolation can be found in [65] and [66].

- **Line 55-60** The time directories created within the case are read and stored. Given that the erosion field needed for the deformation is located in the last time's directory, it is this one that is stored to the variable named `lastTime`. After that, a field consisting of all the points of the mesh before being moved is created and consequently called `zeroPoints`.
- **Line 61-65** Locate and enter directory `lastTime`.
- **Line 66** Include file `erosion.H`, where the erosion field will be manipulated.
- **Line 68-94** Check that the manipulated erosion field exists and if so, read the point displacement (vector field connecting initial and final position of each point), move the points to the final position and write the mesh points to `lastTime` directory. If the erosion field is not found, end program with the message `No erosion Field`.

4.3.3 erosion.H

The file `erosion.H` is the file responsible for the transformations applied to the initial erosion field and also the one in charge of printing out a dictionary that will be used to create a cell set with the deformed cells for an upcoming selective remeshing.

```
//read erosion volScalarField from time directory
```

```
    Info<< "Reading kinematicCloudQ field" << endl;
    volScalarField kinematicCloudQ
    (
        IOobject
        (
            "kinematicCloudFinnie1960",
            runTime.timeName(),
            mesh,
            IOobject::READ_IF_PRESENT,
            IOobject::NO_WRITE
        ),
        mesh
    );


//multiply the erosion scalar field by the scaling factor in order to
// magnify it and be able to see it (if necessary)
volScalarField scaledkinematicCloudQ = scaleFactor*kinematicCloudQ;
//Interpolate the erosion volScalarField to the faces of the cells
surfaceScalarField FkinematicCloudQ =
fvc::interpolate(scaledkinematicCloudQ, "linear");
//The algorithm is going to be executed on the moving-wall patch so we
// look for the right patch
label patchID = mesh.boundaryMesh().findPatchID("incident_wall");

//unitary surface vectors (mesh.Sf()/mesh.magSf()) divided by the face
//area)
const surfaceVectorField Avectors = mesh.Sf()/mesh.magSf();

volVectorField erosion
```

```
    (
        IOobject
        (
            "erosion",
            runTime.timeName(),
            mesh,
            IOobject::NO_READ,
            IOobject::AUTO_WRITE
        ),
        mesh,
dimensionedVector("erosion", dimensionSet(0,0,1,0,0,0,0),
Foam::vector(0,0,0))
    );



forAll( erosion.boundaryField()[patchID], facei)
{
erosion.boundaryField()[patchID][facei] =
Avectors.boundaryField()[patchID][facei]*FkinematicCloudQ.boundaryField()
[patchID][facei];
}
//write the new erosion field created, which is a vectorField with
//magnitude the erosion and the same direction as the face area
//points outside vectors (i.e. in the boundaries of the domain)

erosion.write();

//Finally interpolate that vectorField at each face to each point
//of the face, thus obtaining a pointVectorField
```

```
volPointInterpolation interpolateVolPoint (mesh);

Info<< "Interpolating" << endl;

pointVectorField pointMotionU=interpolateVolPoint.interpolate(erosion);
pointMotionU.write();

//create and fill in list to store modified cells for remeshing
//get list of patches from boundary mesh
const fvPatchList& patches = mesh.boundary();
//from those patches, get the one that is patchID defined at the
//beginning of the file
const fvPatch& currPatch = patches[patchID];

//Initialize variable to count the number of cells deformed
label numDeformedCells = 0;
//Assign number of cells in the patch as size of the list
//loop over all faces of the patch and get the cell number associated
//with each of them, storing them into a list
//set it initially to a very high number so the first comparison does
//only depend on the erosion field being bigger than 0.
label prevCell1=500000000;
forAll (kinematicCloudQ.boundaryField()[patchID], facei)
{
if (kinematicCloudQ.boundaryField()[patchID][facei] >
 0 && currPatch.faceCells()[facei] != prevCell1)
{
//store cell for comparison
//faceCells() gives cell owner of current face inside
//current patch
prevCell1 = currPatch.faceCells()[facei];
```

```
//increase number of deformed cells if conditions
// are met
numDeformedCells=numDeformedCells+1;
}
}
//create a labelList (list of labels) and assign it the size of the
//number of deformed cells, which is calculated in the previous
//forAll loop.
labelList patchDeformedCells (numDeformedCells,0);
//set it initially to a very high number so the first comparison
//does only depend on the erosion field being bigger than 0.
label prevCell2=500000000;
//initialization of counter to start from 0.
label counter = 0;
forAll (kinematicCloudQ.boundaryField()[patchID], faceI)
{
if (kinematicCloudQ.boundaryField()[patchID][faceI] >
 0 && currPatch.faceCells()[faceI] != prevCell2)
{
//position number "counter" of the patchDeformedCells labelList will
// be occupied by the owner of faceI (the cell that owns that cell)
// in case conditions are met (the previous cell in the list is not the
// same one and the erosion field is bigger than 0, i.e., there
//  is deformation.
patchDeformedCells[counter]=currPatch.faceCells()[faceI];
//store the previous cell for comparison with the next one
prevCell2 = patchDeformedCells[counter];
//increment counter
counter = counter+1;
}
}
```

```
//Create and Open the file topoSetDict to write the deformedCells to
//be added to a set with topoSet
const fileName& fName(mesh.time().system()/"topoSetDict");
    OFstream os(fName);
 os << "//Dictionary gathering the deformed cells of the domain.
 Used to create a cellSet with the topoSet utility\n";
 os << "FoamFile\n{\n";
 os <<"    version     2.0;\n";
 os <<"   format      ascii;\n";
 os <<"    class       dictionary;\n";
 os <<"    location    ""system"";\n";
 os <<"    object      topoSetDict;\n}\n";
 os <<"// * * * * * * * * * * * * * * * * * * * * * * * * * * //\n";
     os << "actions\n(\n{\n";
     os << "name   deformedCells;" << endl;
     os << "type   cellSet;" << endl;
     os << "action   new;" << endl;
     os << "source   labelToCell;" << endl;
     os << "sourceInfo\n{" << endl;
 os << "value (";
forAll(patchDeformedCells, i)
{
os << patchDeformedCells[i] << " ";
}
 os << ");\n";
     os << "}\n}\n);";
```

Once the information of the particle impacts is gathered, the variables are operated in order to obtain an erosion field. This field will consist of a scalar value at each boundary face. In the first part of the file, this `volScalarField` is read from the

directory specified in the previous line of `erodedBoundaryCellList.C`, which in this case is the last time stored in memory. After that, the field undergoes a number of transformations. The first step is to multiply it by the scaleFactor defined by the user at the beginning of the program. Normally, the value of this factor should be 1, so one can see the real deformation of the surface if the erosion field is given in the appropriate units. However, not all the formulae used for erosion calculations output the result in meters. In theses cases, this scaling factor could be used to transform the units into meters. Also, given that for a certain amount of time the wear calculated only varies in magnitude but not so much in location or relative difference between points, the erosion field could be subjected to an amplification which would be the equivalent of advancing the simulation in time. This seems reasonable provided that, after a certain number of impacts which can be calculated through sample size determination, the shape of the scar does not vary significantly until the flow does. The line:

```
surfaceScalarField FkinematicCloudQ=
fvc::interpolate(scaledkinematicCloudQ,"linear");
```

interpolates the field of scalars at each cell and at each face of the boundary into what is called in OpenFOAM a `surfaceScalarField` which is a field of scalars that contains a value at each face of the mesh. The word `linear` states that the interpolation scheme to be used will be central differencing. Next, the name of the patch on which the deformation is going to be applied is declared. The next step is to create a field that contains all the surface area vectors. A surface area vector is defined as a vector that points outside of the cell, it is orthogonal to the face and its magnitude is the area of the face. If this is applied to the boundaries, the face area vectors will point outside of the domain and these will give the direction for the point displacement. In OpenFOAM, `mesh.Sf()` and `mesh.magSf()` output the face area vectors and their magnitude respectively. As the magnitude of the displacement of the mesh points will be equal to the magnitude of the erosion, unit normal vectors are required. By dividing the face area vectors by their scalar magnitude, the unit face area normal vectors are obtained as defined in equation 4.1.

$$\hat{\boldsymbol{u}} = \frac{\boldsymbol{u}}{\|\boldsymbol{u}\|} \tag{4.1}$$

Once this stage is completed, a further field of vectors is created containing a unitary vector orthogonal to each face of each of the mesh cells. For each boundary, a set of unit vectors orthogonal to each of its faces will be stored. It is this field of vectors that will be used to multiply it by the erosion field of scalars to have a field of vectors proportional to the magnitude of erosion and pointing orthogonally out of each face at the boundaries. After creating the unit vectors, a field of vectors named erosion is generated and a loop is implemented to assign values to its boundaries. In order to do this, the patch where this loop is to be applied is read (incident-wall is the patch in this particular case) and the values will be the result of multiplying the interpolated erosion field by the unit surface area vectors. The loop will perform the same operation on all the defined patches and all the faces belonging to them while the erosion field of vectors will be written to memory after that.
The final step before being able to move the mesh points is to interpolate the field values contained at each face of the boundaries to each point of those faces using an inverse distance weighting interpolation algorithm. Once the interpolation is completed, the new field will consist of a vector at each point of each face of the boundary, pointing in a direction which will be an interpolation of the directions of the vectors in the adjacent faces and with a value which will be an interpolation of the values of the adjacent faces.

In mathematical form, defining $E(C_i)$ as the calculated erosion field at the face centres of the boundary, the expression of the inverse distance interpolation that yields the values of erosion at the vertices of the boundary takes the form of equation 4.2 [67].

$$E(P_j) = \sum_{i=0}^{n} \lambda_i E(C_i) \tag{4.2}$$

where:

- $E(P_j)$ are the unknown values of the erosion field in location P_j, i.e, the boundary vertices and with $j \neq i$ since the number of points is different to the number of faces.

- $E(C_i)$ are the values of the erosion field at the known locations, i.e, the face centres C_i

- λ_i is the linear combination of the weights and is defined in equations 4.3, 4.4

and 4.5

$$l_i = \frac{1}{d_i} \tag{4.3}$$

$$l_t = \sum_{i=0}^{n} l_i \tag{4.4}$$

$$\lambda_i = \frac{l_i}{l_t} \tag{4.5}$$

Where d_i is the distance from each centre to each point, l_i is the inverse distance and l_t is the sum of inverse distances which defines the weighting factors.
with

$$\sum_{i=0}^{n} \lambda_i = 1 \tag{4.6}$$

Once this first operation is performed, the field composed of a value at each vertex is multiplied (if needed) by a scaling factor and by the unitary surface normal vectors, obtained with equation 4.1, thus obtaining the complete solution shown in equation 4.7. This solution results in a vector field which is the distance to be added to each boundary point.

$$D_j(x, y, z) = K * \hat{u}(x, y, z) * E(P_j)(x, y, z) \tag{4.7}$$

Being K the scaling factor and D_j the distance in the direction of the surface normal vector of each face to be added to each of the boundary points.

The final step in what leads to the mesh deformation is defined in lines 73-77 inside `erodedBoundaryCellList.C` and is available in the original utility from which this new application was created (`deformedGeom.C`):

```
73            pointField newPoints
74        (
75                zeroPoints
76              + pointMotionU
```

```
77          );
```

This operation can be represented as equation 4.8 shows.

$$P_j^1 = P_j^0 + D_j \tag{4.8}$$

Being P_j^1 the location of the new boundary points, P_j^0 their initial location and D_j the distance added to them in the direction of their corresponding surface normal vectors. In OpenFOAM, the field of new points to be written to the mesh is the result of adding to the initial field of points, which was redefined as `zeroPoints`, the field of points that was just created and that is zero everywhere except where erosion takes place. Figure 4.1 shows a very simple mesh of a cube with an erosion value manually set at some of the faces (top and bottom views), an inverse weighted interpolation of the same values and the deformed mesh after the interpolation.

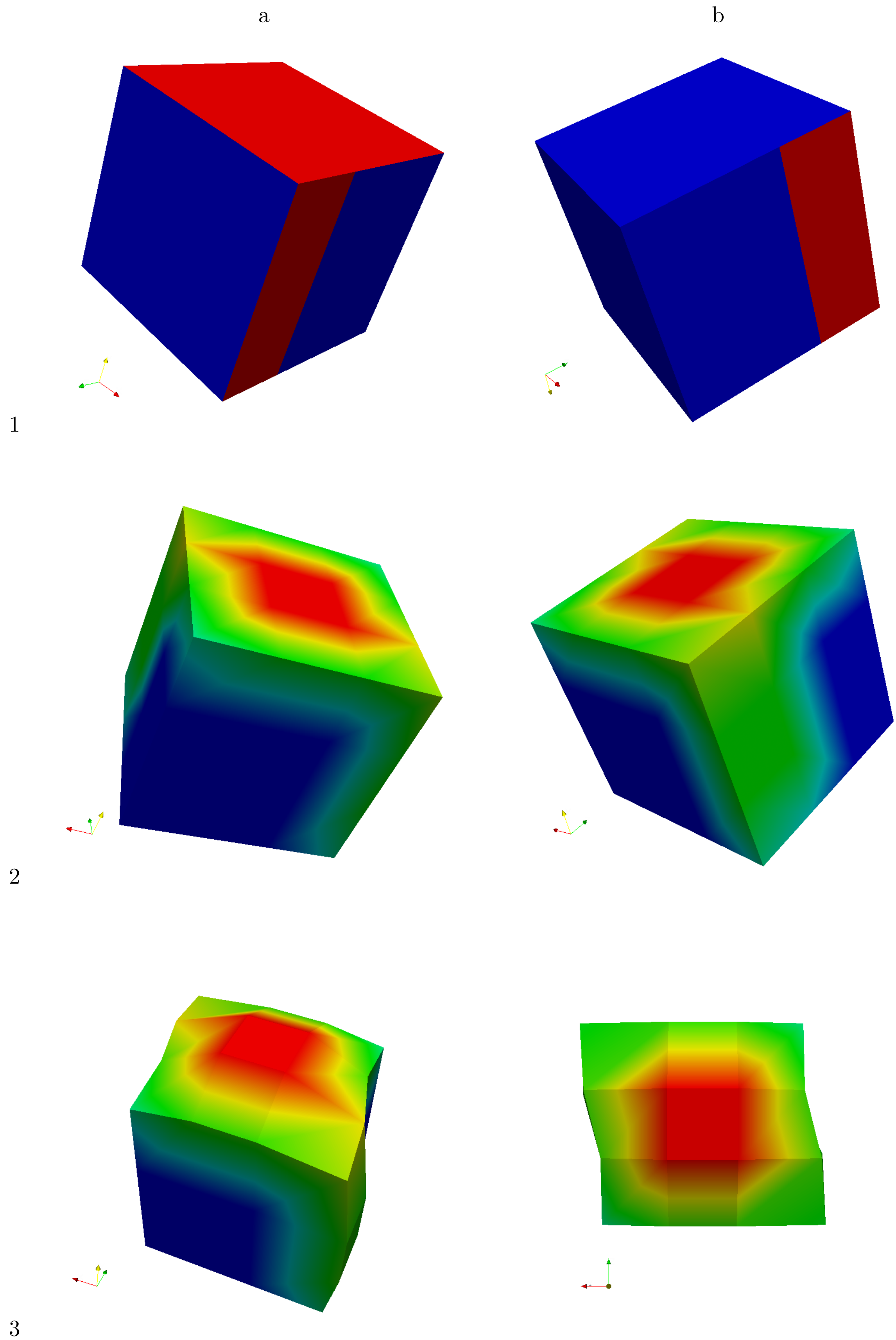


Figure 4.1: Different views of cube with manually set boundary values for erosion (1a and 1b, non-zero values coloured in red), same values interpolated with IDW (2a and 2b, red being the highest values; blue the lowest) and deformation proportional to the interpolated values (3a and 3b.

It might happen that only a small part of the geometry (or patch) is affected by erosion. In that case, the deformation would take place only in a small number of cells within the domain (or patch). It is worthwhile storing the cells where erosion is detected for further manipulation. In this case, a dictionary called `topoSetDict` is created, collecting all the cells where erosion has been stored. When this dictionary is written to memory, it can be used to create a set of cells within the mesh by running the application `topoSet`.The set will be kept inside the `constant/polyMesh/sets`. Creation of this set of cells proves itself very useful when remeshing the deformed parts of the geometry. It is likely that only the cells where erosion has been measured want to be remeshed, and the set stored within the case allows the algorithm to be applied only to those specific elements of the mesh. Thus, the last part of the code consists of a series of loops to store and count all those cells where erosion is present together with the declaration of the class `OFstream`, which allows creating files and writing data out to them. For the dictionary to work with the application it needs to have a particular structure, which accounts for the format used for the last part of the code. Further examples of pre and post-deformation structures are shown in figures 4.2 and 4.3.

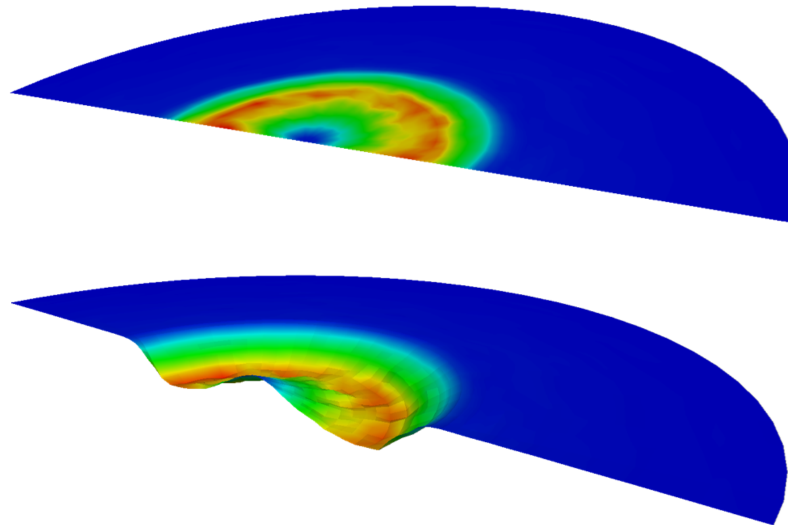


Figure 4.2: Erosion contours in a flat cylindrical probe and the resulting deformed geometry

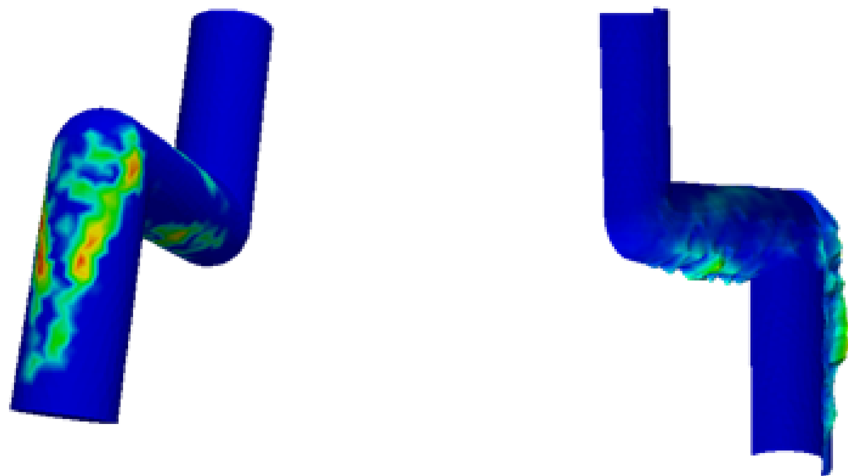


Figure 4.3: Erosion contours in a pipe bend and the resulting deformed geometry

The rest of the code after the mesh deformation in "erosion.H" leads to the storage of the cell numbers affected by the mesh deformation in a set of cells. This set of cells

is automatically introduced in a dictionary generated by the application which, used in combination with `topoSet`, creates a `cellSet` that can be later visualised.

4.4 Implementation of an additional erosion model in OpenFOAM

4.4.1 Introduction

In this section, the procedure to create a new `cloudFunctionObject` for particle erosion processing is described. Given that the variables needed for the new models are the same ones as for the existing `particleErosion`, this template will be used as the starting point.

4.4.2 Implementation of an additional erosion model in OpenFOAM

The new object is going to be placed in the same directory as the existing `particleErosion` so that the compiler is able to find it. In this case, Nandakumar's erosion model will be implemented [68]. By executing in the terminal the following commands, the new erosion model will be created in the aforementioned directory:

```
mkdir NandakumarParticleErosion
cp ParticleErosion/* NandakumarParticleErosion
cd NandakumarParticleErosion
mv ParticleErosion.C NandakumarParticleErosion.C
mv ParticleErosion.H NandakumarParticleErosion.H
```

This series of commands allow us to open a new directory with the name NandakumarParticleErosion and then copy all the files inside `ParticleErosion` into the new directory to finally change their names to `NandakumarParticleErosion.C` and `NandakumarParticleErosion.H`. After this, it is necessary to let the compiler know where to find these new files in order to compile them and add the new erosion model to the list.The file responsible for compiling the Templates inside the intermediate library is `makeParcelCloudFunctionObjects.H` and it can be found inside the

parcels/include directory inside the intermediate library. By adding the following two lines, the compiler will be able to know what to compile and where to get it from:

```
#include "NandakumarParticleErosion.H"
 makeCloudFunctionObjectType(NandakumarParticleErosion, CloudType);\
```

The next step is to change the name of the class by substituting every ParticleErosion by NandakumarParticleErosion, including the header inside the .C file. Once this is completed, recompilation should work properly and the new class will be successfully created. For recompiling libraries in OpenFOAM, the following commands need to be executed in the directory where the Make folder is located:

```
wclean lib
wmake libso
```

Implementation of Nandakumar et al. erosion model in OpenFOAM [68]

For the implementation of this particular model, some parts of the code need to be changed. The name of the command to call the erosion model as well as the name of the file the cloudFunctionObject will output have to be changed. The first part is changed inside NandakumarParticleErosion.H, where the name used to call the cloudFunctionObject, i.e., particleErosion, need to be renamed as something else to avoid conflicts. In this case a suitable name would be NandakumarParticleErosion. Also, a slight modification to the preEvolve function inside the .C file is needed. The name of the file that the program is going to write in the case directory should be changed in case more than one model is being used to avoid overwriting several files or conflicts that might cause a complete crash. In this case the preEvolve function looks like:

```
template<class CloudType>
void Foam::NandakumarParticleErosion<CloudType>::preEvolve()
{
    if (QPtr_.valid())
    {
```

```
        QPtr_->internalField() = 0.0;
    }
    else
    {
        const fvMesh& mesh = this->owner().mesh();

        QPtr_.reset
        (
            new volScalarField
            (
                IOobject
                (
                    this->owner().name() + "Nandakumar",
                    mesh.time().timeName(),
                    mesh,
                    IOobject::READ_IF_PRESENT,
                    IOobject::NO_WRITE
                ),
                mesh,
                dimensionedScalar("zero", dimVolume, 0.0)
            )
        );
    }
}
```

Where `this->owner().name()` ”Q”,+ has been substituted by `this->owner().name()` ”Nandakumar”,+ so that the new output file's name will be `kinematicCloudNandakumar`. The last modification is made in the last few lines inside the `postPatch` function in `NandakumarParticleErosion.C`. The model constants are first defined as C and D and then the rest of the properties such as density mass and diameter. The last step would be to change the `Q` function because, in this model, there is only one formula, independent of the angle of impingement. The initial code of the `postPatch` function

can be found in 4.2 and Nandakumar's erosion model is shown in equation 4.9.

$$\Delta Q = Cm\rho_p^{0.15}(V_0 sin\theta)^{2.3} + Dm^{1.1875}d_p^{-0.0625}V_0^{2.375}(cos\theta)^2(sin\theta)^{0.375} \tag{4.9}$$

And the new code is:

```
template<class CloudType>
void Foam::NandakumarParticleErosion<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)
    {
        vector nw;
        vector Up;

        // patch-normal direction
        this->owner().patchData(p, pp, trackFraction, tetIs, nw, Up);

        // particle velocity relative to patch
        const vector& U = p.U() - Up;
```

```
        // quick reject if particle traveling 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.position()

const scalar C = 7.5e-4;
const scalar D = 0.082;
const scalar rhoP = 2650;
const scalar massP = p.nParticle()*p.mass();
const scalar diamP = p.d();
           const scalar vosintet = magU*sin(alpha);
scalar sinA = sin(alpha);

        const label patchFaceI = pp.whichFace(p.face());
        scalar& Q = QPtr_->boundaryField()[patchI][patchFaceI];

            Q += C*massP*pow(rhoP,0.15)*pow(vosintet,2.3)
+ D*pow(massP,1.1875)*pow(diamP,-0.0625)*pow(magU,
2.375)*sqr(cos(alpha))*pow(sinA,0.375);
    }
}
```

This new function gathers the particle's velocity and impact angle at every impact

and calculates erosion according to Nandakumar's erosion model in order to finally sum up all the erosion produced by all the impacts recorded inside Q, which will have a value for each boundary face and will be stored inside a file with the name `kinematicCloudNandakumar`.

4.5 Patch interaction models

4.5.1 Introduction

OpenFOAM offers the possibility of both defining the interaction with a certain patch of the mesh and recording the information as particles impact one of these patches

4.5.2 Patch interaction models

There are 5 different patch interaction models which are:

- **localInteraction** In this model, the patch interaction is defined on a patch-by-patch basis, i.e., an interaction type is defined for each of the desired patches. A basic condition for the model to work properly is that the particle interaction with all the wall patches is defined. If the model is correctly defined, every iteration, the terminal will print out the number of particles and the mass that sticks to or that escapes the corresponding patch

- **multiInteraction** It runs multiple patch interaction models in turn. It takes a dictionary where all the sub-dictionaries are interaction models.

- **noInteraction** As defined in its header file, it is a dummy class for 'none' option. It will raise an error if any functions are called that require return values.

- **standardWallInteraction** Provides three choices which are rebound, stick or escape and both elasticity and restitution coefficients can be optionally defined. In this case, the interaction model will be the same for all patches.

The definition of which patch interaction model is going to be used is made inside the `kinematicCloudProperties` dictionary, inside the `submodels` section and under

the name of `patchInteractionModel`. There are three different possible interactions with a wall patch:

- **Rebound**
- **Stick**
- **Escape**

Stick

In some environments, particles may reach a boundary and, instead of bouncing off, they may remain stuck to the boundary layer. For this cases, a stick boundary condition can be defined, so that every particle that reaches that boundary will stay stuck on it.

Escape

In cases where an outlet is defined, one may want to let the particles in the domain escape through that patch. For those circumstances, the escape boundary condition is defined, so that when a particle reaches a patch, it will be removed from the domain.

Rebound

Finally,the rebound boundary condition may also be defined. This means that when a particle reaches a distance of `0.5*diameter` or less, it will bounce off the wall and the new velocity will be calculated taking into account the patch velocity (in case it is a moving wall) and reading (if present) the `UFactor` coefficient defined along with the interaction model, which is known as the restitution coefficient.

```
template<class CloudType>
bool Foam::Rebound<CloudType>::correct
(
    typename CloudType::parcelType& p,
    const polyPatch& pp,
    bool& keepParticle,
    const scalar trackFraction,
```

```
    const tetIndices& tetIs
)
{
    vector& U = p.U();

    keepParticle = true;
    p.active() = true;

    vector nw;
    vector Up;

    this->owner().patchData(p, pp, trackFraction, tetIs, nw, Up);

    // Calculate motion relative to patch velocity
    U -= Up;

    scalar Un = U & nw;

    if (Un > 0.0)
    {
        U -= UFactor_*2.0*Un*nw;
    }
```

It is also possible to implement additional rebound models in OpenFOAM. The procedure and modifications to the files necessary to add any rebound models to the code is discussed in AppendixB. In Appendix B, two additional rebound models are implemented in OpenFOAM.

4.6 Patch post processing

As explained in 4.2, `patchPostProcessing` is a utility that provides information about the cloud at the user-specified patches. By default, patchPostProcessing prints out to

file most of the information relevant to each parcel that impacts the specified patches; velocity components , number of particles within the parcel, etcetera. However, it does not provide the angle of impingement of the particle relative to the surface being impacted. In order to do this, the code relative to the angle of impingement and the magnitude of the velocity inside the file `particleErosion.C` is added to `patchPostProcessing.C` so that it will also print those values to the same file. The code inside the modified `PatchPostProcessing` function object is shown below. In this case, the `OStringStream` class is used to print out the required data to a file that will be stored in a folder named `postProcessing`.

```
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_)
    {
        times_[localPatchI].append(this->owner().time().value());

        vector nw;
        vector Up;

        // patch-normal direction
```

```
        this->owner().patchData(p, pp, trackFraction, tetIs, nw, Up);

        // particle velocity relative to patch
        const vector& U = p.U() - Up;

        // quick reject if particle traveling 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.position()

    OStringStream data;
        data<< Pstream::myProcNo() << ' ' << p;
        data<<" Alpha= " <<alpha << " mag(U)= " << mag(U);



        patchData_[localPatchI].append(data.str());
```

The additional variables are `alpha` and `mag(U)`, which represent the angle of impingement and the magnitude of the velocity respectively. This functionality has been used when comparing the velocity and angle average values radially by Lopez et al in [24] and [43].

4.7 Turbulence model

In this section, the effect of the turbulence model will be analysed. In order to do this, two different turbulence models have been assessed. The first one is the $k - \epsilon$ model[69], which has repeatedly been used to predict erosion [9, 14, 60, 70] and the second one is the $k - \omega SST$ model [71]. The $k - \epsilon$ model is a semi-empirical model composed of two equations offering reasonable accuracy at an acceptable computational cost. These reasons make it very popular in industrial applications and over a wide range of flow regimes [72]. On the other hand, the Shear Stress Transport $(SST)k - \omega$ blends k- ω's robust and accurate formulation near the walls with $k - \epsilon$'s free-stream independence in the far field [72]. In a recent publication, the validity and accuracy of some of the most widely used turbulence models (inluding $k - \epsilon$ and $k - \omega SST$) has been assessed by Mackenzie et al in [73], concluding that the most widely used one, which is the $k - \epsilon$ model, is able to capture the general trend of the axial and radial velocities at a reasonable computational cost. With the known limitations of two-equation turbulence models, the $k - \epsilon$ model seems like a reasonable approach, given that it is the general trend of the velocity, the variable that will define the form that the wear scar takes.

4.7.1 Dispersed phase transient simulation

Calculation of the number of impacts per boundary face as well as both angle of impingement and impact velocity averages are carried out through an additional function object that has been implemented for this purpose. The code responsible for the calculation is shown in appendix A. The function sums up the number of impacts on the target, as well as all of the angles and velocities per face and then calculates the averages and standard deviations. In this case, one function object outputs 4 fields at once. Which fields are written to memory at each time step are defined with the line `QPtr1_->write();`. In this case, the fields QPtr1 and QPtr4 to QPtr7 will be written, which correspond to the number of particles, the averages and the standard deviations respectively.

4.8 Statistics of target impacts

4.8.1 General case

In order to calculate how many impacts on the target are needed for an accurate representation of the impact probability on the whole sample the transient simulation is used. Once the steady state in the transient simulation is reached, i.e., the number of particles within the control volume doesn't change or fluctuate around a number, the parameters that allow calculation of the size of the sample needed are obtained. Three new fields are also computed which give some insight into how the mean is evolving during the simulations. These can be written to memory at any time step. For the particular case of a discretised plane, like in the boundary being impinged by particles, the total area (surface of the target) is divided into a set of faces. This means that, as the simulation progresses, a mean and a standard deviation can be calculated for each of the faces of the boundary. By running a transient simulation, a time for which the mean values and the standard deviation stop changing can be found. Once that simulation has been studied and the mean and standard deviation values satisfying the criteria are met, two meaningful values for each of the cells: the mean and the standard deviation are produced. With these two values and the formula used for calculating the size of a sample, the minimum number of particles necessary for a good average can be calculated. In order to calculate the size of the sample that is needed to have a good average of the erosion field, first, the level of confidence has to be specified. In this case this level is set to be 99%, so looking on the Normal Distribution table, it can be inferred that $z_{\frac{\alpha}{2}} = 2.576$. In addition to that, the error needs also be specified. For this variable, a 5 per cent of the mean velocity at each cell is chosen. The procedure is illustrated in equation 4.10 and equation 4.11, which is used for calculating the sample size for each of the faces of the boundary.

$$\delta = \frac{z_{\frac{\alpha}{2}}\sigma}{\sqrt{n}} \tag{4.10}$$

$$n = (\frac{z_{\frac{\alpha}{2}}\sigma}{\delta})^2 \tag{4.11}$$

Where δ is the maximum error of the estimate or the half-width of the confidence interval, n is the size of the sample and σ is the standard deviation [74].

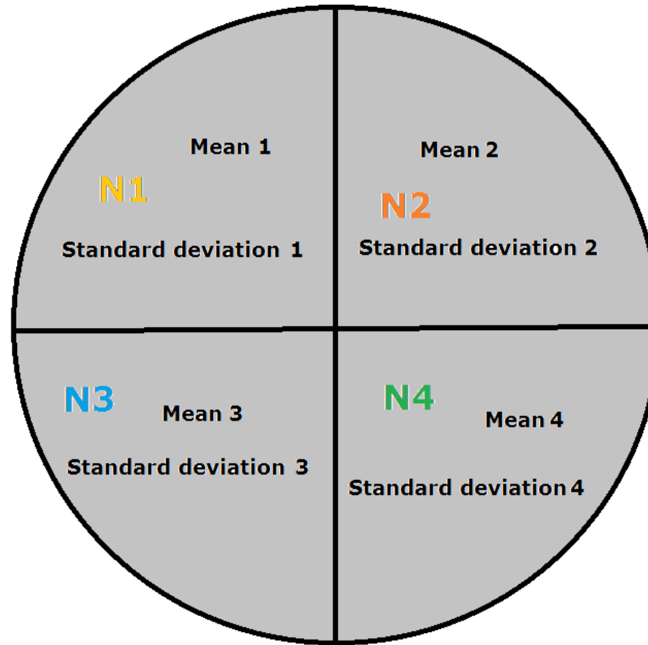


Figure 4.4: Representation of a circular domain divided in four faces, where N1, N2, N3 and N4 are the sample sizes to be obtained for each of the faces

Once the size of the sample has been calculated for each of the faces, all that is needed is to sum up the numbers and see how many particles per face are necessary. However, this process can be made in two different ways: an escape condition can be set up at the target's boundary or a rebound model may be chosen. If the latter is used, the impact velocity average is radically different, as can be observed in images 4.5 and 4.6 since the first impacts have the highest kinetic energy and a big part of it is lost thereafter.

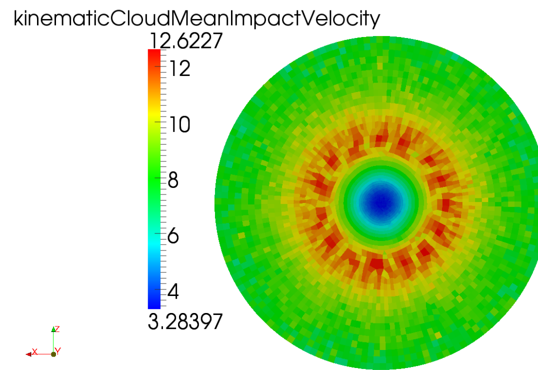


Figure 4.5: Face-wise impact velocity average ($\frac{m}{s}$) after 10 seconds with an escape condition at the target's boundary

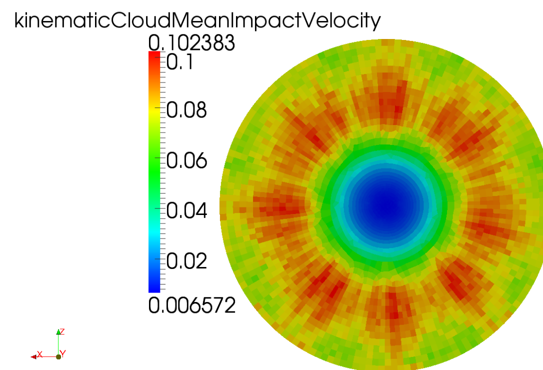


Figure 4.6: Face-wise impact velocity average ($\frac{m}{s}$) after 10 seconds with Forder's [16] rebound model at the target's boundary

Differences are also spotted in the mean angle of impingement and the number of impacts is around $1.5 * 10^5$ times higher, as can be observed in images 4.7, 4.8, 4.9 and 4.10.

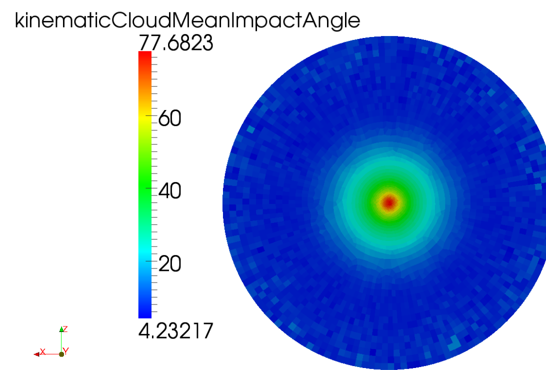


Figure 4.7: Face-wise impact angle average (degrees) after 10 seconds with an escape condition at the target's boundary

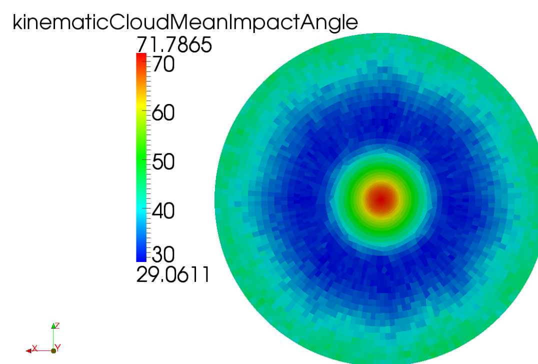


Figure 4.8: Face-wise impact angle average ($\frac{m}{s}$) after 10 seconds with Forder's [16] rebound model at the target's boundary

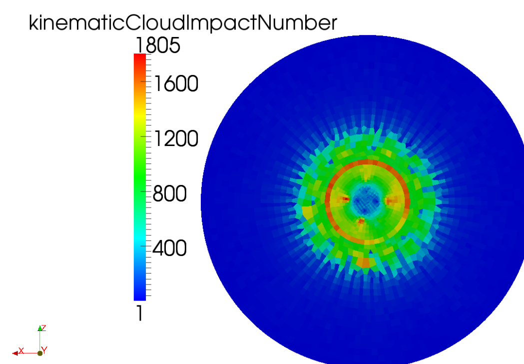


Figure 4.9: Face-wise impact number after 10 seconds with an escape condition at the target's boundary

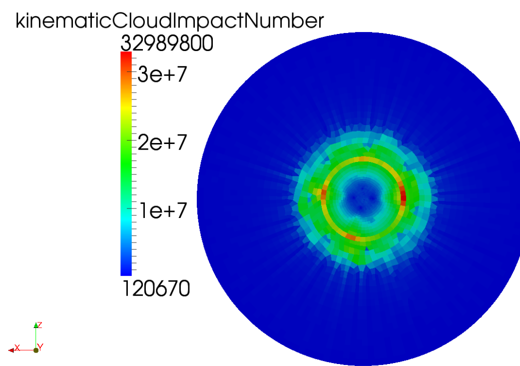


Figure 4.10: Face-wise impact number after 10 seconds with Forder's [16] rebound model at the target's boundary

A total of 10 seconds of transient simulation is set up and the fields are monitored every 0.1 seconds. Values of the fields at 10 seconds are used for calculation of the number of impacts and the velocity and angle averages will be taken from the case with the escape condition, which was previously used by Gnanavelu et al in [9, 14], and which also gives an average which represents more accurately the different wear scar regions, also discussed in [9].

4.9 Implementation of Euler-Lagrange and dynamic mesh solver

An additional solver including dynamic meshing for the deformed boundary has been implemented in both versions of OpenFOAM (2.2.x and 2.3.x). In the latter, dynamic meshing is added to DPMFoam, which is the Euler-Lagrange solver available in that version. The intermediate library in OpenFOAM 2.3.x has an improved efficiency in solving the Lagrangian phase if compared to that of the previous version. The necessary files along with the code implemented to make the dynamic meshing available can be found in Appendix E.

If the code in section 4.3.2 is used, the boundaries of the mesh are deformed according to erosion. This procedure enlarges the cells adjacent to the boundaries which means that they will need remeshing in order to accurately capture the behaviour close to those walls. OpenFOAM has a couple of utilities available for that purpose. One of

them is called `refineWallLayer`. This utility refines the mesh by adding layers of cells on the specified patches. Another possibility is to use the set of cells created with `topoSet` and only refine those. Depending on how complicated the geometry is and the cel distribution at the specified walls, these utilities could create some cells with bad aspect ratios or very skewed, so the results of running the deformed geometry could not be as reliable as one might think. The Laplacian solver used for dynamic meshing, changes the positions of the mesh nodes so that it adapts to the deformation of the boundaries. The algorithm can be set to use inverse distance weighing [66]. In this case the displacement of the mesh nodes is based on the inverse distance of those to the moving boundary, so that the one closest will move more than the ones further away. This methodology is suitable when the mesh deformation is relatively small, as in the present case. Appendix J shows the code developed by the author for a utility that first, calculates the boundary nodes' displacement and then solves the same equations used in dynamic meshing in order to move the adjacent nodes by only one step, with no solution of the fluid flow involved.

In Chapter 6, the methodologies discussed here will be tested and the suitability of each of them for the proposed geometries will be discussed.

Chapter 5

Experimental work

5.1 Introduction

In this chapter the setup of the experimental part of the project will be discussed. During the development phase of the test rig, several nozzle configurations and particle injection methods have been tested, all of which, will be commented in detail.

5.2 Test rig design

The jet impingement test was the test chosen for the experimental work. As indicated by Gnanavelu et al in [9], the jet impingement test is able to reproduce a wide range impact angles and velocities along the target's surface. Trajectories of the particles can be determined using CFD modelling, along with erosion contours and averages of impact angles and velocities. One of the challenges faced during the implementation of the present experiment involved finding a suitable methodology for particle injection. In the literature, when it comes to the injection of the particles into the fluid flow, these are usually circulated through a slurry pump [23, 75–77], which then sends the mixture to the nozzle to erode a metallic sample when a liquid is used. In the case of erosion with air and particles, other methodologies for particle injection may be used such as the ones commented in [5, 12, 35, 37, 78]. The initial approach here, was to avoid circulating the sand used as erodent through the main pump. In order to do this, two different configurations were implemented and tested. First, a venturi was

set up through a bifurcation of the main flow, as shown in figure 5.1. This bifurcation included two valves which allowed control of the mass flow rate going through the venturi. The aim of introducing the venturi was to create a low pressure that would incorporate the sand particles to the flow from the tank where these were isolated. However, it was discovered that, due to the back pressure generated in the contraction situated before the nozzle and represented in figure 5.3, this was not possible as the flow that should be entering the venturi with the particles was in fact, reversed. This was corroborated once the contraction was disconnected and the flow was reversed again and a suction force was measured at the inlet of the pipe situated inmediately after the smallest section of the venturi. A nozzle size of 27 mm, coincident with the size of the inlet and outlet sections of the venturi would have allowed this methodology to work. However, due to the greater size of the nozzle outlet and the specifications of the main pump used for the tests, particle velocity would have been much smaller. This would have involved excessively long testing periods so the configuration was discarded.

The second configuration implemented in the test rig was that of a jet pump, represented in figure 5.6 and figure 5.7. The smallest section of the venturi was enlarged, which created a length of constant section inside the venturi where a 6 mm pipe was introduced after a 90 degrees bend, as figure 5.8 shows. Unnecessary material was also eliminated in order to minimise the weight of the brass part. CFD simulations of the the new configuration confirmed that, at the exit of the 6 mm pipe, a low pressure was again obtained as shown in figures 5.4 and 5.5. However, when connected to the rest of the circuit, the flow was again reversed due to the back pressure created by the contraction situated before the nozzle exit.

The successful introduction of the particles to the flow was achieved though the use of an inexpensive sacrificial pump which injected them after the contraction in the pipe diameter. This configuration is shown in figure 5.9, where a pipe is welded to the nozzle immediately after the contraction. With an initial nozzle diameter of 9.5 mm, the preliminary tests proved successful but excessively long so the diameter of the nozzle was reduced to allow for tests of reduced duration. The decision to reduce the test

Figure 5.1: 3D printed plastic venturi for particle injection

duration was driven by the necessity of having several samples with different depths of the wear scar in order to be able to compare these to the CFD simulations and validate the developed code. The final test rig allowed having different particle concentrations, which ranged from none to around 10% by mass with varying velocities. It is worth noting that the distance between the particle injection and the nozzle exit was higher than 50 mm, while the nozzle diameter was 4.5 mm, allowing more than 10 times the diameter in distance before the exit.

A bypass situated right after the main pump allowed changing the mass flow rate, thus obtaining different velocities at the nozzle. The test rig was composed of three different tanks. The first one had three windows and was where the submerged nozzle was located. The purpose of the windows was for the required imaging and to allow the laser to penetrate the environment for the Particle Image Velocimetry. The second tank contained a smaller cylindrical container inside which the particles were stored.

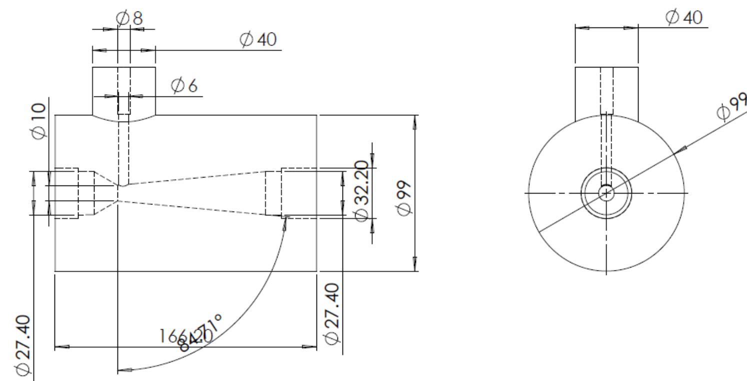


Figure 5.2: Schematic of the venturi for particle injection

A pipe and a valve led directly to this circular container from the nozzle container so that, once these had hit the target (or not) they returned to their tank to be circulated again. The last tank was connected to the second one by means of a pipe with a fine mesh at it's inlet to prevent the few particles that escape the cylindrical container from transferring to the particle-free tank. The particle free tank had two different outlets: the first one was the inlet for the main pump while the second one was an inlet to an auxiliary pump which was used for emptying the whole system at the end of each experiment as well as to help evacuate the particles from the nozzle tank by inducing a flow across its base. A picture of the test rig is shown in figure 5.10 and a schematic is shown in figure 5.11. The last two items of the test rig were a stirrer used to move the particles in order to have a uniform distribution in the tank and a weighing scale which was placed under the nozzle tank which was used as a secondary way of measuring the mass flow rate coming out of the nozzle.

5.2.1 Experimental configuration

The sand used for the experimental work is known as Frac Sand. Frac Sand is a special sand used in the hydraulic fracturing process which is also resistant to crushing. The size distribution of the sand used is shown in figure 5.12. Some of the benefits of using this sand is that the particles can be considered spherical, thus making their numerical simulation less time-consuming as well as being more resistant to break up. The latter is important, since the sand is constantly recirculated. If, instead of spherical Frac Sand, angular sand was used, a high degree of degradation of the sand edges would be

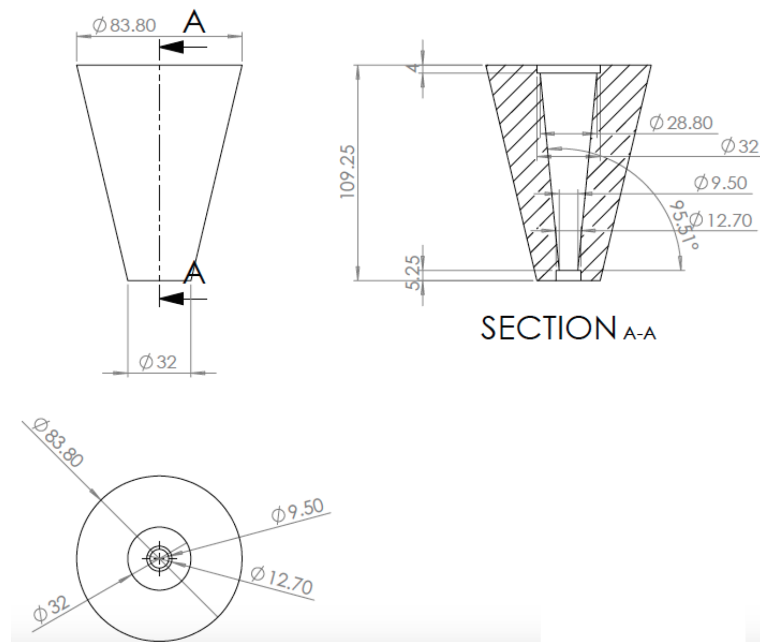


Figure 5.3: Design of the contraction before the nozzle (all dimensions in mm)

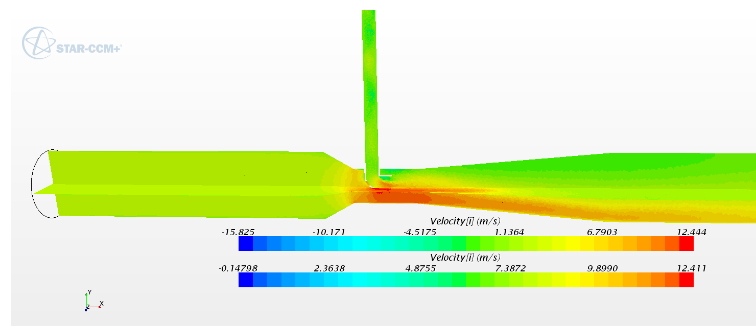


Figure 5.4: Velocity contours of the Jet Pump configuration obtained with Start CCM+

expected. This would imply changing the sand at regular intervals in order to make sure that it keeps its angularity. Since Frac Sand is spherical, no rounding or breaking upon impact was expected.

Once the introduction of the particles was guaranteed by the inclusion of the sacrificial pump, six different tests were run in order to adjust the sand concentration and to check the repeatability of the experiments.

The first three profilometries shown in figure 5.13 correspond to three samples which were eroded for 15 minutes each under the nozzle and scanned in 3 dimensions. The measured concentration of sand was 1.15% by mass and the measured velocity at the

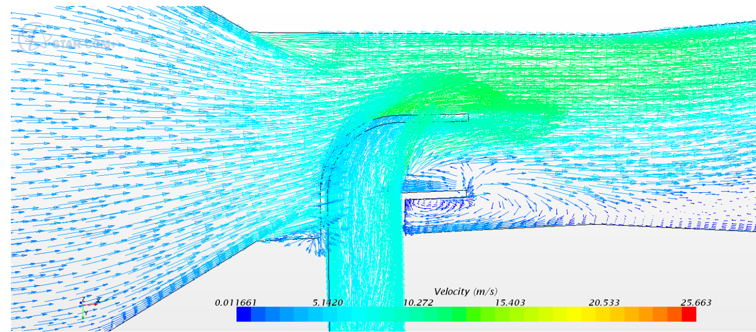


Figure 5.5: Velocity vectors at the pipe inside the Jet Pump

nozzle was 21.8 $\frac{m}{s}$. These samples were scanned with the Alicona Infinite Focus scanner, detailed in 5.3.1. Additionally, three other samples were eroded for 15 minutes with a 7.2% sand concentration and the same fluid velocity at the nozzle exit. These were also scanned and the profilometry is shown in figure 5.14. The higher amount of noise in the profilometry corresponding to the lower sand concentration is due to the difference in the scaling. The samples with lower concentration have an average maximum depth of 40 μm while the samples with 7.2% concentration by mass have an average maximum depth of 350 μm.

5.2.2 Simulation parameters

Sand concentration and fluid velocity were measured directly under the nozzle in order to calculate the parameters necessary for the CFD simulation. These measured values were also checked against the weight change of the scale. Even though the injection of the particles was sufficiently upstream to ensure a fully developed flow at the nozzle exit, an analysis of the wear scar revealed certain asymmetry. This was noticed before running any simulation of the final test rig and it is assumed that the asymmetry is induced by the particle injection. Given the high inertia of the injected particles and their initial velocity (measured velocity of 5.3 $\frac{m}{s}$), a high number of them seemed to travel across the nozzle pipe from the injection point exiting it at the diametrically opposed side, thus creating an asymmetry like the one in figure 5.15 and in the profiles shown in figures 5.13 and 5.14.

For the CFD simulation to be able to represent such a scar, the asymmetry in the particle trajectories must also be included. In order to do this, a length of the pipe

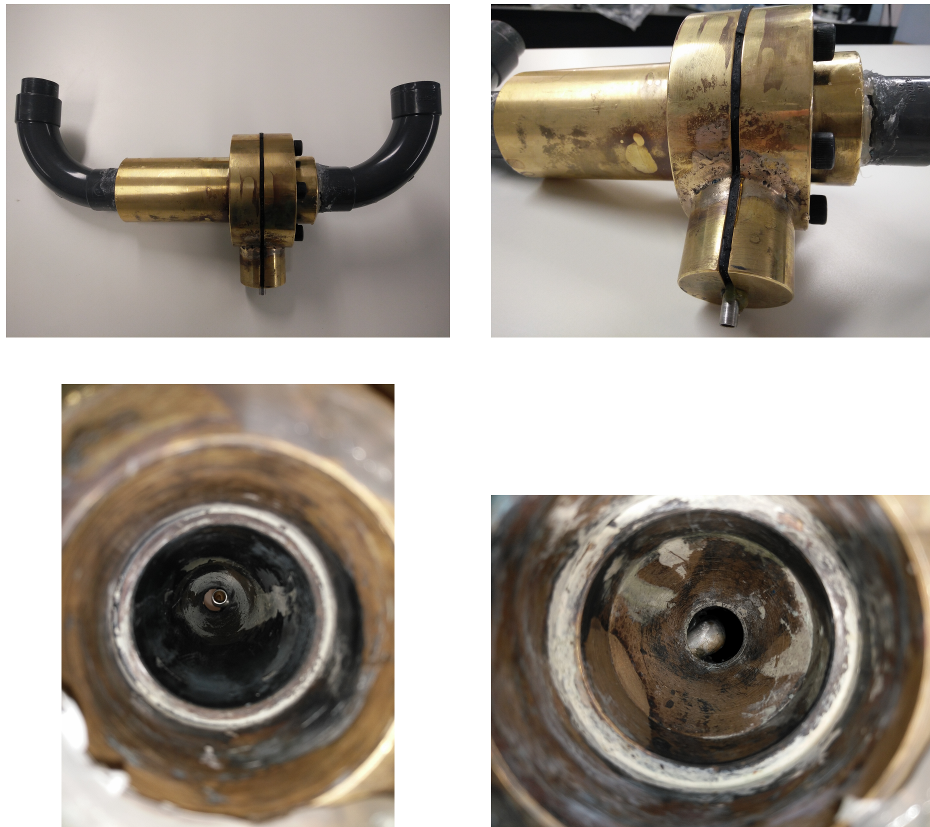


Figure 5.6: Manufactured jet pump configuration

responsible for the particle injection will also be simulated. The measured concentration of sand in that secondary pipe is 26.2% by mass.

5.3 Three Dimensional Scanning

The equipment used for 3D scanning of the eroded samples is part of a set of very advanced materials characterisation equipment located in the Advanced Forming Research Centre, which is a collaborative venture between the University of Strathclyde, Scottish Enterprise, the UK Government and leading multinational engineering firms.

5.3.1 Alicona Infinite Focus IFM G4

The equipment, along with an example of its capabilities is shown in figure 5.16[79]. The most relevant specifications are the following ones:

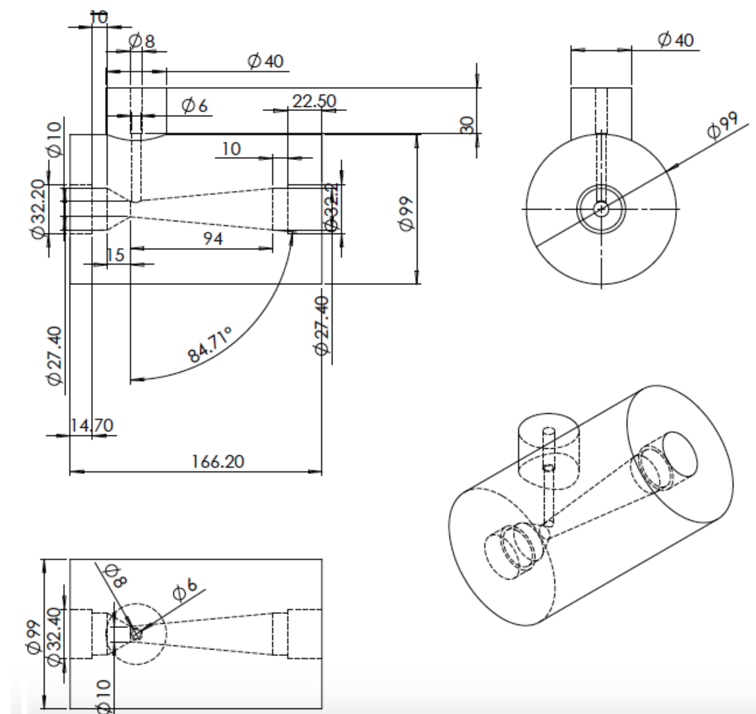


Figure 5.7: Modification of venturi to be adapted to a jet pump configuration (all dimensions in mm)

- Non-contact optical 3 dim. surface digitisation based on Focus-variation
- Surface profile form and roughness measurement
- Minimum vertical repeatability less than 0.12
- Best vertical resolution in the range of 10 to 410 nm
- Fully automated and programmable xyz stages
- 5x, 20x, 50x and 100x objective

5.4 Particle Image Velocimetry

5.4.1 Principles of PIV

Particle Image Velocimetry (PIV) is a non intrusive technique in which some tracer particles are illuminated by a laser sheet. These particles scatter light and the light scattered is then recorded in one or more frames. The displacement of the particle is

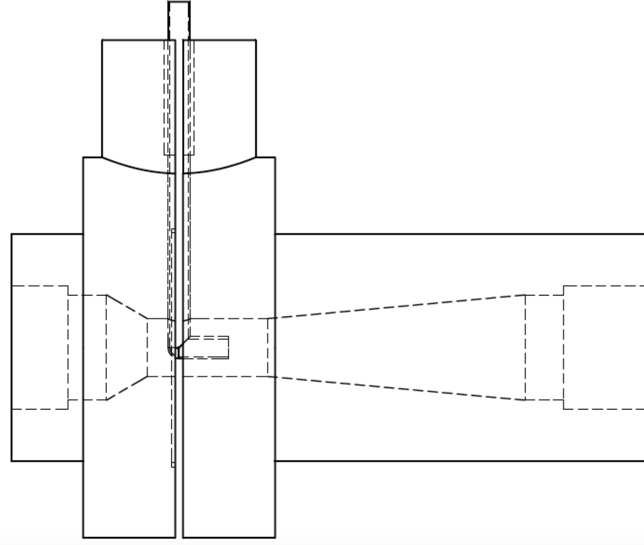


Figure 5.8: Jet pump design

then evaluated through analysis of the recorded images [80]. In an experiment like the one explained here, two different sets of particles can be traced. The tracer particles are assumed to faithfully follow the flow while the Frac Sand can be tracked to validate erodent trajectories. Tracer particles have unit specific gravity, which means it is the same as the water used in the experiments while Frac Sand's specific gravity is around 2.6. Diameters of the sand and the tracer particles also differ: the tracer particles are between 20 and 50 μm in diameter while the Frac Sand's mean diameter is around 500 μm. The efficiency in following the streamlines can be assessed through the stokes number of the particles traveling through the nozzle. Calculation of the stokes number is shown in equation 5.1 [81].

$$S_t = \frac{\rho_p d_p^2}{18\mu_f} \frac{U}{D} \tag{5.1}$$

Where ρ_p and d_p are the density and diameter of the particle respectively, μ_f is the dynamic viscosity of the fluid, U is the mean velocity of the fluid and D is the diameter of the pipe. Values of the Stokes number above 2 indicate highly inertial flow while values below 0.25, the particles are tightly coupled to the fluid flow [42]. Given two equal stokes numbers in two different cases, if the geometrical configurations are similar, this indicates a high similitude in the particle trajectories [82]. If the Stokes

number is calculated for both a tracer particle and a sand particle with diameters of 50 and 500 μm respectively, the numbers obtained are 0.32 for the tracer particles and 82.87 for the sand particles, confirming the predicted difference in inertia.

Once the frames are stored, these can be post-processed with commercial or open source software. This technique was used when different software packages were compared with experimental results in [73]. In this article, a very simple experiment was set up with the flow driven solely by gravity. The aim was to assess how various turbulence models are able to represent the jet impingement test. It was concluded that most of them are able to capture the general trend of the velocity, including k-epsilon model. However, none of the models were able to output an accurate quantitative result when compared to the experiment.

5.4.2 Post-processing the PIV data

There was a number of options for post-processing the images obtained in PIV. One of these options is Open Source Software such as PIVlab [83] which was used for post-processing in [73]. The image obtained through PIV and post-processing the images with PIVlab is shown in figure 5.18. Despite the low velocity at the exit of the nozzle, the method is successfully able to capture the stagnation point right under the nozzle exit as well as the areas where there is higher velocity due to the fluid's change in direction. The standard PIV technique used allowed two velocity components in a plane (the sheet of laser) to be obtained. In order to get a third component, two or more cameras should be used.

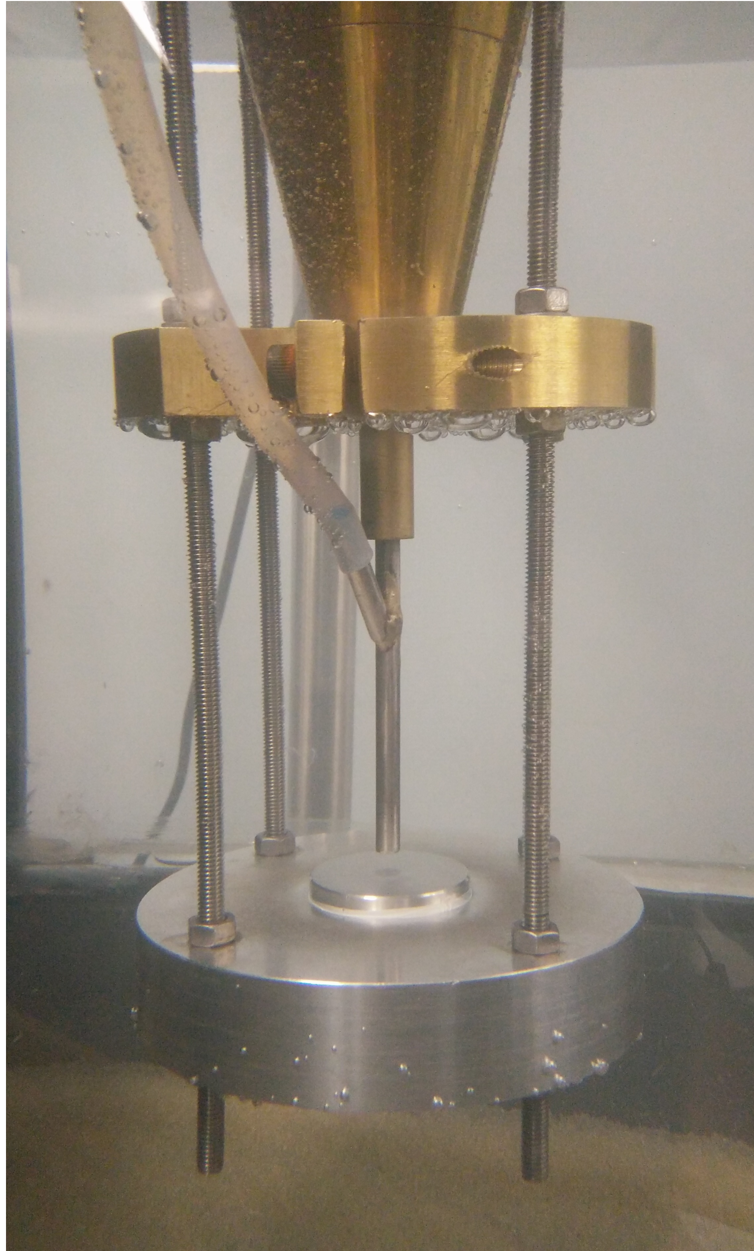


Figure 5.9: Nozzle configuration with sacrificial pump

Figure 5.10: Test rig with sacrificial pump, scale and stirrer

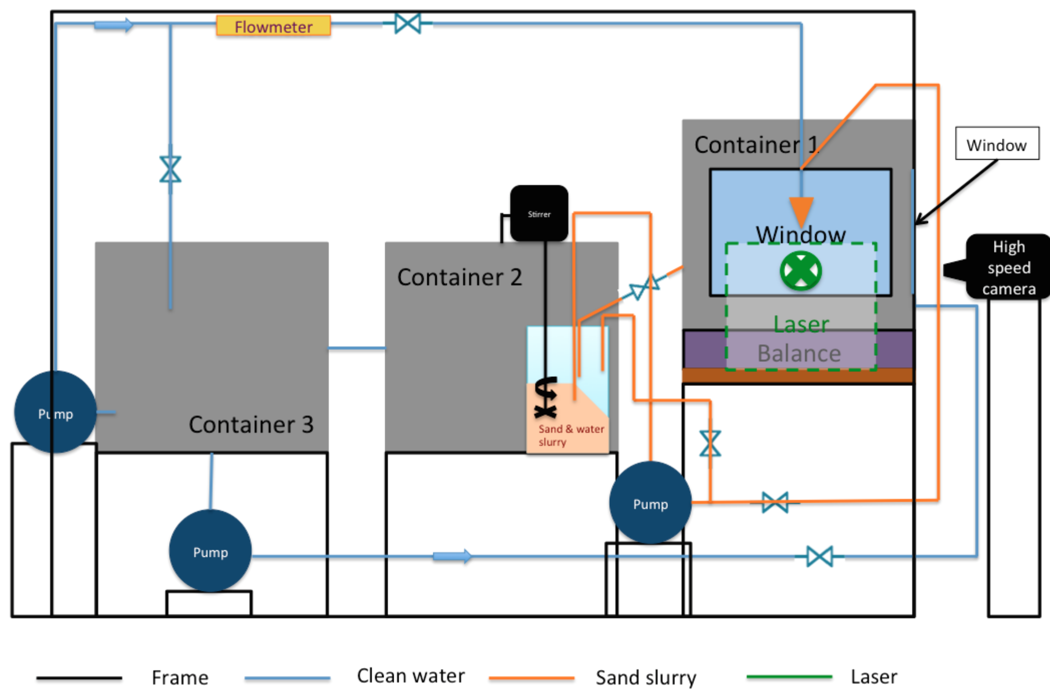


Figure 5.11: Schematic of test rig

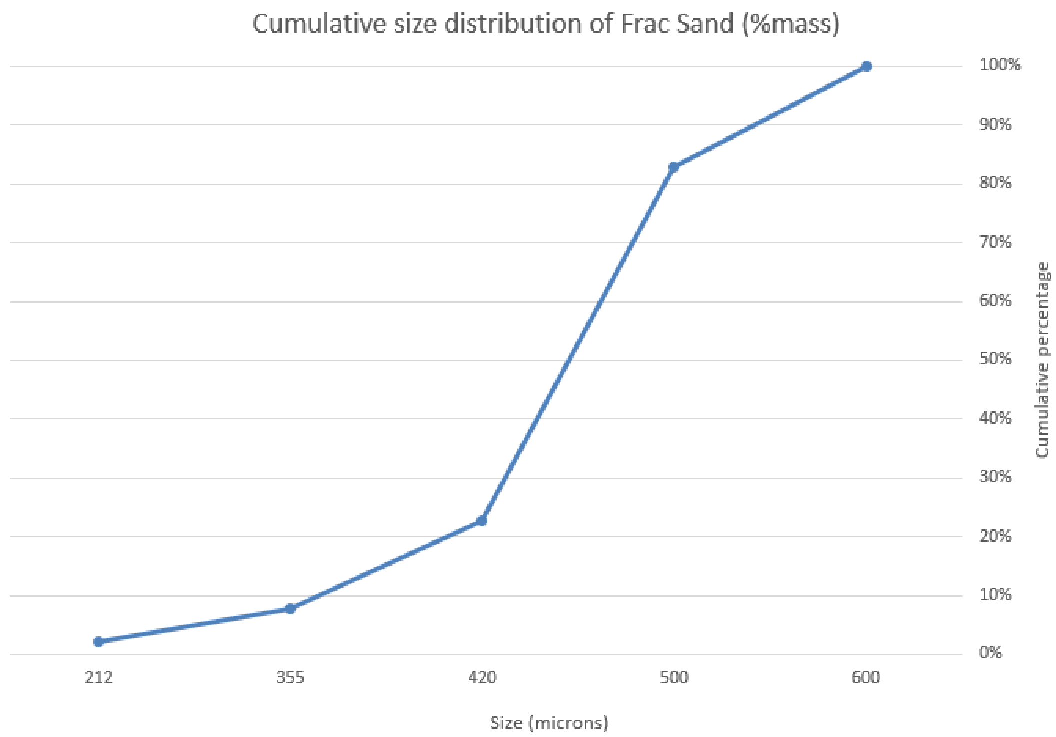


Figure 5.12: Cumulative siize distribution of the Frac Sand used for the experimental work

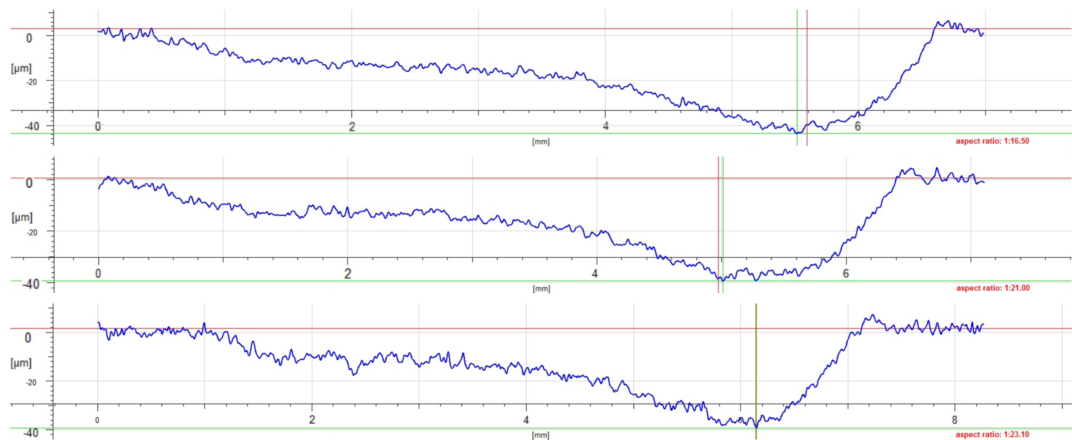


Figure 5.13: Profilometry for three different samples eroded under the same conditions (1.15% sand concentration) for 15 minutes. Depth in μm and horizontal axis in mm

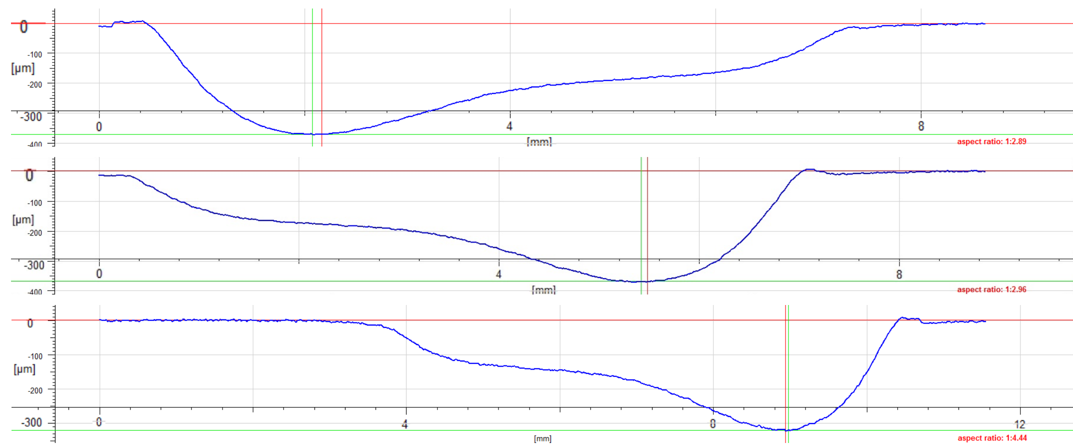


Figure 5.14: Profilometry for three different samples eroded under the same conditions (7.2% sand concentration) for 15 minutes. Depth in μm and horizontal axis in mm

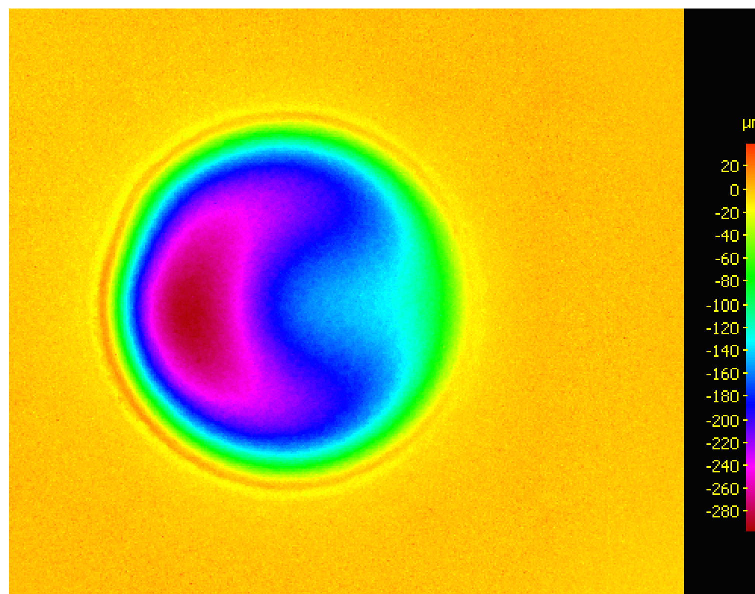


Figure 5.15: Wear scar obtained in a 15 minutes experiment with 7.2% sand concentration

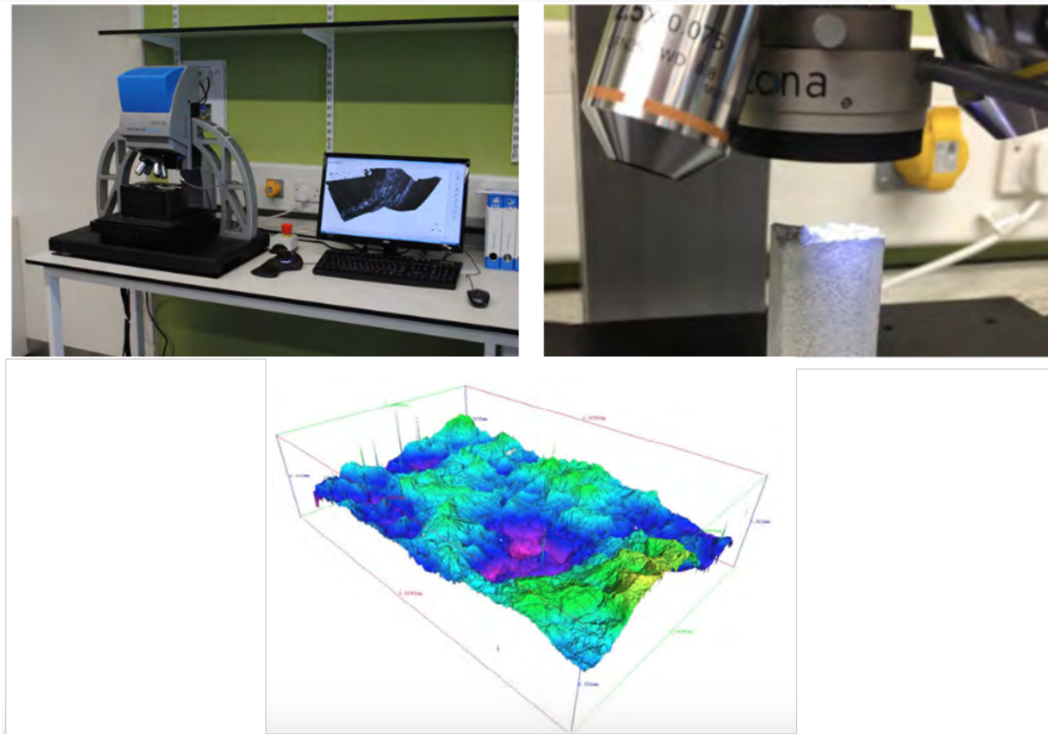


Figure 5.16: Alicona Infinite Focus IFM G4

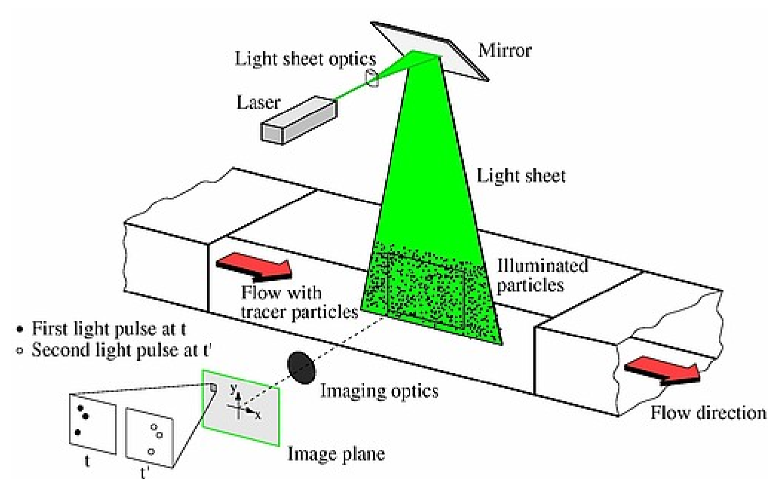


Figure 5.17: PIV setup schematics for a wind tunnel[80]

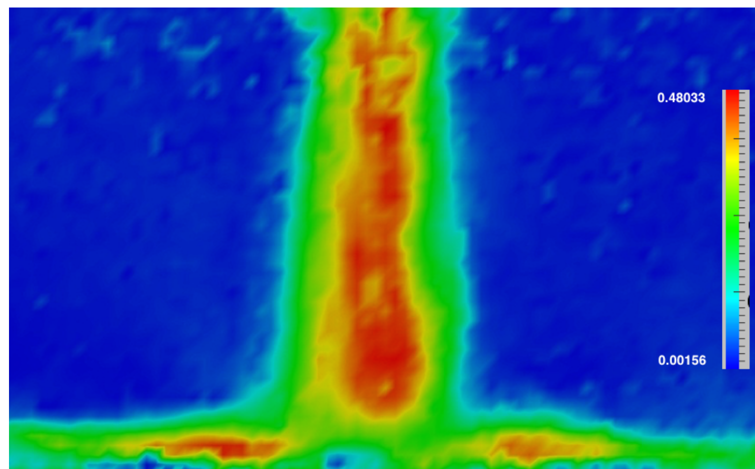


Figure 5.18: Contours of velocity magnitude ($\frac{m}{s}$) obtained with PIVlab [83]

Chapter 6

Results discussion

6.1 Introduction

A series of tests were run with the test rig after ensuring repeatability. These were run for different time intervals, with the aim of obtaining progressively increasing depths in the wear scar under the same conditions. The test rig was drained after each test to ensure that the debris from the previous sample wouldn't affect the new test and that the temperature of the water did not increase excessively. The sand concentration and flow rate were measured in situ both by taking samples directly under the nozzle and using the scale under the impingement tank to verify the obtained values.

6.2 Experimental results

Several experiments have been performed with the designed test rig. Contours of the wear scars for 1.15% mass concentration of sand at 30, 45 minutes, 1 hour and 2 hours test are presented in this section. A 1 hour test with 7% mass concentration was also run and is presented in this section. The wear scars and contour plots of erosion were obtained with the Alincona Infinite Focus detailed in section 5.3.1. For each test, a sample was placed in the holder and eroded for the specified amount of time, after which, the wear scar was measured. The procedure to measure the wear scars was to find the area of maximum depth and then trace a line where the scar profile was measured. It is worth noting that, due to the difference in duration and therefore in

depth between the different tests, the scaling on the contour plots and profiles differ from each other. As the wear scar deepens, some of the material is deformed plastically instead of being removed from the target, having as a result a small area around the scar protruding from the initial surface. Aluminium's ductility makes this visible for the cases of 45 minutes (Figure 6.2), 1 hour (Figure 6.3) and 2 hours duration (Figure 6.4). As a result of this, the colour corresponding to areas with no erosion is slightly shifted. However, when the concentration in a 1 hour test was increased to 7% (Figure 6.5) this feature became no longer visible. The deeper wear scar obtained for that test would substantially modify the fluid flow. This would in turn affect the particle trajectories producing a wear scar with no accumulation of material on the edges as seen in previous cases.

6.2.1 1.15% Concentration tests

Wear scar profiles and contours form 30, 45 minutes, 1 and 2 hours tests are shown in figures 6.1, 6.2, 6.3 and 6.4 respectively.

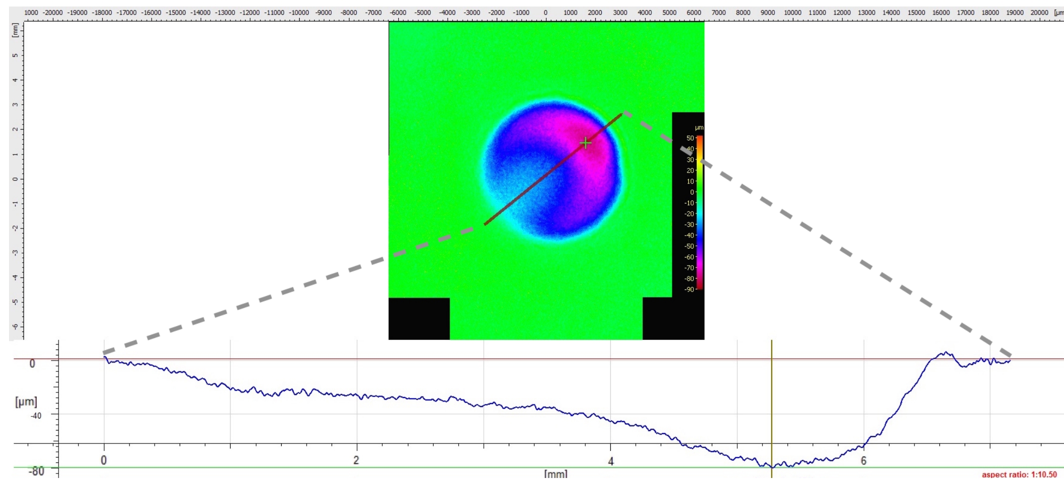


Figure 6.1: Contours of erosion and wear scar profile after 30 mins of experiment at 1.15% sand concentration. Depth in μm and horizontal axis in mm

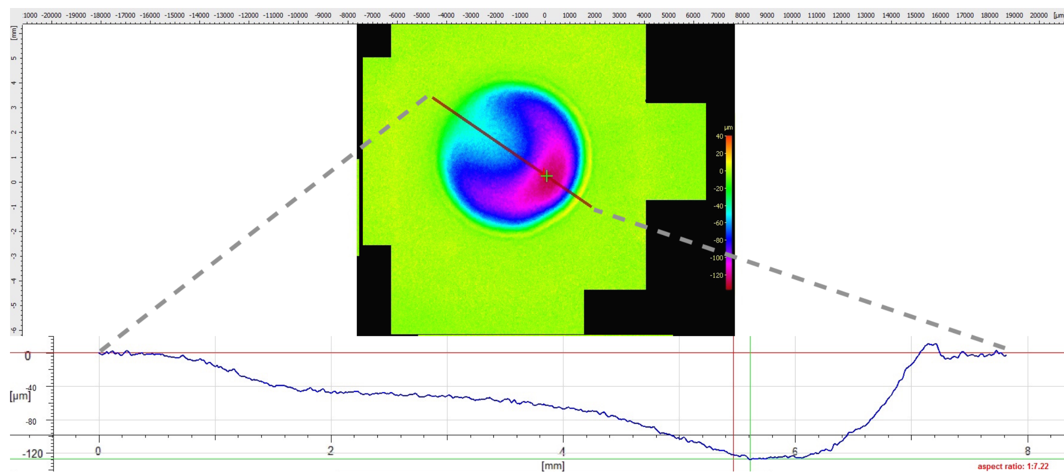


Figure 6.2: Contours of erosion and wear scar profile after 45 mins of experiment at 1.15% sand concentration. Depth in μm and horizontal axis in mm

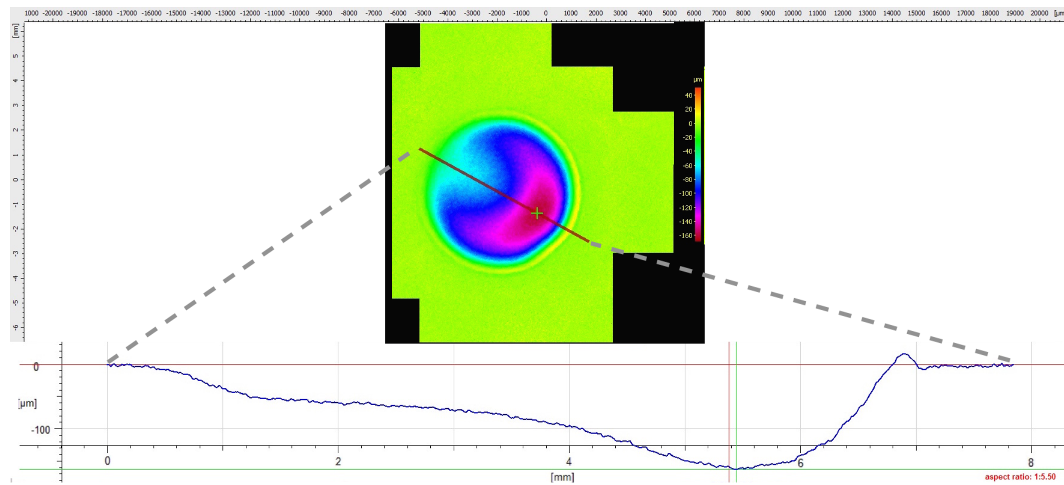


Figure 6.3: Contours of erosion and wear scar profile after 1 hour of experiment at 1.15% sand concentration. Depth in μm and horizontal axis in mm

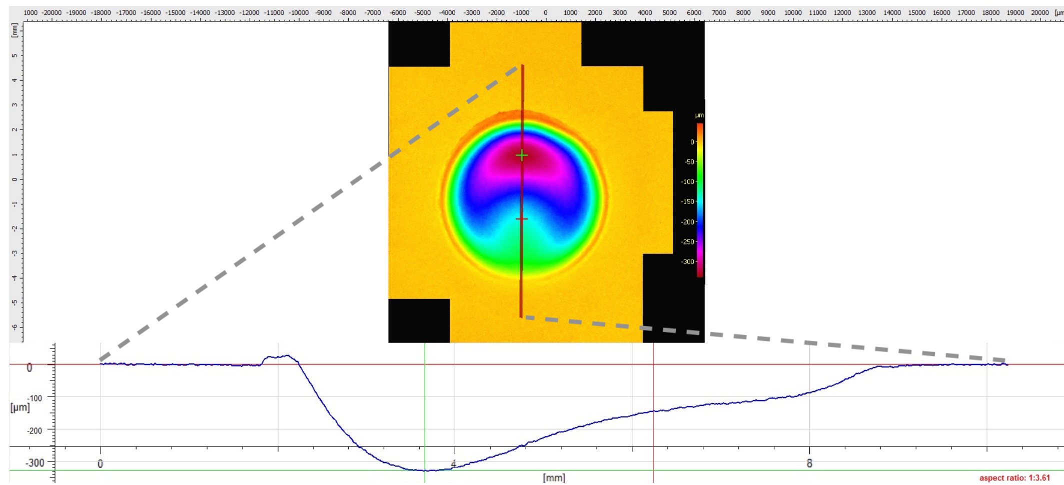


Figure 6.4: Contours of erosion and wear scar profile after 2 hours of experiment at 1.15% sand concentration. Depth in μm and horizontal axis in mm

6.2.2 7% Concentration tests

The wear scar profile and contours of erosion for a 1 hour test at 7% concentration is shown in figure 6.5.

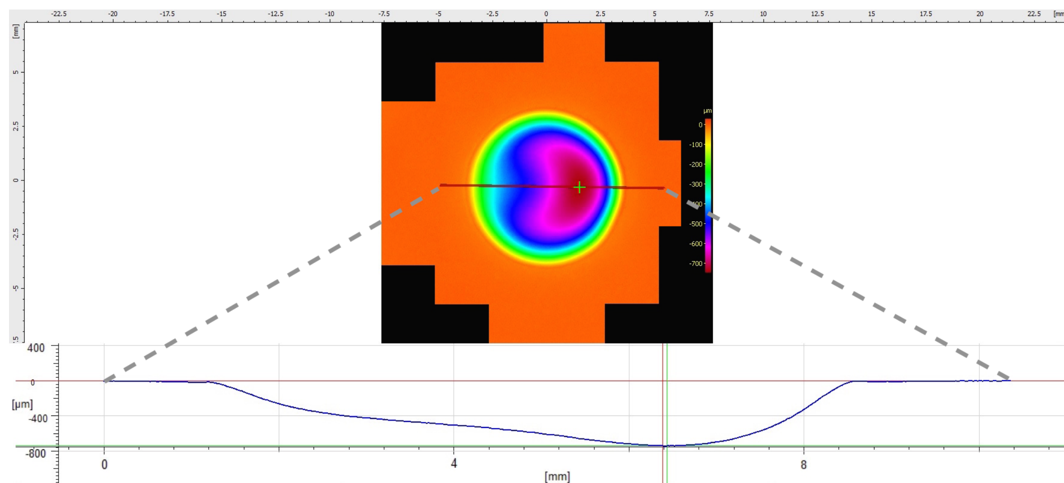


Figure 6.5: Contours of erosion and wear scar profile after 1 hour of experiment at 7% sand concentration. Depth in μm and horizontal axis in mm

6.3 Test rig preliminary CFD simulations

It has been commented in 5.2.2 that an asymmetric wear scar is obtained in the experiments. This asymmetry is attributed mainly to the location of the injection of the particles into the main pipe. A CFD simulation was set up and the averaged values

for the impact angle and the impact velocity were calculated. In order to do this, a steady state simulation was performed first and the particles were later introduced into the flow in a transient Euler-Lagrange simulation. The mesh used for the simulation consisted of 1910673 elements, mostly hexahedral. The velocity contours are shown in Figure 6.6, 6.7 and 6.8 while the pressure contours are displayed in figures 6.9 and 6.10. Figure 6.8 shows the asymmetry in the fluid flow impinging on the target. The averaged magnitude of the impact velocity and angle are shown in Figures 6.11 and 6.12. These contours follow the same trend seen previously in the experimental work, capturing a similar "C" shape on the target's surface.

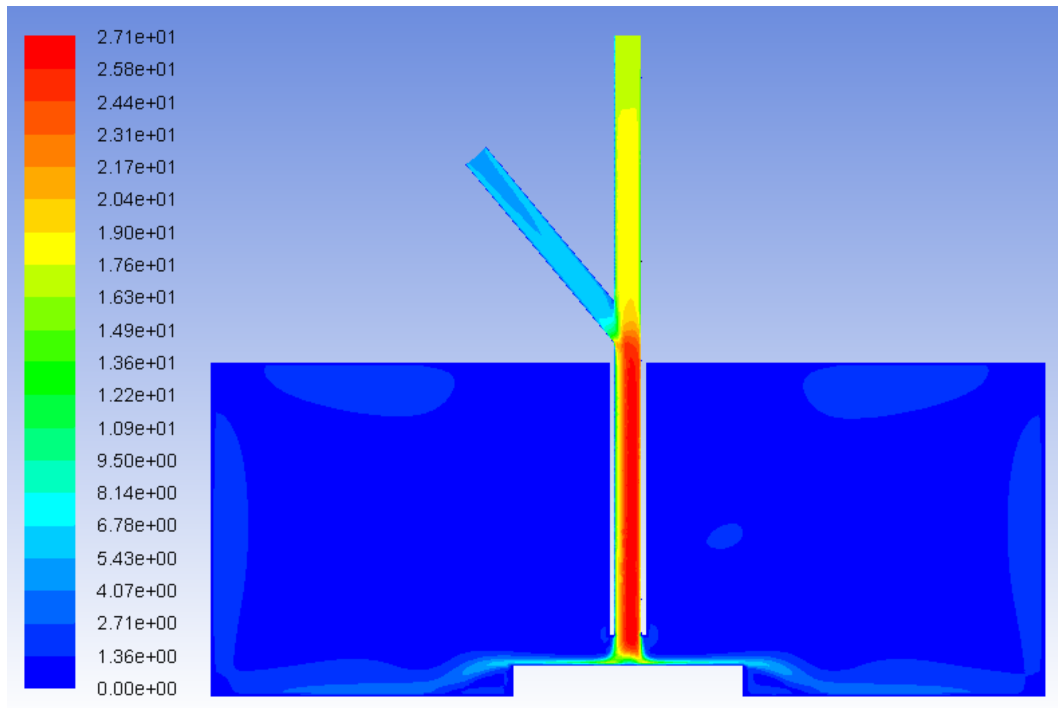


Figure 6.6: Test rig steady state velocity contours($\frac{m}{s}$)

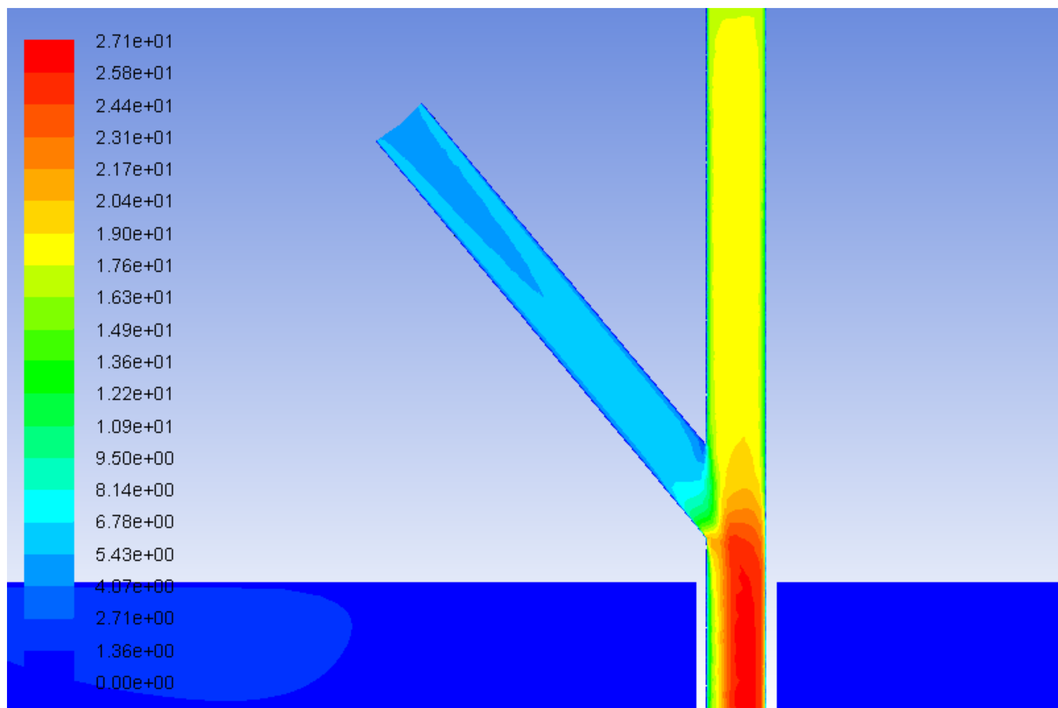


Figure 6.7: Test rig steady state velocity contours($\frac{m}{s}$) around the particle injection

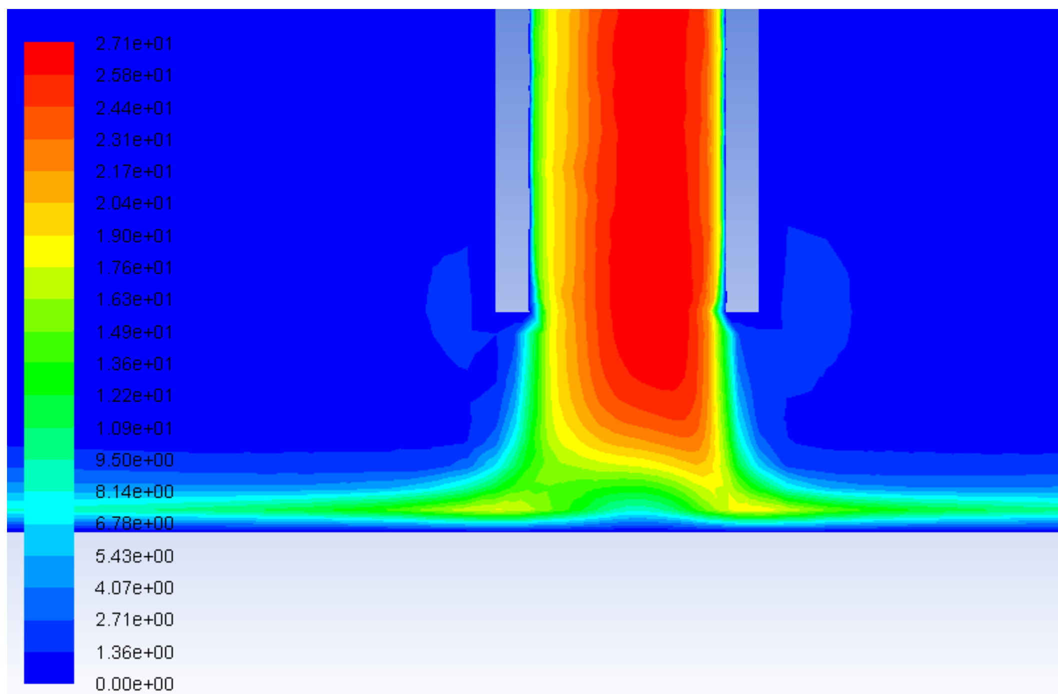


Figure 6.8: Test rig steady state velocity contours($\frac{m}{s}$) around the target

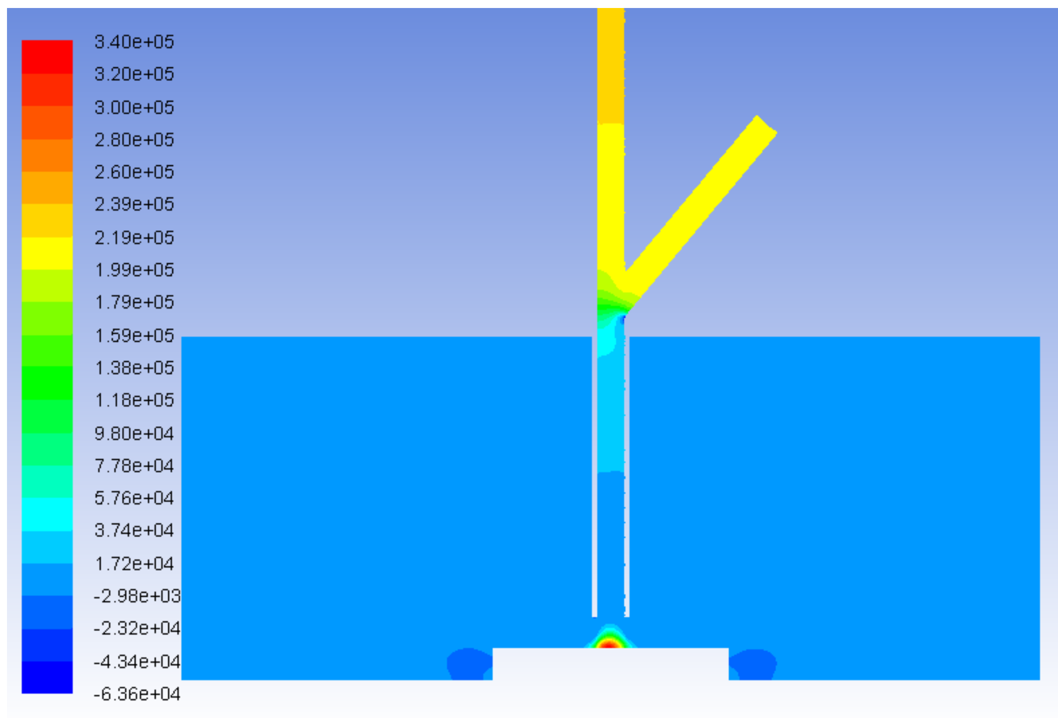


Figure 6.9: Test rig steady state pressure contours(Pa)

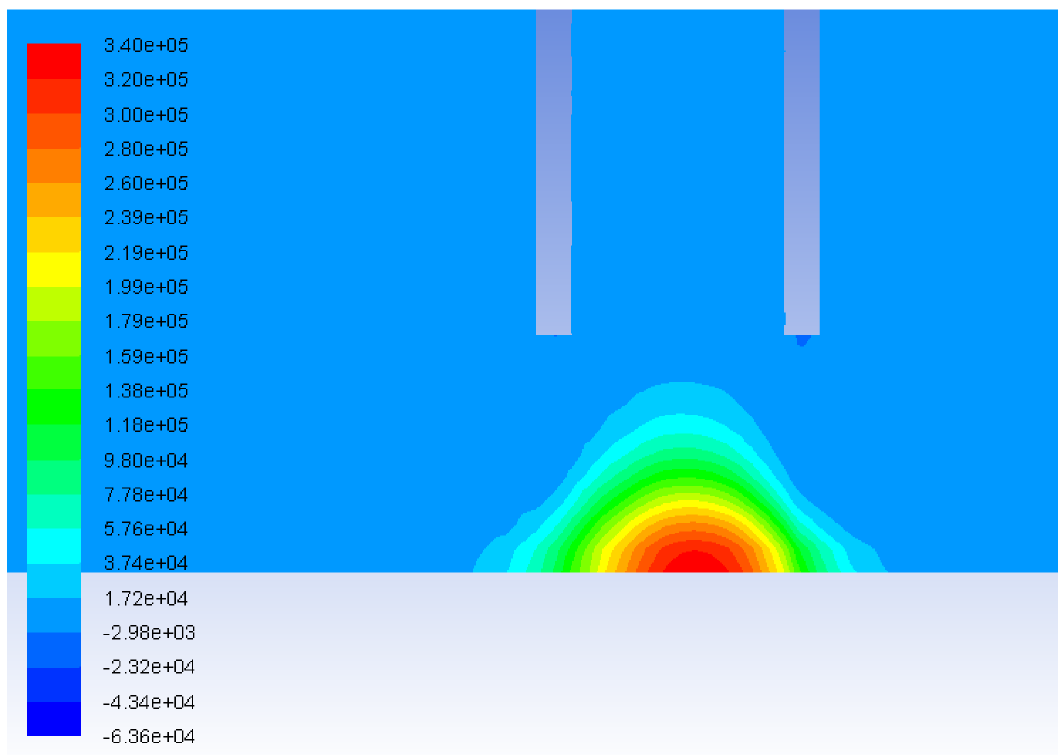


Figure 6.10: Test rig steady state pressure contours(Pa) around the target

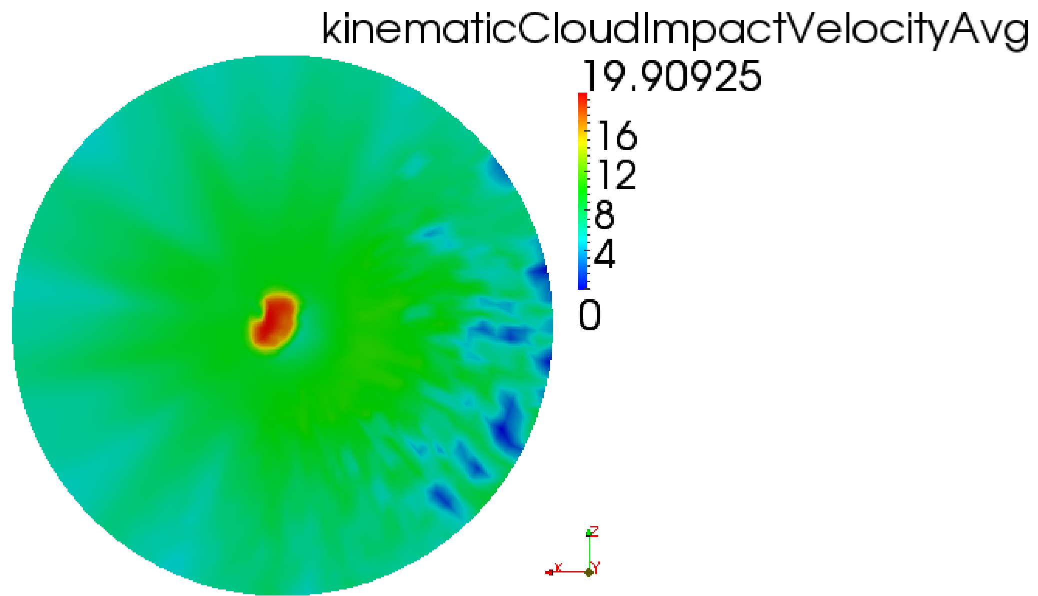


Figure 6.11: Mean particle impact velocity ($\frac{m}{s}$) on test rig's target

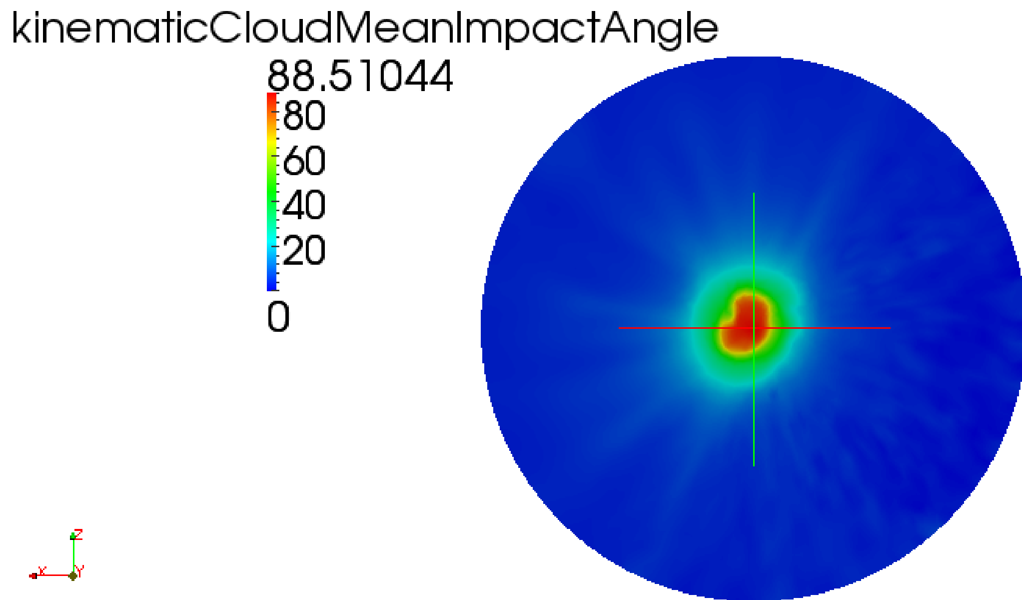


Figure 6.12: Mean particle impact angle (degrees) on test rig's target

These contours confirmed the previously mentioned asymmetry in what respects to the particle impacts. The inertia of the relatively big sand particles used for eroding the geometry was sufficiently high so that their trajectory as they exited the injection was

not sufficiently affected by the flow in the main pipe and a higher number of them ended up impinging on the opposite side of the test sample relative to the injection. This is also illustrated when the particles are represented and coloured by velocity magnitude at the time-step corresponding to 10 seconds. In Figure 6.13, the target is coloured in blue and the asymmetry in the particle distribution is clearly visible, showing that the majority of the particles are driven to one side of the target.The mesh used for this preliminary simulations was relatively coarse. A finer mesh would have been required had the test rig been used for validation given the small size of the nozzle compared to the size of the faces on the target. Three different erosion models were implemented and used to calculate erosion contours for the test rig, namely Tabakoff et al [84], Finnie [12] and Menguturk et al [85]. From these three models, the latter is shown, since it was the one that represented the wear scar measured in the experimental work more closely. The erosion contours on the target's surface obtained with Menguturk's equation [85] are shown in Figure 6.14 and 6.15. Figure 6.15 also shows the mesh faces at the target's surface and the symmetry axes. The wear scar shape obtained in this preliminary simulation is similar to the one obtained in the experiments both showing similar "C" shapes. These results could be improved by refining the mesh around the target in combination with other erosion equations which may better capture the erosive behaviour of the Frac Sand particles.

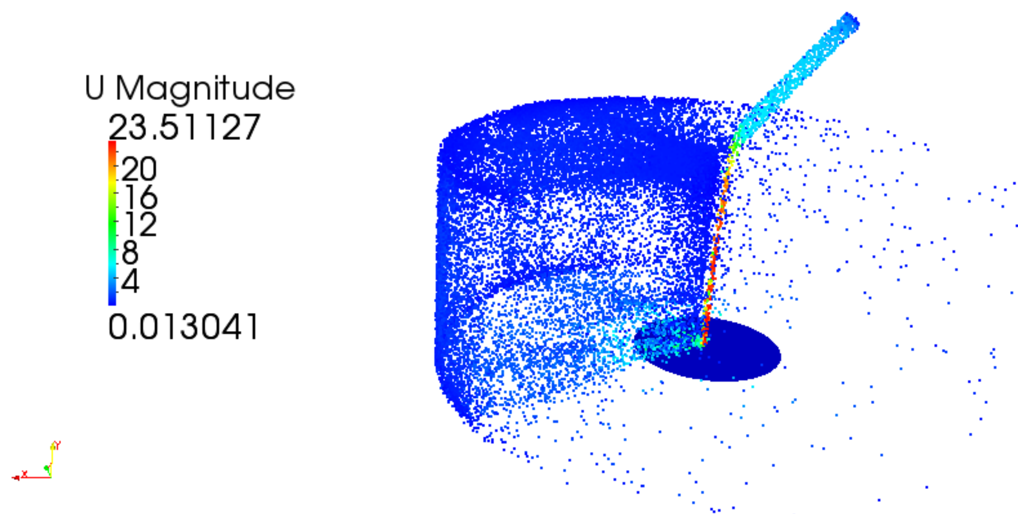


Figure 6.13: Lagrangian particles coloured by velocity magnitude in the test rig domain and target coloured in blue

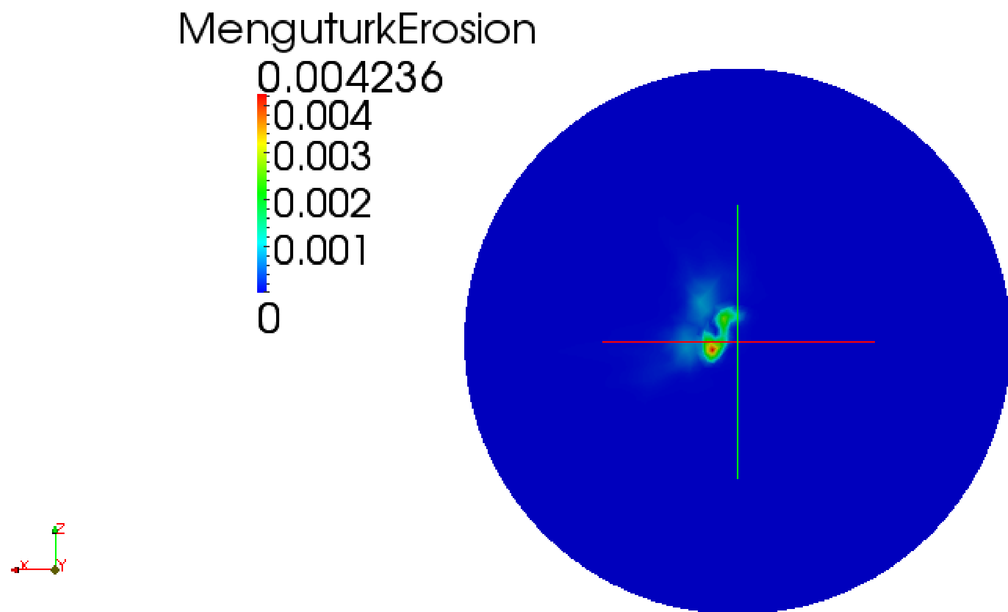


Figure 6.14: Erosion contours at the targets surface showing the symmetry axes

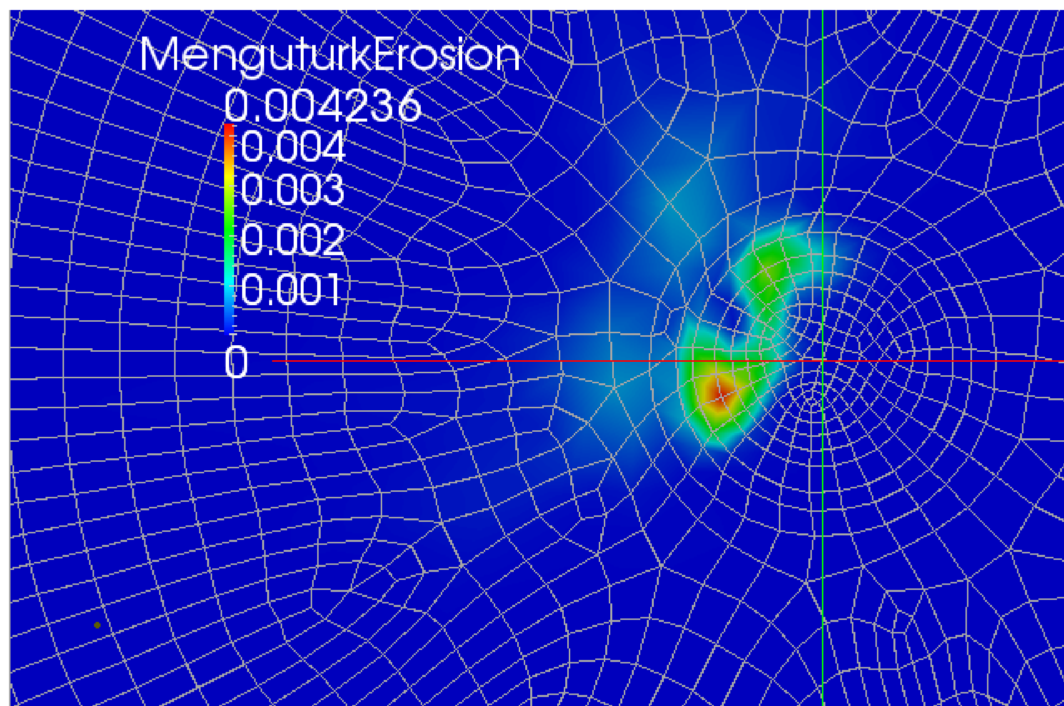


Figure 6.15: Erosion contours at the target's surface showing the symmetry axes and the mesh faces

6.4 Validation fluid flow changes due to wear scar

6.4.1 Introduction

Technical issues with the laser used in the Particle Image Velocimetry prevented validation of the algorithm with this technique. However, the results presented in the work by Nguyen et al [60] seem to predict the creation of a new stagnation area after 30 minutes of erosion with an inlet velocity of 30 $\frac{m}{s}$. In their work, Nguyen et al eroded a target for 30 minutes and used a 3 dimensional scan of the wear scar as the new geometry. Once the steady state was calculated for this new geometry, a new low pressure area appeared as a result of the increased velocity of the fluid induced by the new configuration of the worn surface. In this section, a computational analysis of the experimental work in [60] is presented with the purpose of validating the deformation algorithm. In addition to this, a new approach including mesh deformation is proposed in order to calculate the evolution of any 3 dimensional eroded geometry.

6.4.2 Methodology

With the statistical analysis outlined in section 4.8 the minimum number of impacts needed to obtain both velocity and impact angle averages with a certain degree of confidence can be calculated. In order to do this, 10 seconds of erosion were simulated for the case studied by Nguyen et al [60]. Initially, a visual inspection of the results for different times, showed that the wear scar variations after the first second of simulation were negligible, as shown in figure 6.16, where the wear scar was obtained using the formula developed by Tabakoff et al [84]. These figures show how the contours only increase their values without changing the shape of the scar from the first second of simulation.

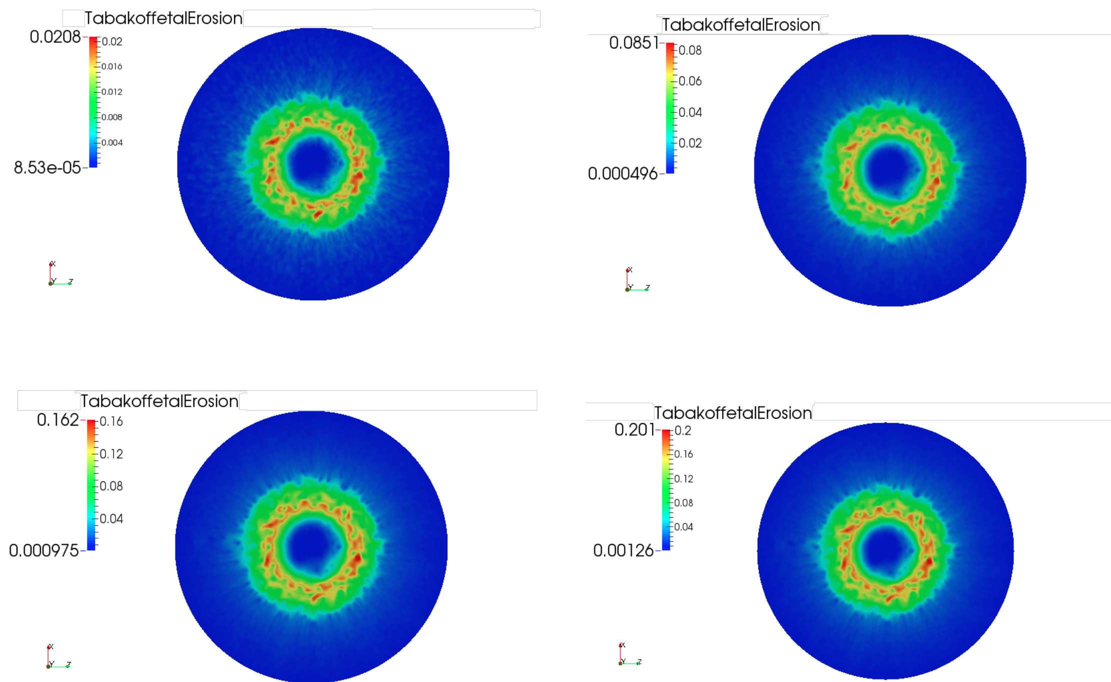


Figure 6.16: Contours of erosion per unit mass of impacting particles at 4 different simulation times. From left to right and from top to bottom: 1, 4, 8 and 10 seconds

Since one of the aims is to optimise the computational time invested in running the simulation, the results between time 0 and 1 were analysed every 0.1 seconds of simulation. In addition to this, with the aid of the fields developed in section 4.8, the minimum number of impacts required per face were calculated. The results after 10 seconds of simulations were used in order to calculate this minimum number of impacts with a confidence level of 95 %. The number of particles released after 10 seconds was 1 million. Additionally, an application was compiled (shown in appendix K) in order to sum up the minimum number of impacts required at each face for the impact angle and impact velocity averages. When the application was run, two numbers were obtained, of which the highest one was chosen as the minimum number of impacts. The value obtained for the simulation was 48677, which corresponded with 0.48677 seconds of simulation. However, this value only accounts for the time at which the particles are released, but 48677 refers to the number of impacts. Thus, the average time that a particle spends inside the domain was added to the 0.48677 seconds. The calculation of the average time a particle spends in the domain was obtained from

the log of OpenFOAM's particle variables and estimated to be 0.02 seconds for this simulation. It was after this simulation time that the number of particles within the domain (incoming minus exiting) fluctuated around 330. Therefore, the total time was rounded up to 0.5 seconds of simulation. A qualitative comparison between the erosion contours at time 0.5 seconds and 10 seconds is shown in figure 6.17. Figure 6.18 shows a comparison of the normalised erosion ratio between the same two time-steps plotted along the radius of the target. The average difference between lines is 1.19%, which validates the statistical accuracy of the erosion contours at 0.5 s.

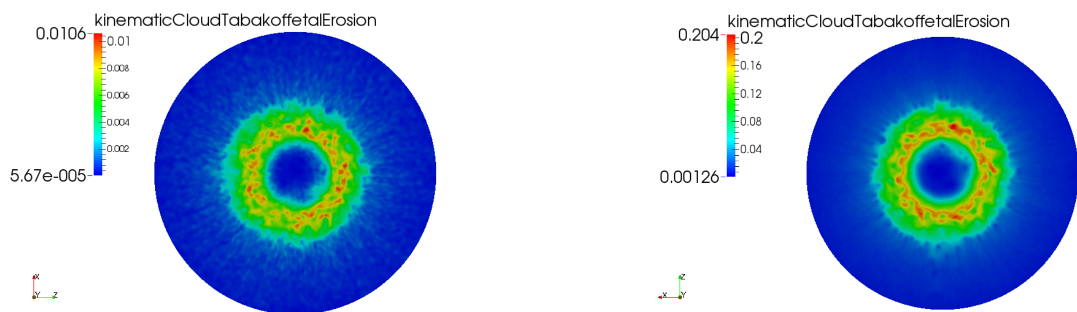


Figure 6.17: Contours of erosion per unit mass of impacting particles at 0.5 and 10 seconds of simulation

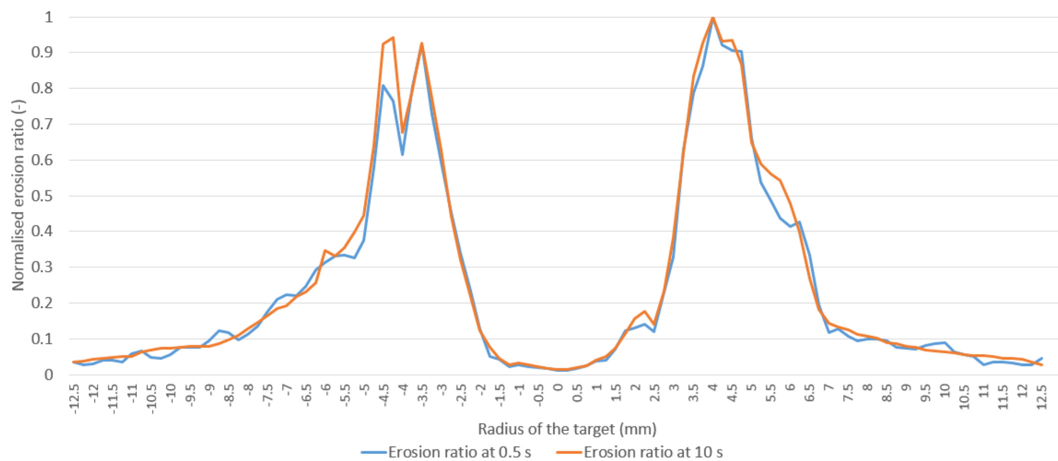


Figure 6.18: Comparison of the normalised erosion ratio over the radius of the probe (mm)

6.4.3 Influence of the rebound model

The computational erosion obtained by Nguyen et al in [60] incorporated a different erosion model than the default one in OpenFOAM. The model used was developed by Forder et al in [16]. Forder's model was implemented in OpenFOAM and a 10 s simulation was obtained for each model. The erosion contours obtained after 10 seconds are shown in figure 6.19. As can be observed, the wear scars obtained with both models are practically identical. The only difference between both scars relates to the magnitude of the erosion. This indicates that a simple model, which would also be less computationally expensive, would be sufficient to calculate erosion with enough accuracy for this configuration.

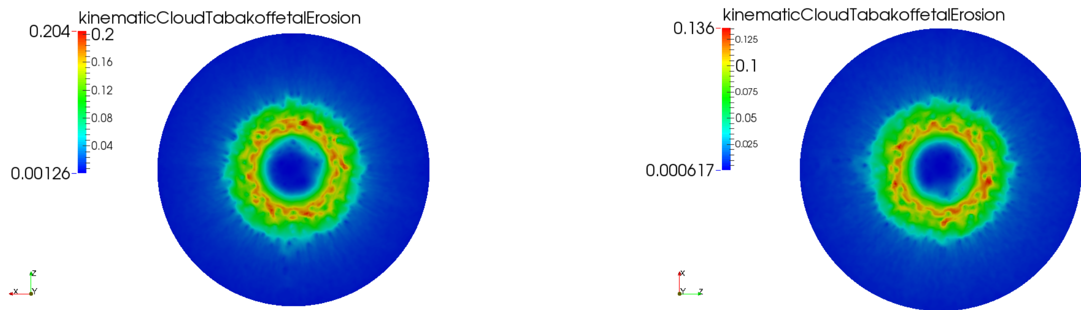


Figure 6.19: Contours of erosion per unit mass of impacting particles for the same erosion model [84] and different rebound models. Forder et al [16] (left) and OpenFOAM's default rebound model (right)

6.4.4 Validation of the 3-dimensional wear scar

An equivalent case to that one of Nguyen et al [60] was set up for validation. Steady state results were computed first and after that, an Euler-Lagrange simulation was run in OpenFOAM in order to calculate erosion induced by solid particle impingement. The parameters of the simulation were set to be the same as the ones used by Nguyen et al in [60]. The formula used for prediction of the erosion contours was developed by Tabakoff et al in [84] and implemented in OpenFOAM. This model was run alongside other models such as the ones developed by Menguturk et in [85] (also discussed in [86] and [87]) and Nandakumar et al [31] producing for all of them very similar erosion profiles and only differing in magnitude. The mesh deformation is applied after 10

seconds of simulation. Once the erosion ratios were calculated, the scar was scaled so that the maximum depth was 542 μm and the steady state was calculated again. The scaling factor which corresponds to 542 μm of depth was 0.0349 for the erosion contours obtained with the proposed formula. Additionally, different scaling factors were applied in order to analyse the evolution of the fluid flow during the steady state erosion. A radial average of the wear scars was obtained for each scaling factor and the different profiles are shown in figure 6.20

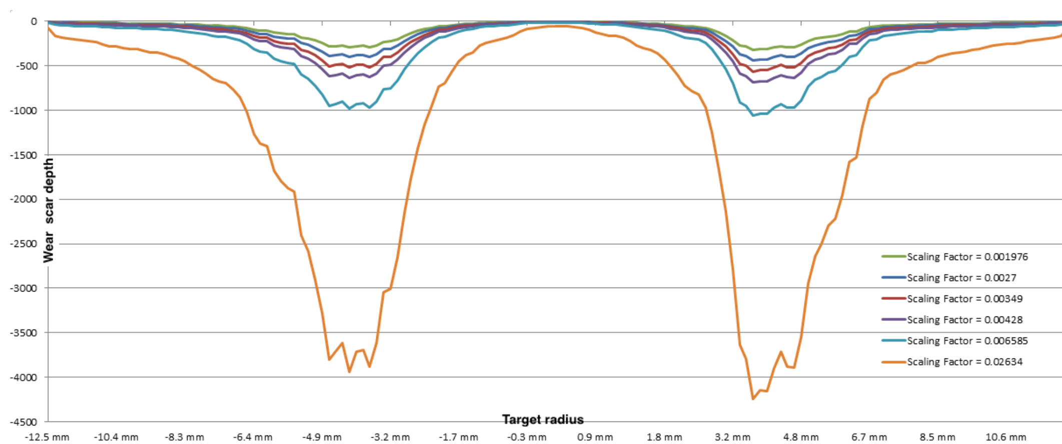


Figure 6.20: Wear scar profile depth comparison (μm) along the radius (mm) for different scaling factors

A quantitative comparison between the wear scar profile obtained with simulations and the ones measured by Nguyen et al is shown in Figure 6.21. It was also found that, as the scar progresses, the new stagnation point is captured, validating the deformation algorithm. As opposed to the work developed by Nguyen et al shown in 2.12, where the wear scar is 3D scanned and introduced into the CFD software again, the results here were obtained entirely computationally after 10 s of simulation . The initial contours of velocity and pressure are shown in figures 6.22 and 6.23 respectively while the same are shown in figures 6.24 and 6.25 respectively for the deformed geometry after being eroded for a value of the scaling factor of 0.00349.

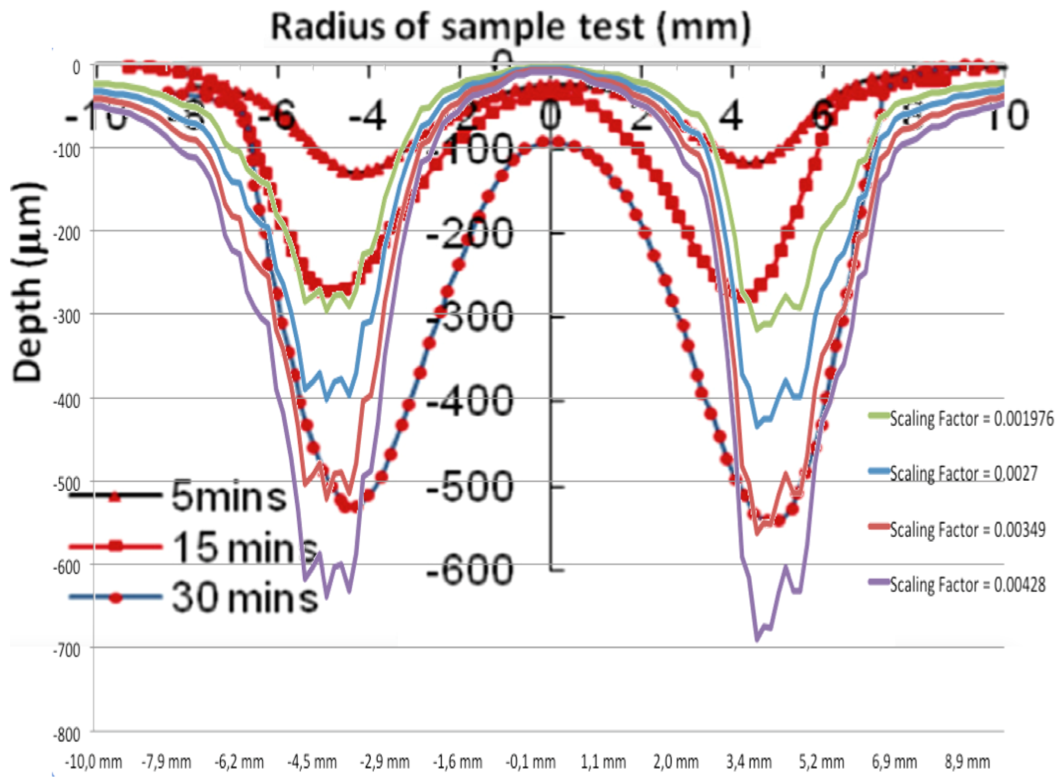


Figure 6.21: Wear scar profile comparison with the experimental scars measured by Nguyen et al in [60]

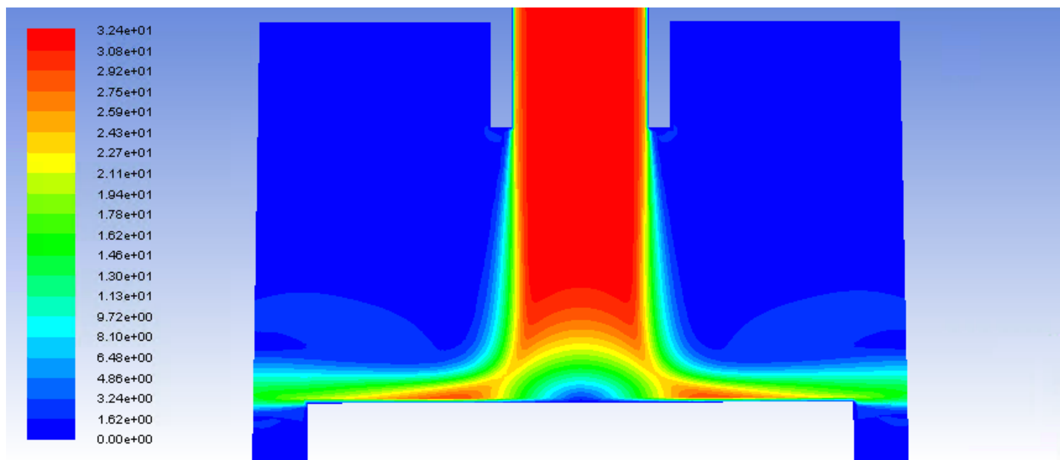


Figure 6.22: Velocity contours of the uneroded geometry ($\frac{m}{s}$)

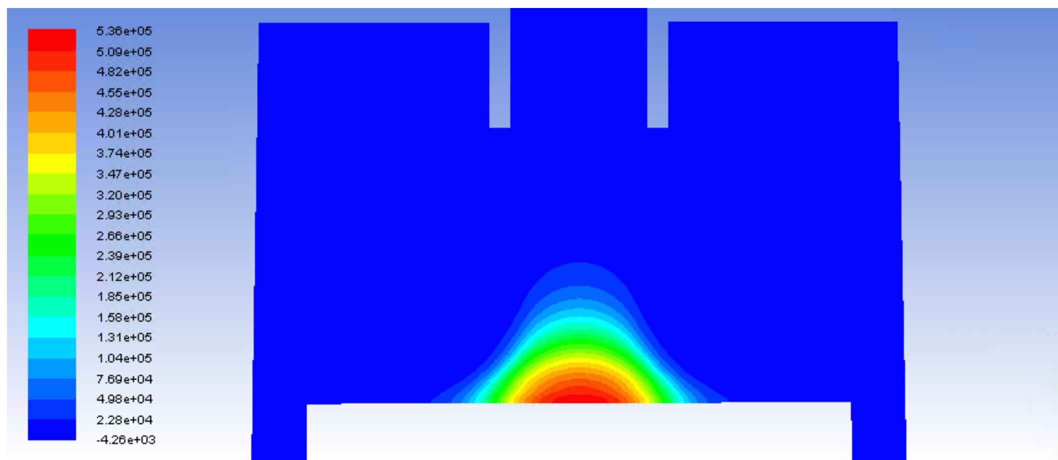


Figure 6.23: Static pressure contours of the uneroded geometry(Pa)

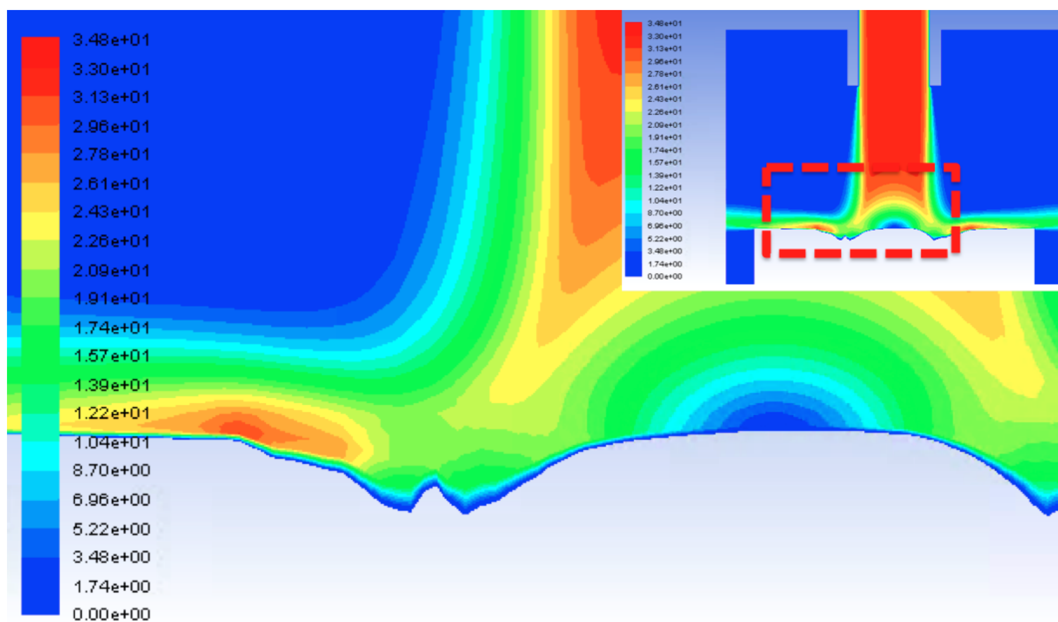


Figure 6.24: Velocity contours of the eroded geometry ($\frac{m}{s}$) for a scaling factor of 0.00349

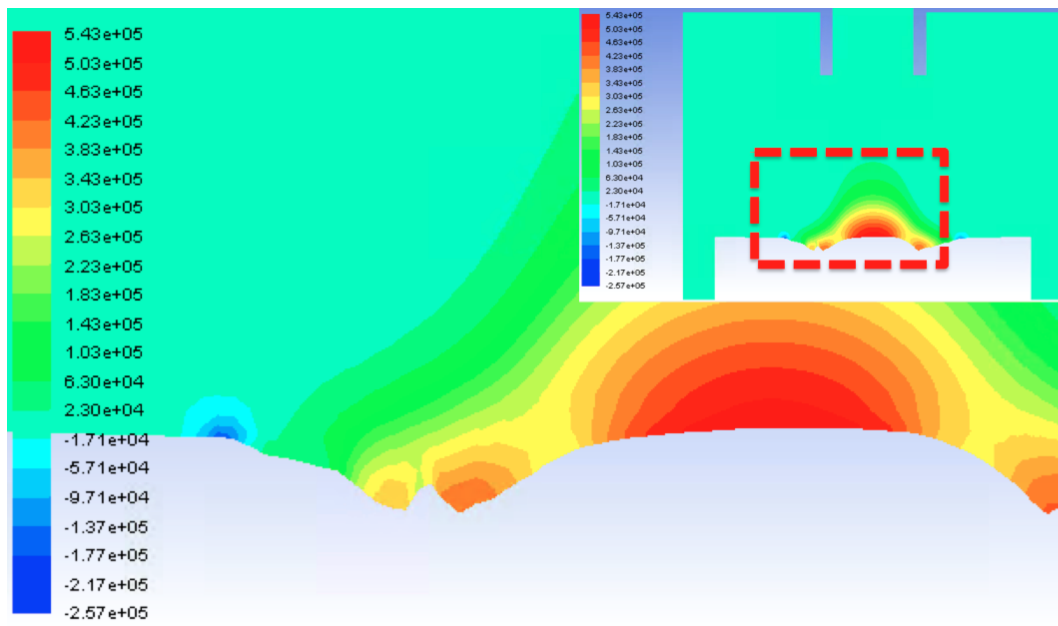


Figure 6.25: Static pressure contours of the eroded geometry(Pa) for a scaling factor of 0.00349

Additionally, the same two images are shown for all the different scaling factors analysed in figure 6.26-6.31.

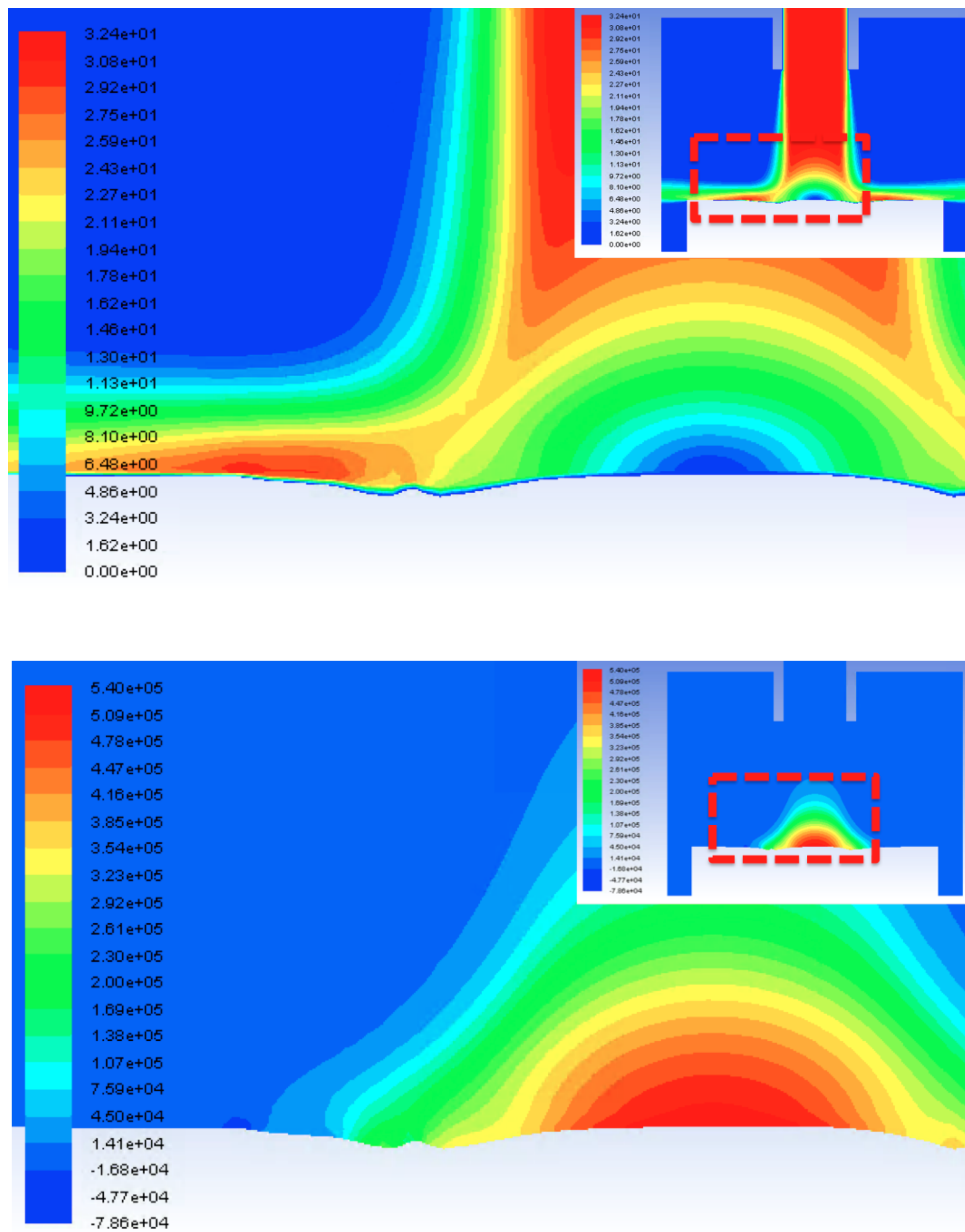


Figure 6.26: Velocity ($\frac{m}{s}$, left) and pressure contours (Pa, right) for scaling factor = 0.001976

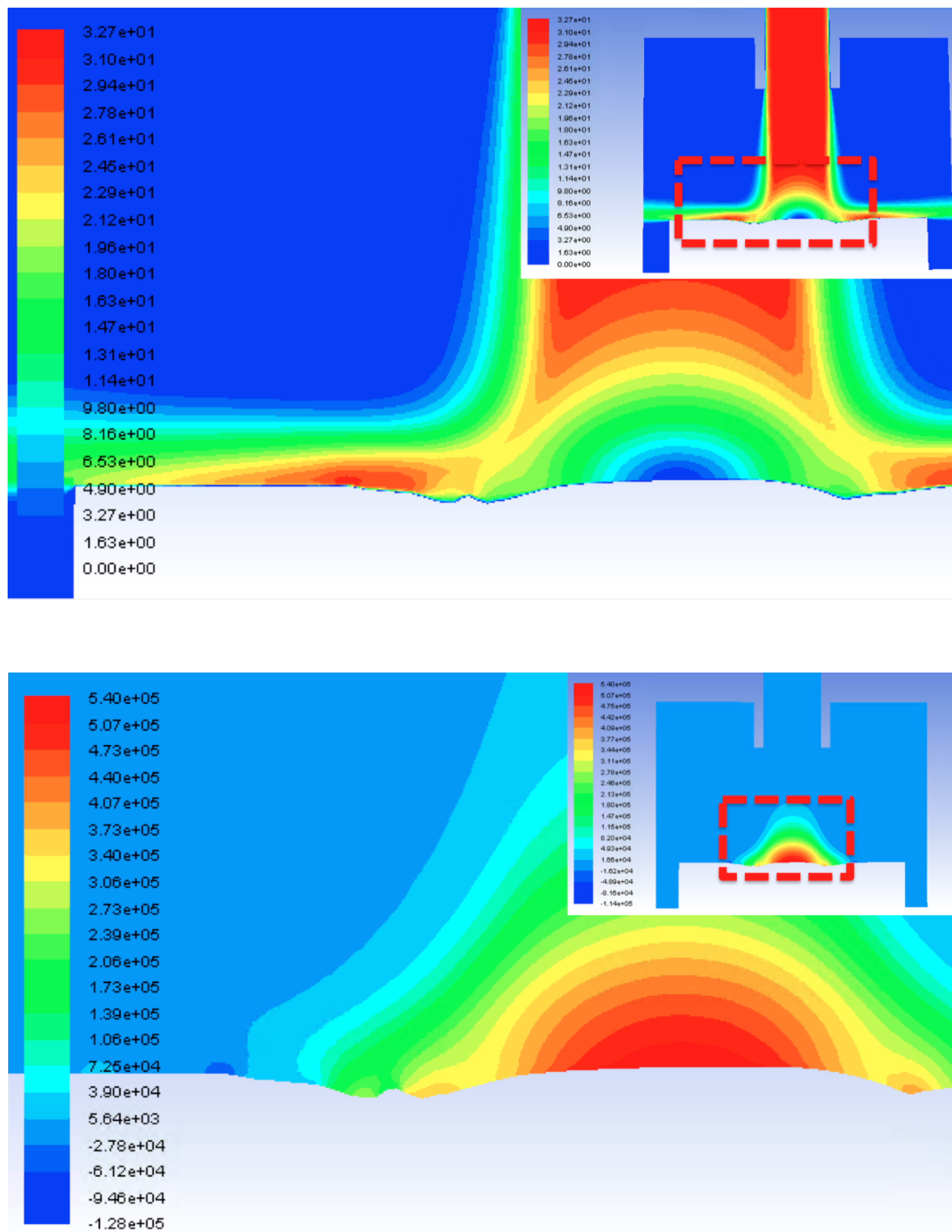


Figure 6.27: Velocity ($\frac{m}{s}$, left) and pressure contours (Pa, right) for scaling factor = 0.0027

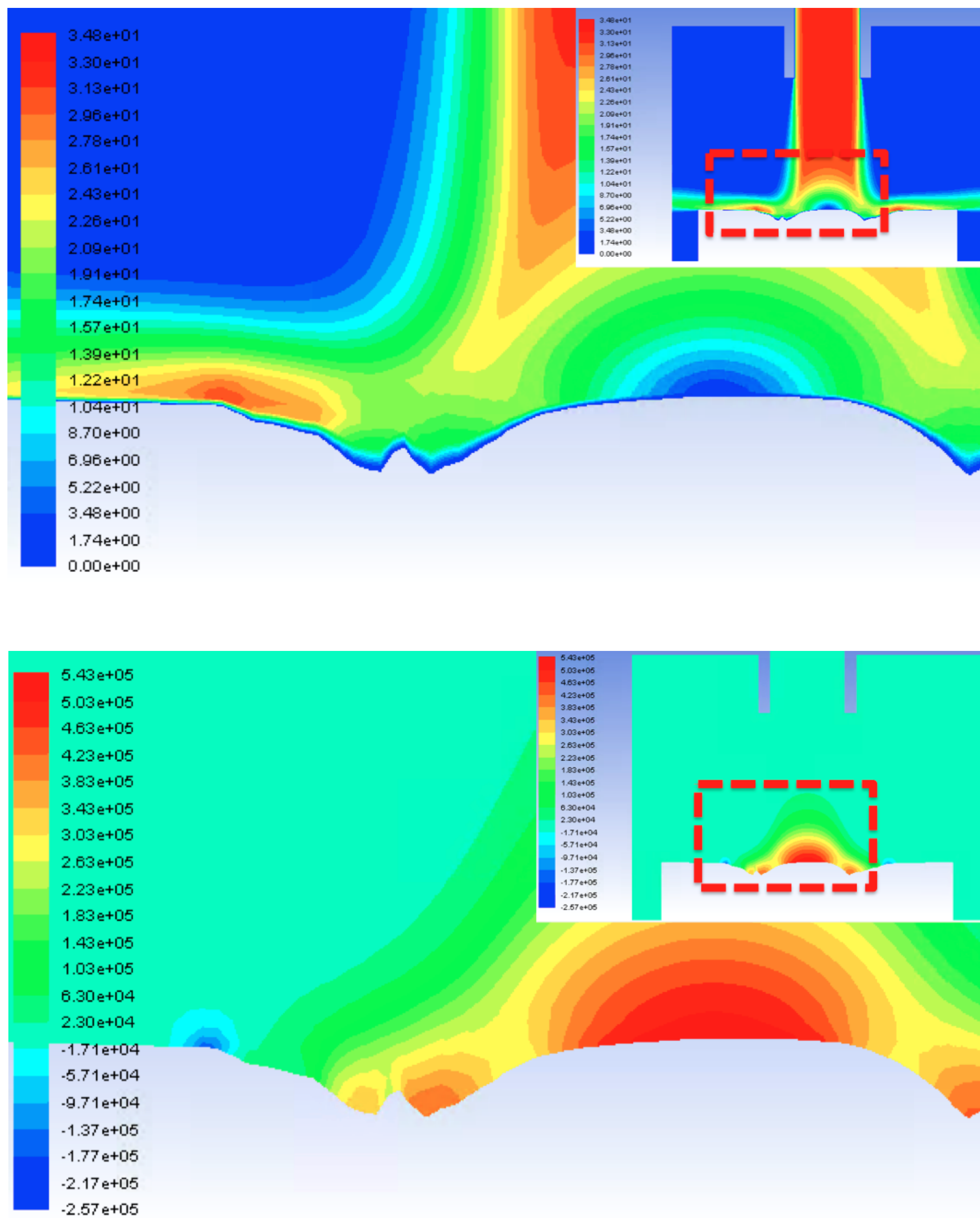


Figure 6.28: Velocity ($\frac{m}{s}$, left) and pressure contours (Pa, right) for scaling factor = 0.00349

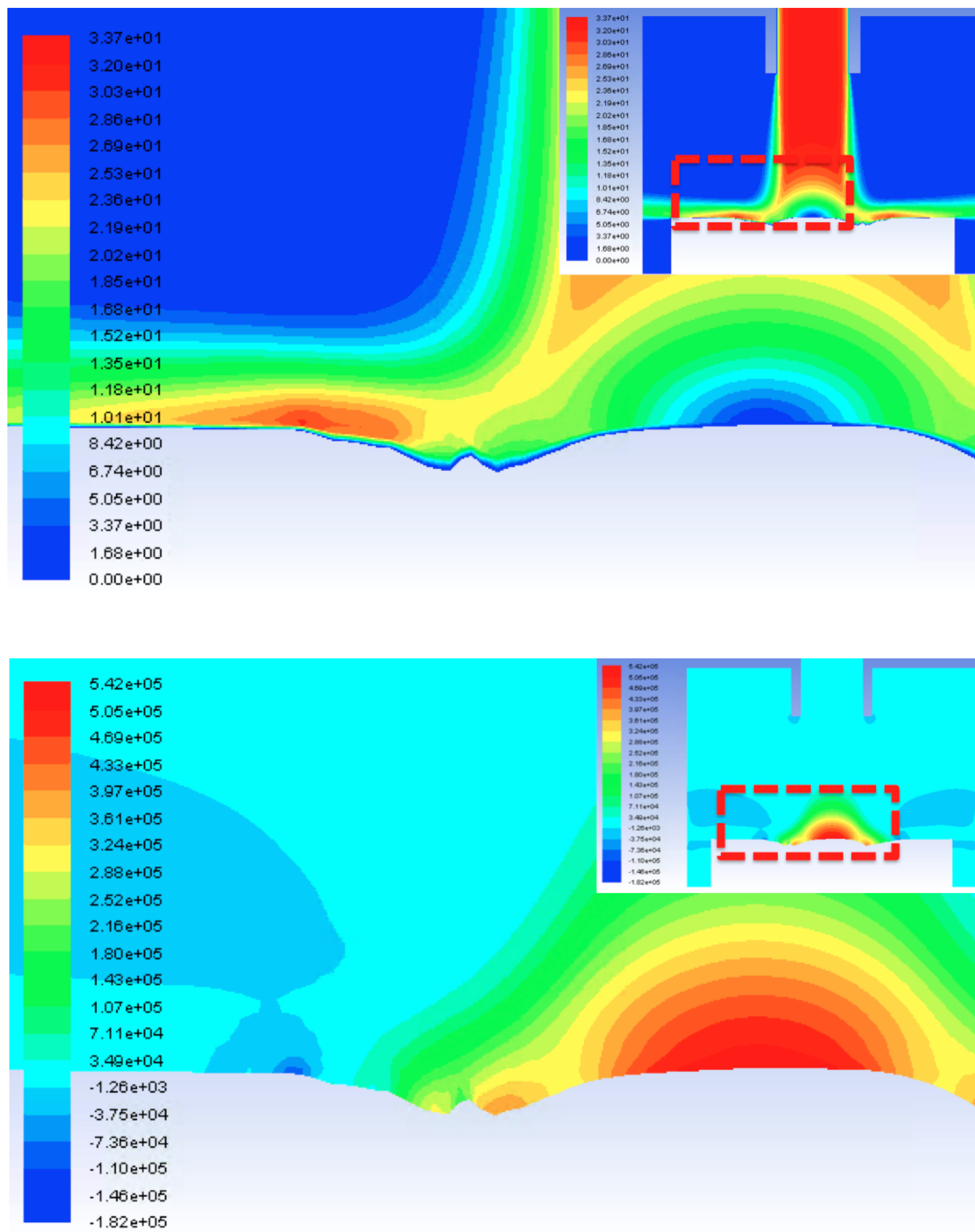


Figure 6.29: Velocity ($\frac{m}{s}$, left) and pressure contours (Pa, right) foscaling factor = 0.00428

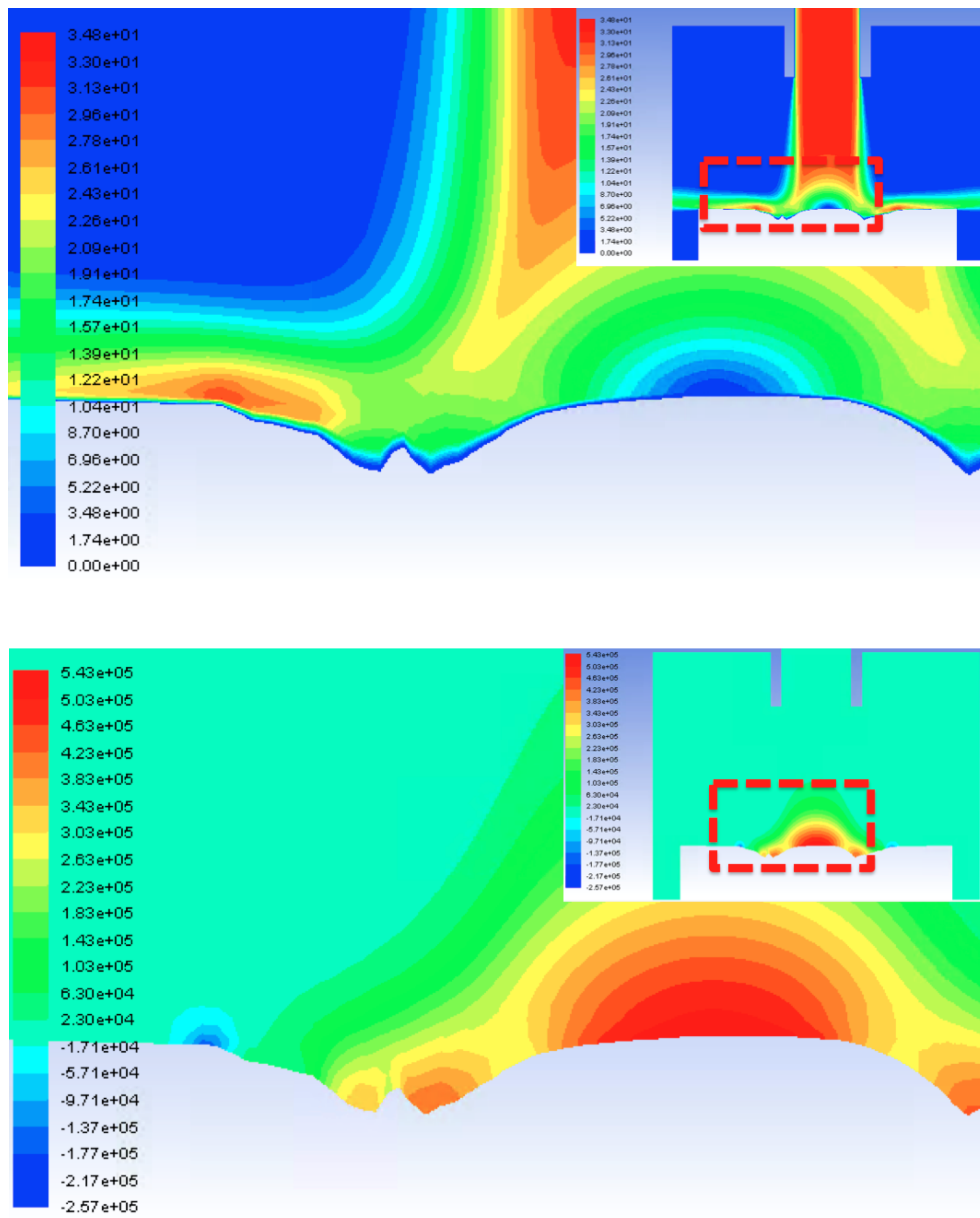


Figure 6.30: Velocity ($\frac{m}{s}$, left) and pressure contours (Pa, right) for scaling factor = 0.006585

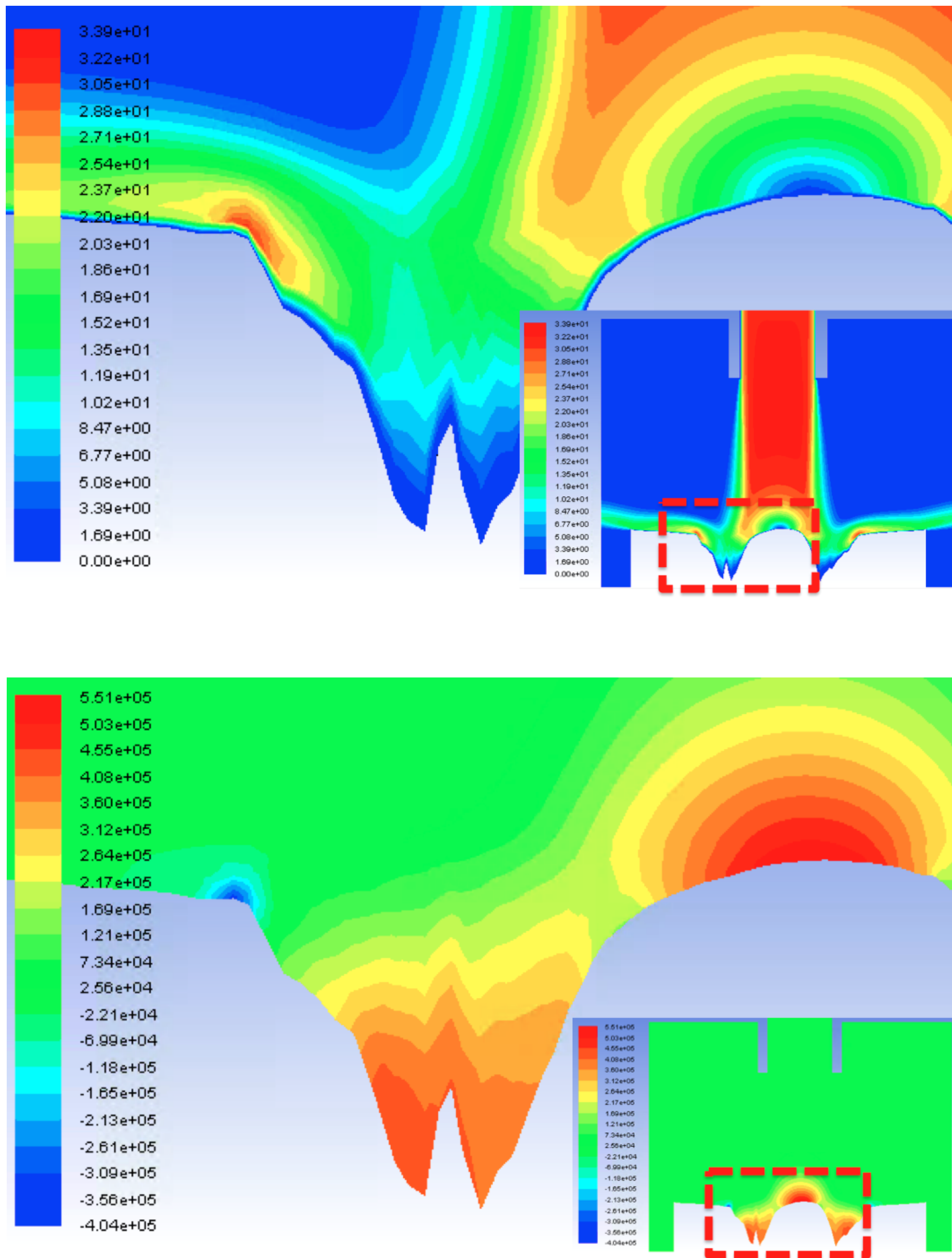


Figure 6.31: Velocity ($\frac{m}{s}$, left) and pressure contours (Pa, right) for all the scaling factor = 0.02634

Figure 6.32 shows the different surfaces obtained after the mesh deformation for the same scaling factors.

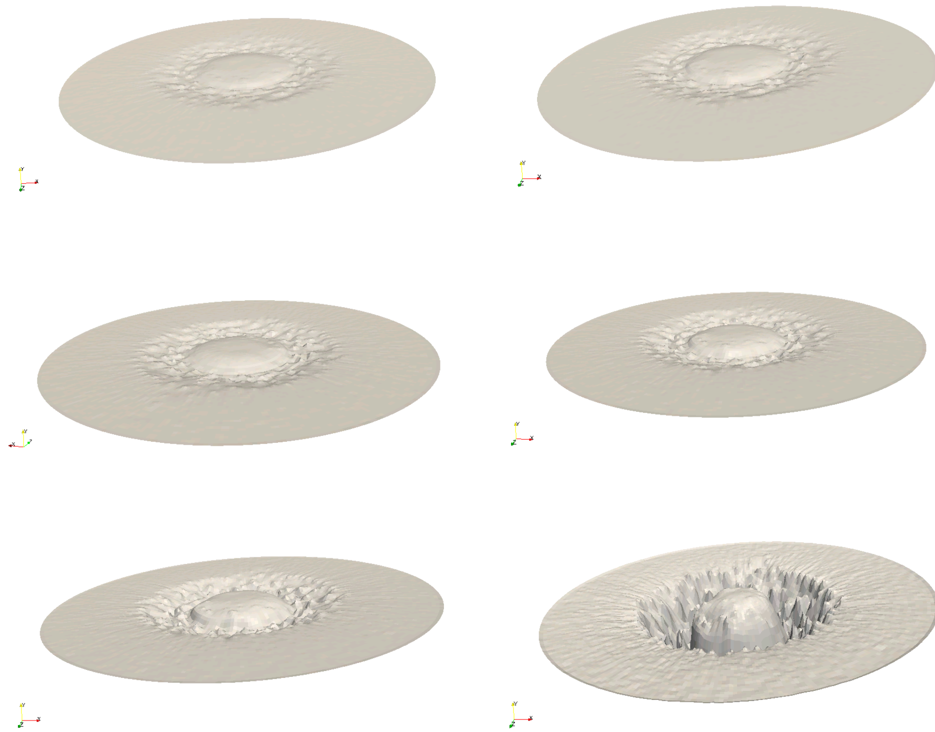


Figure 6.32: Surfaces obtained for all the scaling factors. From top to bottom and left to right: 0.001976, 0.0027, 0.00349, 0.00428, 0.006585 and 0.02634

Figure 6.33 shows the pressure contours at the deformed surfaces for all the scaling factors with different scales for the pressure. This figure seems to indicate that the stagnation point appears even before an equivalent depth to the experiment of Nguyen et al [60]. In the computational calculation, the first appearance of the stagnation point was detected at a scar obtained with a scaling factor which was 1.766 times smaller than that the one equivalent to the scar measured by Nguyen et al in [60].

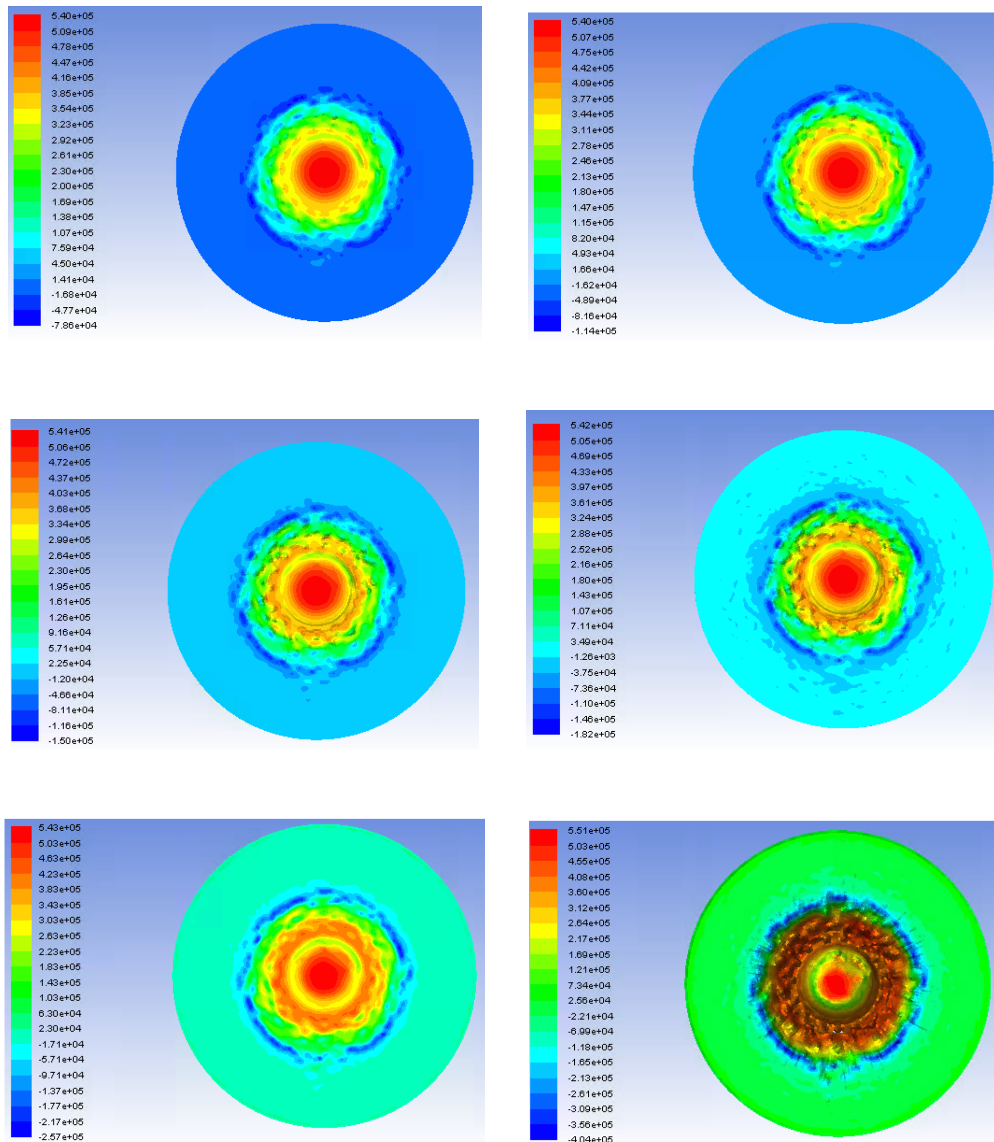


Figure 6.33: Pressure contours (Pa) at the surfaces for all the scaling factors. From top to bottom and left to right: 0.001976, 0.0027, 0.00349, 0.00428, 0.006585 and 0.02634

It is worth noting that the values of the pressure at the stagnation area increase as the wear scar progresses while the location of the stagnation point doesn't change its relative location. However, the maximum velocity generated by the new scars significantly changes it's value at around the same depth analysed by Nguyen et al in [60], which corresponds to a scaling factor of 0.00349. After that point, it fluctuates between different values, not increasing any further. Therefore, it is predicted that the fluid flow

changes would be significant enough to affect particle trajectories once the wear scar is around 540 μm in depth. These results confirm the validity of the approach for the wear scar obtained by Nguyen et al in [60] after 30 minutes of erosion. An analysis of the experimental scars obtained for different experiment durations would be necessary in order to assess how the shape of the wear scar changes as the fluid flow does.

6.4.5 Time-scaling

The proposed methodology introduces three different time-scales. These scales are optimised in order to calculate the deformed state of the geometry once eroded in the minimum amount of time possible.

Lagrangian particles

The Lagrangian time-scale is related to the time-step set in the solver in order to calculate particle trajectories accurately. In this type of simulations the time-step should satisfy a low enough courant-number so that the trajectories are calculated accurately. The lower the courant-number, the more accurately these will be calculated. In theory, the courant number should be kept below a value of 1. In the simulations shown in this work, the courant number was kept between 0.2 and 0.9. The Lagrangian time-step used in the simulations for validation was $1e^{-5}s$.

Erosion and mesh deformation

The time-scale related to both erosion rate and mesh deformation, will be dependent upon the number of impacts necessary in order to calculate the wear scar with the chosen level of confidence. The mesh deformation time-scale will have the same value, as the algorithm will be applied when the wear scar is accurate enough. In this case, as it was outlined in this section, this value is equal to 0.5 seconds.

Fluid flow

Finally, the fluid-flow time-step should have the same value as the erosion and mesh deformation one. The reason behind this is that only after the mesh is deformed, the fluid flow steady state has to be recalculated in order to compute the modified particle

trajectories. Regarding the number of iterations of fluid flow in order to achieve a new steady state, this will depend on the complexity of the geometry on which erosion is calculated.

6.5 Erosion calculation with a dynamic mesh solver

One of the possible techniques to avoid having an increasing size of the cells adjacent to the boundaries is to insert a dynamic mesh solver within the code that moves the points of these adjacent cells and creates new ones where required. Dynamic meshing with a laplacian solver and inverse distance interpolation was implemented in this case although there are other schemes available for dynamic meshing as well as interpolation. The features of this solver have been commented in 4.9 and the code is available in Appendix E. Figures 6.34 and 6.35 represent two series of images of the mesh and velocity contours obtained with this code for the case studied in [60]. In figure 6.34, it can be observed how, as deformation progresses, the whole mesh is adapted to the new shape of the deformed geometry. Figures 6.34 and 6.35 show the solution for the dynamic meshing and the velocity field at each second from left to right. It can be observed how the magnitude of the maximum velocity increases with scar depth. When the dynamic mesh is used, an additional dictionary is included where the number of fluid flow iterations after deforming the mesh is defined as well as other parameters such as the value determining when convergence is reached. In this case, that number was set to 100 and the values for determining when convergence was reached was set to 10^{-4}. The velocity field seemed to have converged well and no inconsistencies were found. However, the pressure field, which is not shown, didn't seem to have reached convergence at each deformation step since its values differed considerably from one time-step to the next one. This could be addressed by increasing the number of iterations for better convergence, which would yield a smooth transition of the pressure field between deformation steps. This solver allows automating the process of erosion calculation, provided that the three different time-scales are introduced in the controlDict and erosionDict dictionaries for the particle time-step and mesh deformation respectively.

It is also worth noting that, the smaller the deformation steps and the higher the damage

each particle causes on the surface, the more irregular the deformed surface becomes, as evidenced by figure 6.36. These surfaces were obtained by increasing the damage the particles cause on the surface and deforming the mesh at every time-step, yielding a very uneven geometry which eventually cased divergence. A complete sequence of these images was developed and presented in [88] by Lopez et al.

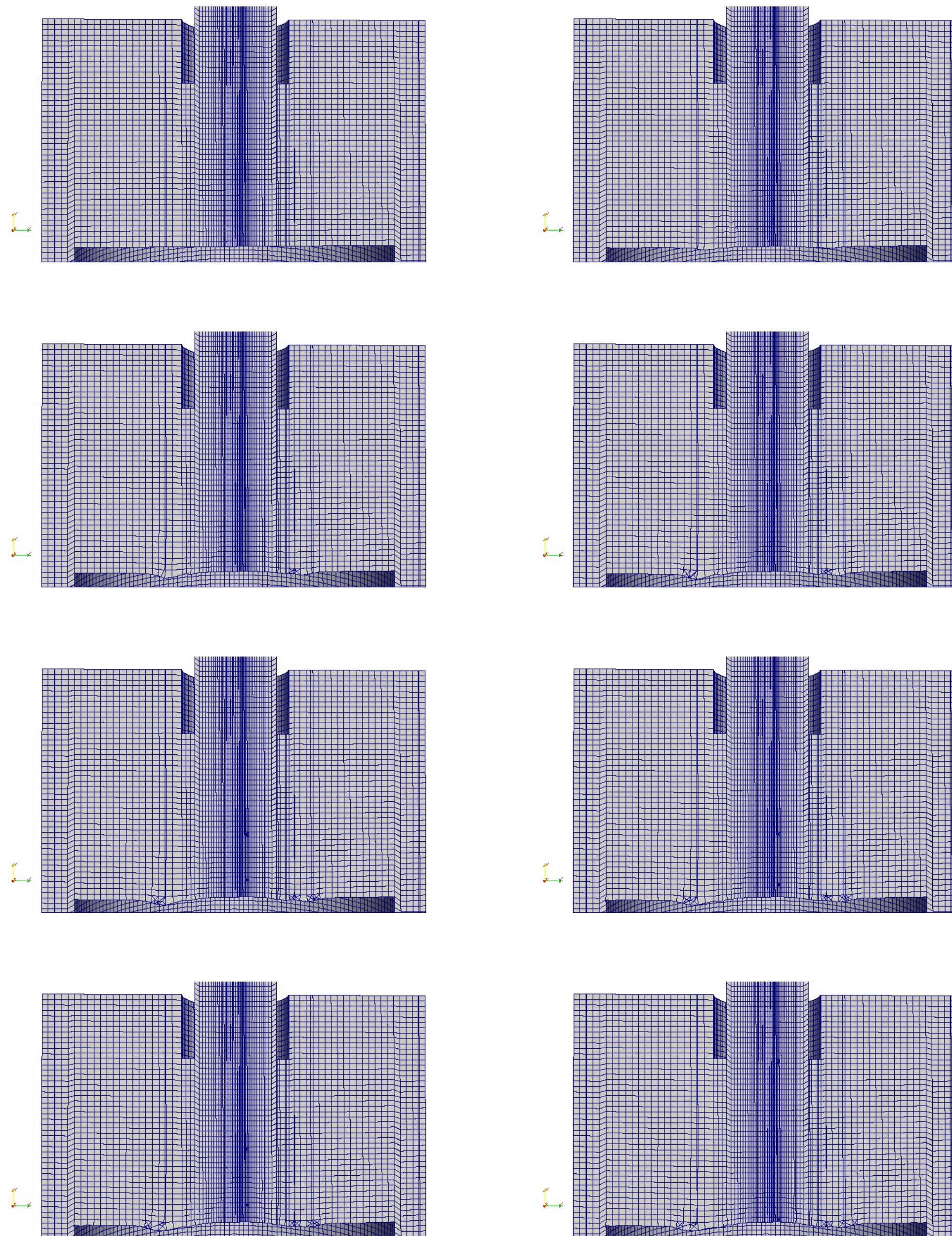


Figure 6.34: Progressive mesh deformation and result of the dynamic meshing

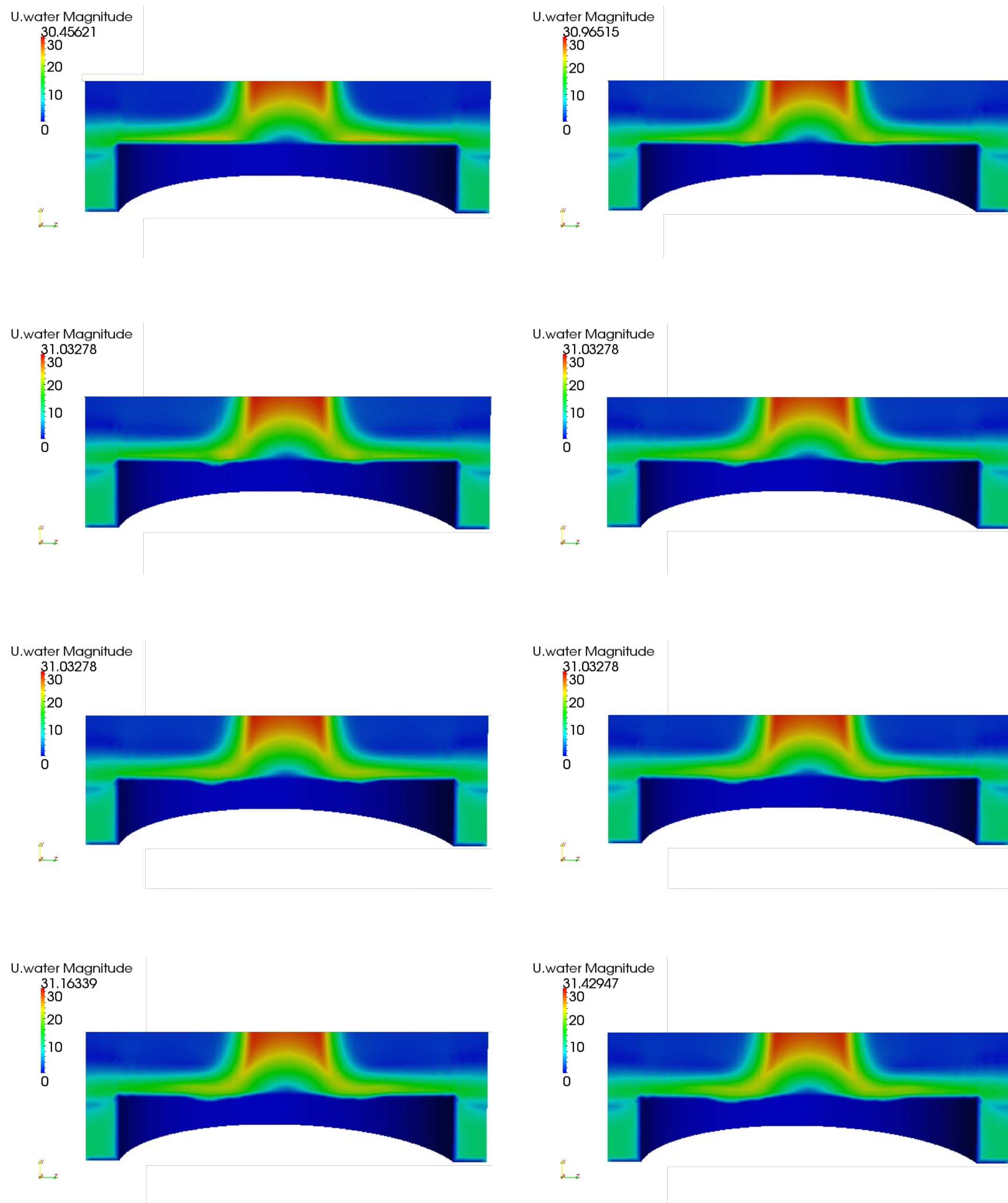


Figure 6.35: Progressive mesh deformation results for the velocity contours

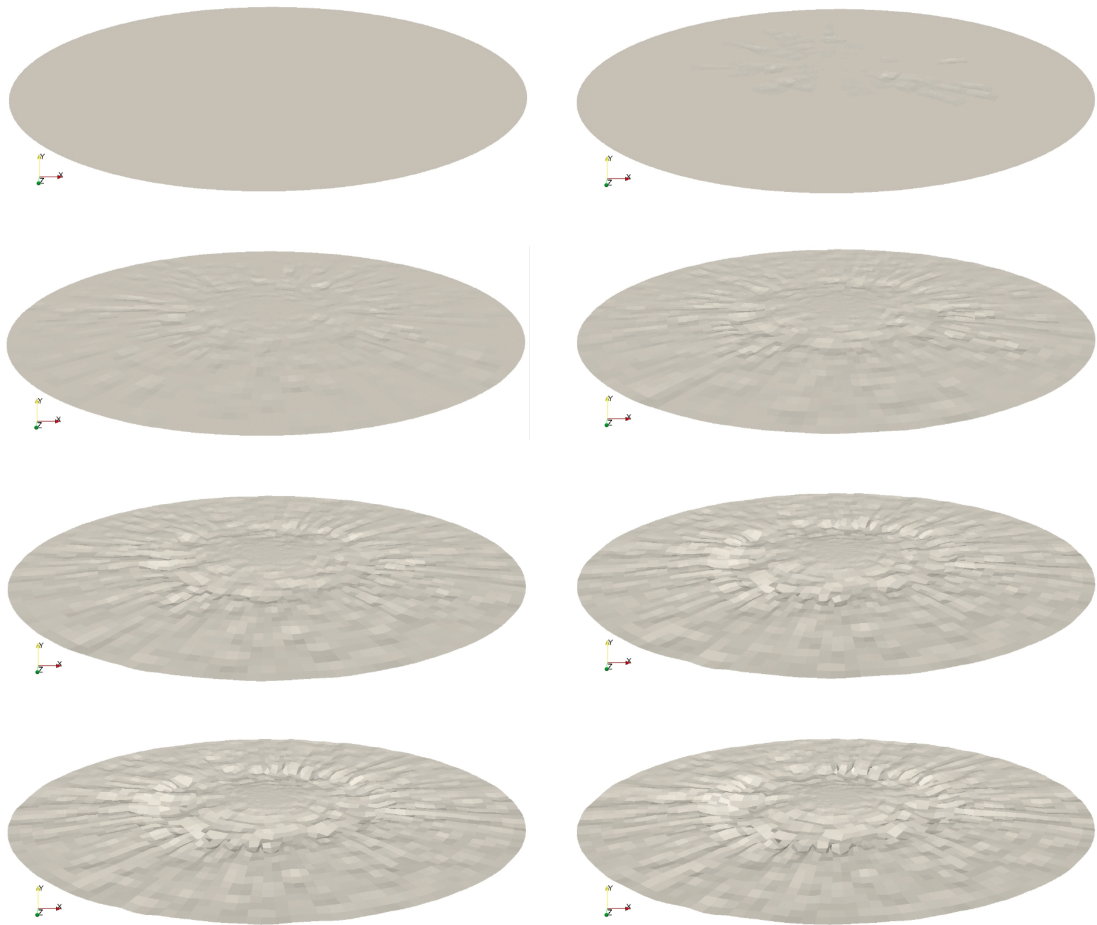


Figure 6.36: Progressive surface deformation with high damaging particles

6.6 Three dimensional implementation of the Wear Map Method

In the work by Gnanavelu et al [9, 14], a methodology is proposed to calculate the erosion rate based on both experimental and computational methods. The CFD methodology in this paper was analysed in chapter 3 and in an article published by Lopez et al [24]. In section 4.8, a methodology was developed for averages calculation of impact angles and velocities. The aim in this section is to implement the Wear Map Method in a three dimensional configuration based on the experimental results published by Nguyen et al in [60]. The erodent used in [60] were angular sand particles with an averaged shape factor of 0.58 and with a concentration of 0.5% by volume, while the

material used in the samples was stainless steel (SUS304).

6.6.1 Equation fitting

Several tools and different fitting methodologies [89, 90] have been investigated. In this section, two of the most accurate ones are presented. In order to obtain the fitting surfaces, contours and residuals, Matlab's fitting toolbox was used [89]. The first step was to run the transient simulation in order to obtain averages for the velocity and the angle of impingement. The profiles along the radius of the sample were easily obtained using Paraview's "Plot over line" feature [91] and the results are presented in figures 6.37 and 6.38 for velocity and impact angle respectively and taken from the contour plots of the velocity and angle of impingement averages shown in figures 6.39 and 6.40. After that, the averages were matched to Nguyen's wear scar after 30 minutes [60] which is represented in figure 6.41 together with the wear scars after 5 and 15 minutes of test.

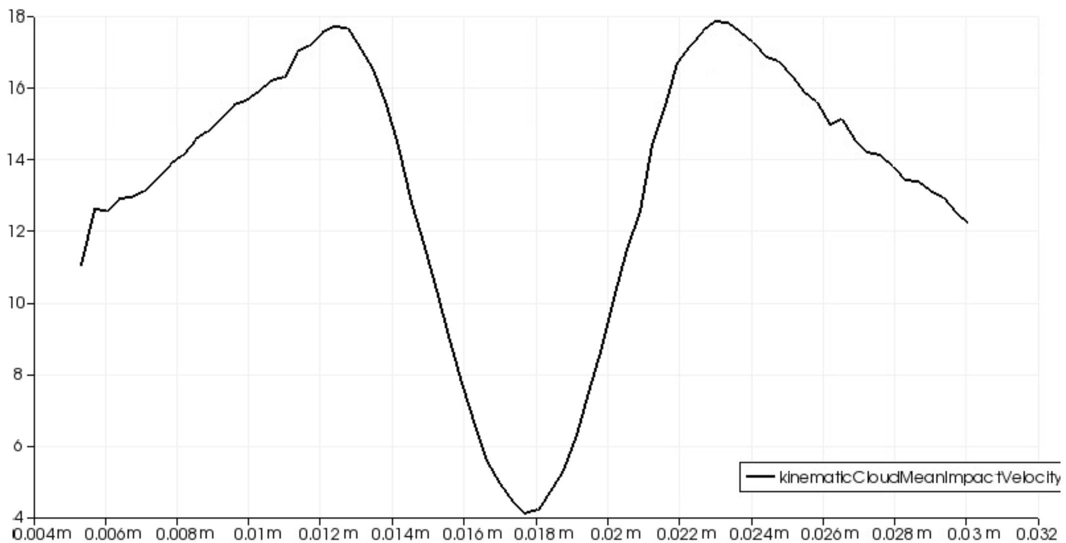


Figure 6.37: Impact velocity average across the radius of the test sample in $\frac{m}{s}$

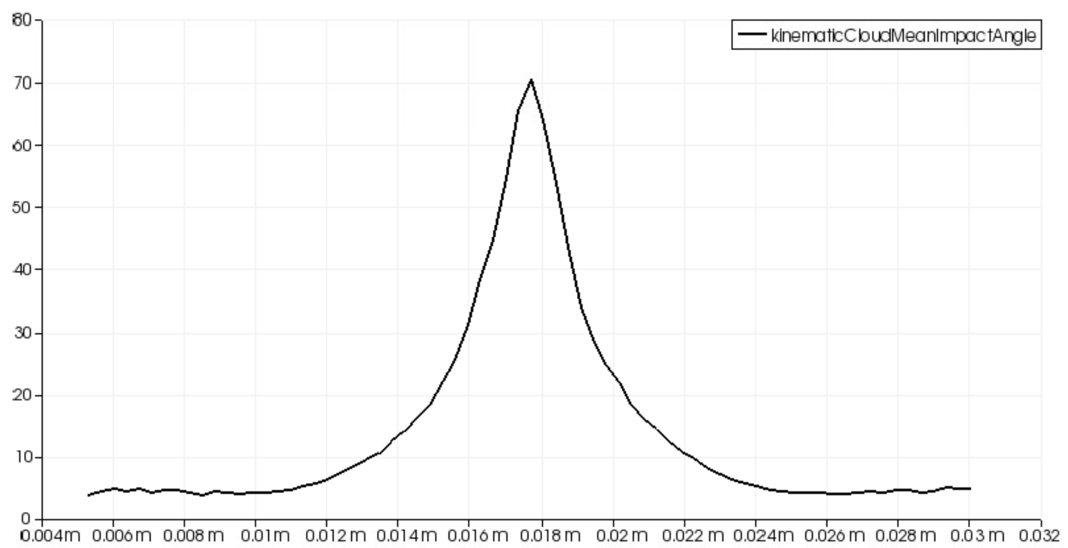


Figure 6.38: Impact angle average across the radius of the test sample in degrees

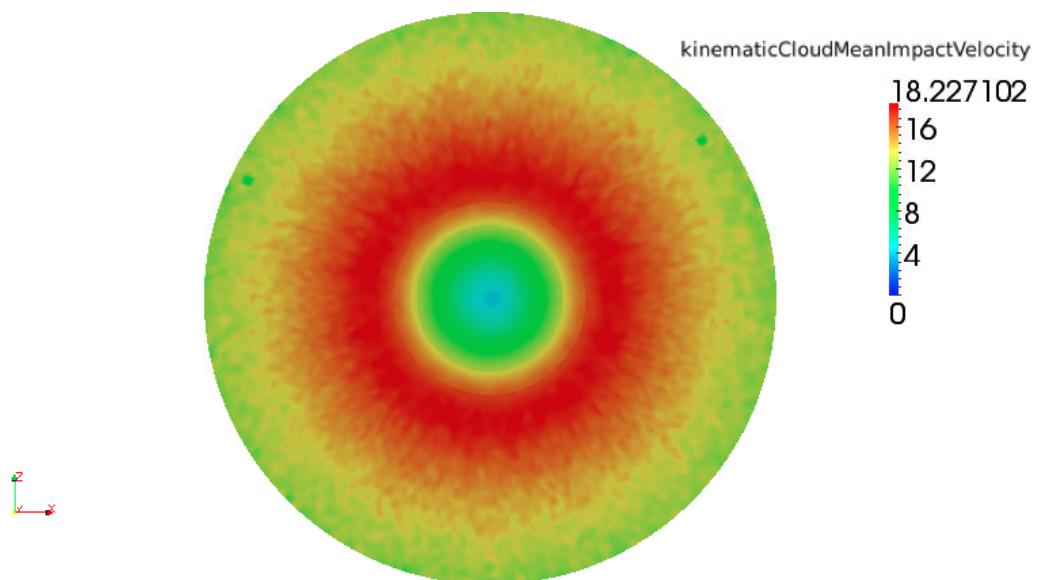


Figure 6.39: Velocity at impingement average in $\frac{m}{s}$

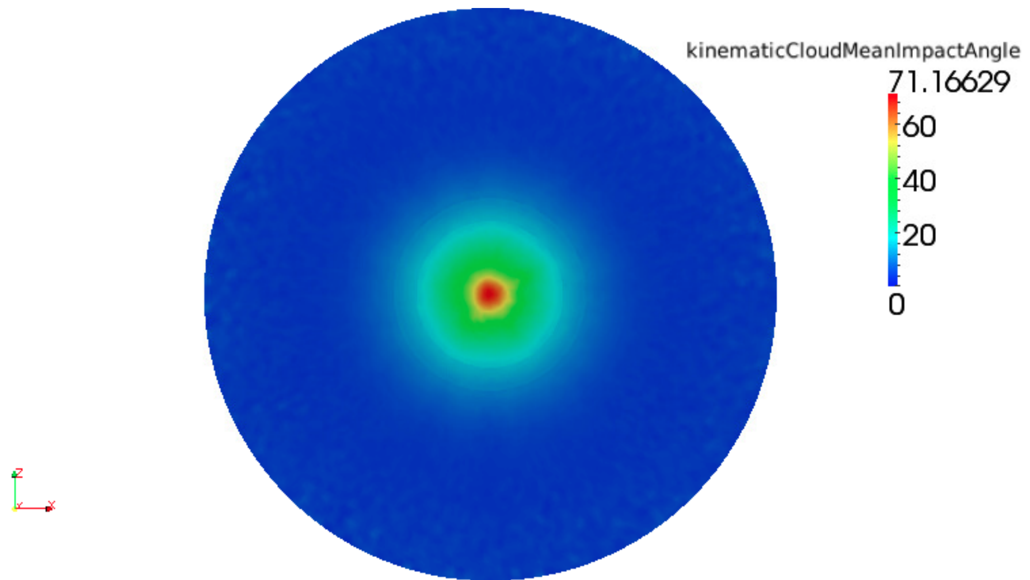


Figure 6.40: Impact angle average in degrees

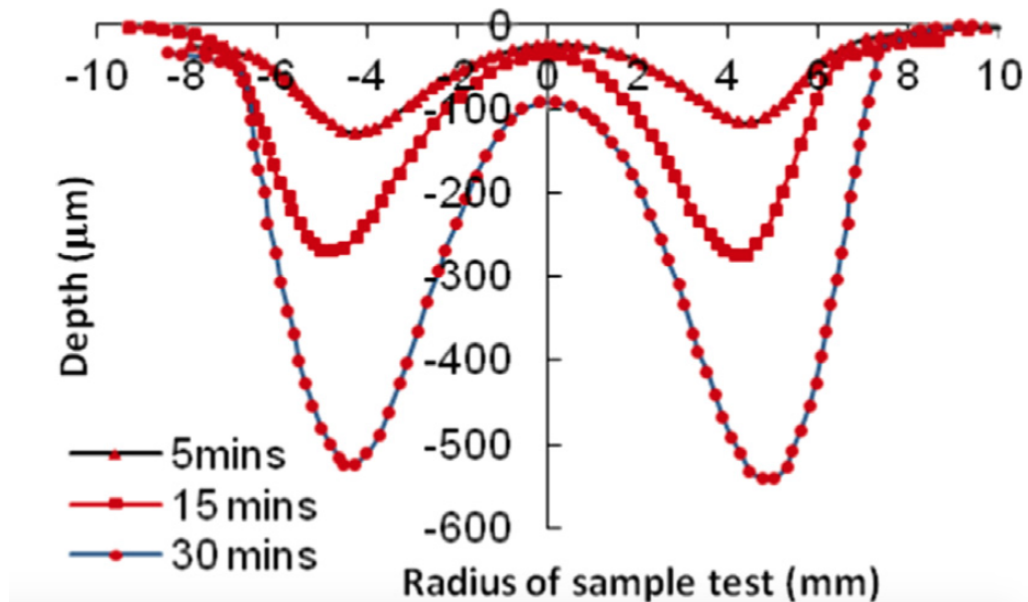


Figure 6.41: Wear scars after 5, 15 and 30 minutes of test [60]

Once the data for impact angle, impact velocity and wear scar depth were put together the wear map was calculated using Matlab.

Two different fits based on different number of points for the equation proposed by Gnanavelu et al in [14], represented in equation 6.1 were tried.

$$ER = V^2(Asin(\theta)^4 + Bsin(\theta)^3 + Csin(\theta)^2 + Dsin(\theta) + E) * F \tag{6.1}$$

Where ER is the erosion ratio (as defined in 2.1), θ is the angle of impingement and V is the velocity at impingement.

6.6.2 Equation fit with 120 points

The surface which represents the best fit for this case is shown in figure 6.42. The residuals after the fitting operation are plotted in figure 6.43 while the resulting Wear Map is shown in figure 6.44 including all the figures the points used for fitting the equation.

The coefficients obtained are represented in table 6.1 along with some parameters that show the goodness of the fit.

A	-4.747
B	10.11
C	-7.813
D	2.995
E	-0.18
F	11.19
SSE	1.375e+04
R-square	0.9939
Adjusted R-Square	0.9936
RMSE	10.98

Table 6.1: Results of the fit for 120 points

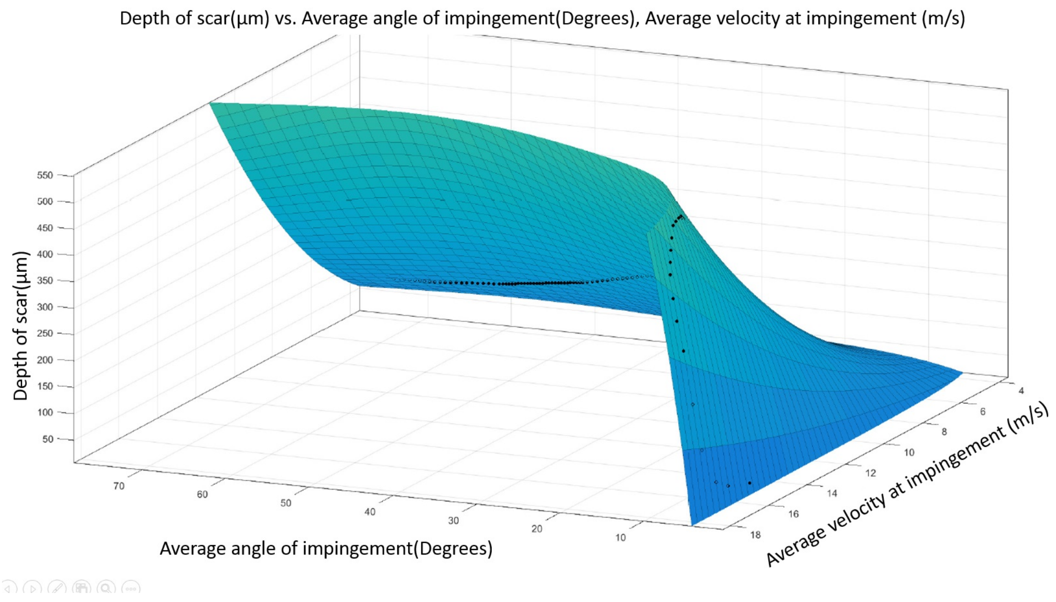


Figure 6.42: Surface fitting for the wear scar and CFD case in [60]

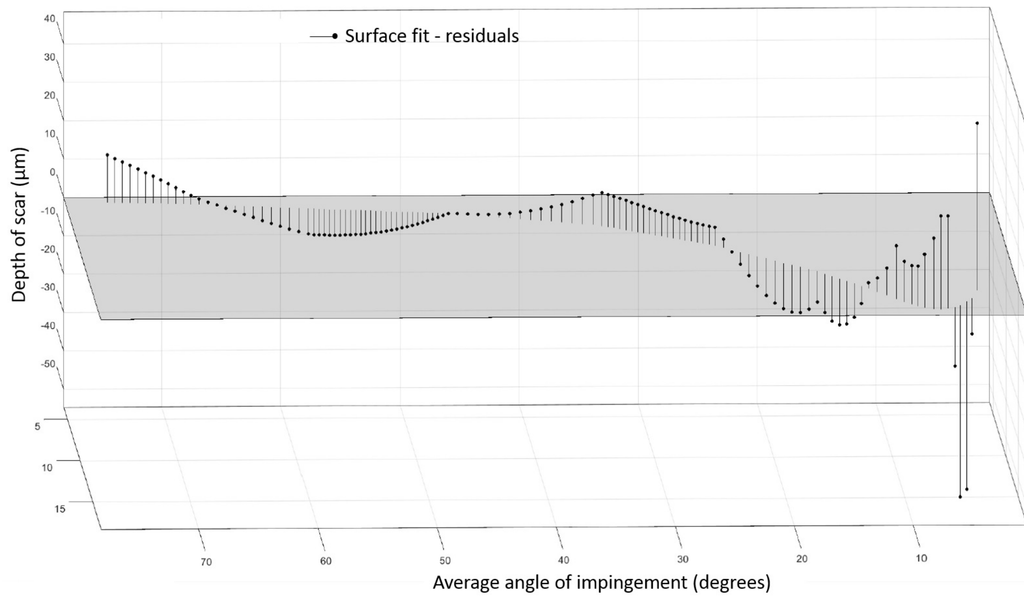


Figure 6.43: Residuals after fitting

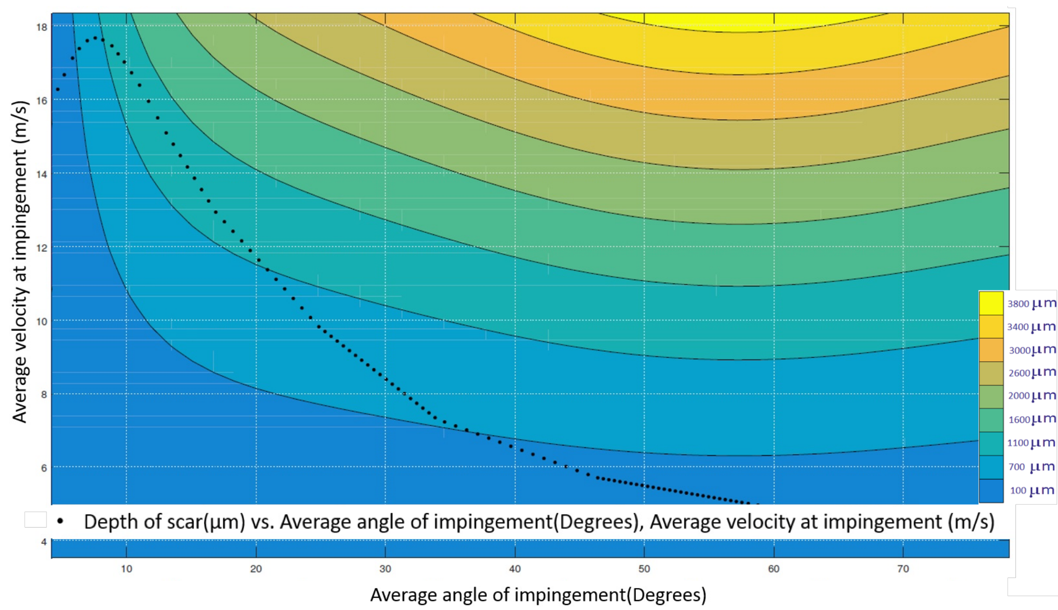


Figure 6.44: Wear map for the wear scar and CFD case in [60]

The equation obtained was implemented in OpenFOAM and the utility described in Appendix F was run in order to obtain the deformed geometry. Once the geometry was deformed, another utility (`refineWallLayer`) was used in order to create new layers at the specified boundary and a steady state reached again. The deformed geometry and the steady state results are shown in figures 6.45, 6.46, 6.47, 6.48, 6.49 and 6.50,.

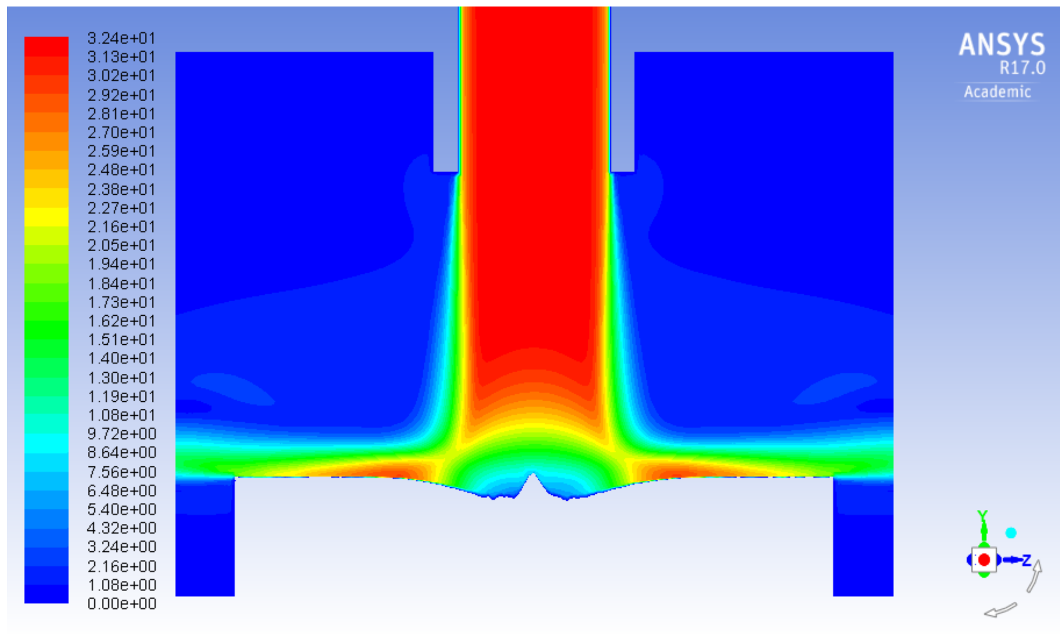


Figure 6.45: Velocity contours in $\frac{m}{s}$

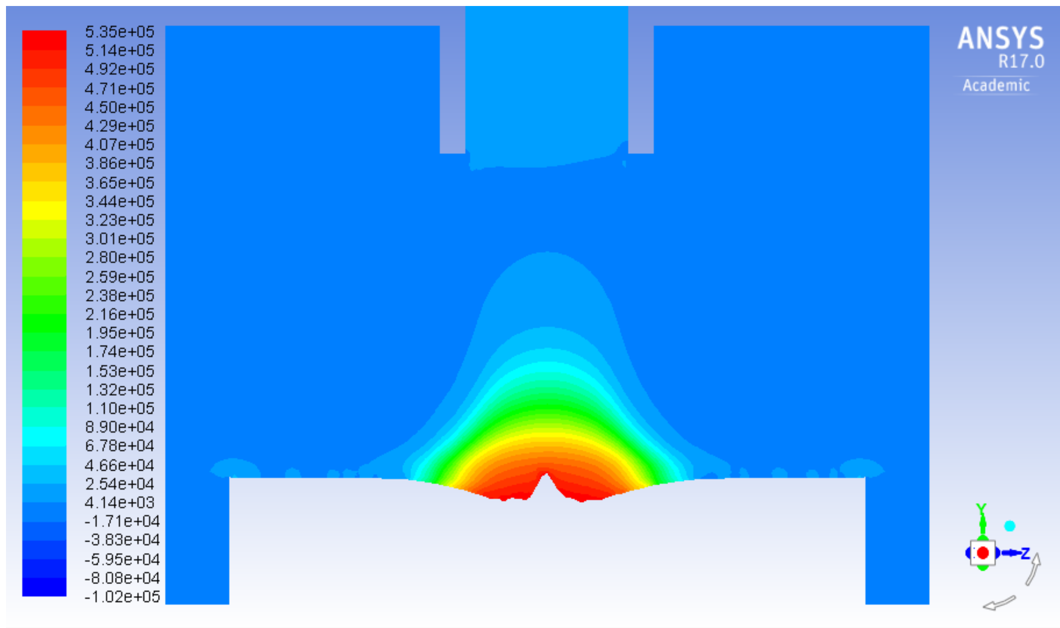


Figure 6.46: Static pressure contours in Pa

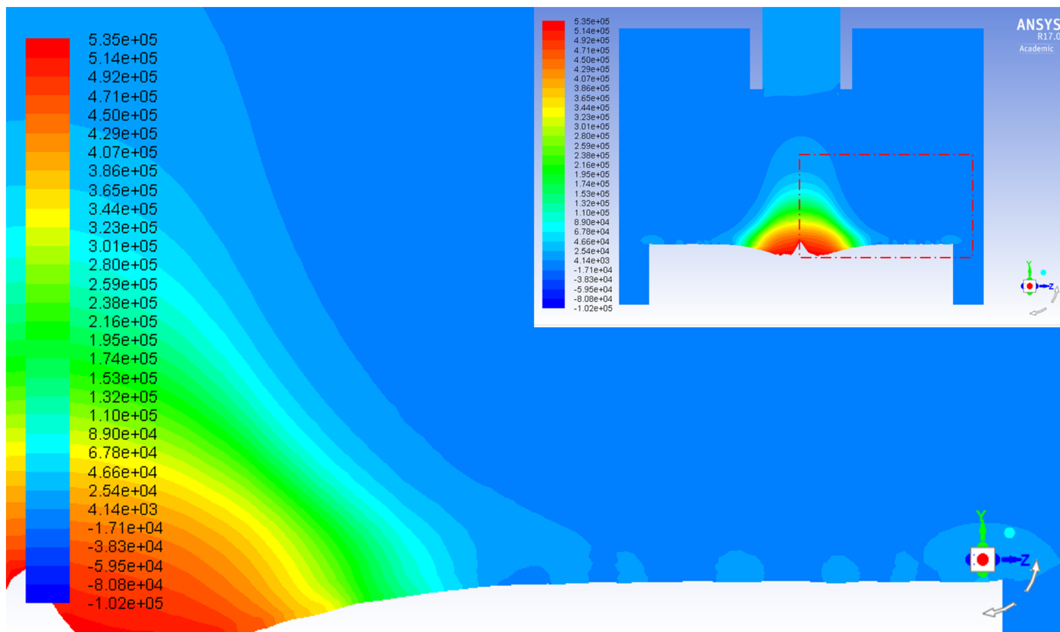


Figure 6.47: Static pressure contours in Pa showing the formation of a possible stagnation point

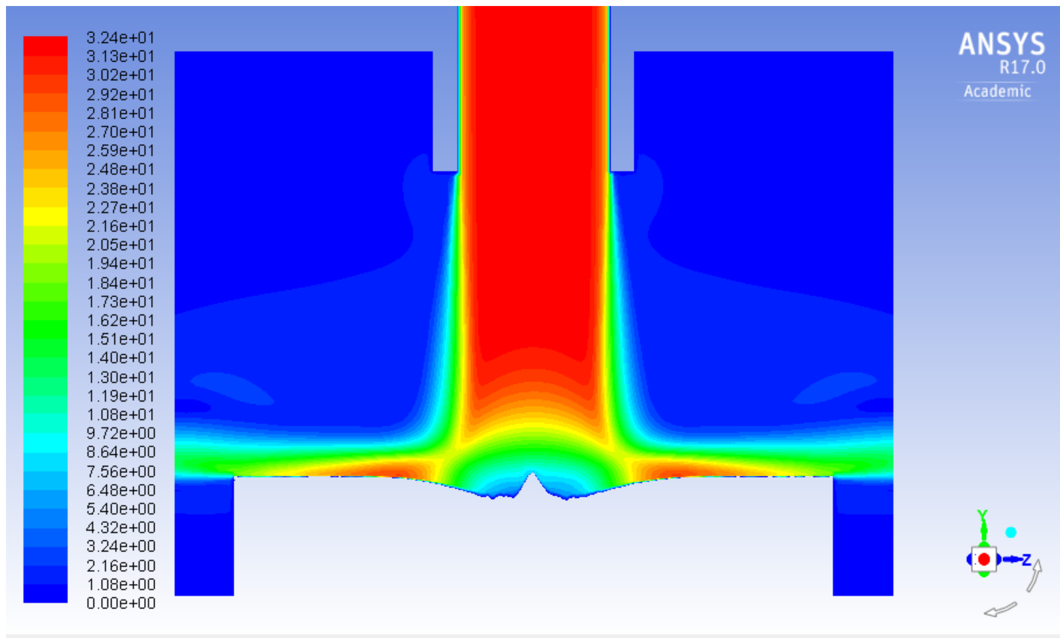


Figure 6.48: Velocity contours in $\frac{m}{s}$

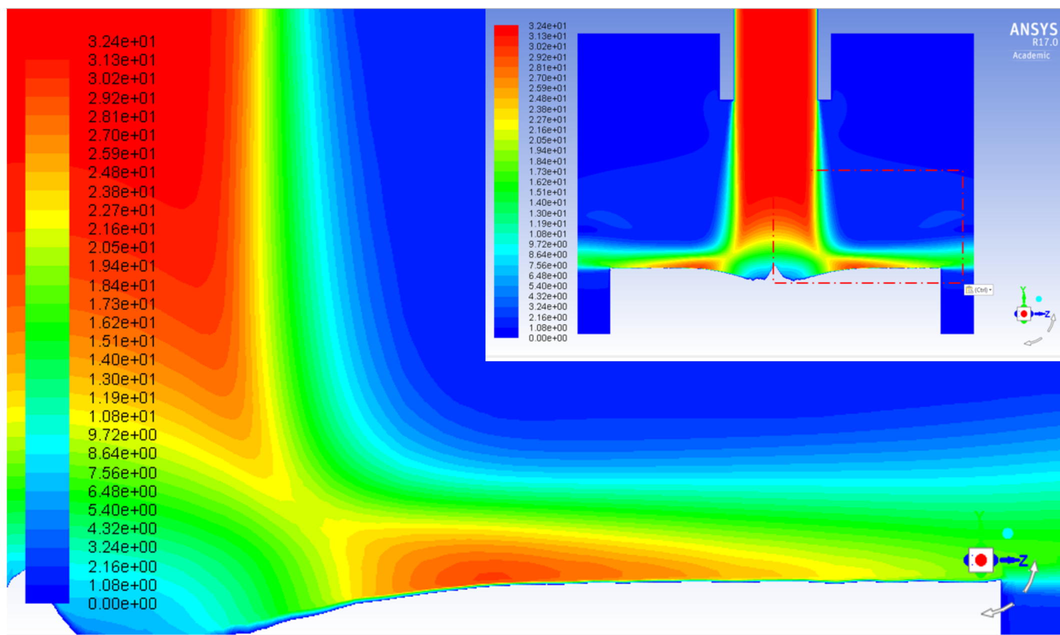


Figure 6.49: Velocity contours in $\frac{m}{s}$ for the edge of the wear scar

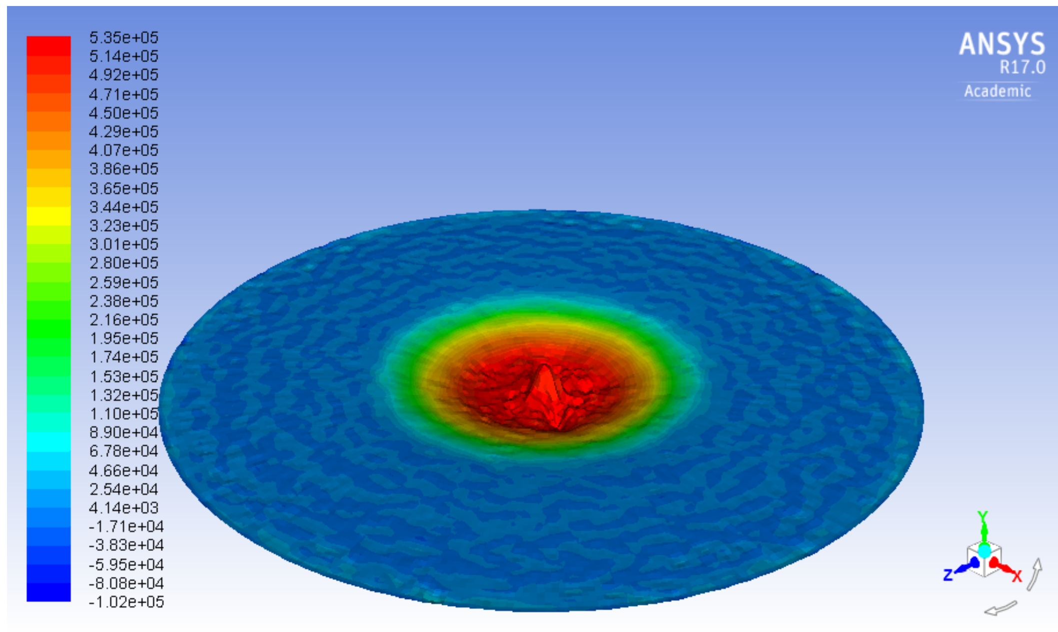


Figure 6.50: Static pressure contours in Pa at the target's surface

As figures 6.46 and 6.47 illustrate, a region with lower pressure indicating the formation of a stagnation point is found in the area where the velocity is higher. However, the central part of the scar is less accurate than the one commented in section 6.4. This may have to do with the accuracy of the fit, which is represented by the residuals in figure 6.43.

6.6.3 Equation fit with 24 points

The surface which represents the best fit for this second case with less points is shown in figure 6.51. The residuals are plotted in figure 6.52 and the resulting Wear Map is shown in figure 6.53. All figures include the points used in the fitting.

The coefficients obtained can be found in table 6.2 along with some parameters showing the goodness of the fit.

A	-7.994
B	18.3
C	-15.43
D	6.494
E	-0.3936
F	4.819
SSE	2.003e+04
R-square	0.9709
Adjusted R-Square	0.9623
RMSE	34.33

Table 6.2: Results of the fit for 24 points

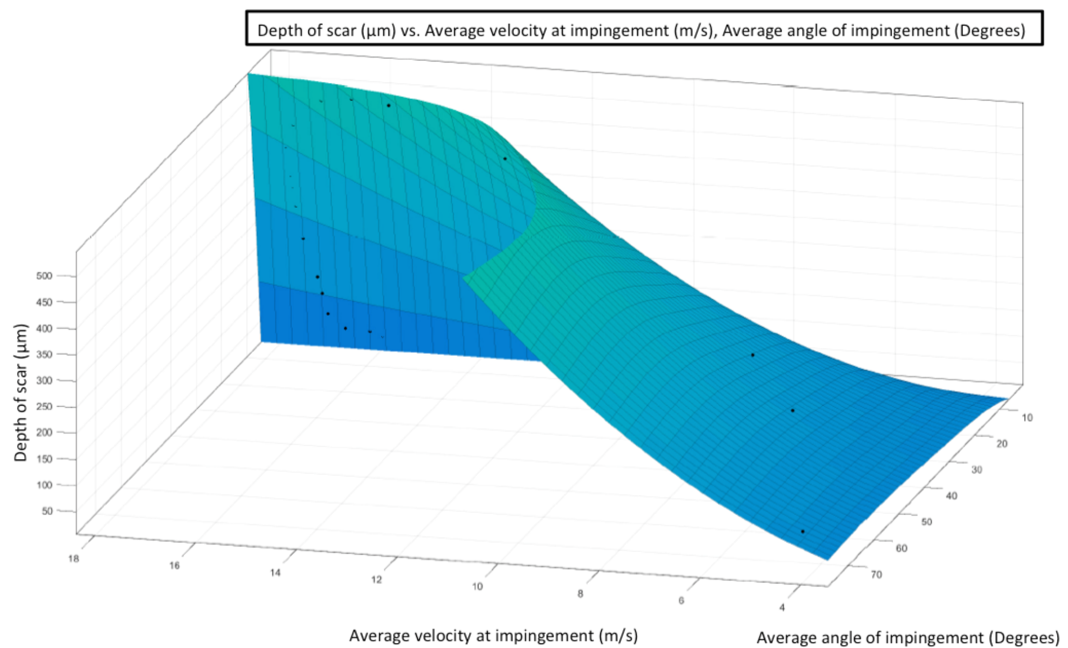


Figure 6.51: Surface fitting for the wear scar and CFD case in [60] with 24 points

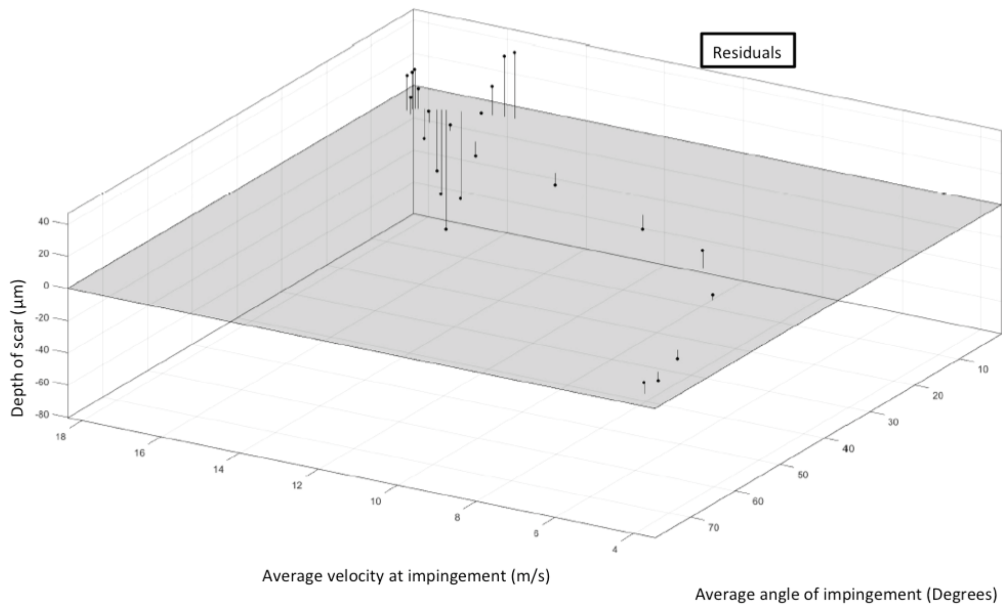


Figure 6.52: Residuals after fitting using 24 points

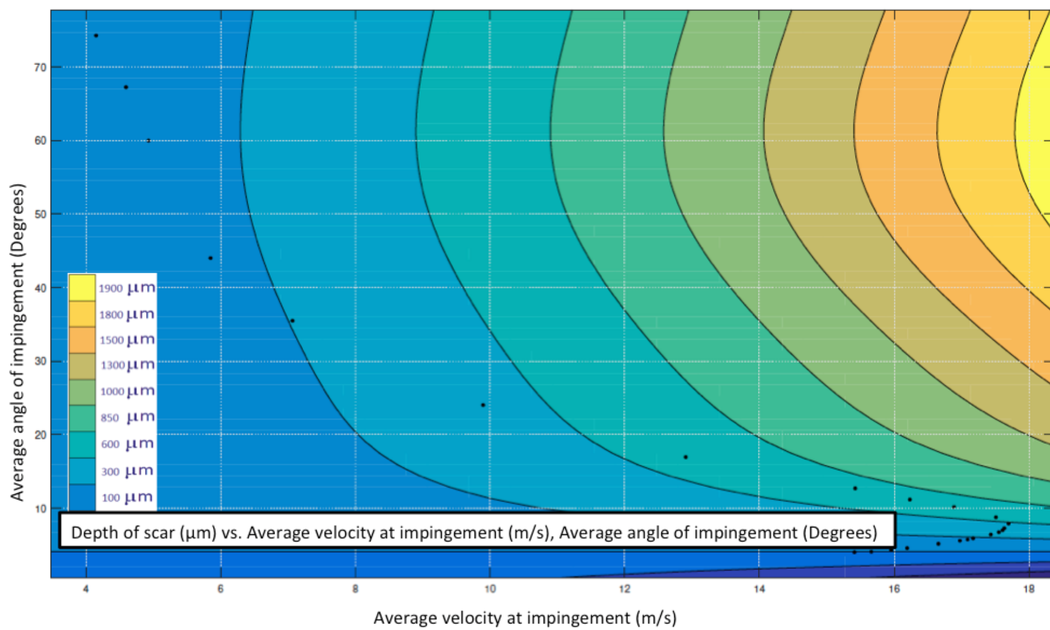


Figure 6.53: Wear map for the wear scar and CFD case in [60] fitted with 24 points

The geometry was deformed according to the new formula obtained and layers of cells were created at the specified boundary and a steady state was reached again. The deformed geometry and the steady state corresponding to this case are shown in figures 6.54, 6.55 and 6.56.

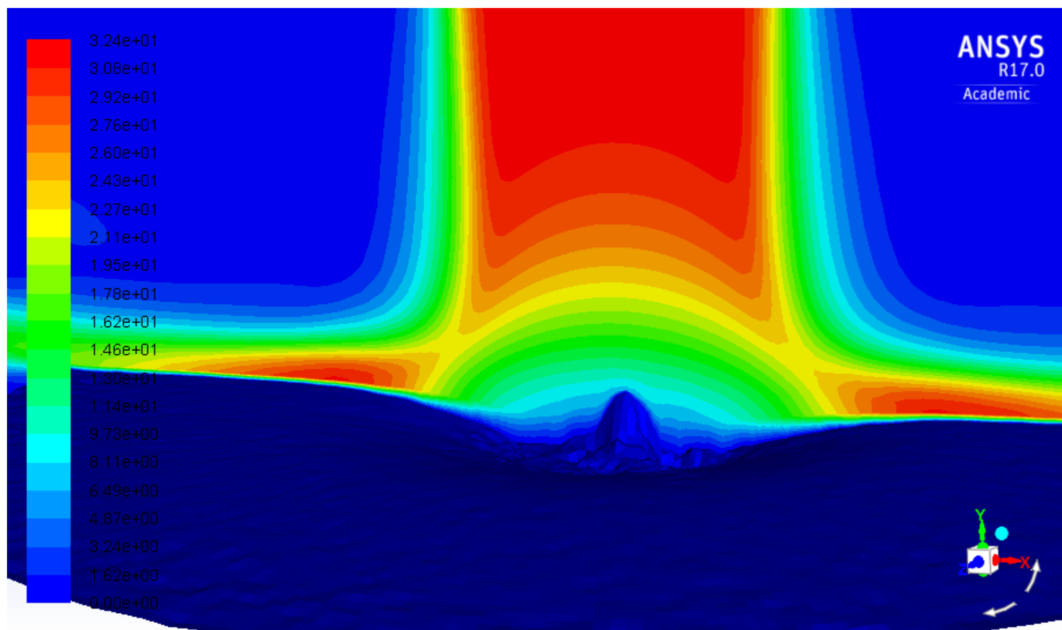


Figure 6.54: Velocity contours and deformed surface in $\frac{m}{s}$

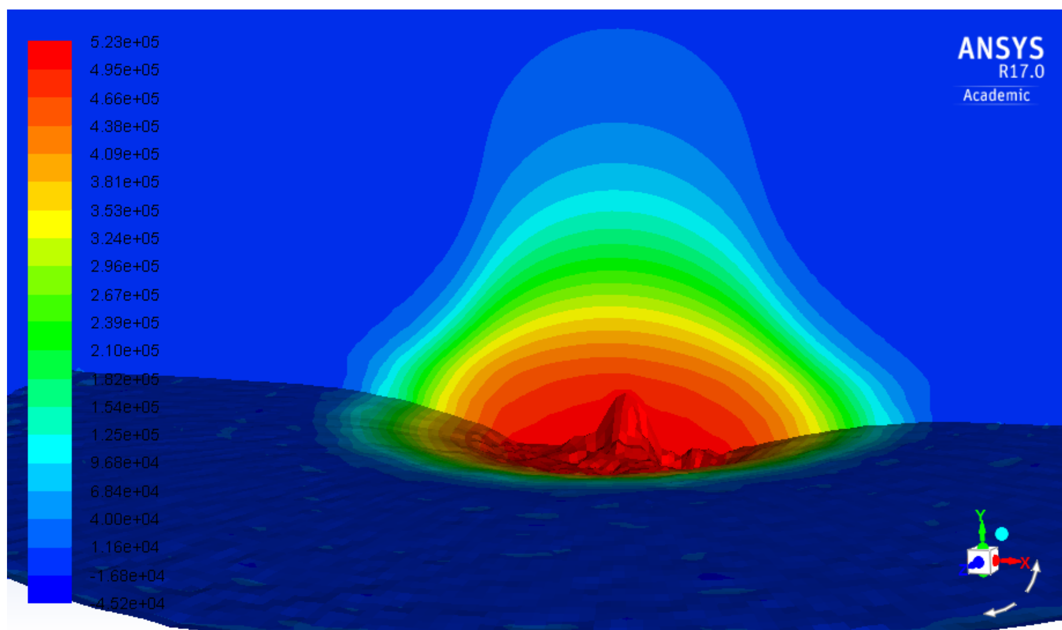


Figure 6.55: Static pressure contours in Pa and deformed surface showing no new stagnation point

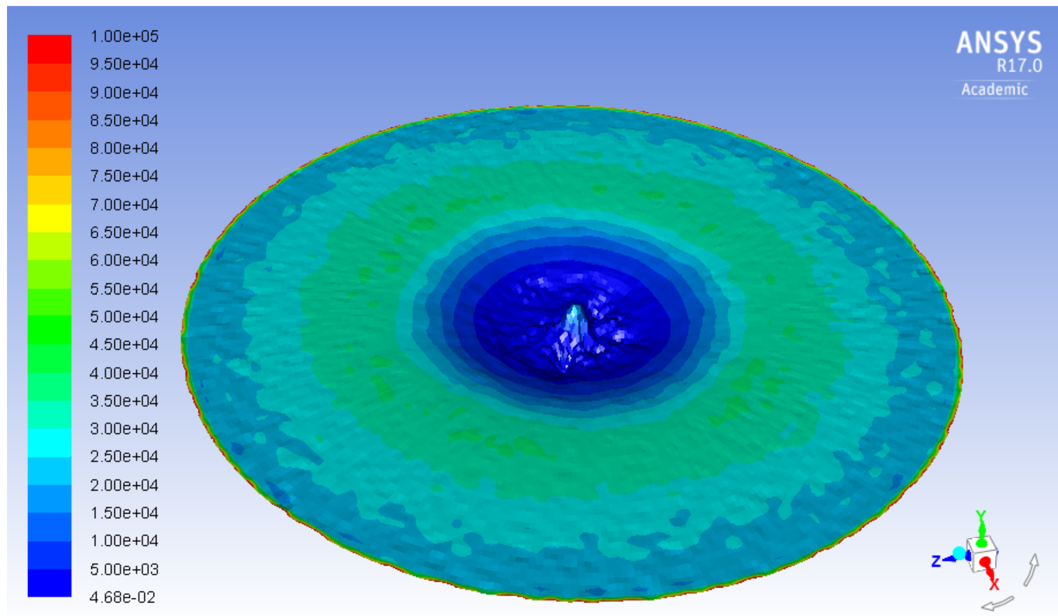


Figure 6.56: Vorticity contours in s^{-1} at the surface of the eroded geometry truncated to a value of 100000

6.6.4 Discussion

As seen in the results, no developing stagnation point is found when applying the Wear Map method. However, this might be due to a relatively poor fit as a result of wide prediction bounds or other factors such as the number of points used in the fitting, distance between those, their corresponding standard deviation, layering after the mesh deformation, etcetera. It is worth noting that the mesh used for both fits was the same one used previously for validation of the algorithm. In the case of the latter, given that the utility used divides the cells at the surface of the specified patch by creating a layer and defining a percentage for the relative thickness between the newly created cells, the cell size at some locations might not be adequate to capture the changes in the variables of the fluid flow. Regarding the goodness of the fit, Figures 6.43 and 6.52 show how well each point adapts to the fitted surface, being in this case the maximum difference of around 40 μm. However, the overall shape of the scar is captured by the method so it is expected that a better fit would yield significantly improved results for the three dimensional wear scar.

6.7 Application to centrifugal pumps

6.7.1 86 AH slurry pump volute

In many cases the erosion field in the volute of a centrifugal slurry pump can be approximated by calculating the square of the velocity field and interpolating it to the faces of the volute's boundary. The immediate implication of this is that erosion behaviour in the volute of the pump is similar to that of the Jet Impingement Test. In the JIT, erosion is calculated through formulae which, in most of the cases, have either the square of the velocity as a variable or a similar power of this magnitude. The code developed gets the magnitude of the velocity field at the cells nearest to the walls. These values are then introduced in the desired formula for calculating erosion and transferred to the boundary faces, which generates an erosion field at the walls of the geometry.

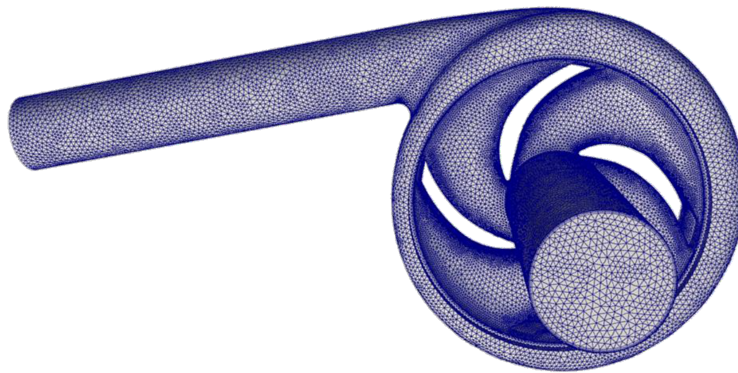


Figure 6.57: Picture of 86AH centrifugal slurry pump's mesh

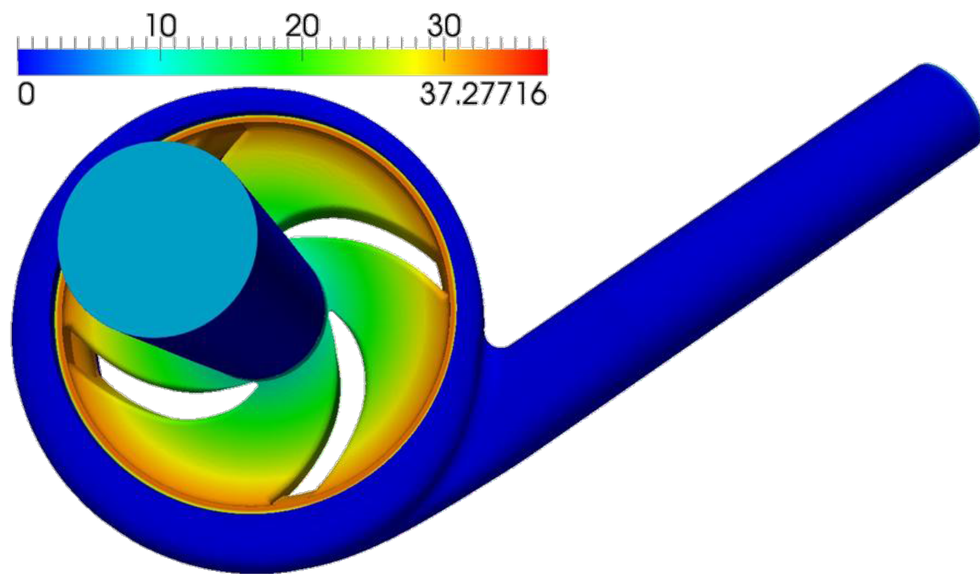


Figure 6.58: Picture of 86AH centrifugal slurry pump's steady state coloured by velocity magnitude $\frac{m}{s}$

Figure 6.59: Picture of 86AH centrifugal slurry pump's volute

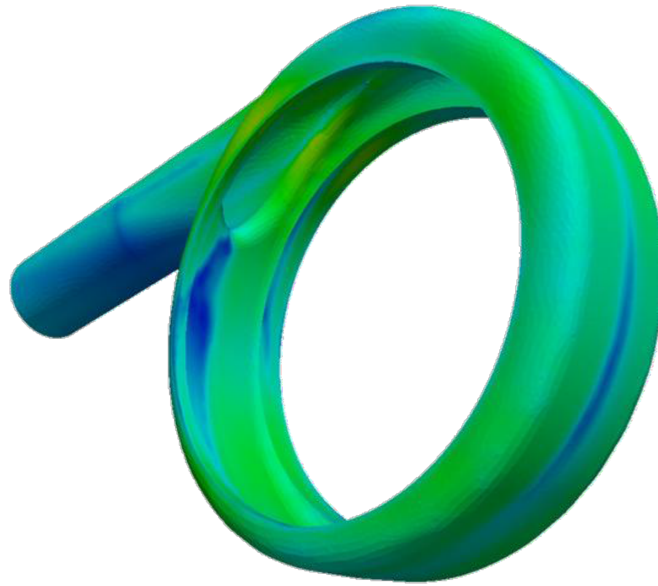


Figure 6.60: Picture of 86AH centrifugal slurry pump's volute coloured by erosion ratio

Figure 6.61: Picture of 86AH centrifugal slurry pump's volute deformed according to erosion

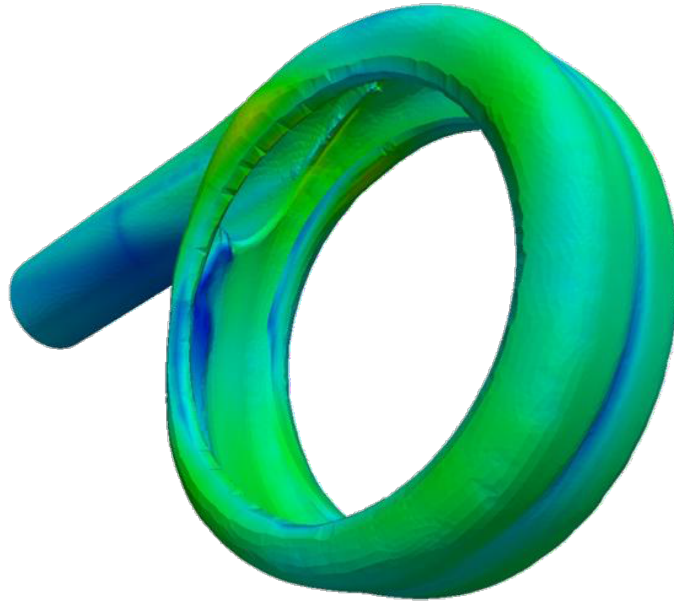


Figure 6.62: Picture of 86AH centrifugal slurry pump's volute deformed according to erosion field with contours of erosion

Figures 6.57 and 6.58 show the pump's mesh and the steady state of the 86AH centrifugal slurry pump, both of which were obtained by David Smith, an engineer in the Weir Group for a volume flow rate of 0.235 $\frac{m^3}{s}$ and a rotational speed of 1100 rpm [92]. Figure 6.59 shows the boundary corresponding to the volute of the pump before being deformed and figure 6.60 shows the same geometry coloured by erosion contours. Figures 6.61 and 6.62 show the deformed geometry with a solid color and with the erosion contours respectively. Field data available for the volute of this model, shown in figure 6.63 compared with the computational eroded volute, confirms the validity of the approach as well as the ability of the algorithm to capture the erosion induced deformation. Erosion contours and the deformed state of the volute were calculated using the code in Appendix I. It must be noted that the impeller used for the simulation had 4 vanes, which is equivalent to the one operating in the pumps shown in figure 6.63. Simulations with different number of vanes would have created a different erosion field.

Figure 6.63: Top: 86AH centrifugal slurry pump's volute operating with Impeller WRT1, Middle:Picture of 86AH centrifugal slurry pump's volute obtained from worn unit operating with F6145WRT1 impeller and F6083 throatbush, Bottom: Computational volute wall deformed according to erosion showing similar erosion pattern

6.7.2 150 WBH slurry pump impeller

A novel approach for calculating an erosion ratio on rotating parts such as impellers in centrifugal pumps was developed by Dr. Luis Moscoso in an internal report for the Weir Group [93] and in his PhD thesis [94] which has not been published and where he developed a formula for erosion calculation that includes the vorticity field. The approach consists of computing the vorticity field, which usually gives a very realistic indication of the erosion pattern in the impeller. The relationship between vorticity and erosion in centrifugal slurry pumps has also been investigated in a more recent paper by Cai et al [95]. Additionally, here, an average of the vorticity field was calculated for the transient of the pump. Once the vorticity field is plotted, if the whole pump is included in the rendering, there may be some areas where vorticity values are higher. However, in this case, focusing on the impeller and with the aid of an application specifically developed for this purpose, a truncated averaged vorticity is calculated. This way, relative values can not only be better visualised, but at the same time this new field was used to show how the eroded impeller would look like comparing it to the new one. The code used for calculating the averages and its truncated field can be found in Appendix G and Appendix H. The idea behind the vorticity field indicating the erosion pattern in rotating parts [93, 94] is based on the definition of the vorticity itself. The magnitude of the vorticity indicates how fast the fluid layers are rotating with respect to each other. In a pump's impeller (and in all rotating machinery), the particles traveling through it can be divided into two main groups. The first group is the one composed mainly by bigger, more inertial particles. The second group contains the smaller particles, which are mainly driven by the fluid's drag and follow it to a further extent. In a slurry pump it is this second group the one that accounts for most of the erosion. These smaller particles are captured in the smaller eddies close to the walls and are the ones responsible for most of the erosion in the impeller [93, 94]. This doesn't mean that the erosion is only caused by small particles. Bigger solids can have an impact on the erosion in the impeller too. A number of these may impact on an area which is not being subjected to much erosion. During the trajectory of the bigger particles and after impact, the path of the smaller particles may be changed by either the bigger ones themselves or the notch they may leave on the impeller's wall. The

smaller particles' behaviour may change in such a way that erosion starts progressing faster in that new area. Once the erosion ratio is calculated, the mesh can be deformed with the aid of the code in Appendix I.

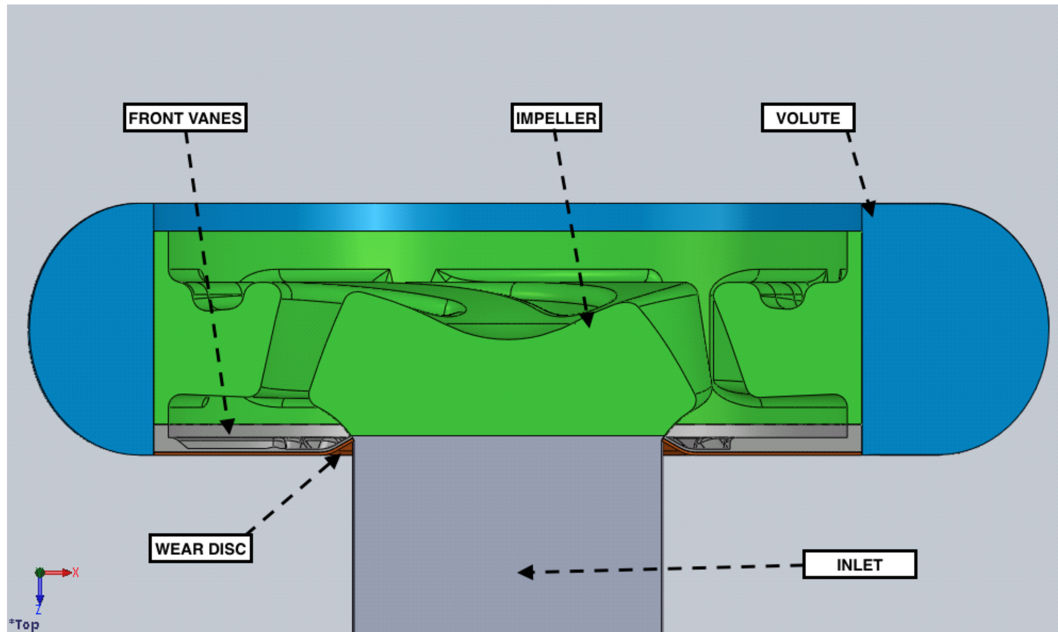


Figure 6.64: Cross section of 150WBH pump showing the different parts

Figure 6.65: Picture of 150WBH centrifugal slurry pump showing uneroded impeller

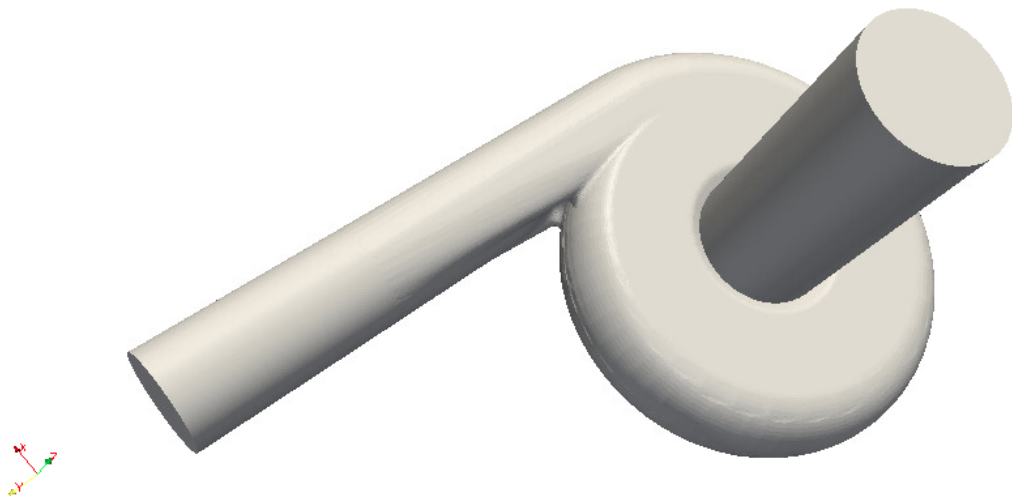


Figure 6.66: Picture of 150WBH centrifugal slurry pump showing volute and wear disc

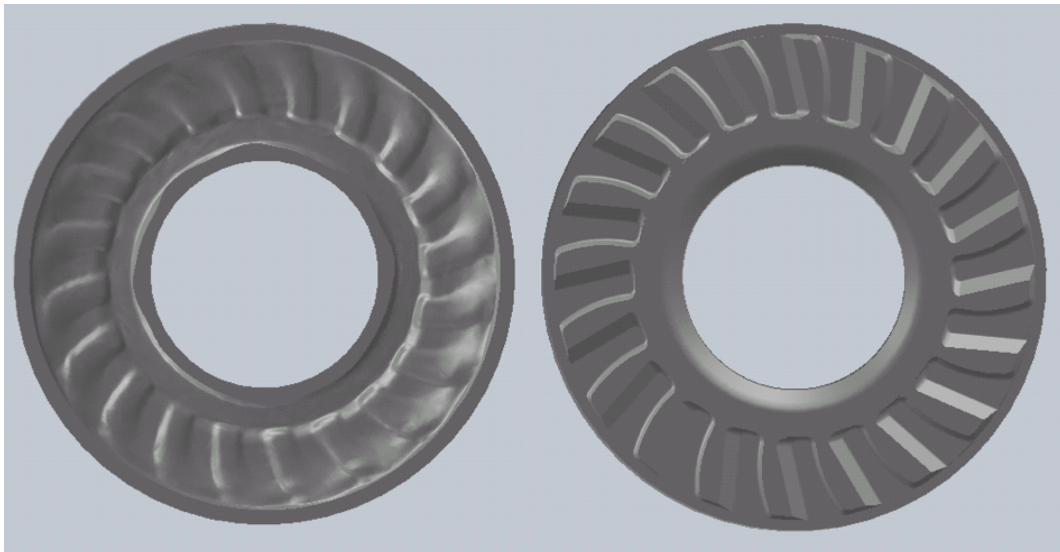


Figure 6.67: Picture of 150WBH centrifugal slurry pump front vanes scan before (right) and after erosion (left)

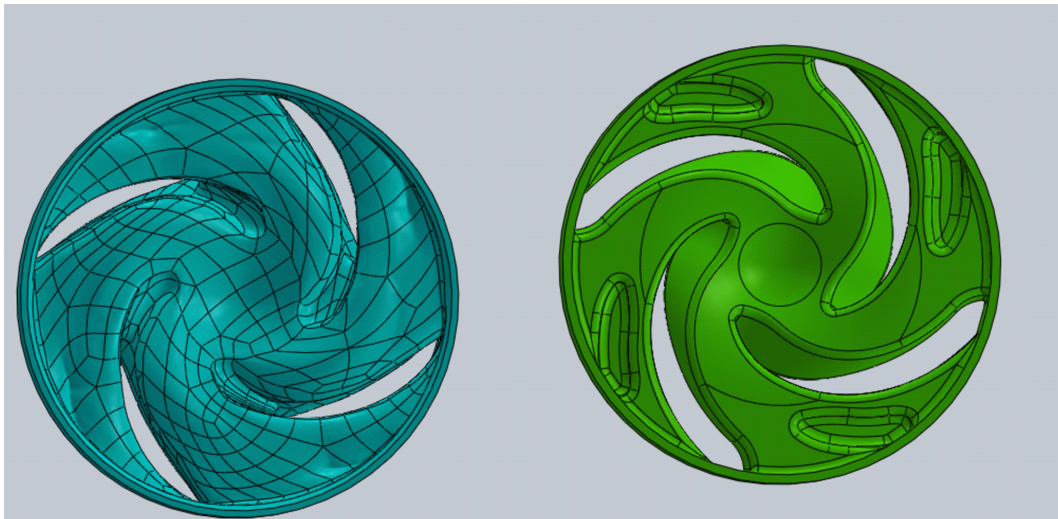


Figure 6.68: Picture of 150WBH centrifugal slurry pump impeller scan before and after erosion

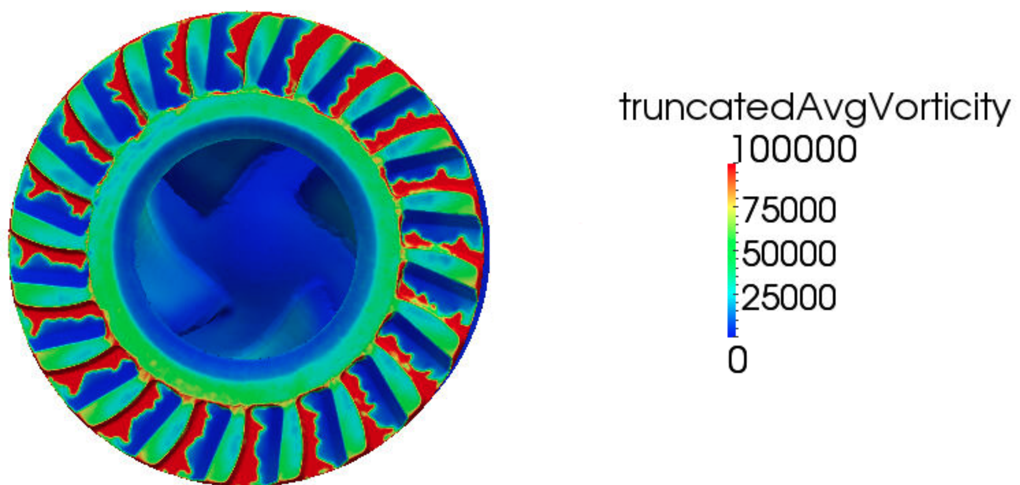


Figure 6.69: Picture of 150WBH centrifugal slurry pump front vanes truncated vorticity field

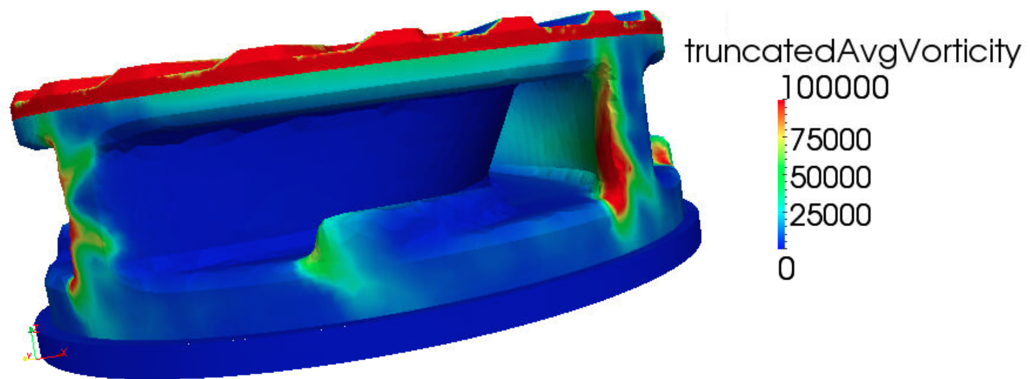


Figure 6.70: Picture of 150WBH centrifugal slurry pump impeller truncated vorticity field

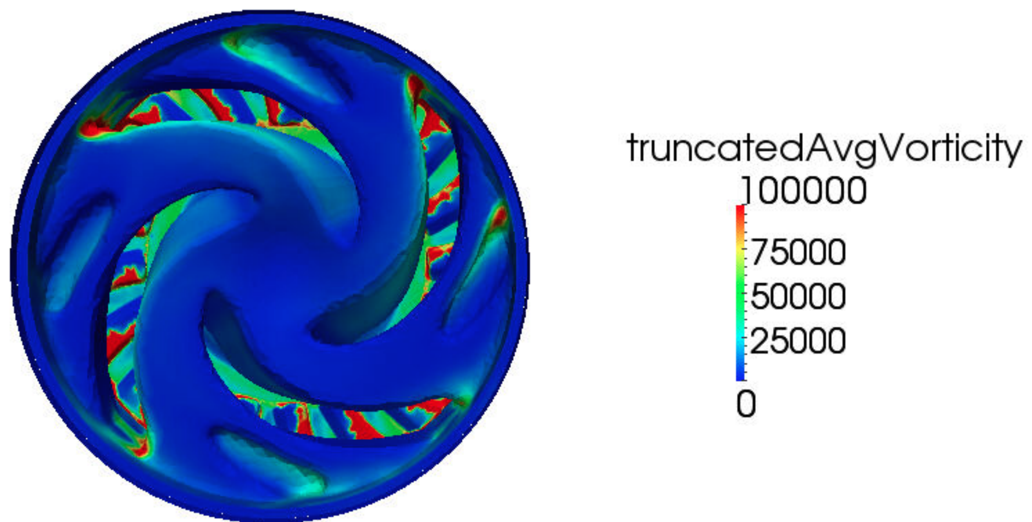


Figure 6.71: Picture of 150WBH centrifugal slurry pump impeller truncated vorticity field

Figure 6.72: Picture of 150WBH centrifugal slurry pump impeller before erosion deformation

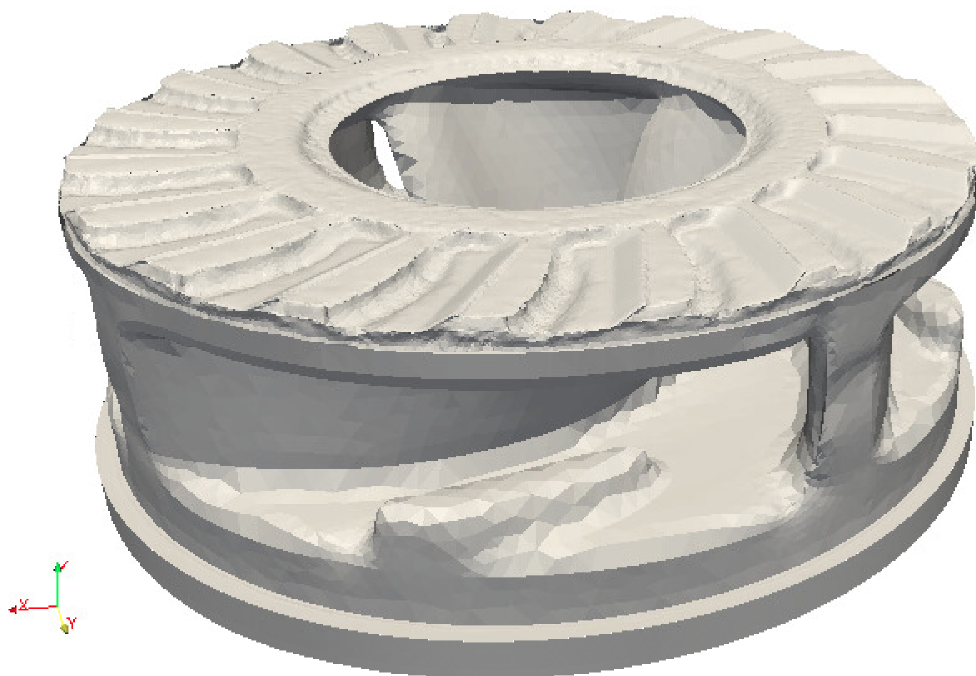


Figure 6.73: Picture of 150WBH centrifugal slurry pump eroded impeller according to calculated erosion magnitude

For the 150 WBH slurry pump, a frozen rotor simulation was set up first in order

to set it as the initial condition for the transient simulation of the pump with dynamic meshing. The pump operation was simulated for a volume flow rate of 125 $\frac{l}{s}$ and a rotational speed of 653 rpm. In this case, the images of the real pump were obtained through a 3D scan of an eroded impeller, front vanes and the wear disc, which is situated between the front vanes and the volute. A cross section of the slurry pump 150 WBH is shown in figure 6.64. Results obtained with the proposed methodology seem to capture well the location of erosion as a comparison between figures 6.67 and 6.69 and figures 6.68 and 6.70-6.71 shows. However, the scanned pump is clearly in a much more advanced state of erosion hence the similarity in the eroded profile is not as good as in the previous case. However, this state could be achieved by applying the method a second time after converging the flow field for the first stage of erosion. This would give as a result, a modified erosion ratio, given the the geometry is progressively changing the flow field, thus obtaining a more accurate representation of the geometry in a more advanced stage of erosion. Figures 6.72 and 6.73 show the new and eroded impellers respectively. The front vanes at the top of figure 6.73 as well as the walls close to the exit of the impeller are expected to be eroded first, as the deformation in the figure shows.

With the developed techniques, it is now possible to calculate, amongst other parameters, the decreased performance of the pump as it is being eroded. In order to do this, first, erosion would be calculated, the mesh would be deformed and remeshed afterwards (or the corresponding cells layered) and the new flow in the pump could be calculated. If this is done for progressively increasing wear scar depth, the trend of the performance could be obtained as well as the new head provided by the pump. Correlating in situ measurements of the pump's performance with the simulated one could give a very good indication of the time the pump will operate before it fails.
In addition to this, many customers change the impellers in their pumps before they reach failure. With the aid of this technique, the expected performance and head of the worn impeller can be calculated along with the expected number of extra hours the impeller would last. This would have a high impact on customers savings.

6.8 Application to other cases

In principle, the mesh deformation algorithm can be applied to any geometry subjected to erosion. In the case of centrifugal pumps and rotating machinery in general, existing tools and fields allow computing the erosion field on the walls of the geometry. A different method may be applied if the fluid pattern changes between different parts. The algorithm is developed to work independently of how the erosion field is calculated. Hence, it allows deforming the geometry according to erosion for any geometrical configuration. The application of the algorithm is only limited when, after applying it, boundaries intersect due to excessive deformation. Even when two different boundaries intersect, there are some alternatives which allow creation of holes in the mesh. One of these alternatives is using some of the commercial meshers which allow deleting and creating individual faces so that the intersecting ones can be deleted and new ones can be created to close the geometry again generating a hole(ICEM and StarCCM+ both have this function). This process, depending on the size of the mesh may become quite long. An algorithm could be implemented that automatically finds intersecting faces, deletes them and then joins opposite boundaries creating the hole automatically. In general, the smaller the faces at the boundary to be eroded and deformed the more accurate the scar will be. Once the eroded geometry is simulated a new simulation can be set up in order to obtain the updated fields and new efficiency of the eroded machine. Finally, application of the algorithm is not restricted to erosion subjected machinery. It can also be used in scouring erosion in bridges, natural erosion processes, river erosion and, in general, any process that implies material removal from the geometry or deformation by impact.

Chapter 7

Conclusions

A new methodology has been developed for erosion calculation including mesh deformation according to the erosion rate. The mesh deformation algorithm is independent of the geometrical configuration and as such, applicable to any geometry subjected to erosion. It was found that, depending on the technique used for interpolation in the dynamic meshing, the smoother or more uneven wear scars are calculated. Amongst the many techniques, a laplacian solver was chosen due to its suitability to solve deformation for smaller displacements. With the aid of the implementation of a number of erosion formulae in OpenFOAM, it was proved that, as erosion progresses in the jet impingement, a new stagnation ring appears around the wear scar as the velocity increased induced by the higher bend the fluid experiences. As opposed to Nguyen et al [60], in this work, the deformed surface was obtained only by computational means and the appearance of the stagnation point was predicted at a scar depth 1.766 times smaller than the one reported in their work. However, the results for the fluid velocity seem to predict that the fluid flow changes become significant enough at a wear scar depth of around 540 μm which is similar to the one analysed by Nguyen et al in [60]

It was also discovered through comparison with OpenFOAM that Ansys Fluent had an erroneous implementation of their particle tracking algorithm, at least in version 14. By systematically comparing results from Star CCM+, Ansys Fluent 14, OpenFOAM and Ansys 15, errors in the calculations of the particle velocities and angles of impingement were successfully identified. This issue with Fluent's Lagrangian particle

tracking has affected a number of articles that is estimated to be quantitatively very important in the literature, not only in erosion but in all articles were Lagrangian particle's trajectories have been analysed. It is unknown if this problem was there from the beginning or if it appeared after an update or even which update. Thus, the number of publications affected can not be calculated.

A working test-rig was developed and is currently being used for erosion experiments. However, due to technical issues it couldn't be used for validation. The test-rig design allows varying the mass concentration of particles between 1% and 7% approximately. Asymmetric scars are produced as a result of the location of the particle injection. However, this issue can be easily solved by moving the injection of the particles to another position such as on the pipe bend directly on top of the nozzle. The samples obtained with the test rig were 3D scanned and the profilometry and wear scars analysed, proving the asymetry, which was also anticipated by the CFD calculations.

Erosion in centrifugal slurry pumps was also investigated and it was found that there is a correlation between the square of the velocity at the cells nearest to the walls and the erosion ratio. It was also proved that erosion in the impeller can be approximated by means of the vorticity calculation. A set of applications have been implemented in OpenFOAM for calculation of the erosion contours and to deform the mesh according to those afterwards. A visual comparison with worn pumps proves the suitability of the technique to calculate performance decay as the pump is being eroded. This technique could also lead to a significant improvement in failure prediction. Further work in this area would include vorticity calculation in the moving parts with increased viscosity similar to that of the slurry. This would in theory give a more accurate vorticity field at the impeller and front vanes, thus enabling a more accurate erosion ratio calculation.

The deformation algorithm can also be used with or without dynamic meshing for many other applications such as simulating progressive pipe blockage or even the blood flow in arteries progressively blocked by accumulating bodies. In these cases, the direction of the surface normal vectors would be inverted as the geometry represented by the fluid path would be becoming smaller instead of bigger. Deformation can be coupled

not only to erosion but also to other fields such as pressure or velocity to simulate other processes.

Regarding the limitations of the mesh deformation algorithm, these are mainly associated to the mesh size and the geometrical configuration. Decreasing values of the face size at the eroded boundary will produce a more accurate representation of the deformed geometry. However, in the simple test case of the JIT, this would imply calculating erosion for a longer period of time since, being the faces smaller, a higher number of impacts will be required to calculate a reasonable average of the velocity and impact angles. Areas with relatively sharp angles could produce face intersection with little erosion applied. However, as discussed in the previous section, this issue could be addressed by creating new surfaces after deleting the intersecting faces.

Some of the proposed further work includes being able to generate holes in the geometry. This feature was investigated and, although not easy, it is feasible. The first thing to do would be to look for intersecting faces and manipulate them by cutting or elimination in order to form a new closed surface with those remaining cuts and new faces created. This would generate a hole connecting two parts of the geometry. This would enable simulating extreme erosion cases in which blades of pumps or turbines continue to operate without failing and the associated effects of this such as vibrations as well as learning how to predict them.

Appendix A

Facewise Average and Standard Deviation calculation

A.1 Procedure

The methodology to be folllowed in this case is the same as the one explained in 4.4.2. The files to be created in this case are named differently and, instead of only one field, several will be written to memory. The means of both impact angle and velocity along with the number of impacts per face, the standard deviations and the sample sizes are calculated with the code in this section.

A.1.1 FacewiseStandardDeviation.C

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | Copyright (C) 2011-2012 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 "FacewiseStandardDeviation.H"

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

template<class CloudType>
Foam::label Foam::FacewiseStandardDeviation<CloudType>::applyToPatch
(
    const label globalPatchI
) const
{
    forAll(patchIDs_, i)
    {
        if (patchIDs_[i] == globalPatchI)
        {
            return i;
        }
```

```
    }

    return -1;
}


template<class CloudType>
void Foam::FacewiseStandardDeviation<CloudType>::write()
{
    if (QPtr1_.valid() && QPtr2_.valid() && QPtr3_.valid())
    {
QPtr1_->write();
QPtr4_->write();
QPtr5_->write();
QPtr6_->write();
QPtr7_->write();
    }
    else
    {
 FatalErrorIn("void Foam::FacewiseStandardDeviation<CloudType>::write()")
            << "QPtr not valid" << abort(FatalError);
    }
}


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

template<class CloudType>
Foam::FacewiseStandardDeviation<CloudType>::FacewiseStandardDeviation
(
    const dictionary& dict,
```

```
    CloudType& owner
)
:
    CloudFunctionObject<CloudType>(dict, owner, typeName),
    QPtr1_(NULL),
    QPtr2_(NULL),
    QPtr3_(NULL),
    QPtr4_(NULL),
    QPtr5_(NULL),
    QPtr6_(NULL),
    QPtr7_(NULL),
    patchIDs_()
{
    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::FacewiseStandardDeviation<CloudType>::FacewiseStandardDeviation"
                "("
                    "const dictionary&, "
                    "CloudType& "
                ")"
         )   << "Cannot find any patch names matching " << patchName[i]
```

```
                << endl;
        }

        uniquePatchIDs.insert(patchIDs);
    }

    patchIDs_ = uniquePatchIDs.toc();

    // trigger ther creation of the Q field
    preEvolve();
}


template<class CloudType>
Foam::FacewiseStandardDeviation<CloudType>::FacewiseStandardDeviation
(
    const FacewiseStandardDeviation<CloudType>& pe
)
:
    CloudFunctionObject<CloudType>(pe),
    QPtr1_(NULL),
    QPtr2_(NULL),
    QPtr3_(NULL),
    QPtr4_(NULL),
    QPtr5_(NULL),
    QPtr6_(NULL),
    QPtr7_(NULL),
    patchIDs_(pe.patchIDs_)
{}
```

```
// * * * * * * * * * * * * * * * * Destructor  * * * * * * * * * * * * *//

template<class CloudType>
Foam::FacewiseStandardDeviation<CloudType>::~FacewiseStandardDeviation()
{}


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

template<class CloudType>
void Foam::FacewiseStandardDeviation<CloudType>::preEvolve()
{
    if (QPtr1_.valid() && QPtr2_.valid() && QPtr3_.valid())
    {
        QPtr1_->internalField() = 0.0;
        QPtr2_->internalField() = 0.0;
QPtr3_->internalField() = 0.0;
        QPtr4_->internalField() = 0.0;
QPtr5_->internalField() = 0.0;
        QPtr6_->internalField() = 0.0;
QPtr7_->internalField() = 0.0;
    }
    else
    {
        const fvMesh& mesh = this->owner().mesh();

        QPtr1_.reset
        (
            new volScalarField
            (
                IOobject
```

```
        (
            this->owner().name() + "ImpactNumber",
            mesh.time().timeName(),
            mesh,
            IOobject::READ_IF_PRESENT,
            IOobject::NO_WRITE
        ),
        mesh,
        dimensionedScalar("zero", dimless, 0.0)
    )
);
QPtr2_.reset
(
    new volScalarField
    (
        IOobject
        (
            this->owner().name() + "SumImpactVelocity",
            mesh.time().timeName(),
            mesh,
            IOobject::READ_IF_PRESENT,
            IOobject::NO_WRITE
        ),
        mesh,
        dimensionedScalar("zero", dimVelocity, 0.0)
    )
);
QPtr3_.reset
(
    new volScalarField
    (
```

```
        IOobject
        (
            this->owner().name() + "SumImpactAngle",
            mesh.time().timeName(),
            mesh,
            IOobject::READ_IF_PRESENT,
            IOobject::NO_WRITE
        ),
        mesh,
        dimensionedScalar("zero", dimless, 0.0)
    )
);
QPtr4_.reset
(
    new volScalarField
    (
        IOobject
        (
            this->owner().name() + "MeanImpactVelocity",
            mesh.time().timeName(),
            mesh,
            IOobject::READ_IF_PRESENT,
            IOobject::NO_WRITE
        ),
        mesh,
        dimensionedScalar("zero", dimVelocity, 0.0)
    )
);
QPtr5_.reset
(
    new volScalarField
```

```
        (
            IOobject
            (
                this->owner().name() + "MeanImpactAngle",
                mesh.time().timeName(),
                mesh,
                IOobject::READ_IF_PRESENT,
                IOobject::NO_WRITE
            ),
            mesh,
            dimensionedScalar("zero", dimless, 0.0)
        )
    );
    QPtr6_.reset
    (
        new volScalarField
        (
            IOobject
            (
                this->owner().name() + "ImpactVelStdDev",
                mesh.time().timeName(),
                mesh,
                IOobject::READ_IF_PRESENT,
                IOobject::NO_WRITE
            ),
            mesh,
            dimensionedScalar("zero", dimless, 0.0)
        )
    );
    QPtr7_.reset
    (
```

```
            new volScalarField
            (
                IOobject
                (
                    this->owner().name() + "ImpactAngleStdDev",
                    mesh.time().timeName(),
                    mesh,
                    IOobject::READ_IF_PRESENT,
                    IOobject::NO_WRITE
                ),
                mesh,
                dimensionedScalar("zero", dimless, 0.0)
            )
        );
    }
}


template<class CloudType>
void Foam::FacewiseStandardDeviation<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)
    {
        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 =
        (180/mathematical::pi)*(mathematical::pi/2.0 - acos(nw & Udir));

        const label patchFaceI = pp.whichFace(p.face());

scalar& Q1 = QPtr1_->boundaryField()[patchI][patchFaceI];
scalar& Q2 = QPtr2_->boundaryField()[patchI][patchFaceI];
scalar& Q3 = QPtr3_->boundaryField()[patchI][patchFaceI];
scalar& Q4 = QPtr4_->boundaryField()[patchI][patchFaceI];
```

```
scalar& Q5 = QPtr5_->boundaryField()[patchI][patchFaceI];
scalar& Q6 = QPtr6_->boundaryField()[patchI][patchFaceI];
scalar& Q7 = QPtr7_->boundaryField()[patchI][patchFaceI];

Q1 += p.nParticle();

        Q2 += magU*p.nParticle();

Q3 += alpha*p.nParticle();

Q4 = Q2/Q1;

Q5 = Q3/Q1;

Q6 = sqrt(sqr(Q4 - magU)/Q1);

Q7 = sqrt(sqr(Q5 - alpha)/Q1);

    }
}


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

A.1.2 FacewiseStandardDeviation.H

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | Copyright (C) 2011-2012 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/>.

Class
    Foam::FacewiseStandardDeviation

Description
    Creates averages for particle variables

SourceFiles
    FacewiseStandardDeviation.C

\*---------------------------------------------------------------------*/

#ifndef FacewiseStandardDeviation_H
```

```
#define FacewiseStandardDeviation_H

#include "CloudFunctionObject.H"
#include "volFields.H"

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

namespace Foam
{

/*---------------------------------------------------------------*\
                       Class FacewiseStandardDeviation Declaration
\*---------------------------------------------------------------*/

template<class CloudType>
class FacewiseStandardDeviation
:
    public CloudFunctionObject<CloudType>
{
    // Private Data

        // Typedefs

            //- Convenience typedef for parcel type
            typedef typename CloudType::parcelType parcelType;


        //- Particle erosion field
        autoPtr<volScalarField> QPtr1_;
        autoPtr<volScalarField> QPtr2_;
        autoPtr<volScalarField> QPtr3_;
```

```
        autoPtr<volScalarField> QPtr4_;
        autoPtr<volScalarField> QPtr5_;
        autoPtr<volScalarField> QPtr6_;
        autoPtr<volScalarField> QPtr7_;

        //- List of patch indices to post-process
        labelList patchIDs_;

        //- Plastic flow stress - typical metal value = 2.7 GPa
        scalar p_;

        //- Ratio between depth of contact and length of cut - default=2
        scalar psi_;

        //- Ratio of normal and tangential forces - default=2
        scalar K_;


protected:

    // Protected Member Functions

        //- Returns local patchI if patch is in patchIds_ list
        label applyToPatch(const label globalPatchI) const;

        //- Write post-processing info
        virtual void write();


public:
```

```
//- Runtime type information
TypeName("FacewiseStandardDeviation");


// Constructors

    //- Construct from dictionary
    FacewiseStandardDeviation
    (const dictionary& dict, CloudType& owner);

    //- Construct copy
    FacewiseStandardDeviation
    (const FacewiseStandardDeviation<CloudType>& pe);

    //- Construct and return a clone
    virtual autoPtr<CloudFunctionObject<CloudType> > clone() const
    {
        return autoPtr<CloudFunctionObject<CloudType> >
        (
            new FacewiseStandardDeviation<CloudType>(*this)
        );
    }


//- Destructor
virtual ~FacewiseStandardDeviation();


// Member Functions

    // Evaluation
```

```
            //- Pre-evolve hook
            virtual void preEvolve();

            //- Post-patch hook
            virtual void postPatch
            (
                const parcelType& p,
                const polyPatch& pp,
                const scalar trackFraction,
                const tetIndices& tetIs,
                bool& keepParticle
            );
};


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

} // End namespace Foam

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

#ifdef NoRepository
#   include "FacewiseStandardDeviation.C"
#endif

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

#endif

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

Appendix B

Implementation of additional rebound models

B.1 Procedure

There is more than one way to implement additional rebound models. The one presented in this Appendix is the one that would require less ”effort”, since only some files need some additions and no additional files are added for compilation. The modified files are shown with the two added rebound models, which are Sommerfeld et al [15] and Forder et al [16]. The implementation of the rebound model is carried out inside the `StandardWallInteraction.C` file so both models will be available if StandardWallInteraction is chosen in the `kinematicCloudProperties` dictionary along with the original rebound, escape and stick conditions. In order to make them available within StandardWallInteraction, `PatchInteractionModel.C` and `PatchInteractionModel.H` need also be modified to include them, as well as `LocalInteraction.H` in case different boundaries have different rebound models. Once the files have been modified, the code is recompiled again.

B.1.1 StandardWallInteraction.C

```
*---------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
```

```
   \\    /   O peration     |
    \\  /    A nd           | Copyright (C) 2011-2012 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 "StandardWallInteraction.H"

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

template<class CloudType>
Foam::StandardWallInteraction<CloudType>::StandardWallInteraction
(
    const dictionary& dict,
    CloudType& cloud
```

```
)
:
    PatchInteractionModel<CloudType>(dict, cloud, typeName),
    interactionType_
    (
        this->wordToInteractionType(this->coeffDict().lookup("type"))
    ),
    e_(0.0),
    mu_(0.0),
    nEscape_(0),
    massEscape_(0.0),
    nStick_(0),
    massStick_(0.0)
{
    switch (interactionType_)
    {
        case PatchInteractionModel<CloudType>::itOther:
        {
       const word interactionTypeName(this->coeffDict().lookup("type"));

            FatalErrorIn
            (
        "StandardWallInteraction<CloudType>::StandardWallInteraction"
                "("
                    "const dictionary&, "
                    "CloudType& cloud"
                ")"
            ) << "Unknown interaction result type "
              << interactionTypeName
              << ". Valid selections are:" << this->interactionTypeNames_
              << endl << exit(FatalError);
```

```
            break;
        }
        case PatchInteractionModel<CloudType>::itRebound:
        {
            e_ = this->coeffDict().lookupOrDefault("e", 1.0);
            mu_ = this->coeffDict().lookupOrDefault("mu", 0.0);
            break;
        }
        default:
        {
            // do nothing
        }
    }
}


template<class CloudType>
Foam::StandardWallInteraction<CloudType>::StandardWallInteraction
(
    const StandardWallInteraction<CloudType>& pim
)
:
    PatchInteractionModel<CloudType>(pim),
    interactionType_(pim.interactionType_),
    e_(pim.e_),
    mu_(pim.mu_),
    nEscape_(pim.nEscape_),
    massEscape_(pim.massEscape_),
    nStick_(pim.nStick_),
    massStick_(pim.massStick_)
```

```
{}


// * * * * * * * * * * * * * * * * Destructor  * * * * * * * * * * *//

template<class CloudType>
Foam::StandardWallInteraction<CloudType>::~StandardWallInteraction()
{}


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

template<class CloudType>
bool Foam::StandardWallInteraction<CloudType>::correct
(
    typename CloudType::parcelType& p,
    const polyPatch& pp,
    bool& keepParticle,
    const scalar trackFraction,
    const tetIndices& tetIs
)
{
    vector& U = p.U();

    bool& active = p.active();

    if (isA<wallPolyPatch>(pp))
    {
        switch (interactionType_)
        {
            case PatchInteractionModel<CloudType>::itEscape:
```

```
{
    keepParticle = false;
    active = false;
    U = vector::zero;
    nEscape_++;
    break;
}
case PatchInteractionModel<CloudType>::itStick:
{
    keepParticle = true;
    active = false;
    U = vector::zero;
    nStick_++;
    break;
}
case PatchInteractionModel<CloudType>::itRebound:
{
    keepParticle = true;
    active = true;

    vector nw;
    vector Up;

    this->owner().patchData(p, pp, trackFraction, tetIs, nw, Up);

    // Calculate motion relative to patch velocity
    U -= Up;

    scalar Un = U & nw;
    vector Ut = U - Un*nw;
```

```
                if (Un > 0)
                {
                    U -= (1.0 + e_)*Un*nw;
                }

                U -= mu_*Ut;

                // Return velocity to global space
                U += Up;

                break;
            }
            case PatchInteractionModel<CloudType>::itForder:
            {
                keepParticle = true;
                active = true;

                vector nw;
                vector Up;

           this->owner().patchData(p, pp, trackFraction, tetIs, nw, Up);

                // Calculate motion relative to patch velocity
                U -= Up;

                scalar Un = U & nw;
                vector Ut = U - Un*nw;

   const scalar magU = mag(U);
   const vector Udir = U/magU;
```

```
   // determine impact angle, alpha
     const scalar alpha = mathematical::pi/2.0 - acos(nw & Udir);

     //determine the normal and tangential restitution coefficients

     //normal coefficient of restitution
     scalar e_n =
0.988-0.78*alpha+0.19*sqr(alpha)-0.024*pow(alpha,3)+0.0027*pow(alpha,4);
     //tangential coefficient of restitution
     scalar e_t =
1.0-0.78*alpha+0.84*sqr(alpha)-0.21*pow(alpha,3)+
0.28*pow(alpha,4)-0.022*pow(alpha,5);

//New velocity is the previous normal and tangential components
//multiplied by their respective restitution coefficients
//calculated above
U -= e_n*(Un*nw) + e_t*Ut;

                // Return velocity to global space
                U += Up;

                break;
            }
            case PatchInteractionModel<CloudType>::itSommerfeldHubber:
            {
                keepParticle = true;
                active = true;

                vector nw;
                vector Up;
```

```
       this->owner().patchData(p, pp, trackFraction, tetIs, nw, Up);

                // Calculate motion relative to patch velocity
                U -= Up;

                scalar Un = U & nw;
                vector Ut = U - Un*nw;

   const scalar magU = mag(U);
   const vector Udir = U/magU;

   // determine impact angle, alpha
     const scalar alpha = mathematical::pi/2.0 - acos(nw & Udir);

     //determine the normal and tangential restitution coefficients

     //normal coefficient of restitution
     scalar e_n =
1.0-0.4159*alpha+0.4994*sqr(alpha)- 0.292*pow(alpha,3);
     //tangential coefficient of restitution
     scalar e_t = 1.0-2.12*alpha+3.0775*sqr(alpha)-1.1*pow(alpha,3);

//New velocity is the previous normal and tangential components
//multiplied by their respective restitution coefficients calculated above
U -= e_n*(Un*nw) + e_t*Ut;

                // Return velocity to global space
                U += Up;

                break;
            }
```

```
            default:
            {
                FatalErrorIn
                (
                    "bool StandardWallInteraction<CloudType>::correct"
                    "("
                        "const polyPatch&, "
                        "const label, "
                        "bool&, "
                        "vector&"
                    ") const"
                )   << "Unknown interaction type "
                    << this->interactionTypeToWord(interactionType_)
                    << "(" << interactionType_ << ")" << endl
                    << abort(FatalError);
            }
        }

        return true;
    }

    return false;
}


template<class CloudType>
void Foam::StandardWallInteraction<CloudType>::info(Ostream& os)
{
    scalar mpe0 = this->template getBaseProperty<scalar>("massEscape");
    scalar mpe = mpe0 + returnReduce(massEscape_, sumOp<scalar>());
```

```
    label npe0 = this->template getBaseProperty<scalar>("nEscape");
    label npe = npe0 + returnReduce(nEscape_, sumOp<label>());

    label nps0 = this->template getBaseProperty<scalar>("nStick");
    label nps = nps0 + returnReduce(nStick_, sumOp<label>());

    scalar mps0 = this->template getBaseProperty<scalar>("massStick");
    scalar mps = mps0 + returnReduce(massStick_, sumOp<scalar>());

    os  << "    Parcel fate (number, mass)" << nl
        << "      - escape            = " << npe << ", " << mpe << nl
        << "      - stick             = " << nps << ", " << mps << nl;

    if (this->outputTime())
    {
        this->setModelProperty("nEscape", npe);
        nEscape_ = 0;

        this->setModelProperty("massEscape", mpe);
        massEscape_ = 0.0;

        this->setModelProperty("nStick", nps);
        nStick_ = 0;

        this->setModelProperty("massStick", mps);
        massStick_ = 0.0;
    }
}


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

B.1.2 StandardWallInteraction.H

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | Copyright (C) 2011-2012 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/>.

Class
    Foam::StandardWallInteraction
```

```
Description
    Wall interaction model. Three choices:
    - rebound - optionally specify elasticity and resitution coefficients
    - stick   - particles assigined zero velocity
    - escape  - remove particle from the domain

    Example usage:

        StandardWallInteractionCoeffs
        {
            type        rebound; // stick, escape
            e           1;       // optional - elasticity coeff
            mu          0;       // optional - restitution coeff
        }

\*---------------------------------------------------------------------------*/

#ifndef StandardWallInteraction_H
#define StandardWallInteraction_H

#include "PatchInteractionModel.H"

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

namespace Foam
{
/*---------------------------------------------------------------------------*\
                   Class StandardWallInteraction Declaration
\*---------------------------------------------------------------------------*/

template<class CloudType>
```

```
class StandardWallInteraction
:
    public PatchInteractionModel<CloudType>
{
protected:

    // Protected data

        //- Interaction type
        typename PatchInteractionModel<CloudType>::interactionType
            interactionType_;

        //- Elasticity coefficient
        scalar e_;

        //- Restitution coefficient
        scalar mu_;


        // Counters for particle fates

            //- Number of parcels escaped
            label nEscape_;

            //- Mass of parcels escaped
            scalar massEscape_;

            //- Number of parcels stuck to patches
            label nStick_;

            //- Mass of parcels stuck to patches
```

```
            scalar massStick_;


public:

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


    // Constructors

        //- Construct from dictionary
 StandardWallInteraction(const dictionary& dict, CloudType& cloud);

        //- Construct copy from owner cloud and patch interaction model
 StandardWallInteraction(const StandardWallInteraction<CloudType>& pim);

        //- Construct and return a clone using supplied owner cloud
 virtual autoPtr<PatchInteractionModel<CloudType> > clone() const
        {
            return autoPtr<PatchInteractionModel<CloudType> >
            (
                new StandardWallInteraction<CloudType>(*this)
            );
        }


    //- Destructor
    virtual ~StandardWallInteraction();
```

```
    // Member Functions

        //- Apply velocity correction
        //  Returns true if particle remains in same cell
        virtual bool correct
        (
            typename CloudType::parcelType& p,
            const polyPatch& pp,
            bool& keepParticle,
            const scalar trackFraction,
            const tetIndices& tetIs
        );


        // I-O

            //- Write patch interaction info to stream
            virtual void info(Ostream& os);
};


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

} // End namespace Foam

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

#ifdef NoRepository
#   include "StandardWallInteraction.C"
#endif
```

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

#endif

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

B.1.3 PatchInteractionModel.C

```
/*---------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | Copyright (C) 2011-2012 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 "PatchInteractionModel.H"
#include "fvMesh.H"
#include "Time.H"
#include "volFields.H"

// * * * * * * * * * * * * * * Static Data Members * * * * * * * * *//

template<class CloudType>
Foam::wordList Foam::PatchInteractionModel
<CloudType>::interactionTypeNames_
(
    IStringStream
    (
        "(rebound stick escape Forder SommerfeldHubber)"
    )()
);

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

template<class CloudType>
Foam::word Foam::PatchInteractionModel
<CloudType>::interactionTypeToWord
(
    const interactionType& itEnum
)
{
```

```
word it = "other";

switch (itEnum)
{
    case itRebound:
    {
        it = "rebound";
        break;
    }
    case itStick:
    {
        it = "stick";
        break;
    }
    case itEscape:
    {
        it = "escape";
        break;
    }
    case itForder:
    {
        it = "Forder";
        break;
    }
    case itSommerfeldHubber:
    {
        it = "SommerfeldHubber";
        break;
    }
    default:
    {
```

```
        }
    }

    return it;
}



template<class CloudType>
typename Foam::PatchInteractionModel<CloudType>::interactionType
Foam::PatchInteractionModel<CloudType>::wordToInteractionType
(
    const word& itWord
)
{
    if (itWord == "rebound")
    {
        return itRebound;
    }
    else if (itWord == "stick")
    {
        return itStick;
    }
    else if (itWord == "escape")
    {
        return itEscape;
    }
    else if (itWord == "Forder")
    {
        return itForder;
    }
    else if (itWord == "SommerfeldHubber")
```

```
    {
        return itForder;
    }
    else
    {
        return itOther;
    }
}


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

template<class CloudType>
Foam::PatchInteractionModel<CloudType>::PatchInteractionModel
(
    CloudType& owner
)
:
    SubModelBase<CloudType>(owner),
    UName_("unknown_UName")
{}


template<class CloudType>
Foam::PatchInteractionModel<CloudType>::PatchInteractionModel
(
    const dictionary& dict,
    CloudType& owner,
    const word& type
)
:
```

```
    SubModelBase<CloudType>(owner, dict, typeName, type),
    UName_(this->coeffDict().lookupOrDefault("UName", word("U")))
{}


template<class CloudType>
Foam::PatchInteractionModel<CloudType>::PatchInteractionModel
(
    const PatchInteractionModel<CloudType>& pim
)
:
    SubModelBase<CloudType>(pim),
    UName_(pim.UName_)
{}


// * * * * * * * * * * * * * * * * Destructor  * * * * * * * * * * *//

template<class CloudType>
Foam::PatchInteractionModel<CloudType>::~PatchInteractionModel()
{}


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

template<class CloudType>
const Foam::word& Foam::PatchInteractionModel<CloudType>::UName() const
{
    return UName_;
}
```

```
template<class CloudType>
bool Foam::PatchInteractionModel<CloudType>::correct
(
    typename CloudType::parcelType&,
    const polyPatch&,
    bool&,
    const scalar,
    const tetIndices&
)
{
    notImplemented
    (
        "bool Foam::PatchInteractionModel<CloudType>::correct"
        "("
            "typename CloudType::parcelType&, "
            "const polyPatch&, "
            "bool&, "
            "const scalar, "
            "const tetIndices& "
        ") const"
    );
    return false;
}


template<class CloudType>
void Foam::PatchInteractionModel<CloudType>::info(Ostream& os)
{
    // do nothing
}
```

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

#include "PatchInteractionModelNew.C"

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

B.1.4 PatchInteractionModel.H

```
/*---------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | Copyright (C) 2011-2012 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/>.

Class
    Foam::PatchInteractionModel

Description
    Templated patch interaction model class

SourceFiles
    PatchInteractionModel.C
    PatchInteractionModelNew.C

\*---------------------------------------------------------------------*/

#ifndef PatchInteractionModel_H
#define PatchInteractionModel_H

#include "IOdictionary.H"
#include "autoPtr.H"
#include "runTimeSelectionTables.H"
#include "polyPatch.H"
#include "wallPolyPatch.H"
#include "tetIndices.H"
#include "SubModelBase.H"

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

```
namespace Foam
{

/*---------------------------------------------------------------------------*\
                    Class PatchInteractionModel Declaration
\*---------------------------------------------------------------------------*/

template<class CloudType>
class PatchInteractionModel
:
    public SubModelBase<CloudType>
{
public:

    // Public enumerations

        // Interaction types
        enum interactionType
        {
            itRebound,
            itStick,
            itEscape,
    itForder,
    itSommerfeldHubber,
            itOther
        };

        static wordList interactionTypeNames_;


private:
```

```
    // Private data

        //- Name of velocity field - default = "U"
        const word UName_;


public:

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

    //- Declare runtime constructor selection table
    declareRunTimeSelectionTable
    (
        autoPtr,
        PatchInteractionModel,
        dictionary,
        (
            const dictionary& dict,
            CloudType& owner
        ),
        (dict, owner)
    );


    // Constructors

        //- Construct null from owner
        PatchInteractionModel(CloudType& owner);
```

```
        //- Construct from components
        PatchInteractionModel
        (
            const dictionary& dict,
            CloudType& owner,
            const word& type
        );

        //- Construct copy
    PatchInteractionModel(const PatchInteractionModel<CloudType>& pim);

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


    //- Destructor
    virtual ~PatchInteractionModel();



    //- Selector
    static autoPtr<PatchInteractionModel<CloudType> > New
    (
        const dictionary& dict,
        CloudType& owner
    );
```

```
// Access

    //- Return name of velocity field
    const word& UName() const;


// Member Functions

    //- Convert interaction result to word
    static word interactionTypeToWord(const interactionType& itEnum);

    //- Convert word to interaction result
    static interactionType wordToInteractionType(const word& itWord);

    //- Apply velocity correction
    //  Returns true if particle remains in same cell
    virtual bool correct
    (
        typename CloudType::parcelType& p,
        const polyPatch& pp,
        bool& keepParticle,
        const scalar trackFraction,
        const tetIndices& tetIs
    );


    // I-O

        //- Write patch interaction info to stream
```

```
            virtual void info(Ostream& os);
};



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

} // End namespace Foam

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

#define makePatchInteractionModel(CloudType)                      \
                                                                  \
    typedef CloudType::kinematicCloudType kinematicCloudType;     \
    defineNamedTemplateTypeNameAndDebug                           \
    (                                                             \
        PatchInteractionModel<kinematicCloudType>,                \
        0                                                         \
    );                                                            \
    defineTemplateRunTimeSelectionTable                           \
    (                                                             \
        PatchInteractionModel<kinematicCloudType>,                \
        dictionary                                                \
    );


#define makePatchInteractionModelType(SS, CloudType)              \
                                                                  \
    typedef CloudType::kinematicCloudType kinematicCloudType;     \
    defineNamedTemplateTypeNameAndDebug(SS<kinematicCloudType>, 0); \
                                                                  \
    PatchInteractionModel<kinematicCloudType>::                   \
```

```
        adddictionaryConstructorToTable<SS<kinematicCloudType> >    \
            add##SS##CloudType##kinematicCloudType##ConstructorToTable_;


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

#ifdef NoRepository
#   include "PatchInteractionModel.C"
#endif

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

#endif

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

B.1.5 LocalInteraction.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 "LocalInteraction.H"

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

template<class CloudType>
Foam::LocalInteraction<CloudType>::LocalInteraction
(
    const dictionary& dict,
    CloudType& cloud
)
:
    PatchInteractionModel<CloudType>(dict, cloud, typeName),
    patchData_(cloud.mesh(), this->coeffDict()),
    nEscape_(patchData_.size(), 0),
    massEscape_(patchData_.size(), 0.0),
    nStick_(patchData_.size(), 0),
```

```
    massStick_(patchData_.size(), 0.0),
    writeFields_(this->coeffDict().lookupOrDefault("writeFields", false)),
    massEscapePtr_(NULL),
    massStickPtr_(NULL)
{
    if (writeFields_)
    {
   word massEscapeName(this->owner().name() + ":massEscape");
   word massStickName(this->owner().name() + ":massStick");
   Info<< "    Interaction fields will be written to " << massEscapeName
            << " and " << massStickName << endl;

        (void)massEscape();
        (void)massStick();
    }
    else
    {
        Info<< "    Interaction fields will not be written" << endl;
    }

    // check that interactions are valid/specified
    forAll(patchData_, patchI)
    {
        const word& interactionTypeName =
            patchData_[patchI].interactionTypeName();
 const typename PatchInteractionModel<CloudType>::interactionType& it =
            this->wordToInteractionType(interactionTypeName);

        if (it == PatchInteractionModel<CloudType>::itOther)
        {
            const word& patchName = patchData_[patchI].patchName();
```

```
   FatalErrorIn("LocalInteraction(const dictionary&, CloudType&)")
        << "Unknown patch interaction type "
        << interactionTypeName << " for patch " << patchName
        << ". Valid selections are:"
        << this->PatchInteractionModel<CloudType>::interactionTypeNames_
        << nl << exit(FatalError);
        }
    }
}


template<class CloudType>
Foam::LocalInteraction<CloudType>::LocalInteraction
(
    const LocalInteraction<CloudType>& pim
)
:
    PatchInteractionModel<CloudType>(pim),
    patchData_(pim.patchData_),
    nEscape_(pim.nEscape_),
    massEscape_(pim.massEscape_),
    nStick_(pim.nStick_),
    massStick_(pim.massStick_),
    writeFields_(pim.writeFields_),
    massEscapePtr_(NULL),
    massStickPtr_(NULL)
{}


// * * * * * * * * * * * * * * * * Destructor  * * * * * * * * * * * *//
```

```
template<class CloudType>
Foam::LocalInteraction<CloudType>::~LocalInteraction()
{}



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

template<class CloudType>
Foam::volScalarField& Foam::LocalInteraction<CloudType>::massEscape()
{
    if (!massEscapePtr_.valid())
    {
        const fvMesh& mesh = this->owner().mesh();

        massEscapePtr_.reset
        (
            new volScalarField
            (
                IOobject
                (
                    this->owner().name() + ":massEscape",
                    mesh.time().timeName(),
                    mesh,
                    IOobject::READ_IF_PRESENT,
                    IOobject::AUTO_WRITE
                ),
                mesh,
                dimensionedScalar("zero", dimMass, 0.0)
            )
        );
    }
```

```
    return massEscapePtr_();
}


template<class CloudType>
Foam::volScalarField& Foam::LocalInteraction<CloudType>::massStick()
{
    if (!massStickPtr_.valid())
    {
        const fvMesh& mesh = this->owner().mesh();

        massStickPtr_.reset
        (
            new volScalarField
            (
                IOobject
                (
                    this->owner().name() + ":massStick",
                    mesh.time().timeName(),
                    mesh,
                    IOobject::READ_IF_PRESENT,
                    IOobject::AUTO_WRITE
                ),
                mesh,
                dimensionedScalar("zero", dimMass, 0.0)
            )
        );
    }

    return massStickPtr_();
```

```
}


template<class CloudType>
bool Foam::LocalInteraction<CloudType>::correct
(
    typename CloudType::parcelType& p,
    const polyPatch& pp,
    bool& keepParticle,
    const scalar trackFraction,
    const tetIndices& tetIs
)
{
    label patchI = patchData_.applyToPatch(pp.index());

    if (patchI >= 0)
    {
        vector& U = p.U();
        bool& active = p.active();

        typename PatchInteractionModel<CloudType>::interactionType it =
            this->wordToInteractionType
            (
                patchData_[patchI].interactionTypeName()
            );

        switch (it)
        {
            case PatchInteractionModel<CloudType>::itEscape:
            {
                scalar dm = p.mass()*p.nParticle();
```

```
        keepParticle = false;
        active = false;
        U = vector::zero;
        nEscape_[patchI]++;
        massEscape_[patchI] += dm;
        if (writeFields_)
        {
            label pI = pp.index();
            label fI = pp.whichFace(p.face());
            massEscape().boundaryField()[pI][fI] += dm;
        }
        break;
    }
    case PatchInteractionModel<CloudType>::itStick:
    {
        scalar dm = p.mass()*p.nParticle();

        keepParticle = true;
        active = false;
        U = vector::zero;
        nStick_[patchI]++;
        massStick_[patchI] += dm;
        if (writeFields_)
        {
            label pI = pp.index();
            label fI = pp.whichFace(p.face());
            massStick().boundaryField()[pI][fI] += dm;
        }
        break;
    }
```

```
    case PatchInteractionModel<CloudType>::itRebound:
    {
        keepParticle = true;
        active = true;

        vector nw;
        vector Up;

 this->owner().patchData(p, pp, trackFraction, tetIs, nw, Up);

        // Calculate motion relative to patch velocity
        U -= Up;

        scalar Un = U & nw;
        vector Ut = U - Un*nw;

        if (Un > 0)
        {
            U -= (1.0 + patchData_[patchI].e())*Un*nw;
        }

        U -= patchData_[patchI].mu()*Ut;

        // Return velocity to global space
        U += Up;

        break;
    }
case PatchInteractionModel<CloudType>::itForder:
    {
        keepParticle = true;
```

```
          active = true;

          vector nw;
          vector Up;

this->owner().patchData(p, pp, trackFraction, tetIs, nw, Up);

          // Calculate motion relative to patch velocity
          U -= Up;

          scalar Un = U & nw;
          vector Ut = U - Un*nw;

         const scalar magU = mag(U);
   const vector Udir = U/magU;

   // determine impact angle, alpha
     const scalar alpha = mathematical::pi/2.0 - acos(nw & Udir);

     //determine the normal and tangential restitution coefficients

     //normal coefficient of restitution
     scalar e_n =
     0.988-0.78*alpha+0.19*sqr(alpha)-
     0.024*pow(alpha,3)+0.0027*pow(alpha,4);
     //tangential coefficient of restitution
     scalar e_t =
     1.0-0.78*alpha+0.84*sqr(alpha)-0.21*pow(alpha,3)+
     0.28*pow(alpha,4)-0.022*pow(alpha,5);

 //New velocity is the previous normal and tangential components
```

```
//multiplied by their respective restitution
//coefficients calculated above
U -= e_n*(Un*nw) + e_t*Ut;

        // Return velocity to global space
        U += Up;

        break;
    }
case PatchInteractionModel<CloudType>::itSommerfeldHubber:
    {
        keepParticle = true;
        active = true;

        vector nw;
        vector Up;

this->owner().patchData(p, pp, trackFraction, tetIs, nw, Up);

        // Calculate motion relative to patch velocity
        U -= Up;

        scalar Un = U & nw;
        vector Ut = U - Un*nw;

       const scalar magU = mag(U);
  const vector Udir = U/magU;

  // determine impact angle, alpha
    const scalar alpha = mathematical::pi/2.0 - acos(nw & Udir);
```

```
//determine the normal and tangential restitution coefficients

        //normal coefficient of restitution
        scalar e_n = 1.0-0.4159*alpha+0.4994*sqr(alpha)- 0.292*pow(alpha,3);
        //tangential coefficient of restitution
        scalar e_t = 1.0-2.12*alpha+3.0775*sqr(alpha)-1.1*pow(alpha,3);

    //New velocity is the previous normal and tangential components
    //multiplied by their respective restitution coefficients
    //calculated above
    U -= e_n*(Un*nw) + e_t*Ut;

            // Return velocity to global space
            U += Up;

            break;
        }
        default:
        {
            FatalErrorIn
            (
                "bool LocalInteraction<CloudType>::correct"
                "("
                    "typename CloudType::parcelType&, "
                    "const polyPatch&, "
                    "bool&, "
                    "const scalar, "
                    "const tetIndices&"
                ") const"
            )   << "Unknown interaction type "
        << patchData_[patchI].interactionTypeName()
```

```
            << "(" << it << ") for patch "
            << patchData_[patchI].patchName()
            << ". Valid selections are:" << this->interactionTypeNames_
            << endl << abort(FatalError);
            }
        }

        return true;
    }

    return false;
}


template<class CloudType>
void Foam::LocalInteraction<CloudType>::info(Ostream& os)
{
    // retrieve any stored data
    labelList npe0(patchData_.size(), 0);
    this->getModelProperty("nEscape", npe0);

    scalarList mpe0(patchData_.size(), 0.0);
    this->getModelProperty("massEscape", mpe0);

    labelList nps0(patchData_.size(), 0);
    this->getModelProperty("nStick", nps0);

    scalarList mps0(patchData_.size(), 0.0);
    this->getModelProperty("massStick", mps0);

    // accumulate current data
```

```
labelList npe(nEscape_);
Pstream::listCombineGather(npe, plusEqOp<label>());
npe = npe + npe0;

scalarList mpe(massEscape_);
Pstream::listCombineGather(mpe, plusEqOp<scalar>());
mpe = mpe + mpe0;

labelList nps(nStick_);
Pstream::listCombineGather(nps, plusEqOp<label>());
nps = nps + nps0;

scalarList mps(massStick_);
Pstream::listCombineGather(mps, plusEqOp<scalar>());
mps = mps + mps0;


forAll(patchData_, i)
{
    os  << "    Parcel fate (number, mass)      : patch "
        <<  patchData_[i].patchName() << nl
        << "      - escape                      = " << npe[i]
        << ", " << mpe[i] << nl
        << "      - stick                       = " << nps[i]
        << ", " << mps[i] << nl;
}

if (this->outputTime())
{
    this->setModelProperty("nEscape", npe);
    nEscape_ = 0;
```

```
        this->setModelProperty("massEscape", mpe);
        massEscape_ = 0.0;

        this->setModelProperty("nStick", nps);
        nStick_ = 0;

        this->setModelProperty("massStick", mps);
        massStick_ = 0.0;
    }
}


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

Appendix C

Matlab Script for Scar comparison

C.1 Procedure

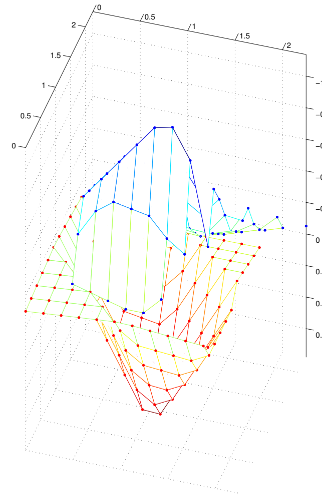
It is clear that the two scars to be compared (CFD and Experimantal) will not have the same number of points. What is needed then is to be able to compare the two sets of points. To achieve this, both scars are interpolated onto a new grid, whose size can be chosen and then the points can be compared. The only condition is that the coordinate system's origin is the same. However, having the same origin is the easiest part, since once both origins are chosen, one set of points can be changed to the second coordinate system by applying a simple transformation which displaces all the points and adapts them to the new coordinate system. In this appendix, the script includes an example of two sets of points with the same coordinate system but different coordinates. The figures show the results of running such script for this particular case.

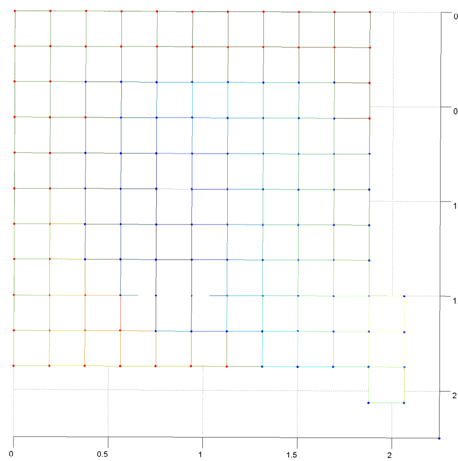
C.2 Matlab Script for Scar comparison

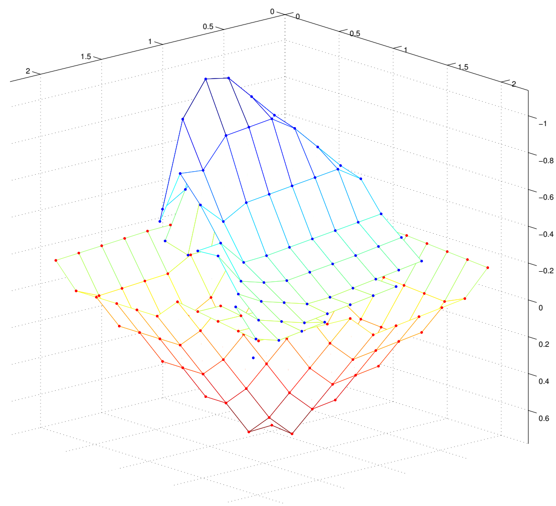
```
X1=[0 1 2 0 1 2 0 1 2]';%append transpose sign ' to  create column vector
Y1=[0 0 0 1 1 1 2 2 2]';
Z1=[-0.1 -0.15 -0.1 -0.15 0.9 0 -0.1 -0.1 0]';
```

```
X2=[0.25 0.75 1.75 0.4 0.75 1.75 0.4 0.75 2.25 0.5 1.25 1.25 0.575]';
Y2=[0.25 0.25 0.25 1.25 1.25 1.25 1.6 1.6 2.25 0.5 0.25 1.5 1.425]';
Z2=[0.15 -0.5 -0.15 -0.1 -1.2 -0.05 -0.3 -0.85 -0.05 -0.5 -0.4 -0.3 -0.35]';
%find
Mx=max(max(X1),max(X2));
My=max(max(Y1),max(Y2));
mx=min(min(X1),min(X2));
my=min(min(Y1),min(Y2));
%Create a grid with liear spacements between our max and min values
%of both sets of points in X and Y
xlin=linspace(mx,Mx,13);
ylin=linspace(my,My,13);
%Construct the common grid
[X,Y]=meshgrid(xlin,ylin);
%interpolate our Z coordinates to our new grid for both sets of points
Z_1=griddata(X1,Y1,Z1,X,Y,'linear')
Z_2=griddata(X2,Y2,Z2,X,Y,'linear')
mesh(X,Y,Z_1) %interpolated
mesh(X,Y,Z_2) %interpolated

axis tight;hold on
plot3(X,Y,Z_1,'.','MarkerSize',10,'color','red')
plot3(X,Y,Z_2,'.','MarkerSize',10,'color','blue')
```


0
0.5
1
1.5
2
2
1.5
1
0.5
0
−1
−0.8
−0.6
−0.4
−0.2
0
0.2
0.4
0.6


0
0.5
1
1.5
2
0
0.5
1
1.5
2


2
1.5
1
0.5
0
0
0.5
1
1.5
2
−1
−0.8
−0.6
−0.4
−0.2
0
0.2
0.4
0.6

Appendix D

LPT for Erosion Modelling in OpenFOAM

D.1 Introduction

The following peer reviewed report was developed for the CFD Course with Open Source Software taught by Håkan Nilsson in Charmers University of Technology and it can be found in `http://www.tfd.chalmers.se/~hani/kurser/OS_CFD_2013/`. The aim of this report is to provide information about the Lagrangian libraries available in OpenFOAM 2.2.x as well as a tutorial and instructions on how to implement an Euler-Lagrange solver in OpenFOAM. The following versions of OpenFOAM (2.3.x and 3.0) include an Euler-Lagrange solver for incompressible flows called DPMFoam. This solver implements the same linkage between the intermediate library and pimpleFoam as the one explained in this section. DPMFoam has an improved efficiency in the calculation of the variables of the discrete phase, allowing the implementation of a much higher number of particles. All the libraries, solvers and utilities developed within the frame of this work for erosion calculation and mesh deformation have been also upgraded to be used with OpenFOAM 2.3.x. This upgrade involved adding additional functions and changing some of the names of the templates in order for the libraries to work with the newer version of OpenFOAM.

D.2 Report

CFD WITH OPENSOURCE SOFTWARE

A COURSE AT CHALMERS UNIVERSITY OF TECHNOLOGY

TAUGHT BY HÅKAN NILSSON

Project work:

LPT for erosion modeling in OpenFOAM

Differences between solidParticle and kinematicParcel, and how to add erosion modeling

Developed for OpenFOAM-2.2.x

Author:
ALEJANDRO LÓPEZ

Peer reviewed by:
ABOLFAZL ASNAGI
OLIVIER PETIT

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.

February 4, 2014

D.2.1 Theoretical Background

D.2.2 Introduction

The aim of this tutorial is to effectively describe the available possibilities in OpenFOAM to simulate lagrangian inert particles and the different classes used for their modeling. The tutorial also tries to give a description of how to create an incompressible multiphase solver and how to pre-process, run and post-process a case involving an incompressible flow with inert Lagrangian particles in a three-dimensional domain.

D.2.3 Lagrangian Particle Tracking

When dealing with the movement of a group of particles inside a fluid, there are basically two different ways to approach the problem. In the Eulerian-Eulerian models, the particles are treated as a continuous phase and conservation equations are solved for the particulate phase. This method is suitable for large particle concentrations, where two-way coupling between the fluid and the particulate phases as well as particle-particle collisions are important. On the other hand, in the Eulerian-Lagrangian approach, the Eulerian continuum equations are solved for the fluid phase, while Newton's equations for motion are solved for the particulate phase in order to determine the trajectories of the particles (or groups of particles). The trajectories are obtained once the following equation for the particles has been solved:

$$m_p \frac{d\mathbf{u}_p}{dt} = \mathbf{F}_p \tag{D.1}$$

Once the force has been calculated, the trajectories are calculated by means of integration of the particle velocity:

$$\frac{dx_p}{dt} = \mathbf{u}_p \tag{D.2}$$

There are three different possibilities when constructing the equations to solve particle motion:

- One way coupling: particle influence on the fluid phase is neglected
- Two way coupling: the force the particles exert on the fluid is no longer neglected
- Four way coupling: also particle-particle collisions are taken into acount

For a more accurate selection of the correct approach, the particle mass loading, β, and the Stokes number, S_t might also be calculated [97].
The particle mass loading is expressed as follows:

$$\beta = \frac{\text{particulate mass per unit volume of flow}}{\text{fluid mass per unit volume of flow}} = \frac{r_p \rho_p}{r \rho} \tag{D.3}$$

Where r is a volume fraction and ρ is a density. Significant two-way coupling is expected for particle mass loadings greater than 0.2 and values greater than 0.6 indicate that particle collisions are likely in, at least, some parts of the domain.

The Stokes number defines the degree to which particle motion is tied to fluid motion:

$$S_t = \frac{\rho_p d_p^2 V_s}{18 \mu L_s} \tag{D.4}$$

Where d_p is the particle diameter, μ is the dynamic viscosity of the fluid and V_s and L_s are the characteristic velocity and length scales of the flow. For values of $S_t > 2.0$ the flow will be dominated by particle-wall interactions, whereas for $S_t < 0.25$ the effect of particle-wall interactions is negligible and the particles are tightly coupled to the fluid through viscous drag. At $S_t < 0.05$ the particles and the fluid are strongly coupled, while for $S_t << 0.01$ the particles are expected to respond almost instantaneously to any changes in the fluid flow.

The force balance on a spherical particle inside a viscous fluid is written:

$$\mathbf{F}_p = m_p \frac{d\mathbf{V}_p}{dt} = \mathbf{F}_D + \mathbf{F}_P + \mathbf{F}_g + \mathbf{F}_A \tag{D.5}$$

The drag force on spherical particles is then calculated as:

$$\mathbf{F}_D = m_p \frac{18\mu}{\rho_p d_p^2} \frac{C_D Re(Re)}{24} (\mathbf{u} - \mathbf{u}_p) \tag{D.6}$$

Where the drag coefficient C_D is obtained from the following equation:

$$C_D = \begin{cases} \frac{24}{Re} & \text{if } Re_p < 1 \\ \frac{24}{Re_p}(1 + 0.15 Re_p^{0.687}) & \text{if } 1 \leq Re_p \leq 1000 \\ 0.44 & \text{if } Re_p > 1000 \end{cases}$$

The gravity and buoyancy force is:

$$\mathbf{F}_g = m_p \mathbf{g}(1 + \frac{\rho}{\rho_p}) \tag{D.7}$$

The pressure gradient force is:

$$\mathbf{F}_P = \frac{1}{6}\pi d_p^3 \nabla P \tag{D.8}$$

And the added mass force:

$$\mathbf{F}_A = \frac{1}{12}\pi d_p^3 \rho_p \frac{d\mathbf{V}_p}{dt} \tag{D.9}$$

D.2.4 Erosion

Throughout the years many authors have published a very large amount of papers and literature on erosion, having most of them developed their own equations for predicting erosive wear taking into account different approaches and factors that may influence erosion. One of the most important authors in erosion literature and responsible for one of the most commonly used equations for erosion prediction is Iain Finnie [12]. The equation developed by Finnie yields the volume of material, Q removed by a single abrasive grain of mass, m, and velocity, V.

$$Q = \frac{mV^2}{p\psi K}\left(sin2\alpha - \frac{6}{K} sin^2\alpha\right) \quad \text{if } \tan\alpha \leq \frac{K}{6} \tag{D.10}$$

$$Q = \frac{mV^2}{p\psi K}\left(\frac{K cos^2\alpha}{6}\right) \quad \text{if } \tan\alpha > \frac{K}{6} \tag{D.11}$$

Where p is the plastic flow stress of the material being eroded, ψ is the ratio of the depth of contact to the depth of cut and K is the ratio of vertical to horizontal force

components acting on the particle.

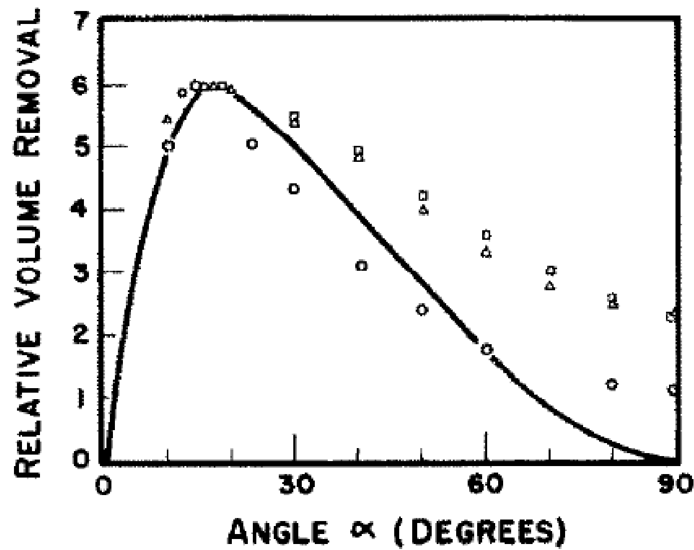


Figure D.1: Predicted variation of volume removal with angle of impingement for a single abrasive grain. Experimental points for erosion by many grains (△ copper, □ SAE I020 steel, ○ aluminium) are plotted so that the maximum erosion is the same in all cases.

Although this equation predicts no erosion for angles of impingement close to 90 degrees (figure D.1), it serves its purpose as a first approach, and, implementing an additional term which takes into account particle rotation at impingement, erosion at normal angles can also be predicted [13].

D.2.5 Implementation of LPT in OpenFOAM

D.2.6 Introduction

OpenFOAM provides the user with a number of possibilities in order to represent lagrangian particles, two of which are going to be commented here: The `/lagrangian/intermediate` library and the `/lagrangian/solidParticle` library. Some of the available examples and references are [100], [101] and [102], as well as the material at the course homepage [103].

D.2.7 SolidParticle Class

The `solidParticle` class enables the user to implement solid particles and to couple those to a given solver. Some examples on coupled solvers can be found in the references

mentioned above. This library can be found by typing the following in the terminal window:

```
cd $WM_PROJECT_DIR/src/lagrangian/solidParticle
```

This folder contains the files `solidParticle.C` and `solidParticleCloud.C`, which define how the particles are implemented and their behaviour. The `solidParticle.C` starts with a definition of the `solidParticleCloud` class, which is, in fact, a templated Cloud of solid particles. The `Foam::solidParticle::move` function includes the implementation of the Reynolds number, needed for the calculation of the drag force and the new velocity. This new velocity will be affected by the parameters yielded by the operations performed on the eulerian phase. The carrier phase properties are represented inside the code by `rhoc` for the density,`Uc` for the velocity and `nuc` for the viscosity. Also some additional functions are implemented that determine what happens when a patch is hit by a particle and these are different depending on whether it is a processor patch or a wall patch for example. Regarding the file `solidParticleCloud.C`, the constructor for the cloud of solid particles is defined first. This constructor reads the properties from a dictionary called `particleProperties` and initially only the density of the particles (`rhop`) the restitution ratio (`e`) and the friction coefficient (`mu`) are required. In the examples and tutorials cited above, various modifications are implemented to this class such as the addition of an injector and modification of particle shape among others, for which additional properties need to be defined. Also a more specific definition of the class and the functions of its different members can be found in those references.

D.2.8 The intermediate library

The `lagrangian/intermediate` library in OpenFOAM consists of a series of models, forces and CloudFunctionObjects templated for each of the classes derived from the parcel class. The classes available are the following ones:

- `CollidingParcel`
- `ReactingParcel`

- ReactingMultiphaseParcel
- ThermoParcel
- KinematicParcel

This report is mainly focused on the description of the KinematicParcel class and it will also try to briefly introduce the additional features of the CollidingParcel class. Before making any changes in the code, it is highly recommended to recompile the files needed in the $WM_PROJECT_USER_DIR/src/lagrangian directory. In order to do this, we can run the following commands in the terminal window:

```
cd $WM_PROJECT_USER_DIR
mkdir -p src/lagrangian
cp -r $FOAM_SRC/lagrangian/intermediate $WM_PROJECT_USER_DIR/src/lagrangian
```

Once this is done, the necessary files will be copied into the user source directory and the next step would be to recompile them. It is highly reccommended to recompile only the necessary files into the user directory both to save disk space and to make the compilation faster. In this case, only the necessary files to compile both kinematic libraries (kinematicCloud and kinematicClollidingCLoud) are recompiled. This way, the user is able to run one, two or four-way coupled simulations of inert particles.

D.2.9 KinematicParcel Class

While in the solidParticle class the particles are tracked individually, in the KinematicParcel class, a set of particles or computational parcel is tracked. This construction is made because it is usually too expensive in computational terms to simulate all the real particles. In order to capture the behaviour of the real particles, some real case properties are defined. Thus, the main difference between the solidParticle class and the kinematicParcel class is that the solidparticle class contains no parcel treatment, but only real particles. However, both classes share some of their member functions and both are derived from the particle class and their clouds are both templates of the Cloud class. Nevertheless, the kinematicParcel class complexity lies far beyond the solidParticle class one.

D.2.10 KinematicCloudProperties dictionary

In order to set up the properties of the parcel as well as the additional submodels a dictionary called `kinematicCloudProperties` is needed. An example of such a file can be found on Appendix 1. Additional examples can also be found by typing in the terminal window the following commands:

```
run
find tutorials/ -name kinematicCloudProperties
```

By running this command, the search is made inside the tutorials folder within the `run` directory. If one wishes to find examples inside the `$FOAM_TUTORIALS` directory, the commands to run in the terminal window would be the following ones:

```
find $FOAM_TUTORIALS -name kinematicCloudProperties
```

and once the appropriate tutorial has been found, it can be copied to the run directory by typing:

```
cp -r desiredTutorial/ $FOAM_RUN
```

The properties of our Lagrangian particles are going to be defined within this dictionary, such as the injection model, the forces on the particles and the cloudFunctionObjects, which will enable the user to output erosion rates. The first dictionary entry (`solution`) consists of a series of switches, `sourceTerms`, `interpolationSchemes`, and `integrationSchemes` The `coupled` option can be set to true or false and the user may choose between transient or steady-state solution by switching the boolean `transient` to yes or no respectively.

```
template<class CloudType>
template<class TrackData>
void Foam::KinematicCloud<CloudType>::evolveCloud(TrackData& td)
{
    if (solution_.coupled())
    {
```

```
        td.cloud().resetSourceTerms();
    }

    if (solution_.transient())
    {
        label preInjectionSize = this->size();

        this->surfaceFilm().inject(td);

        // Update the cellOccupancy if the size of the cloud has changed
        // during the injection.
        if (preInjectionSize != this->size())
        {
            updateCellOccupancy();

            preInjectionSize = this->size();
        }

        injectors_.inject(td);


        // Assume that motion will update the cellOccupancy as necessary
        // before it is required.
        td.cloud().motion(td);
    }
    else
    {
//        this->surfaceFilm().injectSteadyState(td);

        injectors_.injectSteadyState(td, solution_.trackTime());
```

```
        td.part() = TrackData::tpLinearTrack;
        CloudType::move(td,  solution_.trackTime());
    }
}
```

As it can be seen in the code above, depending on whether we choose the solution to be coupled, uncoupled, transient or steady-state, the `evolveCloud` function will perform different operations. For instance, in case the solution is transient, the function

```
Foam::InjectionModel<CloudType>::inject(TrackData& td)
```

will inject the parcels. However, if it is a steady-state solution, the injection will be performed by the function

```
Foam::InjectionModel<CloudType>::injectSteadyState
```

In the file `InjectionModel.C`, which can be found in

```
$WM_PROJECT_USER_DIR/src/lagrangian/intermediate/submodels/Kinematic/
InjectionModel/InjectionModel
```

both functions are implemented and, as an example, one of the main differences between them has to do with the mass that the injector is going to introduce inside our computational domain. In a steady state case, the total mass to be injected is equal to the mass flow rate that the user specifies inside the `kinematicCloudProperties` dictionary, while in a transient case, the time-step, the duration of the injection, the mass flow rate and other properties defined are taken into account to calculate the number of parcels the injector is going to introduce in the system per time-step in order to fulfill the user's requirements.

In the particular case of a coupled simulation, the `sourceTerms` dictionary entry allows the user to specify what kind of scheme to use, which can be set to `explicit` or `semiImplicit`, as well as the relaxation factors, which have to be preceded by the name of the field they are going to be applied on (default value is 1). All this data introduced by the user will be processed by `cloudSolution.C`, located inside the following directory:

`$WM_PROJECT_USER_DIR/src/lagrangian/intermediate/clouds/Templates/KinematicCloud/cloudSolution`

It is also inside this file where it can be found what the switch `active` does. In fact, if it is set to true, `cloudSolution` will call the function `read()` in order to set up how is the solution going to be obtained according to the rest of parameters specified in the dictionary.
The `cellValueSourceCorrection` switch, when set to on, activates the correction of the momentum transferred from the lagrangian phase to the carrier phase in case the simulation is coupled. This is done by manipulating the updated momentum for the lagrangian phase and dividing its cell value by the mass of the cell. The function is implemented in the following file:

`$WM_PROJECT_USER_DIR/src/lagrangian/intermediate/parcels/Templates/KinematicParcel/KinematicParcel.C`

The `interpolationSchemes` entry is used for the definition of how the fields relative to the lagrangian phase have to be interpolated. The last entry inside `solution` states which schemes going to be used when integrating the lagrangian fields. Options for the schemes are `Euler` or `analytical`. In what concerns the `constantProperties` dictionary entry, at least `rho0`, `youngsModulus` and `poissonsRatio` have to be specified.
Other parameters such as `parcelTypeId`, `rhoMin` and `minParticleMass` will be set to their default values in case they are not found (1e-15 is the default value for both density and mass). While the meaning of the last two parameters is fairly clear, the first one might not be so obvious. The `parcelTypeId` is just a form of identification of the particles belonging to this particular cloud. This might be useful if two or more different clouds with different properties are being post-processed. In this case, setting different `parcelTypeId` numbers, the different clouds will be flagged with different numbers (default is 1).

D.2.11 Submodels

The `submodels` directory contains the different templated models that can be added to the lagrangian particle cloud classes. The structure of the `submodels` directory in

what concerns the `KinematicCloud` class is as follows:

```
submodels/
---- CloudFunctionObjects
    ---- CloudFunctionObject
    ---- CloudFunctionObjectList
    ---- FacePostProcessing
    ---- ParticleCollector
    ---- ParticleErosion
    ---- ParticleTracks
    ---- ParticleTrap
    ---- PatchPostProcessing
    ---- VoidFraction
---- ForceTypes
    ---- ParticleForceList
---- Kinematic
    ---- CollisionModel
    ---- DispersionModel
    ---- InjectionModel
    ---- ParticleForces
    ---- PatchInteractionModel
    ---- SurfaceFilmModel
---- SubModelBase.C
---- SubModelBase.H
```

This structure is what should be seen if one wants to recompile the intermediate library only taking the `kinematicCloud` class into account. This is useful in case this class is the only one being used since the compilation will be much faster.

Injection Model

The different injection models available are the following ones:

- `cellZoneInjection`

- `coneInjection`
- `coneNozzleInjection`
- `fieldActivatedInjection`
- `inflationInjection`
- `injectionModel`
- `kinematicLookupTableInjection`
- `manualInjection`
- `noInjection`
- `patchInjection`
- `patchFlowRateInjection`

A quick description of the injection along with the entries needed in the dictionary can be found at each `.H` file inside the correspondent injection model folder, which can be reached by typing the following line in the terminal

```
cd $WM_PROJECT_USER_DIR/src/lagrangian/intermediate/submodels/Kinematic/
InjectionModel
```

Particle Forces

OpenFOAM allows the user to choose the forces to be included in the model. The forces inside the `ParticleForces` directory are:

- `Drag` Two possible drag models are available: `SphereDrag` and `NonSphereDrag`. The first one is a drag model assuming spherical particles while the latter is used for non-spherical particles and it is based on a coefficient obtained by dividing the area of a sphere with the same volume by the particle area. The Drag force for both cases is basically calculated as in equation D.6.

- `Lift` Both Saffman-Mei for spherical particles and Tomiyama for deformable bubbles lift models are implemented. The lift coefficient is calculated with the chosen model and then the force is calculated inside `LiftForce.C`, after which it is stored in `forceSuSp` This class is a helper container for both explicit and implicit terms where `Su` is the explicit contribution (directly calculated as a force), and `Sp` is the implicit contribution (calculated as $\frac{force}{velocity}$) so that the total contribution is calculated as $F = S_p(u - u_p) + S_u$.

- `Gravity` The Gravity force is calculated with equation D.7 and in the standard `KinematicParcel` class, a file named `g` with both units and value of the gravitational acceleration is needed inside the constant directory. The constructor of the cloud will ask for this file, which will be used for the calculation of the gravity force.

- `Paramagnetic` This model calculates the particle paramagnetic (magnetic field) force. Both the field and the magnetic susceptibility of the material are needed for the calculations.

- `PressureGradient` Function that calculates the pressure grandient force on the particles. An additional interpolation scheme has to be included for the `DUcDt` field, used for the calculations.

- `VirtualMass` Calculates the virtual mass force in coupled simulations in conjunction with the pressure gradient force. A dictionary with the virtual mass coefficient (`Cvm`) must be specified, along with the interpolation scheme for `DUcDt`. A typical value for the virtual mass coefficient is 0.5.

- `NonInertialFrame` Used to calculate the non inertial frame of reference force. The solver will look up inside the `kinematicCloudProperties` dictionary for the linear acceleration, angular velocity and angiular acceleration. If the user does not specify a name, default values will be chosen. If no value for accelerations and velocity is specified, zero values will be used as default, so the reference frame will remain static.

- `SRF` Allows calculation of the SRF (Simple Reference Frame) force. The class

contains a pointer to the SRF model being used so that the values needed will be extracted from the SRF properties

Distribution Models

In what concerns the injection of the lagrangian parcels, OpenFOAM contains a library of runtime-selectable distribution models.A distribution model is a function that, for a particular property defines quantitatively how the values of that property are distributed among the particles in the entire population. The current distribution models include:

- `exponential`
- `fixedValue`
- `general`
- `multiNormal`
- `normal`
- `RosinRammler`
- `uniform`

The equation used by each one of these distributioin models is specified inside the `.H` file inside OpenFOAM's `distributionModels` directory, which can be reached by running in the terminal window:

```
cd $FOAM_SRC/lagrangian/distributionModels
```

As an example, the Normal and the Rosin-Rammler distristributions are plotted below:

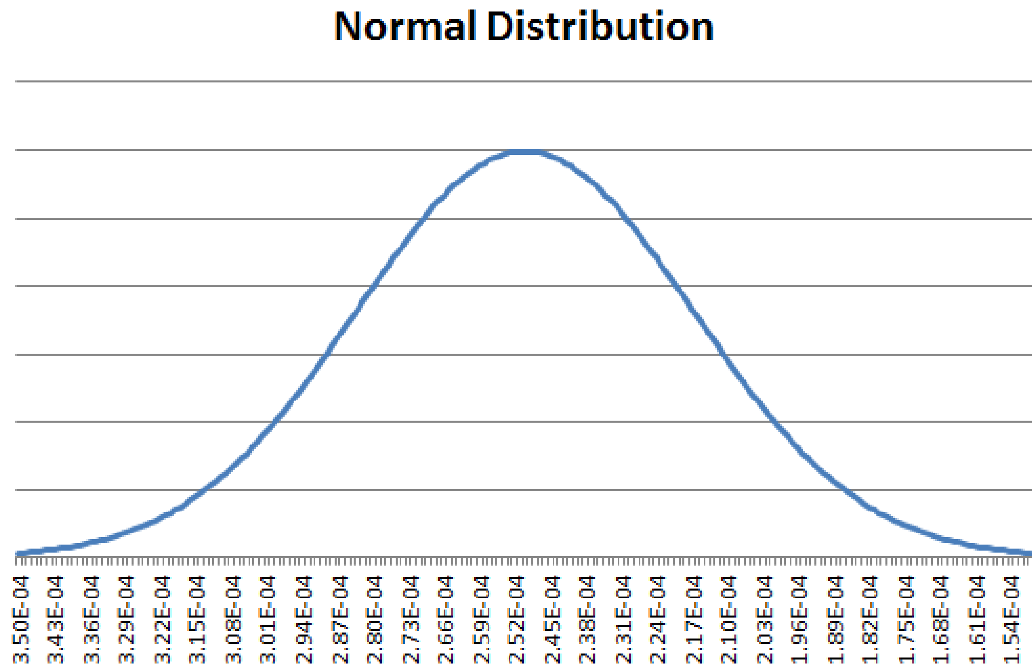


Figure D.2: Normal Distribution for Particle diameters between $150\mu m$ and $350\mu m$

CloudFunctionObjects

One of many useful classes OpenFOAM provides the users with is the `CloudFunctionObjects`. These are library functions that provide additional capabilities to the cloud-based solvers. The available ones in OpenFOAM 2.2.x are the following ones:

- `facePostProcesing` It records particle face quantities on user-specified face zone. It supports accumulated mass and average mass flux calculations.
- `particleCollector` Function that collects the parcel-mass and mass flow rate over a set of polygons, defined as a list of points.
- `particleTracks` It records all particle variables one each call to postFace.
- `particleTrap` Traps the particles within a given phase fraction for multiphase cases.
- `patchPostProcessing` Standard post-processing. It outputs the desired information at the user-specified patches.

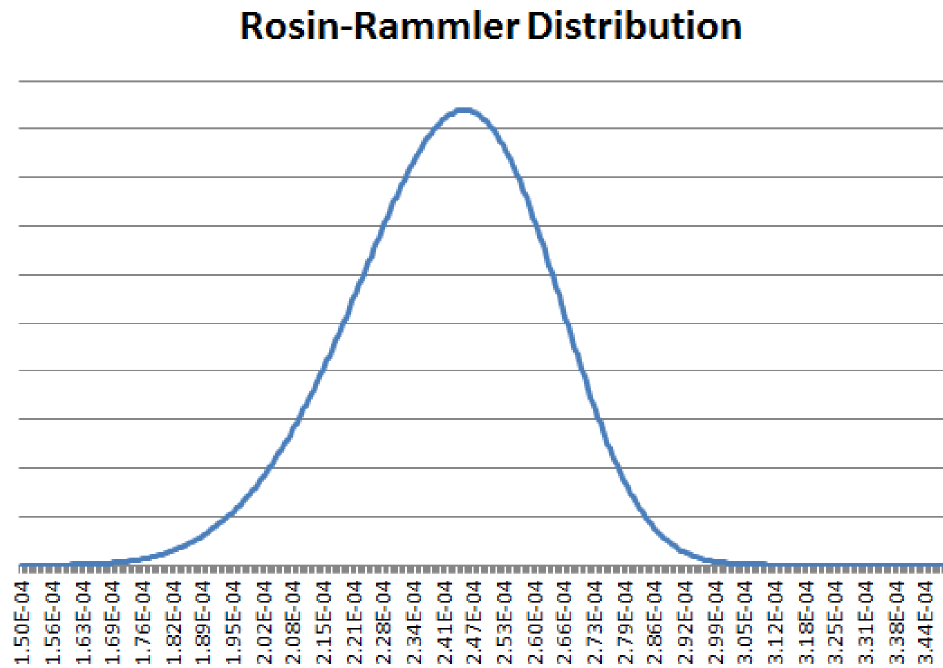


Figure D.3: Rosin-Rammlerl Distribution for Particle diameters between $150\mu m$ and $350\mu m$

- `voidFraction` Creates the particle void fraction on the carrier phase.
- `particleErosion` This function creates the particle erosion field on the user-specified patches. It outputs a `volScalarField` which, at each face, it will be the sum of the volume eroded by all the particle hits.

A set up example of the `particleErosion` `CloudFunctionObject` can be found in the `kinematicCloudProperties` dictionary in Appendix 1. For the rest of the functions, the set up is basically the same. Only some of the variables have to be changed because each function requires different input information. However, the information needed can be found inside the code, by entering the respective `.H` file inside each `CloudFunctionObject` folder. This folders can be reached by typing in the terminal:

```
cd $WM_PROJECT_USER_DIR/src/
```

And then,

```
cd lagrangian/intermediate/submodels/CloudFunctionObjects
```

In case the intermediate folder has been recompiled in the user directory.

D.2.12 Erosion modeling

Impingement information, such as impact speed and impact angle, is gathered as particles hit the wall of the geometry.

D.2.13 Implementation of Erosion Modelling in OpenFOAM

Taking a look inside the `ParticleErosion.C` file, the constructor is implemented and it requires, in this case, the names of the patches where it is going to be applied and the plastic flow stress of the material being eroded. Both ratios, depth of contact to length of cut and normal and tangential forces, are also read but in this case, if they are not found, the default ones are used (2 is the default value for both of them).

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


template<class CloudType>
void Foam::ParticleErosion<CloudType>::preEvolve()
{
    if (QPtr_.valid())
    {
        QPtr_->internalField() = 0.0;
    }
    else
    {
        const fvMesh& mesh = this->owner().mesh();

        QPtr_.reset
        (
            new volScalarField
            (
                IOobject
                (
                    this->owner().name() + "Q",
```

```
                    mesh.time().timeName(),
                    mesh,
                    IOobject::READ_IF_PRESENT,
                    IOobject::NO_WRITE
                ),
                mesh,
                dimensionedScalar("zero", dimVolume, 0.0)
            )
        );
    }
}
```

The `preEvolve` member function, as seen above, initializes the field. It can be seen that the name the field will be given is going to be the name of the cloud that is being tracked with a Q at the end. In case the `kinematicCloud` is being used, the erosion field will have the name of `kinematicCloudQ`. The member function in charge of gathering all the necessary information, manipulate it and store it inside the erosion field is called `postPatch`. Here is where the user can set up his own erosion model (different from the one that is already implemented) just by taking the necessary particle variables and changing the function into the desired one.

```
template<class CloudType>
void Foam::ParticleErosion<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)
{
    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;
    }

    //Calculate magnitude of the particle velocity at impingement
 const scalar magU = mag(U);
    //Udir is the velocity unitary vector, i.e, the direction of
    //the particle at impingement.
 const vector Udir = U/magU;

    // determine impact angle, alpha
 const scalar alpha = mathematical::pi/2.0 - acos(nw & Udir);

    const scalar coeff = p.nParticle()*p.mass()*sqr(magU)/(p_*psi_*K_);
```

```
        const label patchFaceI = pp.whichFace(p.face());
        scalar& Q = QPtr_->boundaryField()[patchI][patchFaceI];
        if (tan(alpha) < K_/6.0)
        {
            Q += coeff*(sin(2.0*alpha) - 6.0/K_*sqr(sin(alpha)));
        }
        else
        {
            Q += coeff*(K_*sqr(cos(alpha))/6.0);
        }
    }
}


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

If the user-specified patch is hit, the magnitude of the velocity of the impinging particle at that moment is calculated with the expression `mag(u)` and stored inside `magU`. In order to calculate the angle of impingement, the direction of the particle velocity is determined first ad stored in the `Udir` vector. The angle of impingement is the one between `Udir` and the patch normal direction, and it is stored inside `alpha`. the scalar `coeff` is the number of particles inside the parcel times the mass of those particles (i.e., the total mass) multiplied by the velocity and the constant coefficients: plastic flow stress and the two ratios. The field this `CloudFunctionObject` writes to the case directory is going to be zero everywhere but in the specified patches, where is is going to print a `nonuniform List<scalar>`, which will be the erosion rate at each face of the boundary patches specified. The equation used for the calculation of the erosion field is exactly equations D.10 and D.11.

D.2.14 Templating in OpenFOAM

Due to the fact that templates are very common within the `KinematicParcel` class, an introduction to templating could be of some use in order to be able to understand and customize Lagrangian simulations in OpenFOAM.
C++ requires the declaration of variables, functions, and most other kinds of entities using specific types. However, a lot of code looks the same for different types. Templates are functions or classes that are written for one or more types not yet specified. When using a template, the types are passed as arguments, explicitly or implicitly.

D.2.15 Function Templates [104] [98] [62]

A function template is used to represent a family of functions. The difference with ordinary functions is that, in the template, some of the parameters are left undetermined, i.e., parametrized. For example, in case one wants to create a function that returns the greater one of two objects:

```
template <typename T>
inline T const& max (T const& a, T const& b)
{
return a<b?b:a;
}
```

The keyword `typename` introduces a type parameter. This is the most common one but it is not the only possible kind of parameter. The template parameters have to be announced with a syntax like:

```
template<parameters separated by commas>
```

In the example, the type parameter is `T`. This type parameter can have any identifier. It represents an arbitrary type that is going to be specified when the function is called. The condition to use a particular type is that it supports the operation that is being done in the function. For instance, in the example, the type specified has to support the operator `<`. The word `typename` may also be substituted by the word `class`. This keyword was actually the only way to introduce type parameters until the use of the

keyword `typename` was enabled and it still remains valid. Normally, templates aren't compiled into single entities that can handle any type. Instead, different entities are generated from the template for every type for which the template is used. The process of replacing the template parameters by concrete types is called instantiation. In the example above the template can be instantiated:

```
inline int const& max (int const& a, int const& b)
{
// if a < b then use b else use a
return a<b?b:a;
}
```

for integers,

```
const double& max (double const&, double const&);
```

for double,

```
const std::string& max (std::string const&, std::string const&);
```

for strings and so on.

OpenFOAM contains a very large amount of templates. As an example, the following functions are used inside the `KinematicParcel` class to interpolate the density and the velocity. However,they have different types, being the density a scalar and the velocity a vector. That is why the generic type of `Foam::interpolation` is redefined for each one of the functions: scalar for the density and vector for the velocity.

```
template<class ParcelType>
template<class CloudType>
inline const Foam::interpolation<Foam::scalar>&
Foam::KinematicParcel<ParcelType>::TrackingData
<CloudType>::rhoInterp() const
{
    return rhoInterp_();
}
```

```
template<class ParcelType>
template<class CloudType>
inline const Foam::interpolation<Foam::vector>&
Foam::KinematicParcel<ParcelType>::TrackingData
<CloudType>::UInterp() const
{
    return UInterp_();
}
```

An attempt to instantiate a template for a type that doesn't support all the operations used within it will result in a compile-time error. Thus, templates are compiled twice: without instantiation, the template code is checked for correct syntax and when instantiated, it is checked that the calls are valid. When function templates are called for some arguments, the template parameters are determined by the arguments passed. If integers are passed as arguments, the compiler must conclude that the parameters are integers. In templating, there are two different kinds of parameters:

- Template parameters: These are declared in angle brackets (`template<typename T>`) before the function template name.
- Call parameters: Declared in parentheses (`max (T const& a, T const& b)`)after the function template name.

Also, like ordinary functions, function templates can be overloaded. There can be different function definitions with the same function name so that when that name is used in a function call, a C++ compiler must decide which one of the various candidates to call.

D.2.16 Class Templates [104] [98] [62]

Similar to functions, classes can also be parameterized with one or more types. Container classes, which are used to manage elements of a certain type, are a typical example of this feature. The declaration procedure is very similar to function templates.

Just before the declaration, a statement declares an identifier as a type parameter. Inside the class, the type parameter can be used like any other type in oder to declare members and member functions. Because templates are compiled when required, the implementation of a template class function must be in the same file as its declaration. The class template declaration starts with the same syntax as the function templates:

```
template<class T>
class Item
```

The keyword class is used twice. The first one defines the template type specification, and the second one is the C++ class declaration. While in funciton templates, it is the compiler the one that deduces the template type arguments, in class templates the user must explicitly pass the template type (in angle brackets `<>`).
An additional feature of templates is specialization. This feature enables the user to define a different implementation for a template when a specific type is passed as template parameter. An example of specialization could be as follows:

```
 template <> class myclass <char> { ... };
```

This allows the definition of a specific implementation when the argument is of type char. The fact that the angle brackets are empty (`<>`) allows the identification of this structure as a template specification.

D.2.17 Coupling of the kinematicCloud class and an incompressible solver

D.2.18 Uncoupled Lagrangian Particle Tracking

First of all, it is best if the necessary libraries to couple the incompressible solver are recompiled inside the `$WM_PROJECT_USER_DIR` directory. An `src/lagrangian` directory should have been created inside the user directory by typing:

```
cd $WM_PROJECT_USER_DIR
mkdir -p src/lagrangian/
```

As explained at the beginning of this chapter. The -p argument is used in this case to create the nested directories. Once this has been done, the next step is to copy the necessary files for compilation.

```
cp -r $FOAM_SRC/lagrangian/intermediate
$WM_PROJECT_USER_DIR/src/lagrangian
```

Now all the files inside the intermediate library have been copied to the user directory and the next step is to recompile them. Here, there are two possibilities: Recompile them without changing anything inside the `\Make` directory or change a few lines there and then recompile the library inside the user's `src/` directory. The basic difference is the compilation time. If one is going to be constantly working with this library (or any other library) and assuming that, in the process of adding new lines to the source code and changing the functions, the library is going to be recompiled, it is more efficient to just recompile the part of the code being used, since this will save a lot of time in compilation time. In this case, as the libraries that are going to be used are the ones that have the word `kinematic` on them, the rest of the parcel types can be deleted from the `Make/files` directory, leaving only the following:

```
PARCELS=parcels
BASEPARCELS=$(PARCELS)/baseClasses
DERIVEDPARCELS=$(PARCELS)/derived

CLOUDS=clouds
BASECLOUDS=$(CLOUDS)/baseClasses
DERIVEDCLOUDS=$(CLOUDS)/derived



/* Cloud base classes */
$(BASECLOUDS)/kinematicCloud/kinematicCloud.C

/* kinematic parcel sub-models */
KINEMATICPARCEL=$(DERIVEDPARCELS)/basicKinematicParcel
```

```
$(KINEMATICPARCEL)/defineBasicKinematicParcel.C
$(KINEMATICPARCEL)/makeBasicKinematicParcelSubmodels.C

KINEMATICCOLLIDINGPARCEL=$(DERIVEDPARCELS)/basicKinematicCollidingParcel
$(KINEMATICCOLLIDINGPARCEL)/defineBasicKinematicCollidingParcel.C
$(KINEMATICCOLLIDINGPARCEL)/makeBasicKinematicCollidingParcelSubmodels.C

submodels/Kinematic/PatchInteractionModel/
LocalInteraction/patchInteractionData.C
submodels/Kinematic/PatchInteractionModel/
LocalInteraction/patchInteractionDataList.C

KINEMATICINJECTION=submodels/Kinematic/InjectionModel
$(KINEMATICINJECTION)/KinematicLookupTableInjection/
kinematicParcelInjectionData.C
$(KINEMATICINJECTION)/KinematicLookupTableInjection/
kinematicParcelInjectionDataIO.C
$(KINEMATICINJECTION)/KinematicLookupTableInjection/
kinematicParcelInjectionDataIOList.C

/* integration schemes */
IntegrationScheme/makeIntegrationSchemes.C

/* phase properties */
phaseProperties/phaseProperties/phaseProperties.C
phaseProperties/phaseProperties/phasePropertiesIO.C
phaseProperties/phasePropertiesList/phasePropertiesList.C

/* Additional helper classes */
clouds/Templates/KinematicCloud/cloudSolution/cloudSolution.C
```

```
LIB = $(FOAM_USER_LIBBIN)/libkinematiclagrangianIntermediate
```

Inside the `\Make` directory, the `files` file should be similar to the one above. It is important to remember changing the location directory into the user's one. In this case the library is written inside `username-2.2.x/plattforms/linux64GccDPOpt/lib` (the folder containing the lib and bin directories may a slightly different name).

After this, the library can be recompiled by typing the following two commands in the terminal window:

```
wclean lib
wmake libso
```

Once the library is compiled, the next step is to link it to the incompressible solver that is going to be created. In this first case, an incompressible transient uncoupled solver is going to be created. For low particle concentrations, this is a sufficiently accurate assumption. The steps to create the solver are the following ones:

- Create the solver directory inside the user's `applications/solvers` directory and copy the original pimpleFoam solver into that directory by doing:

  ```
  cd $WM_PROJECT_USER_DIR/ applications/solvers/
  mkdir pimpleKinematicFoam
  ```

- Copy the pimpleFoam solver into the created directory

  ```
  cd pimpleKinematicFoam
  cp -r $WM_PROJECT_DIR/ applications/solvers/incompressible/pimpleFoam/* .
  ```

  This will copy all the files inside the pimpleFoam directory into the user directory (also pimpleDyMFoam and SRFPimpleFoam solvers will be copied, but one can get rid of them easily, if they are not going to be used by doing
  `rm -r pimpleDyMFoam SRFPimpleFoam` The next step is to change the name of the solver to the desired one by doing:

```
mv pimpleFoam.C pimpleKinematicFoam.C
```

- Modify the `Make/files` and `Make/options`

  Inside files, the user should have:

```
pimpleKinematicFoam.C
EXE = $(FOAM_USER_APPBIN)/pimpleKinematicFoam
```

Which tells the compiler which files to compile and to store the app created inside the user directory.

The options file should contain the following lines for the compiler to know where to look for the files:

```
LIB_USER_SRC = $(WM_PROJECT_USER_DIR)/src


EXE_INC = \
-I$(LIB_SRC)/turbulenceModels/incompressible/turbulenceModel \
-I$(LIB_SRC)/transportModels \
-I$(LIB_SRC)/transportModels/incompressible/singlePhaseTransportModel \
-I$(LIB_SRC)/finiteVolume/lnInclude\
-I$(LIB_SRC)/meshTools/lnInclude \
-I$(LIB_SRC)/fvOptions/lnInclude \
-I$(LIB_SRC)/sampling/lnInclude \
-I$(LIB_SRC)/lagrangian/basic/lnInclude \
-I$(LIB_SRC)/regionModels/surfaceFilmModels/lnInclude \
-I$(LIB_SRC)/regionModels/regionModel/lnInclude \
-I$(LIB_SRC)/lagrangian/intermediate/lnInclude \
-I$(LIB_USER_SRC)/lagrangian/intermediate/lnInclude \
#swap LIB_USER_SRC for LIB_SRC in case the $FOAM_SRC library is used
-I$(LIB_SRC)/lagrangian/distributionModels/lnInclude \
-I$(LIB_SRC)/thermophysicalModels/specie/lnInclude \
-I$(LIB_SRC)/thermophysicalModels/basic/lnInclude \
```

```
-I$(LIB_SRC)/thermophysicalModels/properties/liquidProperties/
lnInclude \
-I$(LIB_SRC)/thermophysicalModels/properties/liquidMixtureProperties/
lnInclude \
-I$(LIB_SRC)/thermophysicalModels/properties/solidProperties/
nInclude \
-I$(LIB_SRC)/thermophysicalModels/properties/solidMixtureProperties/
lnInclude \
-I$(LIB_SRC)/thermophysicalModels/reactionThermo/lnInclude \
-I$(LIB_SRC)/thermophysicalModels/SLGThermo/lnInclude \
-I$(LIB_SRC)/thermophysicalModels/radiationModels/lnInclude \
-I$(LIB_SRC)/dynamicFvMesh/lnInclude \
-I$(LIB_SRC)/sampling/lnInclude



EXE_LIBS = \
    -lincompressibleTurbulenceModel \
    -lincompressibleRASModels \
    -lincompressibleLESModels \
    -lincompressibleTransportModels \
    -lfiniteVolume \
    -lmeshTools\
    -lfvOptions \
    -llagrangian\
    -llagrangianIntermediate \
    -lkinematiclagrangianintermediate \
    -lthermophysicalFunctions \
    -lsurfaceFilmModels \
    -ldistributionModels \
    -lregionModels \
```

```
    -lspecie \
    -lfluidThermophysicalModels \
    -lliquidProperties \
    -lliquidMixtureProperties \
    -lsolidProperties \
    -lsolidMixtureProperties \
    -lreactionThermophysicalModels \
    -lSLGThermo \
    -lradiationModels \
    -lLESdeltas \
    -lcompressibleTurbulenceModel \
    -lcompressibleRASModels \
    -lcompressibleLESModels \
    -lregionModels \
    -lsurfaceFilmModels \
    -ldynamicFvMesh \
    -lsampling
```

It is essential to tell the compiler where to look for the user's library. The purpose of the first line is to let the compiler know that when it reads, `LIB_USER_SRC`, it should go to the `WM_PROJECT_USER_DIR/src` directory, and there, look for the intermediate library.

- Once this is ready, it is time to modify the `createFields.H` file:

  At the beginning of the file the following lines are added in order to be able to look for some properties defined in the `transportProperties` dictionary for further manipulation:

```
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")
);
```

Where `rhoInf` is the density of the carrier phase defined in the mentioned dictionary.

- A couple of fields for the carrier phase density and viscosity have to be created too. It is an incompressible solver and they are not going to vary, but the `KinematicCloud` class needs those for the constructor. The first of those fields is the density:

```
volScalarField rhoInf
(
    IOobject
    (
        "rho",
        runTime.timeName(),
        mesh,
        IOobject::NO_READ,
```

```
            IOobject::AUTO_WRITE
        ),
        mesh,
        rhoInfValue
    );
```

And the second one is the dynamic viscosity:

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

The dynamic viscosity is to be added just after the following lines:

```
autoPtr<incompressible::turbulenceModel> turbulence
(
    incompressible::turbulenceModel::New(U, phi, laminarTransport)
);
```

- The last modification inside the `createFields.H` file is to include the constructor for the `KinematicCloud` class:

```
    word kinematicCloudName("kinematicCloud");
```

```
    args.optionReadIfPresent("cloudName", kinematicCloudName);

Info<< "Constructing kinematicCloud " << kinematicCloudName << endl;
    basicKinematicCloud kinematicCloud
    (
        kinematicCloudName,
        rhoInf,
        U,
        mu,
        g
    );
```

As seen in this last piece of code, the constructor needs the density of the carrier phase and the dynamic viscosity, which were created in the previous step. This constructor will also ask for the gravity, and that means, that the user must have a `g` file inside the `constant` directory with the value and units of the gravity force so that the `readGravitationalAcceleration.H` header, which is going to be included later, is able to read it from that location.

- The modifications to the `pimpleKinematicFoam.C` file are the following ones:

  Copy the following line after `#include "IOMRFZoneList.H"`:

```
#include "basicKinematicCloud.H"
```

  This tells the compiler which cloud is going to be used. If four way coupling was necessary, the file to paste inside the `#include` statement would be `basicCollidingCloud.H`.

  Include after `int main(int argc, char *argv[])` the following lines:

```
 argList::addOption
    (
        "cloudName",
        "name",
```

```
        "specify alternative cloud name. default is 'kinematicCloud'"
    );
```

Which adds the option for the user to specify an alternative cloud name.

Then, after `#include "createMesh.H"`:

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

So the solver knows how and where from to read the gravitational acceleration to use it for both calculations and the construction of the cloud.

- Finally, just before `runTime.write();` the following lines are added too:

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

- Once all this is completed, the solver can be compiled and, hopefully, the process will not output any errors. In order to compile, inside the solver directory run, as always:

```
wclean
wmake
```

This is the procedure to create the solver. In the next chapter, the necessary files for running a case will be specified.

D.2.19 Coupled Lagrangian Particle Tracking

As it can be seen in the `calc` function displayed below, the term responsible for the coupling is `Su()`.

```
template<class ParcelType>
template<class TrackData>
void Foam::KinematicParcel<ParcelType>::calc
```

```
(
    TrackData& td,
    const scalar dt,
    const label cellI
)
{
    // Define local properties at beginning of time step
    // ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    const scalar np0 = nParticle_;
    const scalar mass0 = mass();

    // Reynolds number
    const scalar Re = this->Re(U_, d_, rhoc_, muc_);



    // Sources
    //~~~~~~~~

    // Explicit momentum source for particle
    vector Su = vector::zero;

    // Linearised momentum source coefficient
    scalar Spu = 0.0;

    // Momentum transfer from the particle to the carrier phase
    vector dUTrans = vector::zero;



    // Motion
    // ~~~~~~
```

```
    // Calculate new particle velocity
this->U_ = calcVelocity(td, dt, cellI, Re, muc_, mass0, Su, dUTrans, Spu);



    // Accumulate carrier phase source terms
    // ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    if (td.cloud().solution().coupled())
    {
        // Update momentum transfer
        td.cloud().UTrans()[cellI] += np0*dUTrans;

        // Update momentum transfer coefficient
        td.cloud().UCoeff()[cellI] += np0*Spu;
    }
}
```

And the function to calculate the new particle velocity (`calcVelocity`):

```
        //- Calculate new particle velocity
        template<class TrackData>
        const vector calcVelocity
        (
            TrackData& td,
            const scalar dt,        // timestep
            const label cellI,      // owner cell
            const scalar Re,        // Reynolds number
            const scalar mu,        // local carrier viscosity
            const scalar mass,      // mass
            const vector& Su,       // explicit particle momentum source
            vector& dUTrans,        // momentum transfer to carrier
            scalar& Spu             // linearised drag coefficient
        ) const;
```

If the solution is properly coupled, the solver will write out two fields:

- nameoftheCloud:Ucoeff It is of type volscalarField::DimensionedInternalField It is a `nonuniform List<vector>`
- nameoftheCloud:Utrans It is of type volVectorField::DimensionedInternalField It is a `nonuniform List<scalar>`

However, if the solution is not coupled, these fields will be written to the `runTime` but instead of being non-uniform lists, they will be `uniform 0;` and `uniform (0 0 0);` respectively. First, and provided that the `KinematicClass` is being coupled to an incompressible solver, the momentum will have to be divided by the density of the carrier phase. In pimpleFoam and the rest of the incompressible solver in OpenFOAM, the equations are divided by the carrier phase density. An easy way of doing this is creating the inverse of the density and then, in the file`UEqn.H`, multiply the momentum by that inverse, so there will be no mismatched units. Inside the `createFields.H` the following line is added after the density is read from the `particleProperties` dictionary:

```
dimensionedScalar invrhoInf("invrhoInf",(1.0/rhoInfValue));
```

Then, in order to create the coupled version of the solver it is necessary to include this momentum transfer into the `UEqn.H`, so the file will look now like:

```
// Solve the Momentum equation


tmp<fvVectorMatrix> UEqn
(
    fvm::ddt(U)
  + fvm::div(phi, U)
  + turbulence->divDevReff(U)
 ==
     fvOptions(U)
    +invrhoInf*kinematicCloud.SU(U)
```

```
);

UEqn().relax();

fvOptions.constrain(UEqn());

volScalarField rAU(1.0/UEqn().A());

if (pimple.momentumPredictor())
{
    solve
    (
        UEqn()
         ==
        -fvc::grad(p)
    );

    fvOptions.correct(U);
}
```

Finally, once all the changes are done and the files are saved the final step is to run in the terminal (inside the solver directory):

```
wclean
wmake
```

Given that the two-way coupling is going to be switched on and off inside the `kinematicCloudProperties` dictionary, it is probably a good option to create a unique solver and decide whether the simulation is going to be coupled or uncoupled switching from on to off in the mentioned dictionary.

D.2.20 Preprocessing

D.2.21 Geometry definition

The geometry chosen for this tutorial is a very simple one, consisting of a 100 mm length quadrangular pipe with a 90 degrees bend (figure D.4).

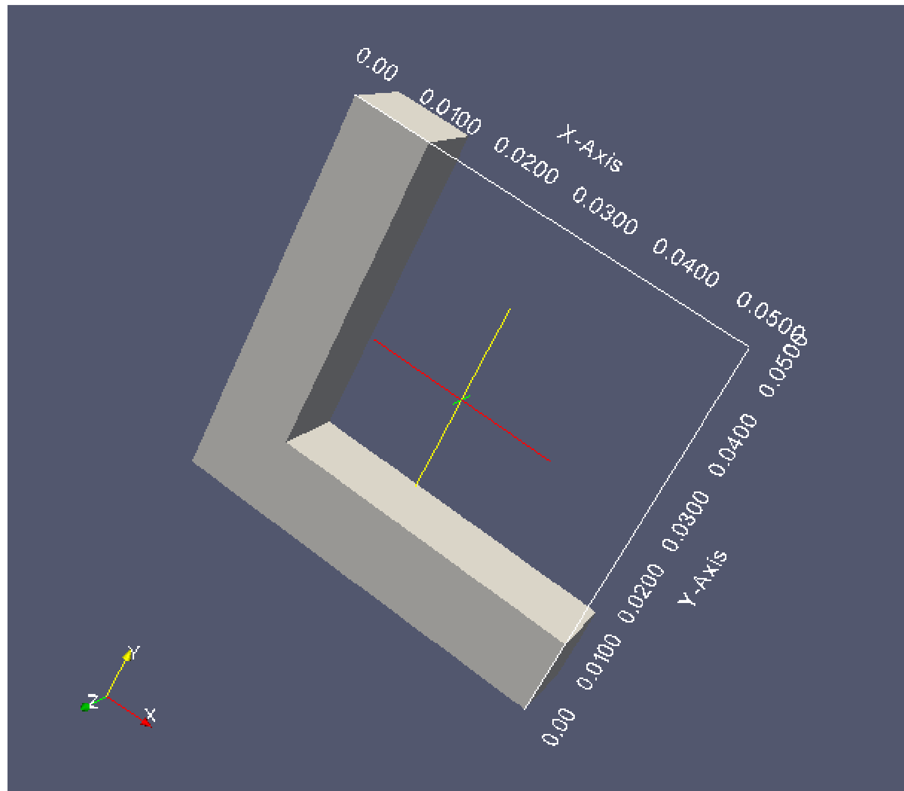


Figure D.4: Geometry of the pipe used for the tutorial case.

The geometry is defined in the `blockMeshDict` dictionary and in order to generate the rest of the mesh files needed by OpenFOAM, the `blockMesh` command has to be run in the case directory.

D.2.22 The 0/ directory

In order to have a more efficient case solution it is recommended to run the case without the particles until it reaches the steady state (`simpleFoam` is the most suitable one for this simulation) and then use the command,

```
mapFields /Path-To-Steady-State-Case-Directory -consistent
```

to map the fields obtained for the fluid flow into the transient case with particles and use them as initial conditions in our 0 directory. Regarding the application settings, before running `mapFields`, the starting time in the transient case has to be the same one as the time step being mapped from the steady state solution, and the directory created by the application and containing the non-uniform scalar and vector fields has to be renamed as `0`, once `mapFields` has finished the transfer.
For the steady-state case, three directories are necessary: `0`, `constant` and `system`. Once the steady-state case has been defined, the command to run it is as follows:

```
simpleFoam >&log&
```

The calculation process can also be viewed (and stopped with ctrl + C) in the terminal typing:

```
tail -f log
```

The case can also be killed by typing,

```
pidof simpleFoam
```

Which outputs the PID of the process and then, being PID the number obtained in the terminal window, run,

```
kill PID
```

Once the steady state case fields are correctly mapped into the transient case directory, it is time to set up the necessary files for the lagrangian simulation.

D.2.23 The constant/ directory

Inside the `constant` directory the user must specify the properties of the lagrangian simulation in a dictionary called `kinematicCloudProperties`. The example used for this tutorial can also be found in Appendix 1. A file named g is necessary for the construction of the cloud, as explained before. This file must, at least, contain:

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       uniformDimensionedVectorField;
    location    "constant";
    object      g;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *//

dimensions      [0 1 -2 0 0 0 0];
value           ( 0 -9.81 0 );
```

Where the gravity vector is specified, depending on which reference is being used. For this case, the gravity is in the -$\hat{j}$ direction.

In this particular case, a $k-\epsilon$ model has been chosen. For this, a `RASProperties` file is also required containing:

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "constant";
    object      RASProperties;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
```

```
RASModel        kEpsilon;


turbulence      on;


printCoeffs     on;
```

And as explained before, the `transportProperties` file, which should look similar to:

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "constant";
    object      transportProperties;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *//


transportModel  Newtonian;


nu              nu [ 0 2 -1 0 0 0 0 ] 1e-06;
rhoInf        rhoInf  [ 1 -3 0 0 0 0 0] 1000;
```

D.2.24 The system/ directory

The files inside the system directory will be the usual ones, i.e., `fvSchemes`, `fvSolution` and `controlDict`. However, it is good to pay some attention in this kind of simulations to the Courant number. The Courant number is defined as:

$$C = \frac{u\Delta t}{\Delta x} \tag{D.12}$$

Being u the velocity, Δx the space interval and Δt the time interval. Keeping the courant number under the value of 1 will help the solution converge, specially when

dealing with coupled simulations.

D.2.25 Running the case

Once the case is properly set up, it can be run by typing:

```
pimpleKinematicFoam >&log&
```

which will allow later the use of `foamLog` to extract the residuals relative to each of the variables for plotting.

D.2.26 Postprocessing

D.2.27 Lagrangian Particles in Paraview

There are actually two ways for visualization of the lagrangian cloud in paraview. The first one is to transform the case data into VTK format by doing:

```
foamToVTK
```

The second one is to run `paraFoam` in the terminal window and then click on "Skip Zero Time" (no parcels have been released at zero time. That is why no lagrangian fields or cloud are available for display). Once this is done, any of the lagrangian fields as well as the kinematic cloud can be displayed with paraview just by checking the box relative to each one of them.

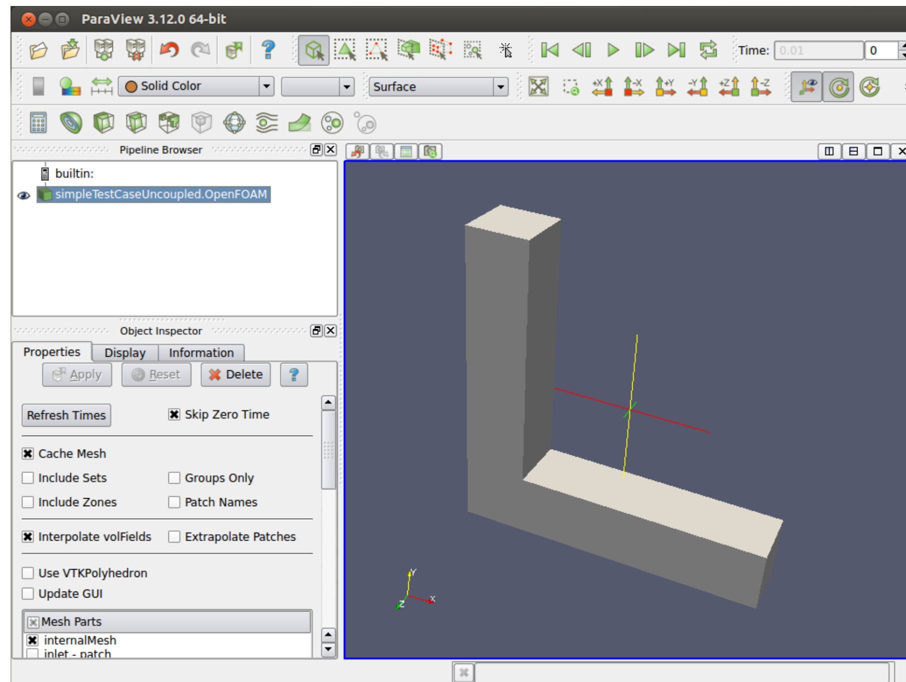


Figure D.5: Check "Skip Zero Time" box

D.2.28 Results of Coupled and Uncoupled Simulations

As expected, in this particular case, the case with two-way coupling is practically identical to the one-way coupled. This is due to the fact that the momentum transferred between phases is actually negligible, being the values very close to zero. In case of the uncoupled simulation, the solver allows the user to set up a higher courant number (`maxCo` in `controlDict`), which, consequently will enlarge the time step, and will be reflected in a much faster simulation. However, in the coupled case, instabilities might appear when trying to set a high courant number, causing the solver to diverge. Thus, the courant number to be set up in the `controlDict` dictionary must be carefully chosen, taking into account the size of our mesh the velocity and the time-step.

D.2.29 Post-processing erosion in Paraview 3.12.0

In order to postprocess erosion in Paraview, the "kinematicCloudQ" field box has to be checked ito display the erosion field as shown in picture D.8.

What paraview is representing in the erosion contours is the volume of material eroded, as the sum of the material eroded by each of the individual impacts of the

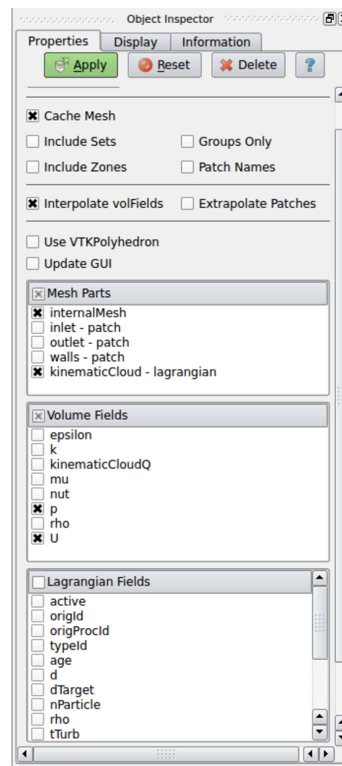


Figure D.6: Check "kinematicCloud-lagrangian" and any of the available lagrangian fields

particles at each face.

D.2.30 Report Appendix 1

D.2.31 kinematicCloudProperties Dictionary

```
/*--------------------------------*- C++ -*--------------------*\
| =========                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
|  \\    /   O peration     | Version:  2.2.0
|   \\  /    A nd           | Web:      www.OpenFOAM.org
|    \\/     M anipulation  |
\*-------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
```

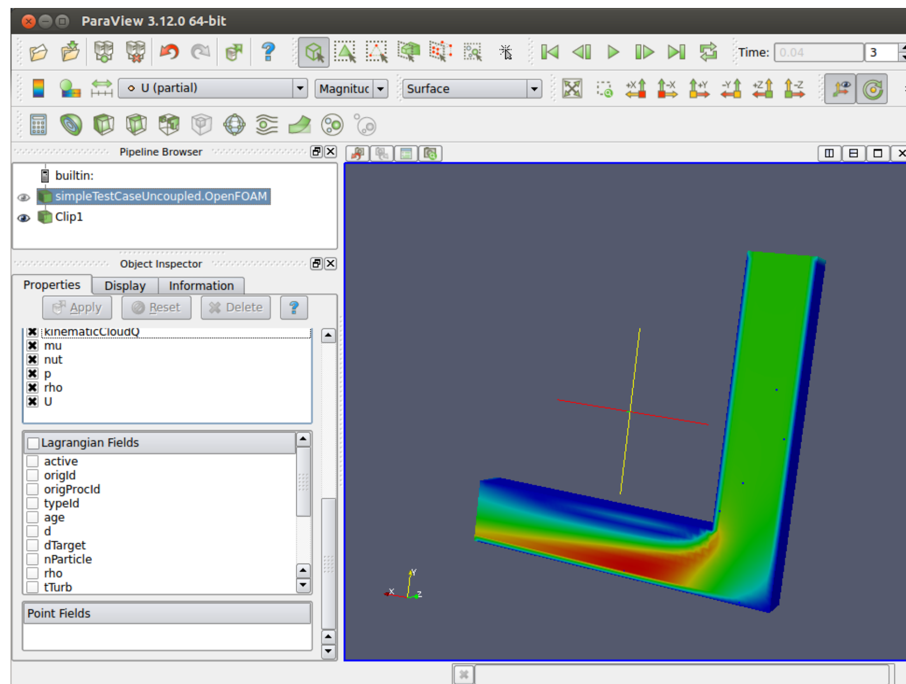


Figure D.7: Visualization of the particles in paraview

```
    format      ascii;
    class       dictionary;
    location    "constant";
    object      kinematicCloudProperties;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *//


solution
{
    active   true; //can be set to true or false
    coupled  false; //true or false for coupled or uncoupled simulations
    transient yes; //yes or no, no for steady-state calculations
    cellValueSourceCorrection off; //when set to on it activated the
//correction of the momentum tranferred to the eulerian phase


    sourceTerms
    {
```

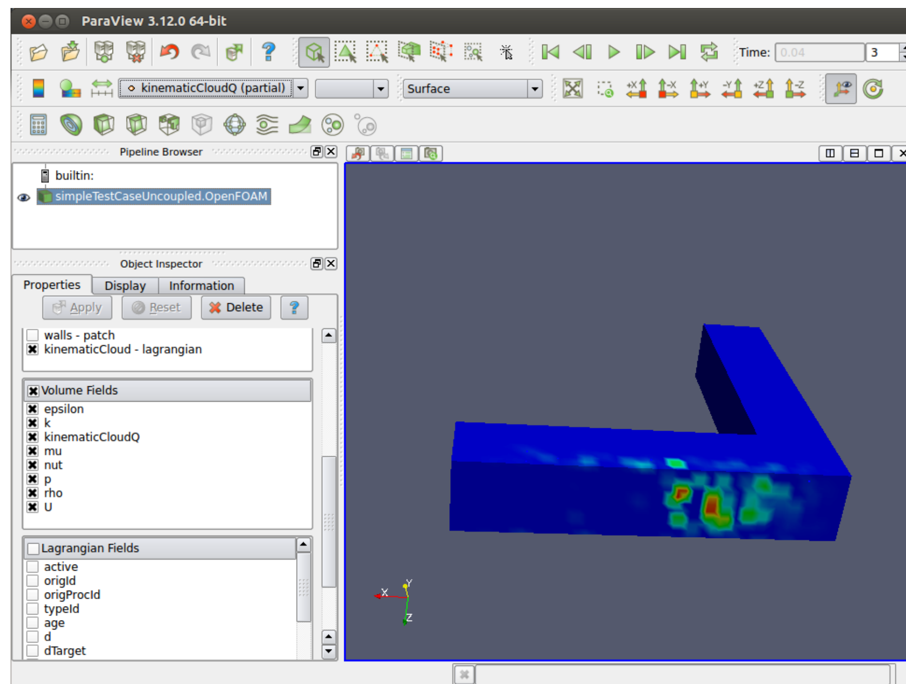


Figure D.8: Erosion contours in paraview

```
        schemes
        {
U           semiImplicit    1;//explicit or semiImplicit
     //ALSO  specify relaxCoeff for each of the fields
        }
    }


    interpolationSchemes
    {
        rho                cell;
        U                  cellPoint;
        mu                 cell;
      //curlUcDt cell; //field used for Lift force calculations
      //DucDt cell;//filed used for pressureGradient calculations
/*Available schemes are:
cell
cellPatchConstrained
```

```
cellPoint
cellPointFace
cellPointWallModified
pointMVC
*/
    }

    integrationSchemes
    {
        U               Euler;
/*Available schemes are:A dictionary with the value of
Euler
analytical
*/
    }
}


constantProperties
{
    //parcelTypeId 1:
    //rhoMin 1e-15;
    //minParticleMass 1e-15:
    rho0            3217;
    youngsModulus   700e9;
    poissonsRatio   0.187;
}


subModels
{
```

```
    particleForces
    {
        sphereDrag;
        gravity;
/*SaffmanMeiLiftForce        //TomiYamaLift may be chosen instead
{
U           U;
}
*/
      /*paramagnetic
{
magneticSusceptibility         -6.3e-9; //m^3/kg for graphite
HdotGradH           U;
*/ }

/*pressureGradient
{
U        U;
}
*/

/*virtualMass
{
Cvm        0.5;
}
*/

    /*nonInertialFrame
        {
             linearAccelerationName   linearAc;
                     linearAcceleration     10;
```

```
               angularVelocityName    angVelo;
                     angularVelocity         5;
                     angularAccelerationName  angAcc
                     angularAcceleration    5;
          }
      */

//SRF;


    }


    injectionModels
    {
 model1
     {
         type            patchInjection;
         patchName       inlet;
  SOI 0; //Start of injection
  massFlowRate 0.01;
  massTotal 0.2;
  parcelBasisType mass;
  flowRateProfile 0.01;
  sizeDistribution
{
type RosinRammler;
RosinRammlerDistribution
{
minValue 200e-6;
maxValue 300e-6;
d 250e-6;
n 3;
```

```
}

}
         duration            20;
         parcelsPerSecond   500000;
         U0              ( 0 -15 0 );
     }
    }

    dispersionModel none;

    patchInteractionModel standardWallInteraction;

    heatTransferModel none;

    surfaceFilmModel none;

    collisionModel none;

    radiation off;

    pairCollisionCoeffs
    {
        // Maximum possible particle diameter expected at any time
       /* maxInteractionDistance  0.006;

        writeReferredParticleCloud no;

        pairModel pairSpringSliderDashpot;

        pairSpringSliderDashpotCoeffs
```

```
{
    useEquivalentSize   no;
    alpha               0.12;
    b                   1.5;
    mu                  0.52;
    cohesionEnergyDensity 0;
    collisionResolutionSteps 12;
};

wallModel    wallLocalSpringSliderDashpot;

wallLocalSpringSliderDashpotCoeffs
{
    useEquivalentSize no;
    collisionResolutionSteps 12;
    walls
    {
        youngsModulus   1e10;
        poissonsRatio   0.23;
        alpha           0.12;
        b               1.5;
        mu              0.43;
        cohesionEnergyDensity 0;
    }
    frontAndBack
    {
        youngsModulus   1e10;
        poissonsRatio   0.23;
        alpha           0.12;
        b               1.5;
        mu              0.1;
```

```
                cohesionEnergyDensity 0;
            }
        };*/
    }

    standardWallInteractionCoeffs
    {
        type            rebound;
    }
}


cloudFunctions
{
      particleErosion
     {

          functionObjectLibs ("libcloudFunctionObjects.so");
          enabled         true;
          outputControl   outputTime;
          log             true;
         valueOutput     true;
         p 11000000; //yield stress
//for aluminium = 11000000 Pa or 11 MPa
psi 2;//Ratio of the depth of contact to the depth of cut
//(default value = 2 )
K 2; //Ratio of vertical to horizontal force components
//(2 for angular abrassive grains)
         patches
         (
             moving-wall
```

```
        );
    }
}

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

D.2.32 Report Appendix 2

D.2.33 blockMeshDict

```
/*--------------------------------*- C++ -*----------------------*\
| =========                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
|  \\    /   O peration     | Version:  2.2.0
|   \\  /    A nd           | Web:      www.OpenFOAM.org
|    \\/     M anipulation  |
\*---------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      blockMeshDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

convertToMeters 0.001;

vertices
(
    (0 0 0)//0
    (0 50 0)//1
```

```
    (0 50 -10)//2
    (0 0 -10)//3
    (10 10 0)//4
    (10 50 0)//5
    (10 50 -10)//6
    (10 10 -10)//7
    (50 0 0)//8
    (50 10 0)//9
    (50 10 -10)//10
    (50 0 -10)//11
    (10 0 -10)//12
    (10 0 0)//13
    (0 10 -10)//14
    (0 10 0)//15
);

blocks
(
    hex (13 4 9 8 12 7 10 11) (10 40 10) simpleGrading (1 1 1)
    hex (0 15 4 13 3 14 7 12) (10 10 10) simpleGrading (1 1 1)
    hex (15 1 5 4 14 2 6 7) (40 10 10) simpleGrading (1 1 1)
);

edges
(
);

boundary
(
    inlet
    {
```

```
        type patch;
        faces
        (
            (1 5 6 2)
        );
    }
    outlet
    {
        type patch;
        faces
        (
            (9 8 11 10)
        );
    }
    walls
    {
        type wall;
        faces
        (
            (4 9 10 7)
            (4 13 8 9)
            (12 11 8 13)
            (7 10 11 12)
            (3 12 13 0)
            (0 13 4 15)
            (0 15 14 3)
        (3 14 7 12)
            (15 4 5 1)
            (5 4 7 6)
            (6 7 14 2)
        (14 15 1 2)
```

```
        );
    }
);

mergePatchPairs
(

);

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

Appendix E

Implementation of E-L solver with Dynamic meshing

E.1 introduction

In this seciton, an Euler-Lagrange coupled solver with Dynamic Meshing is implemented. Since the structure of the solver is different for the two versions of OpenFOAM, the code needed for both versions is commented here. Only the modified files will be listed here, since the rest of them are the default ones for `pimpleFoam` in version 2.2.x and `DPMFoam` in version 2.3.x. For both solvers, an additional dictionary is required. The name of this dictionary should be erosionDict and it should be placed in the constant directory with the following entries which will be read by the top level solver:

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  2.2.0                                 |
|   \\  /    A nd           | Web:      www.OpenFOAM.org                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
```

```
{
    version     2.0;
    format      ascii;
    class       dictionary;
    location    "constant";
    object      erosionDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
patches
(
incident-wall
);
erosionStep 6e-8;//minimum required erosion to deform the mesh.
//This is multiplied by nDef to get the second, and following conditions.
//If these are satisfied at runTime, the mesh will be deformed and the
//displacement of the mesh nodes calculated with the dynamic mesh solver.
nDef   1; //Initialise to 1. It counts the number of deformation steps.
scaleFactor     10; //Factor by which the erosion field is scaled up
// ************************************************************* //
```

E.2 Implementation of an Euler-lagrange solver with Dynamic meshing in OpenFOAM 2.2.x

The implementation of such a solver is similar in OpenFOAM 2.2.x and 2.3.x. The main difference is that, as commented before, OpenFOAM 2.3.x already has a builtin lagrangian solver. Therefore, only the implementation of the solver in OpenFOAM 2.3.x will be shown, as the same solver in 2.2.x can be easily implemented by comparing it to the one in 2.3.x.

E.3 Implementation of an Euler-lagrange solver with Dynamic meshing in OpenFOAM 2.3.x

E.3.1 DPMErosionFOAM.C

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | Copyright (C) 2013-2014 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/>.

Application
    DPMErosionFOAM

Description
```

```
    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 and dynamic mesh deformation according to erosion

\*---------------------------------------------------------------------------*/

#include "fvCFD.H"
#include "dynamicFvMesh.H"
#include "singlePhaseTransportModel.H"
#include "PhaseIncompressibleTurbulenceModel.H"
#include "pimpleControl.H"
#include "fvIOoptionList.H"
#include "fixedFluxPressureFvPatchScalarField.H"
#include "volPointInterpolation.H"
#ifdef MPPIC
    #include "basicKinematicMPPICCloud.H"
    #define basicKinematicTypeCloud basicKinematicMPPICCloud
#else
    #include "basicKinematicCollidingCloud.H"
    #define basicKinematicTypeCloud basicKinematicCollidingCloud
#endif

int main(int argc, char *argv[])
{
    argList::addOption
    (
        "cloudName",
        "name",
        "specify alternative cloud name. default is 'kinematicCloud'"
    );
```

```
#include "setRootCase.H"
#include "createTime.H"
#include "createDynamicFvMesh.H"
#include "readGravitationalAcceleration.H"
#include "initContinuityErrs.H"

pimpleControl pimple(mesh);

#include "createFields.H"
#include "createUf.H"
#include "createFvOptions.H"
#include "readTimeControls.H"
#include "createPcorrTypes.H"
#include "CourantNo.H"
#include "setInitialDeltaT.H"

Info<< "\nStarting time loop\n" << endl;

while (runTime.run())
{
    #include "readControls.H"
    #include "CourantNo.H"
    #include "setDeltaT.H"

    runTime++;

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

    continuousPhaseTransport.correct();
    muc = rhoc*continuousPhaseTransport.nu();
```

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

// Update continuous phase volume fraction field
alphac = max(1.0 - kinematicCloud.theta(), alphacMin);
alphac.correctBoundaryConditions();
alphacf = fvc::interpolate(alphac);
alphaPhic = alphacf*phic;

fvVectorMatrix cloudSU(kinematicCloud.SU(Uc));
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;
```

```
//check that there is erosion

Info<< "Checking kinematicCloudMenguturk field conditions" << endl;
        IOobject erosionHeader
        (
         "kinematicCloudMenguturk",
         runTime.timeName(),
         mesh,
         IOobject::NO_READ
        );

if (erosionHeader.headerOk())
{

//create labels to go over the patches that experience erosion
label patchesDef;
patchesDef = mesh.boundaryMesh().findPatchID("incident-wall");
Info<< "Reading kinematicCloudMenguturk field" << endl;
     volScalarField kinematicCloudQ
     (
       IOobject
               (
               "kinematicCloudMenguturk",
           runTime.timeName(),
               mesh,
           IOobject::READ_IF_PRESENT,
               IOobject::NO_WRITE
       ),
       mesh
     );
```

```
//scalar minErVal= min(mag(kinematicCloudQ)).value();
scalar maxErVal= max(mag(kinematicCloudQ)).value();
Info<< "Maximum Erosion is "<< maxErVal << endl;
//If the value found for the maximum erosion is bigger
//than nDef (integer that counts the number of times
//he mesh has been deformed) * erosionStep
//value read from dictionary transportProperties
//for the minimum erosion that we consider for
//deformation to be applied) then it will enter the
//loop and deform the mesh. If not it will just run
//the particles.
if(maxErVal > nDef*erosionStep)
{

nDef=nDef+1;

#include "erosion.H"

mesh.update();

        // Calculate absolute flux from the mapped surface velocity
                       phic = mesh.Sf() & Uf;

       if (mesh.changing() && correctPhi)
       {
             #include "correctPhi.H"
         }
         // Make the fluxes relative to the mesh motion
         fvc::makeRelative(phic, Uc);

         if (mesh.changing() && checkMeshCourantNo)
```

```
        {
            #include "meshCourantNo.H"
        }

        // --- Pressure-velocity PIMPLE corrector loop
        while (pimple.loop())
        {
            #include "UcEqn.H"

            // --- PISO loop
            while (pimple.correct())
            {
                #include "pEqn.H"
              }

               if (pimple.turbCorr())
                 {
                continuousPhaseTurbulence->correct();
                 }
            }
}
}
else
{
Info<< "No deformation is applied in this loop" << endl;
}

        runTime.write();

        Info<< "ExecutionTime = " << runTime.elapsedCpuTime() << " s"
            << "  ClockTime = " << runTime.elapsedClockTime() << " s"
```

```
            << nl << endl;
    }

    Info<< "End\n" << endl;

    return 0;
}


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

E.3.2 erosion.H

```
volPointInterpolation pInterp(mesh);

pointField zeroPoints(mesh.points());

//read erosion volScalarField from time directory
//scalar scaleFactor = 10000;

//multiply the erosion scalar field by the scaling factor in
//order to magnify it and be able to see it (if necessary)
volScalarField scaledkinematicCloudQ =
scaleFactor*kinematicCloudQ;

//Interpolate the erosion volVectorField to the faces of the cells
surfaceScalarField FkinematicCloudQ =
fvc::interpolate(scaledkinematicCloudQ, "linear");

//unitary surface vectors (mesh.Sf()/mesh.magSf())
//divided by the face area to get erosion in meters instead of m.
const surfaceVectorField Avectors = mesh.Sf()/mesh.magSf();
```

```
volVectorField erosion
    (
        IOobject
        (
            "erosion",
            runTime.timeName(),
            mesh,
            IOobject::NO_READ,
            IOobject::AUTO_WRITE
        ),
        mesh,
dimensionedVector
("erosion", dimensionSet(0,1,0,0,0,0,0), Foam::vector(0,0,0))
    );



forAll(erosion.boundaryField()[patchesDef], facei)
{
  erosion.boundaryField()[patchesDef][facei] =
  Avectors.boundaryField()[patchesDef][facei]*
  FkinematicCloudQ.boundaryField()[patchesDef][facei];
}

/*forAll( erosion.boundaryField()[patchesDef[1]], facei)
{
  erosion.boundaryField()[patchesDef[1]][facei] =
  Avectors.boundaryField()[patchesDef[1]][facei]*
  FkinematicCloudQ.boundaryField()[patchesDef[1]][facei];
}
forAll( erosion.boundaryField()[patchesDef[2]], facei)
{
```

```
  erosion.boundaryField()[patchesDef[2]][facei] =
  Avectors.boundaryField()[patchesDef[2]][facei]*
  FkinematicCloudQ.boundaryField()[patchesDef[2]][facei];
}
*/

//Finally interpolate that vectorField at each face to
//each point of the face, thus obtaining a pointVectorField
volPointInterpolation interpolateVolPoint (mesh);

Info<< "Interpolating" << endl;

pointVectorField pointMotionU = interpolateVolPoint.interpolate(erosion);

pointMotionU.write();

pointField newPoints
(
 zeroPoints
 + pointMotionU
);



mesh.movePoints(newPoints);



//const pointMesh& pMesh= pointMesh::New(mesh);
pointVectorField &pointDisplacement = const_cast<pointVectorField&>
( mesh.lookupObject<pointVectorField>("pointDisplacement") );

pointDisplacement.boundaryField() = pointMotionU.boundaryField();
```

```
pointDisplacement.write();
```

Appendix F

Implementation of Gnanavelu's methodology

F.1 Introduction

This utility is based on OpenFOAM's deformedGeom.C. The utility is renamed and an additional header file is included (erosion.H) which is where the averaged fields are called and the method is implemented. The results from field manipulation are then passed over to the deformation algorithm, which takes a field of vectors as an input. This version is working on OpenFOAM 2.3.x.

F.2 gnanaveluErosion.C

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | Copyright (C) 2011 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
    erodedBoundaryCellList
Description
    Implementation of Gnanavelu et al's methodology in OpenFOAM
    including mesh deformation


\*---------------------------------------------------------------------------*/
#include "OFstream.H"
#include "fvCFD.H"
#include "argList.H"
#include "fvMesh.H"
#include "pointFields.H"
#include "IStringStream.H"
#include "volPointInterpolation.H"

using namespace Foam;
```

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

int main(int argc, char *argv[])
{

#   include "setRootCase.H"

#   include "createTime.H"
#   include "createMesh.H"

    volPointInterpolation pInterp(mesh);

    // Get times list
    instantList Times = runTime.times();
    label lastTime=Times.size()-1;
    pointField zeroPoints(mesh.points());

    // skip "constant" time
    if (label timeI = lastTime)
    {
        runTime.setTime(Times[timeI], timeI);

#   include "erosion.H"

         //Check that the erosion field was created successfully
        if (erosion.headerOk())
        {
            Info<< "Reading point displacement" << endl;
//Calculate the new point field for the new mesh
           pointField newPoints
```

```
            (
                zeroPoints
+ pointMotionU
            );
            mesh.polyMesh::movePoints(newPoints);
            mesh.write();
   Info<< "Writing new points in Time = " << runTime.timeName() << endl;
        }
        else
        {
            Info<< "    No erosion Field" << endl;
        }

        Info<< endl;
    }

    Info<< "End\n" << endl;

    return 0;
}


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

F.3 erosion.H

```
 //Read fields of average velocities and angles of impingement

    Info<< "Reading Averages fields" << endl;

    volScalarField velocityAverage
    (
```

```
        IOobject
        (
            "kinematicCloudMeanImpactAngle",
            runTime.timeName(),
            mesh,
            IOobject::READ_IF_PRESENT,
            IOobject::NO_WRITE
        ),
        mesh
    );

    volScalarField angleAverage
    (
     IOobject
(
"kinematicCloudMeanImpactVelocity",
runTime.timeName(),
mesh,
IOobject::READ_IF_PRESENT,
IOobject::NO_WRITE
),
mesh
    );

//unitary surface vectors (mesh.Sf()/mesh.magSf())
//divided by the face area
const surfaceVectorField Avectors = mesh.Sf()/mesh.magSf();



//Get the label identifying the patch
label patchID =
```

```
mesh.boundaryMesh().findPatchID("incident-wall-part-solid");

volVectorField erosion
    (
        IOobject
        (
            "erosion",
            runTime.timeName(),
            mesh,
            IOobject::NO_READ,
            IOobject::AUTO_WRITE
        ),
        mesh,
dimensionedVector("erosion", dimensionSet(0,1,0,0,0,0,0),
Foam::vector(0,0,0))
    );


forAll( erosion.boundaryField()[patchID], facei)
{

scalar Vel = velocityAverage.boundaryField()[patchID][facei];

scalar angle = angleAverage.boundaryField()[patchID][facei]*
(constant::mathematical::pi/180);

scalar depth = (1e-6)*Vel*Vel*(-7.156*((Foam::sin(scalar(angle)))*
(Foam::sin(scalar(angle)))*
(Foam::sin(scalar(angle)))*(Foam::sin(scalar(angle))))
+17.05*((Foam::sin(scalar(angle)))*(Foam::sin(scalar(angle)))*
(Foam::sin(scalar(angle))))
```

```
-15.02*((Foam::sin(scalar(angle)))*(Foam::sin(scalar(angle))))
+6.565*(Foam::sin(scalar(angle)))-0.4047)*4.693;


erosion.boundaryField()[patchID][facei] =
Avectors.boundaryField()[patchID][facei]*depth;

}


//write the new erosion field created,
//which is a vectorField with magnitude
the erosion and the same direction as the face area vectors
(i.e. in the boundaries points outside of the domain)
erosion.write();

volPointInterpolation interpolateVolPoint (mesh);

Info<< "Interpolating" << endl;

pointVectorField pointMotionU = interpolateVolPoint.interpolate(erosion);
pointMotionU.write();
```

Appendix G

Average velocity field calculation

G.1 Introduction

This utility calculates the average of the velocity field in the pump for all the specified time-steps.

G.2 avgVelocity.C

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | Copyright (C) 2011 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
    avgVelocity

Description
    Calculates the magnitude of the velocity at each time and its average
     over all time-steps

\*---------------------------------------------------------------------------*/
#include "fvCFD.H"
#include "argList.H"
#include "fvMesh.H"
#include "pointFields.H"
#include "IStringStream.H"
#include "volPointInterpolation.H"

using namespace Foam;

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

int main(int argc, char *argv[])
{
```

```
#   include "setRootCase.H"
#   include "createTime.H"
#   include "createMesh.H"

    volPointInterpolation pInterp(mesh);

    // Get times list
    instantList Times = runTime.times();


volVectorField averagedVelocity
    (
        IOobject
        (
            "averagedVelocity",
            runTime.timeName(),
            mesh,
            IOobject::READ_IF_PRESENT,
            IOobject::AUTO_WRITE
        ),
        mesh,
Foam::vector(0,0,0)
    );

volVectorField sumVelocity
    (
        IOobject
        (
            "sumVelocity",
            runTime.timeName(),
            mesh,
```

```
            IOobject::READ_IF_PRESENT,
            IOobject::AUTO_WRITE
        ),
        mesh,
Foam::vector(0,0,0)
    );

    // skip "constant" time

    for (label timeI = 1; timeI < Times.size(); ++timeI)
    {
        runTime.setTime(Times[timeI], timeI);

    Info<< "Reading U field at Time = " << runTime.timeName() << endl;

    volVectorField U
    (
        IOobject
        (
            "U",
            runTime.timeName(),
            mesh,
            IOobject::READ_IF_PRESENT,
            IOobject::NO_WRITE
        ),
        mesh
    );

forAll(sumVelocity.internalField(), cellI)
{
  sumVelocity.internalField()[cellI] =
```

```
  U.internalField()[cellI] + sumVelocity.internalField()[cellI];
}


    }


forAll( averagedVelocity.internalField(), cellI)
{
   averagedVelocity.internalField()[cellI] =
  sumVelocity.internalField()[cellI]/Times.size();
}



    Info<< "Writing new fields..." << endl;


    averagedVelocity.write();


        Info<< endl;


    Info<< "End\n" << endl;


    return 0;
}



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

Appendix H

Average truncated vorticity field calculation

H.1 Introduction

This utility calculates the average of the vorticity field and then truncates the maximum value to a specified one. The vorticity field is not automatically calculated in OpenFOAM so first, the corresponding utility to calculate it (`vorticity`)would have to be run so the field is available for operating with it.

H.2 avgTruncVorticity.C

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | Copyright (C) 2011 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
    deformedGeom

Description
    Deforms a polyMesh using a displacement field U and a scaling factor
    supplied as an argument.

\*---------------------------------------------------------------------------*/
#include "fvCFD.H"
#include "argList.H"
#include "fvMesh.H"
#include "pointFields.H"
#include "IStringStream.H"
#include "volPointInterpolation.H"

using namespace Foam;

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

```
int main(int argc, char *argv[])
{
    argList::validArgs.append("truncating factor"); //value at which
     the averaged vorticity gets truncated

#   include "setRootCase.H"

    const scalar truncatingFactor = args.argRead<scalar>(1);

#   include "createTime.H"
#   include "createMesh.H"

    volPointInterpolation pInterp(mesh);

    // Get times list
    instantList Times = runTime.times();

    pointField zeroPoints(mesh.points());

volScalarField truncatedAvgVorticity
    (
        IOobject
        (
            "truncatedAvgVorticity",
            runTime.timeName(),
            mesh,
            IOobject::READ_IF_PRESENT,
            IOobject::AUTO_WRITE
        ),
        mesh,
```

```
dimensionedScalar("truncatedAvgVorticity", dimensionSet(0,0,0,0,0,0,0),
 Foam::scalar(0))
    );

volScalarField averagedVorticity
    (
        IOobject
        (
            "averagedVorticity",
            runTime.timeName(),
            mesh,
            IOobject::READ_IF_PRESENT,
            IOobject::AUTO_WRITE
        ),
        mesh,
dimensionedScalar("averagedVorticity", dimensionSet(0,0,0,0,0,0,0),
 Foam::scalar(0))
    );

volScalarField sumVorticity
    (
        IOobject
        (
            "sumVorticity",
            runTime.timeName(),
            mesh,
            IOobject::READ_IF_PRESENT,
            IOobject::AUTO_WRITE
        ),
        mesh,
dimensionedScalar("sumVorticity", dimensionSet(0,0,0,0,0,0,0),
```

```
 Foam::scalar(0))
    );

    // skip "constant" time
    for (label timeI = 1; timeI < Times.size(); ++timeI)
    {
        runTime.setTime(Times[timeI], timeI);

    Info<< "Reading magVorticity field at Time = "
     << runTime.timeName() << endl;

    volScalarField magVorticity
    (
        IOobject
        (
            "magVorticity",
            runTime.timeName(),
            mesh,
            IOobject::READ_IF_PRESENT,
            IOobject::NO_WRITE
        ),
        mesh
    );


label patchID1 = mesh.boundaryMesh().findPatchID("imp-walls");
label patchID2 = mesh.boundaryMesh().findPatchID("wall-impeller");
label patchID3 = mesh.boundaryMesh().findPatchID("fv-walls");

forAll( sumVorticity.boundaryField()[patchID1], facei)
{
```

```
  sumVorticity.boundaryField()[patchID1][facei] =
  magVorticity.boundaryField()[patchID1][facei] +
  sumVorticity.boundaryField()[patchID1][facei];
}

forAll( sumVorticity.boundaryField()[patchID2], facei)
{
   sumVorticity.boundaryField()[patchID2][facei] =
 magVorticity.boundaryField()[patchID2][facei] +
 sumVorticity.boundaryField()[patchID2][facei];
}

forAll( sumVorticity.boundaryField()[patchID3], facei)
{
  sumVorticity.boundaryField()[patchID3][facei] =
  magVorticity.boundaryField()[patchID3][facei] +
  sumVorticity.boundaryField()[patchID3][facei];
}



    }
label patchID1 = mesh.boundaryMesh().findPatchID("imp-walls");
label patchID2 = mesh.boundaryMesh().findPatchID("wall-impeller");
label patchID3 = mesh.boundaryMesh().findPatchID("fv-walls");

forAll( averagedVorticity.boundaryField()[patchID1], facei)
{
   averagedVorticity.boundaryField()[patchID1][facei] =
 sumVorticity.boundaryField()[patchID1][facei]/Times.size();
}
```

```
forAll( averagedVorticity.boundaryField()[patchID2], facei)
{
   averagedVorticity.boundaryField()[patchID2][facei] =
 sumVorticity.boundaryField()[patchID2][facei]/Times.size();
}

forAll( averagedVorticity.boundaryField()[patchID3], facei)
{
   averagedVorticity.boundaryField()[patchID3][facei] =
 sumVorticity.boundaryField()[patchID3][facei]/Times.size();
}

    Info<< "Creating truncating vorticity field at a value of"
    << truncatingFactor << endl;

forAll( truncatedAvgVorticity.boundaryField()[patchID1], facei)
{
   if (averagedVorticity.boundaryField()[patchID1][facei] >=
  truncatingFactor)
{
truncatedAvgVorticity.boundaryField()[patchID1][facei]=truncatingFactor;
}
   else if (averagedVorticity.boundaryField()[patchID1][facei] <
 truncatingFactor)
{
truncatedAvgVorticity.boundaryField()[patchID1][facei]=
averagedVorticity.boundaryField()[patchID1][facei];
}
}

forAll( truncatedAvgVorticity.boundaryField()[patchID2], facei)
```

```
{
   if (averagedVorticity.boundaryField()[patchID2][facei] >=
 truncatingFactor)
{
truncatedAvgVorticity.boundaryField()[patchID2][facei]=truncatingFactor;
}
   else if (averagedVorticity.boundaryField()[patchID2][facei] <
 truncatingFactor)
{
truncatedAvgVorticity.boundaryField()[patchID2][facei]=
averagedVorticity.boundaryField()[patchID2][facei];
}
}

forAll( truncatedAvgVorticity.boundaryField()[patchID3], facei)
{
   if (averagedVorticity.boundaryField()[patchID3][facei] >=
 truncatingFactor)
{
truncatedAvgVorticity.boundaryField()[patchID3][facei]=truncatingFactor;
}
   else if (averagedVorticity.boundaryField()[patchID3][facei] <
 truncatingFactor)
{
truncatedAvgVorticity.boundaryField()[patchID3][facei]=
averagedVorticity.boundaryField()[patchID3][facei];
}
}

    Info<< "Writing new fields..." << endl;
```

```
    truncatedAvgVorticity.write();
    averagedVorticity.write();

        Info<< endl;

    Info<< "End\n" << endl;

    return 0;
}


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

Appendix I

Erosion calculation in a pump

I.1 Introduction

This utility calculates an approximation of the erosion field in a centrifugal pump. It first transfers the values of the velocity at closest cells to the walls to the boundary and then uses the second power of the velocity for the static parts of the pump. For the rotating parts, the truncated vorticity field is used instead. Both fields are computed and the whole geometry is then deformed by calling the corresponding patches. For the mesh deformation, the application is based on the existing utility `defomedGeom`.

I.2 pumpErosion.C

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | Copyright (C) 2011 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
    pumpErosion
Description
    Deforms a polyMesh using a displacement field erosion and two scaling
    factors defined at the beginning of the file. The erosion field is
    proportional to the vorticity field for the rotating parts and
    proportional to the square of the averaged velocity
    (previously calculated with avgVelocity app) for the non
    rotating parts.

\*---------------------------------------------------------------------------*/
#include "OFstream.H"
#include "fvCFD.H"
#include "argList.H"
#include "fvMesh.H"
#include "pointFields.H"
#include "IStringStream.H"
#include "volPointInterpolation.H"
```

```
using namespace Foam;

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

int main(int argc, char *argv[])
{

#   include "setRootCase.H"

    const scalar scaleFactor1 = 0.00007;
    const scalar scaleFactor2 = 4e-8;

#   include "createTime.H"
#   include "createMesh.H"

    volPointInterpolation pInterp(mesh);

    // Get times list
    instantList Times = runTime.times();
    label lastTime=Times.size()-1; //Get the last saved time-step
    pointField zeroPoints(mesh.points());

    // skip "constant" time
    if (label timeI = lastTime)
    {
        runTime.setTime(Times[timeI], timeI);

/////////////////////////////////////////////////////////////////////

    Info<< "Reading Averaged Velocity field" << endl;
```

```
    volVectorField averagedVelocity
    (
        IOobject
        (
            "averagedVelocity",
            runTime.timeName(),
            mesh,
            IOobject::READ_IF_PRESENT,
            IOobject::NO_WRITE
        ),
        mesh
    );

    Info<< "Reading Averaged Vorticity field" << endl;
    volScalarField averagedVorticity
    (
        IOobject
        (
            "averagedVorticity",
            runTime.timeName(),
            mesh,
            IOobject::READ_IF_PRESENT,
            IOobject::NO_WRITE
        ),
        mesh
    );

//Normalised surface normal vectors
const surfaceVectorField Avectors = mesh.Sf()/mesh.magSf();

//150 WBH PUMP WALLS
```

```
//ROTATING_WALLS
//wall-impeller
//imp-walls
//fv-walls
//imp-back

//STATIC-WALLS
//volute-tb-walls

label patchID1 = mesh.boundaryMesh().findPatchID("wall-impeller");
label patchID2 = mesh.boundaryMesh().findPatchID("imp-walls");
label patchID3 = mesh.boundaryMesh().findPatchID("fv-walls");
label patchID4 = mesh.boundaryMesh().findPatchID("imp-back");

label patchID5 = mesh.boundaryMesh().findPatchID("volute-tb-walls");


//Velocity of each closest cell to each boundary face computed
 at each boundary face
volScalarField magErosion
    (
        IOobject
        (
            "magErosion",
            runTime.timeName(),
            mesh,
            IOobject::READ_IF_PRESENT,
            IOobject::NO_WRITE
        ),
        mesh,
```

```
dimensionedScalar
("magErosion", dimensionSet(0,0,0,0,0,0,0), Foam::scalar(0.0))
    );

const fvPatchList& patches = mesh.boundary();

//ROTATING_WALLS
//wall-impeller patchID1
//imp-walls patchID2
//fv-walls patchID3
//imp-back patchID4

forAll(magErosion.boundaryField()[patchID1], facei)
{
    magErosion.boundaryField()[patchID1][facei] =
scaleFactor2*averagedVorticity.boundaryField()[patchID1][facei];
}
forAll(magErosion.boundaryField()[patchID2], facei)
{
   magErosion.boundaryField()[patchID2][facei] =
 scaleFactor2*averagedVorticity.boundaryField()[patchID1][facei];
}
forAll(magErosion.boundaryField()[patchID3], facei)
{
   magErosion.boundaryField()[patchID3][facei] =
 scaleFactor2*averagedVorticity.boundaryField()[patchID1][facei];
}
forAll(magErosion.boundaryField()[patchID4], facei)
{
   magErosion.boundaryField()[patchID4][facei] =
scaleFactor2*averagedVorticity.boundaryField()[patchID1][facei];
```

```
}

//STATIC-WALLS
//volute-tb-walls patchID5

forAll( magErosion.boundaryField()[patchID5], facei)
{
  const fvPatch& currPatch = patches[patchID5];
  label intCell = currPatch.faceCells()[facei];
   vector vecVel = averagedVelocity.internalField()[intCell];
   scalar avgMagU = mag(vecVel);

   magErosion.boundaryField()[patchID5][facei] = scaleFactor1*sqr(avgMagU);
}

magErosion.write();




volVectorField erosion
    (
        IOobject
        (
            "erosion",
            runTime.timeName(),
            mesh,
            IOobject::NO_READ,
            IOobject::AUTO_WRITE
        ),
        mesh,
dimensionedVector
```

```
("erosion", dimensionSet(0,0,1,0,0,0,0), Foam::vector(0,0,0))
    );



forAll( erosion.boundaryField()[patchID1], facei)
{
  erosion.boundaryField()[patchID1][facei] =
   Avectors.boundaryField()[patchID1][facei]*
   (magErosion.boundaryField()[patchID1][facei]);
}
forAll( erosion.boundaryField()[patchID2], facei)
{
  erosion.boundaryField()[patchID2][facei] =
  Avectors.boundaryField()[patchID2][facei]*
  (magErosion.boundaryField()[patchID2][facei]);
}
forAll( erosion.boundaryField()[patchID3], facei)
{
  erosion.boundaryField()[patchID3][facei] =
  Avectors.boundaryField()[patchID3][facei]*
  (magErosion.boundaryField()[patchID3][facei]);
}
forAll( erosion.boundaryField()[patchID4], facei)
{
  erosion.boundaryField()[patchID4][facei] =
  Avectors.boundaryField()[patchID4][facei]*
  (magErosion.boundaryField()[patchID4][facei]);
}
forAll( erosion.boundaryField()[patchID5], facei)
{
```

```
  erosion.boundaryField()[patchID5][facei] =
  Avectors.boundaryField()[patchID5][facei]*
  (magErosion.boundaryField()[patchID5][facei]);
}

//write the new erosion field created, which is a vectorField with
//magnitude the erosion and the same direction as the face
//area vectors (i.e. in the boundaries points outside of the
//domain)
erosion.write();

//Finally interpolate that vectorField at each face to each point
//of the face, thus obtaining a pointVectorField

volPointInterpolation interpolateVolPoint (mesh);

Info<< "Interpolating" << endl;

pointVectorField pointMotionU = interpolateVolPoint.interpolate(erosion);
pointMotionU.write();

////////////////////////////////////////////////////////////////////////////

      // Check that the erosion field was created successfully
      if (erosion.headerOk())
      {
          Info<< "Reading point displacement" << endl;
//Calculate the new point field for the new mesh
         pointField newPoints
          (
              zeroPoints
```

```
+ pointMotionU
            );
            mesh.polyMesh::movePoints(newPoints);
            mesh.write();
   Info<< "Writing new points in Time = " << runTime.timeName() << endl;
        }
        else
        {
            Info<< "    No erosion Field" << endl;
        }

        Info<< endl;
    }

    Info<< "End\n" << endl;

    return 0;
}


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

Appendix J

Mesh deformation with dynamic meshing

J.1 Introduction

The utility calculates the displacement of the boundary according to erosion. Instead of just moving the nodes, a dynamic mesher is included in this application so that after deforming it will iterate and move the adjacent nodes accordingly. For this utility to work, a field in the `0` directory (pointDisplacement) and a dynamicMeshDict in the `constant` directory are needed.

J.2 erodedBoundaryAdaptive.C

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | Copyright (C) 2011 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
    deformedGeomAdaptive
    For this application to work we need the directory with
    0/pointDisplacement, constant directory with
    dynamicMeshDict and System directory with cellDisplacement
    in fvSchemes and fvSolution (to tell the OpenFOAM how
     to solve the cell displacements of the dynamic mesh)

Description
    Deforms a polyMesh using a displacement field U and a scaling factor
    supplied as an argument.

\*---------------------------------------------------------------------------*/
#include "fvCFD.H"
#include "argList.H"
#include "fvMesh.H"
#include "pointFields.H"
```

```
#include "IStringStream.H"
#include "volPointInterpolation.H"
#include "dynamicFvMesh.H"

using namespace Foam;

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

int main(int argc, char *argv[])
{

//scaling factor to magnify erosion and be able to see the deformation

    argList::validArgs.append("scaling factor");

#   include "setRootCase.H"

    const scalar scaleFactor = args.argRead<scalar>(1);

#   include "createTime.H"
#   include "createDynamicFvMesh.H"

    volPointInterpolation pInterp(mesh);

    // Get times list
    instantList Times = runTime.times();

    pointField zeroPoints(mesh.points());

    // skip "constant" time
    if (label timeI = 1)
```

```
    {
        runTime.setTime(Times[timeI], timeI);

#   include "erosion.H"

        // Check that the erosion field was created successfully
        if (erosion.headerOk())
        {
            Info<< "Reading point displacement" << endl;
//Calculate the new point field for the new mesh
           pointField newPoints
            (
                zeroPoints
+ pointMotionU
            );
mesh.movePoints(newPoints);
mesh.update();



            mesh.polyMesh::movePoints(newPoints);
            mesh.write();
        Info<< "Writing new points in Time = " << runTime.timeName() << endl;
        }
        else
        {
            Info<< "No erosion Field" << endl;
        }

        Info<< endl;
    }
```

```
    Info<< "End\n" << endl;

    return 0;
}


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

J.3 erosion.H

```
//read erosion volScalarField from time directory

    Info<< "Reading kinematicCloudQ field" << endl;
    volScalarField kinematicCloudQ
    (
        IOobject
        (
            "kinematicCloudHutchings",
            runTime.timeName(),
            mesh,
            IOobject::READ_IF_PRESENT,
            IOobject::NO_WRITE
        ),
        mesh
    );

//multiply the erosion scalar field by the scaling factor
 in order to magnify it and be able to see it (if necessary)
volScalarField scaledkinematicCloudQ = scaleFactor*kinematicCloudQ;
//Interpolate the erosion volVectorField to the faces of the cells
surfaceScalarField FkinematicCloudQ =
```

```
fvc::interpolate(scaledkinematicCloudQ, "linear");

//The algorithm is going to be executed on the
//moving-wall patch so we look for the right patch
label patchID = mesh.boundaryMesh().findPatchID("incident_wall");

const surfaceVectorField Avectors = mesh.Sf()/mesh.magSf();

volVectorField erosion
    (
        IOobject
        (
            "erosion",
            runTime.timeName(),
            mesh,
            IOobject::NO_READ,
            IOobject::AUTO_WRITE
        ),
        mesh,
dimensionedVector("erosion", dimensionSet(0,0,1,0,0,0,0),
Foam::vector(0,0,0))
    );



forAll( erosion.boundaryField()[patchID], facei)
{
  erosion.boundaryField()[patchID][facei] =
  Avectors.boundaryField()[patchID][facei]*
  FkinematicCloudQ.boundaryField()[patchID][facei];
}
```

```
//write the new erosion field created, which is a
vectorField with magnitude the erosion and the
 same direction as the face area vectors
 (i.e. in the boundaries points outside of the domain)
erosion.write();

//Finally interpolate that vectorField at each face
 to each point of the face, thus obtaining a pointVectorField

volPointInterpolation interpolateVolPoint (mesh);

Info<< "Interpolating" << endl;

pointVectorField pointMotionU = interpolateVolPoint.interpolate(erosion);
//const pointMesh& pMesh= pointMesh::New(mesh);
pointVectorField &pointDisplacement = const_cast<pointVectorField&>
( mesh.lookupObject<pointVectorField>("pointDisplacement") );

pointDisplacement.boundaryField() = pointMotionU.boundaryField();

pointDisplacement.write();
```

Appendix K

Application to count the minimum number of impacts

K.1 Introduction

This utility calculates the number of impacts needed to calculate the wear scar with the desired level of confidence. The application takes the two fields with those values for the impact angle and the impact velocity and calculates the sum over all the faces of the specified boundary after rounding the obtained values up to the nearest integer. The result consists of two numbers, one for the impact angle and one for the impact velocity, of which the highest one should be chosen

K.2 ecountParticles.C

```
/*---------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     |
    \\  /    A nd           | Copyright (C) 2011 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
    countParticles

Description
    Counts necessary number of impacts for an average
     with a certain degree of confidence

\*---------------------------------------------------------------------------*/
#include "fvCFD.H"
#include "argList.H"
#include "fvMesh.H"
#include "pointFields.H"
#include "IStringStream.H"
#include "volPointInterpolation.H"

using namespace Foam;
```

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

int main(int argc, char *argv[])
{

#   include "setRootCase.H"

#   include "createTime.H"
#   include "createMesh.H"

    volPointInterpolation pInterp(mesh);

    // Get times list
    instantList Times = runTime.times();

    pointField zeroPoints(mesh.points());

    // skip "constant" time
    if (label timeI = 1)
    {
        runTime.setTime(Times[timeI], timeI);


//read sample sizes for angle and impact velocity from time directory

    Info<< "Reading kinematicCloudQ field" << endl;
    volScalarField kinematicCloud1
    (
        IOobject
        (
```

```
            "kinematicCloudVelocitySampleSize",
            runTime.timeName(),
            mesh,
            IOobject::READ_IF_PRESENT,
            IOobject::NO_WRITE
        ),
        mesh
    );
    Info<< "Reading kinematicCloudQ field" << endl;
    volScalarField kinematicCloud2
    (
        IOobject
        (
            "kinematicCloudImpactAngleSampleSize",
            runTime.timeName(),
            mesh,
            IOobject::READ_IF_PRESENT,
            IOobject::NO_WRITE
        ),
        mesh
    );

//We look for the right patch
label patchID = mesh.boundaryMesh().findPatchID("incident-wall-part-solid");

//initialise variables
scalar numImpVel=0;
scalar numImpAng=0;

//sum all the sample sizes for all faces of the patch after rounding
//up to the nearest integer with std::ceil
```

```
forAll( kinematicCloud1.boundaryField()[patchID], facei)
{
  numImpVel=numImpVel+std::ceil(kinematicCloud1.boundaryField()
  [patchID][facei]);
}

forAll( kinematicCloud1.boundaryField()[patchID], facei)
{
  numImpAng=numImpAng+std::ceil(kinematicCloud2.boundaryField()
  [patchID][facei]);
}
//Print both numbers, of which we take the highest one

Info<< "Number of Particles for Impact Velocity" << numImpVel << endl;
Info<< "Number of Particles for Impact Angle" << numImpAng << endl;



    }

    Info<< "End\n" << endl;

    return 0;
}


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

References

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

[2] I. M. Hutchings. A model for the erosion of metals by spherical particles at normal incidence. *Wear*, 70: 269–281, 1981.

[3] R. M. Davies. The determination of static and dynamic yield stresses using a step ball. *Proceedings of the Royal Society of London*, 197A: 416–432, 1949.

[4] A. V. Riemsdijk and J. G. A. Bitter. Erosion in gas-solid systems. *Fifth World Petroleum Congress, Section VII, Engineering, Equipment and Materials*, pages 43–58, 1959.

[5] G. Grant and W. Tabakoff. An experimental investigation of the erosion characteristics of 2024 aluminum alloy. *Department of Aerospace Engineering Tech. Rep., University of Cincinnati*, pages 73–77, 1973.

[6] J. G. A. Bitter. A study of erosion phenomena part i. *Wear*, 6: 5–21, 1963.

[7] J. G. A. Bitter. A study of erosion phenomena part ii. *Wear*, 6: 169–190, 1963.

[8] J. H. Neilson and A. Gilchrist. Erosion by a stream of solid particles. *Wear*, 11: 111–122, 1968.

[9] A. Gnanavelu et al. An integrated methodology for predicting material wear rates due to erosion. *Wear*, 267: 1935–1944, 2009.

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

[11] M. A. Al-Bukhaiti et al. Effect of impingement angle on slurry erosion and mechanisms of 1017 steel and high-chromium white cast iron. *Wear*, 262: 1187—1198, 2007.

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

[13] I. Finnie. Some observations on the erosion of ductile metals. *Wear*, 19: 81–90, 1972.

[14] A. Gnanavelu et al. An investigation of a geometry independent integrated method to predict erosion rates in slurry erosion. *Wear*, 271: 712–719, 2011.

[15] M. Sommerfeld and N. Hubber. Experimental analysis and modelling of particle-wall collisions. *International Journal of Multiphase Flow*, 25(6-7): 1457—1489, 1999.

[16] A. Forder, M. Thew and D. Harrison. A numerical investigation of solid particle erosion experienced within oilfield control valve. *Wear*, 216: 184—193, 1998.

[17] S. Wiederhorn and B. Hockey. Effect of material parameters on the erosion resistance of brittle materials. *Journal of Materials Science*, 18: 766–780, 1983.

[18] G. L. Sheldon and A. Kanhere. An investigation of impingement erosion using single particles. *Wear*, 21: 195—209, 1972.

[19] P. Shipway and I. Hutchings. The role of particle properties in the erosion of brittle materials. *Wear*, 193: 105–113, 1996.

[20] T. Deng, M. Bingley and M. Bradley. The influence of particle rotation on the solid particle erosion rate of metals. *Wear*, 256: 1037–1049, 2004.

[21] T. Deng, M. Bingley and M. Bradley. Understanding particle dynamics in erosion testers: A review of influences of particle movement on erosion test conditions. *Wear*, 267: 2132–2140, 2009.

[22] D. W. Nicholls. *Private Communication on Molecular Dynamics.* University of Strathclyde, 2013.

[23] M. T. Benchaita, P. Griffith and E. Rabinowicz. Erosion of metallic pate by solid partiles entrained in a liquid jet. *Journal of engineering for industry*, 105: 215–222, 1983.

[24] A. Lopez, M. Stickland, W. Dempster and W. Nicholls. Cfd study of jet impingement test erosion using ansys fluent and openfoam. *Computer Physics Communications*, 197: 88–95, 2015.

[25] Y. Zhong and K. Minemura. Measurement of erosion due to particle impingement and numerical prediction of wear in pump casing. *Wear*, 199: 36–44, 1996.

[26] G. F. Truscott. A literature survey on abrasive wear on hydraulic machinery. *Wear*, 20: 29–50, 1972.

[27] W. Jennings, W. Head and J. C.R. Manning. A mechanistic model for the prediciton of ductile erosion. *Wear*, 40: 93–112, 1976.

[28] Y. Oka, H. Ohnogi, T. Hosokawa and M. Matsumura. The impact angle dependenceof erosion damage caused by solid particle impact. *Wear*, 203-204: 573–579, 1997.

[29] S. J. Cummins, M. D. Sinnott and P. W. Cleary. Dem modelling of wear in high shear mixers. *Eleventh International Conference on CFD in the Minerals and Process Industries*, 2015.

[30] M. Stack, C. Telfer and B. Jana. Particle concentration and size effects on the erosion-corrosion of pure metals in aqueous slurries. *Tribology International*, 53: 35–44, 2012.

[31] K. Nandakumar et al. A phenomenological model for erosion of material in a horizontal slurry pipeline flow. *Wear*, 269: 190–196, 2010.

[32] M. S. Patil et al. Study of the parameters affecting erosion wear of ductile material in solid-liquid mixture. *Proceedings of the World Congress on Engineering*, 3, 2011.

[33] S. Rajahram, T. Harvey and R. Wood. Evaluation of a semi-empirical model in predicting erosion-corrosion. *Wear*, 267: 1883–1893, 2009.

[34] H.-J. Bart and M. Azimian. Erosion investigations by means of centrifugal accelerator erosion tester. *Wear*, 328-329: 249–256, 2015.

[35] G. Tilly and W. Sage. The interaction of particle and material behaviour in erosion processes. *Wear*, 16: 447–465, 1970.

[36] W. Head, L. Lineback and C. Manning. Modification and extension of a model for predicting the erosion of ductile metals. *Wear*, 23: 291–298, 1973.

[37] J. S. Mason and B. V. Smith. The erosion of bends by pneumatically conveyed suspensions of abrasive particles. *Powder Technology*, 6: 323–335, 1972.

[38] B. F. Levin, K. Vecchio, J. N. Dupont and A.R.Marder. Modeling solid-particle erosion of ductile alloys. *Metallurgical and materials transactions*, 30A: 1763–1774, 1999.

[39] S. Srinivasan and R. O. Scattergood. Effect of erodent hardness on erosion of brittle materials. *Wear*, 128: 139–152, 1988.

[40] H. M. Clark. Specimen diameter, impact velocity, erosion rate and particle density in a slurry pot erosion tester. *Wear*, 162-164: 669–678, 1993.

[41] N. A. Gorbushin and Y. V. Petrov. Dynamic fragmentation of solid particles interacting with a rigid barrier. *Technical Physics*, 59: 194–198, 2014.

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

[43] A. Lopez, M. Stickland and W. Dempster. Comparative study of different erosion models in an eulerian-lagrangian frame using open source software. *Proceedings of the 12th European Fluid Machinery Congress*, 2014.

[44] W. Tabakoff and B. V. R. Vittal. High temperature erosion study of inco 600 metal. *Wear*, 86: 89–99, 1983.

[45] F. T. Barwell. Wear of metals. *Wear*, 1: 317–332, 1957.

[46] I. Kleis and P. Kulu. *Solid Particle Erosion: Occurrence, Prediction and Control.* Springer, 2008.

[47] P. A. Schweitzer. *Fundamentals of Corrosion: Mechanisms, Causes and Preventive Methods.* CRC Press, 2010.

[48] C. Lhymn and P. Wapner. Slurry erosion of polyphenilene sulfide-glass composites. *Wear*, 119: 1–12, 1987.

[49] S.Johansson, F. Eriscon and J. Schweitz. Solid particle erosion - a statistical method for evaluation of strength properties of semiconducting materials. *Wear*, 115: 107–120, 1987.

[50] A. G. Evans. Impact damage mechanics: solid projectiles in c. m. preece (ed.). *Treatise on Material Science and Technology*, 16: 1, 1979.

[51] A. W. Ruff and S. M. Wieherhorn. Erosion by solid particle impact in c. m. preece (ed.). *Treatise on Material Science and Technology*, 16: 69, 1979.

[52] G. Sundararajan and P. Shewmon. A new model for the erosion of metals at normal incidence. *Wear*, 84: 237–258, 1983.

[53] C. E. Smeltzer et al. Mechanisms of sand and dust erosion in gas turbine engines. *USAAVLABS Tech. Rep. 70-36, U.S. Army Air Mobility Research and Development Laboratory*, 1970.

[54] L. L. Shreir. *Corrosion*, volume 1. Newnes-Butterworths, 1963.

[55] Y. Li, G. Burstein and I. M. Hutchings. The influence of corrosion on the erosion of aluminium by aqueous silica slurries. *Wear*, 186-187: 515–522, 1995.

[56] S. Dosanjh and J. A. C. Humphrey. The influence of turbulence on erosion by a particle-laden fluid jet. *Wear*, 102: 309–330, 1985.

[57] S. Morsi and A. Alexander. An investigation of particle trajectories in two-phase flow systems. *Journal of Fluid Mechanics*, 55(2): 193–208, 1972.

[58] 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.

[59] H. M. Clark. The influence of the flowfield in slurry erosion. *Wear*, 152: 223–240, 1992.

[60] V.B. Nguyen et al. A combined numerical-experimental study on the effect os surface evolution on the water-sand multiphase flow characteristics and the material erosion behavior. *Wear*, 319: 96–109, 2014.

[61] T. Maric, J. Höpken and K. Mooney. *The OpenFOAM Technology Primer*. Sourceflux, 2014.

[62] OpenFOAM Foundation. Openfoam user guide, 2011. `http://www.openfoam.org/docs/user/`.

[63] K. Sugiyama, K. Harada and S. Hattori. Influence of impact angle of solid particles on erosion by slurry jet. *Wear*, 265: 713–720, 2008.

[64] OpenFOAM Foundation. Openfoam programmer's guide, 2011. `http://foam.sourceforge.net/docs/Guides-a4/ProgrammersGuide.pdf`.

[65] D. Shepard. A two-dimensional interpolation function for irregularly spaced data. *Proceedings 1968 ACM National Conference*, 1: 517–524, 1968.

[66] G.Allasia. Some physical and mathematical properties of inverse distance weighted methods for scattered data interpolation. *Calcolo*, 29: 97–109, 1992.

[67] G. Y. Lu and D. W. Wong. An adaptive inverse-distance weighting spatial interpolation technique. *Computers and Geosciences*, 34: 1044–1055, 2008.

[68] K. Nandakumar et al. A comprehensive phenomenological model for erosion of materials in jet flow. *Powder Technology*, 187: 273–279, 2008.

[69] B. E. Launder and D. Spalding. *Lectures in Mathematical Models of Turbulence.* Academic Press, 1972.

[70] K. Abdellatif et al. Predicting initial erosion during the hole erosion test by using turbulent flow cfd simulation. *Applied Mathematical Modellling*, 36: 3359–3370, 2012.

[71] F. Menter. Two-equation eddy-viscosity turbulence models for engineering applications. *AAIA Journal*, 32: 1598–1605, 1994.

[72] Ansys Inc. *Ansys Fluent User Guide.* Ansys, Inc., 2009.

[73] A. Mackenzie, A. Lopez, K. Ritos, M. Stickland and W. Dempster. A comparison of cfd software packages' ability to model a submerged jet. *Eleventh International Conference on CFD in the Minerals and Process Industries, CSIRO, Melbourne, Australia*, 2015.

[74] P. Mathews. *Sample Size Calculations: Practical Methods for Engineers and Scientists.* Mathews Malnar and bailey, Inc., 2010.

[75] L. J. W. Graham, D. R. Lester and J. Wu. Quantification of erosion distributions in complex geometries. *Wear*, 268: 1066–1071, 2010.

[76] R. Okita, Y. Zhang, B. S. McLaury and S. A. Shirazi. Experimental and computational investigations to evaluate the effects of fluid viscosity and particle size on erosion damage. *Journal of Fluids Engineering*, 134: 0631301, 2012.

[77] R. J. K. Wood and T. F. Jones. Investigations of sand-water induced erosive wear of aisi 304l stainless steel pipes by pilot-scale and laboratory-scale testing. *Wear*, 255: 206–218, 2003.

[78] W. J. Head and M. E. Harr. The development of a model to predict the erosion of materials by natural contaminants. *Wear*, 15: 1–46, 1970.

[79] A.F.R.C. Advanced forming research centre equipment directory, 2015. `https://www.strath.ac.uk/media/departments/dmem/afrc/afrc-refurbwebsite/equipment/AFRC_equipment_brochure.pdf`.

[80] M. Raffel, C. E. Willert, S. T. Wereley and J. Kompehans. *Particle Image Velocimetry: A practical guide.* Springer, 2007.

[81] L. Nicolaou and T. A. Zaki. On the stokes number and characterization of aerosol deposition in the respiratory airways. *4th International Conference on Computational and Mathematical Biomedical Engineering - CMBE2015*, 1: 224–227, 2015.

[82] R. C. Flagan and J. H. Seinfeld. *Fundamentals of air pollution engineering.* Prentice Hall, Inc., 1988.

[83] W. Thielicke and E. J. Stamhuis. Pivlab - time-resolved digital particle image velocimetry tool for matlab (version: 1.4), 2014.

[84] W. Tabakoff, R. Kotwal and A. Hamed. Erosion study of different materials affected by coal ash particles. *Wear*, 52: 161–173, 1979.

[85] M. Menguturk and E. F. Sverdrup. Calculated tolerance of a large electric utility gas turbine to erosion damage by coal gas ash particles. *ASTM Special Technical Publications*, 664: 193–224, 19.

[86] K. Sun, L. Lu and H. Jin. Modeling and numerical analysis of the solid particle erosion in curved ducts. *Abstract and Applied Analysis*, 2013: 1–8, 2013.

[87] X. Song, J. Z. Lin, J. Zhao and T. Y. Shen. Research on reducing erosion by adding ribs on the wall in particulate two-phase flows. *Wear*, 193: 1–7, 1996.

[88] A. Lopez, M. Stickland and W. Dempster. Predicting surface evolution in erosion processes with openfoam, 2015. `http://openfoam-extend.sourceforge.net/OpenFOAM_Workshops/OFW10_2015_AnnArbor/?page_id=146`.

[89] The MathWorks Inc., Massachusetts, United States. Matlab and statistics toolbox release 2016a, 2016.

[90] OriginLab, Northampton, US. Origin 2016, 2016.

[91] J. Ahrens, B. Geveci and C. Law. Paraview: An end-user tool for large data visualization. *Visualization Handbook, Elsevier*, 1, 2005.

[92] D. Smith. *Modelling cavitation in centrifugal pumps using OpenFOAM.* University of Strathclyde, 2016.

[93] L. Moscoso. Internal report/communication. pump technology centre, weir minerals, australia.

[94] L. Moscoso, University of Sydney. *Computational Investigation of Wear in Centrifugal Slurry Pumps.* University of Sydney, 2010.

[95] X. Cai, S. Zhou and S. Li. Study on the influence of back blade shape on the wear characteristics of centrifugal slurry pump. *IOP Conf. Series: Materials Science and Engineering*, 129: 12–58, 2016.

[96] J. Stijnen, A. Heemink and H. Lin. An efficient 3d particle transport model for use in stratified flow. *International journal for numerical methods in fluids*, 51: 331–350, 2006.

[97] G. Brown. Erosion prediciton in slurry pipeline tee-junctions. *Applied Mathematical Modelling*, 26: 155–170, 2002.

[98] OpenFOAM Foundation. Openfoam c++ documentation, 2011. http://www.openfoam.org/docs/cpp/.

[99] Cplusplus.com. C++ tutorial, 2012. http://www.cplusplus.com/doc/tutorial/.

[100] A. Vallier. Tutorial lagrangian particle tracking, 2011. http://www.tfd.chalmers.se/~hani/kurser/OS_CFD_2009/AureliaVallier/oscfd09_present_aureliapdf.pdf.

[101] A. Vallier. Coupling of vof with lpt in openfoam, 2011. http://www.tfd.chalmers.se/~hani/kurser/OS_CFD_2011/OF_kurs_LPT_120911.pdf.

[102] S. I. G. on Multiphase Flows (SIG Multiphase). Tutorials for particle based methods, 2011. http://openfoamwiki.net/index.php/Tutorials_for_particle_based_methods.

[103] H. Nilsson. Msc/phd course in cfd with opensource software, 2013 and previous years, 2013. http://www.tfd.chalmers.se/~hani/kurser/OS_CFD/.

[104] D. Vandevoorde and N. M. Josuttis. *C++ Templates*. Addison Wesley, 2002.