A Numerical Study of Flow and Particle Deposition in Bifurcating Tubes

By M. K. Harsha Perera

Master of Philosophy Degree Thesis

School of Engineering Sciences, Department of Aeronautics and Astronautics

September 2002

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

SCHOOL OF ENGINEERING SCIENCES

DEPARTMENT OF AERONAUTICS AND ASTRONAUTICS

Master of Philosophy

A Numerical Study of Flow and Particle Deposition in Bifurcating Tubes

By M. K. Harsha Perera

Asthma drugs are administered through the mouth and the drug particles enter the upper airways. Most of these particles do not go further into the central airways, because they get deposited in the airway walls. The aim of the project is to simulate the flow field within a human airway segment and identify the proportion of particles that get deposited. A laminar steady flow model was used after making some relevant approximations on the flow behaviour due to inspiratory flow in central airways. A segment of airway in central region of the airway tree was chosen, since the flow simulation in whole airways computationally expensive and very time consuming. The geometrical parameters of a typical airway branch was investigated and double airway bifurcation that lie in the 5^{th} to 7^{th} generation of the Weibel model A airway model was chosen for flow simulation and particle tracking.

The Computational Fluid Dynamical (CFD) model in Fluent (commercial software) was chosen to well represent the physical flow model, maintain high degree of numerical stability, and faster convergence of residuals below at least four orders of magnitude less than the initial residual in first couple of iterations. A structured grid generation scheme was chosen for its accuracy and speed of solution. A strategy for creating multi-block grid is explained and some two- and three-dimensional multi-block designs were created that gave the best possible grid quality. Due to the length of time taken for particle tracking and flow simulation two-dimensional simulation was conducted in detail, instead of the three-dimensional simulation. Grid independent results were used to analyse some interesting flow features for various Reynolds numbers and various branch outlet static pressure variations.

Particle equations of motion were chosen out of most probable forces that particle could encounter when it moves with the fluid in laminar flow field. Time advance of particle equations of motion was done by 4^{th} order Runge-Kutta scheme and the interpolation of fluid velocity using the discrete field (in CFD) at particle position was done by Shape Function interpolation scheme. Vortical flow field and potential flow were used as test cases and the combination of Runge-Kutta time advance with Shape Function interpolation gave adequate accuracy for particle tracking.

Particle tracking program was written in C++ language, which uses the Fluent solver dump data in "Tecplot" format, and is also able to do multi-particle runs on a structured grid. A data structure was created for the particle for tracking in structured grid, which took into consideration the different data structures present in commercial flow solvers. The particle deposition sites were found and particle deposition efficiency was calculated. It was found that when particles were released away from the centre of the parent branch they had more chances of reaching the outlet.

Table of Contents

1 INTRODUCTION 11

1.1 SUMMARY 14
1.2 REFERENCES 14

2 FLOW THROUGH THE AIRWAYS 15

2.1 INTRODUCTION 15
2.2 DESCRIPTION AND MODELS OF THE AIRWAYS 15
2.2.1 Airway models 15
2.2.2 Latest Models 20
2.2.3 Typical airway bifurcation 20
2.3 FLOW IN THE AIRWAYS 22
2.4 FLOW MODEL 24
2.4.1 Airway model 24
2.4.2 Flow conditions 24
2.4.3 Flow equations 26
2.5 CONCLUSION 27
2.6 REFERENCES 27

3 GEOMETRY AND THE GRID 29

3.1 INTRODUCTION 29
3.2 BIFURCATION GEOMETRIES 29
3.3 GRID GENERATION 31
3.3.1 Algebraic and elliptic grid generation 32
3.3.2 Geometrical surfaces and edges 33
3.4 MULTI-BLOCK DESIGN 35
3.4.1 Controlling the grid 38
3.5 CONCLUSION 38
3.6 REFERENCES 39

4 CFD RESULTS 40

4.1 INTRODUCTION 40
4.2 NUMERICAL SOLUTION PROCESS 40
4.2.1 Evaluating the fluxes 41
4.2.2 Solving the pressure-velocity coupling 41
4.2.3 Solving the discrete system of equations 42
4.3 COMPUTATIONAL RUNS 42
4.3.1 Validation 46
4.4 GENERAL FLOW FEATURES WITHIN BIFURCATIONS 57
4.4.1 Entrance flow 61
4.4.2 First bifurcation 61
4.4.3 Entrance to first set of daughter tubes 61
4.4.4 Second bifurcation 62
4.5 THREE DIMENSIONAL FLOW MODEL 69
4.6 CONCLUSION 70
4.7 REFERENCES 70
4.8 VELOCITY MAGNITUDE PLOTS FOR VARIOUS REYNOLDS NUMBERS 72

5 PARTICLE EQUATIONS OF MOTION AND INTERPOLATION METHODS 76

5.1 INTRODUCTION 76
5.2 CHARACTERISTICS OF AEROSOL PARTICLES 76

5.3 PARTICLE MOTION 77
5.3.1 *Drag force* 78
5.3.2 *Basset force* 78
5.3.3 *Saffman force* 79
5.3.4 *Magnus force* 79
5.3.5 *Particle equations of motion* 80
5.4 NUMERICAL SCHEMES TO SOLVE THE SECOND ORDER ODE 81
5.5 INTERPOLATION SCHEMES 82
5.6 CONCLUSION 85
5.7 REFERENCES 85

6 PARTICLE TRACKING ON A STRUCTURED GRID 86

6.1 INTRODUCTION 86
6.2 SEARCH AND LOCATE PROCESS 86
6.2.1 *Solution procedure in Computational Domain* 87
6.2.2 *Solution procedure in Physical Domain* 89
6.3 DATA STRUCTURES 91
6.3.1 *Structured single block data structure* 91
6.3.2 *Structured multi-block data structure* 92
6.3.3 *Unstructured data structure* 93
6.4 TRACKING PROCESS 95
6.5 VALIDATION PROCESS 96
6.5.1 *Fluid particle tracks in Vortical flow* 96
6.5.2 *Fluid particle tracks in potential flow* 98
6.5.3 *Three dimensional fluid particle tracks* 100
6.6 PARTICLE DEPOSITION 101
6.7 CONCLUSION 105
6.8 REFERENCES 106

7 CONCLUSIONS 108

7.1 REFERENCES 110

8 APPENDIX A 111

8.1 WEIBELS MODEL A 111

9 APPENDIX B 112

9.1 INPUT PARAMETERS USED IN THE PARTICLE TRACKING CODE 112
9.1.1 *Multi-Block connectivity in file "mod.txt"* 112
9.2 PROGRAMS FOR THREE-DIMENSIONAL PARTICLE TRACKING 113
9.2.1 *Introduction* 113
9.2.2 *bgather.cpp* 113
9.2.3 *calc_gridnum.cpp* 113
9.2.4 *direction.cpp* 115
9.2.5 *face.cpp* 116
9.2.6 *grid_vertex.cpp* 116
9.2.7 *makesure.cpp* 117
9.2.8 *move_check.cpp* 117
9.2.9 *nameblock.cpp* 117
9.2.10 *namegrid.cpp* 118
9.2.11 *part_rel.cpp* 118
9.2.12 *reading.cpp* 119
9.2.13 *runge.cpp* 120
9.2.14 *side_vector.cpp* 120
9.2.15 *track.cpp* 120

9.2.16 functions.h *122*
9.2.17 tophead.h *122*
9.2.18 variables.h *123*

10 APPENDIX C **124**

10.1 PRE-PROCESSING FLUENT UNSTRUCTURED DATA – I 124
10.1.1 find_six.cpp *124*
10.1.2 Flu_man.cpp *124*
10.1.3 ini_read.cpp *124*
10.1.4 print_fun.cpp *125*
10.1.5 store_cell.cpp *125*
10.1.6 flu_struct.h *125*
10.1.7 struct.h *126*
10.1.8 things.h *126*
10.2 PRE-PROCESSING FLUENT UNSTRUCTURED DATA – II 126
10.2.1 find_six.cpp *126*
10.2.2 flu_man.cpp *126*
10.2.3 ini_read.cpp *126*
10.2.4 print_fun.cpp *127*
10.2.5 flu_struct.cpp *127*
10.2.6 struct.cpp *127*
10.2.7 things.h *127*
10.3 SOLVER 127
10.3.1 cart_curvi.cpp *127*
10.3.2 initial_part.cp *128*
10.3.3 initial_search.cpp *128*
10.3.4 Interpol.cpp *129*
10.3.5 main_prog.cpp *130*
10.3.6 reading.cpp *132*
10.3.7 search_locate.cpp *133*
10.3.8 tstep.cpp *134*
10.3.9 write_dump.cpp *135*
10.3.10 write_dump2.cpp *135*
10.3.11 ext_struct.h *135*
10.3.12 structures.h *135*
10.3.13 tophead.h *136*

11 APPENDIX D **137**

11.1 REFERENCE 138

12 APPENDIX E **139**

List of Figures

FIGURE 2-1 MAIN SEGMENTS OF THE HUMAN AIRWAYS (WEIBEL 1991) 16
FIGURE 2-2 CAST OF THE HUMAN AIRWAYS SHOWING THE MULTITUDE OF THE BRANCHING (WEB REF: 01) 16
FIGURE 2-3 A PERIPHERAL SEGMENT OF THE LOWER HUMAN AIRWAY (WEIBEL 1991) 17
FIGURE 2-4 REGULAR AND IRREGULAR BRANCHING MODELS 18
FIGURE 2-5 WEIBEL'S SYMMETRICAL MODEL OF THE AIRWAYS 18
FIGURE 2-6 CROSS-SECTIONAL CHANGES 'A' TO 'E' AS SEEN ON THE BIFURCATION PLANE AND THE LENGTH OF THE FLOW DIVIDER OR THE TRANSITION ZONE, T 21
FIGURE 2-7 DIFFERENT SHAPES OF CARINA A, B AND C, AND D IS THE CROSS-SECTION PERPENDICULAR TO THE BIFURCATION PLANE OR THE VERTICAL PLANE, (HORSFIELD, DART ET AL. 1971) 21
FIGURE 2-8 IDENTIFYING KEY BRANCHES OF THE UPPER AIRWAYS, (COHEN 1993). 23
FIGURE 2-9 APPROXIMATE FLOW RATE PATTERN FOR QUIET BREATHING. 26
FIGURE 3-1 THREE-DIMENSIONAL SINGLE BIFURCATION GEOMETRY, DRAWN USING THE PARAMETRIC EQUATIONS DESCRIBED IN APPENDIX E IN MATLAB SOFTWARE. 30
FIGURE 3-2 PHYSICAL AND COMPUTATIONAL REPRESENTATION OF AN ANNULUS, (THOMPSON, WARSI ET AL. 1985) 32
FIGURE 3-3 COMPLEX SURFACE SHAPE OF THE TRANSITION ZONE OF A BIFURCATION. THE SURFACE IS MADE MORE APPARENT BY THE YELLOW CURVES, OR RIBS ON THE SURFACE. 33
FIGURE 3-4 A GOOD SURFACE GRID PRODUCED AS A RESULT OF THE RIBS IN FIGURE 3-3. 34
FIGURE 3-5 THE PERCENTAGE OF CROSS-SECTIONAL ERROR BETWEEN CALCULATED AND EXACT, $2D^2/D^2$. 34
FIGURE 3-6 O-TYPE EDGE DESIGN ON A CROSS-SECTION OF A PIPE. 35
FIGURE 3-7 MULTI-BLOCK DESIGN OF THE THREE-DIMENSIONAL BIFURCATION 36
FIGURE 3-8 MULTI-BLOCK DESIGN OF TWO DIMENSIONAL DOUBLE BIFURCATION FACING UP37
FIGURE 3-9 THE COMPLETE TWO DIMENSIONAL BLOCK DESIGN 37
FIGURE 3-10 THE STRUCTURED GRID AT THE FIRST BIFURCATION REGION OF THE TWO DIMENSIONAL GEOMETRY 38
FIGURE 4-1 CELL NUMBERS PER EDGE FOR THREE GRID SIZES 44
FIGURE 4-2 RESIDUAL PLOT FOR THE FINEST GRID, G1 44
FIGURE 4-3 RESIDUAL PLOT FOR THE MEDIUM SIZE GRID, G2 45
FIGURE 4-4 RESIDUAL PLOT FOR THE COARSEST GRID, G3 45
FIGURE 4-5 CROSS-SECTIONAL VELOCITY DATA EXTRACTION LINES FOR GRID INDEPENDENCE. 47
FIGURE 4-6 WALL LABELS USE TO EXTRACT STATIC PRESSURES 47
FIGURE 4-7 MAGNITUDE OF VELOCITY ON LINE-1 48
FIGURE 4-8 MAGNITUDE OF VELOCITY ON LINE-2 48
FIGURE 4-9 MAGNITUDE OF VELOCITY ON LINE-3 49
FIGURE 4-10 MAGNITUDE OF VELOCITY ON LINE-4 49

FIGURE 4-11 MAGNITUDE OF VELOCITY ON LINE-5 50
FIGURE 4-12 MAGNITUDE OF VELOCITY ON LINE-6 50
FIGURE 4-13 MAGNITUDE OF VELOCITY ON LINE-7 51
FIGURE 4-14 MAGNITUDE OF VELOCITY ON LINE-8 51
FIGURE 4-15 MAGNITUDE OF VELOCITY ON LINE-9 52
FIGURE 4-16 MAGNITUDE OF VELOCITY ON LINE-10 52
FIGURE 4-17 MAGNITUDE OF VELOCITY ON LINE-11 53
FIGURE 4-18 MAGNITUDE OF VELOCITY ON LINE-12 53
FIGURE 4-19 MAGNITUDE OF VELOCITY ON LINE-13 54
FIGURE 4-20 STATIC PRESSURE ALONG WALL 1 54
FIGURE 4-21 STATIC PRESSURE ALONG WALL 2 55
FIGURE 4-22 STATIC PRESSURE ALONG WALL 3 55
FIGURE 4-23 STATIC PRESSURE ALONG WALL 4 56
FIGURE 4-24 STATIC PRESSURE ALONG WALL 5 56
FIGURE 4-25 CLOSE UP AT WALL 1, OUTER WALL OF FIRST BIFURCATION 57
FIGURE 4-26 THE PRESSURE DIFFERENCE BETWEEN EACH BRANCH OUTLET AND THE INLET 58
FIGURE 4-27 MASS FLOW RATE THROUGH THE OUTLETS, WHERE MASS1 IS THE MASS FLOW RATE OUT OF THE OUTERMOST BRANCH, WHILE MASS2 IS THE MASS FLOW RATE OUT OF THE INNERMOST BRANCH. 60
FIGURE 4-28 SUM OF PRESSURE DIFFERENCES BETWEEN TWO OUTLETS AND THE INLET 60
FIGURE 4-29 VELOCITY WITH OUTERMOST BRANCH OUTLET PRESSURE OF 1PA 63
FIGURE 4-30 PRESSURE WITH OUTERMOST BRANCH OUTLET PRESSURE OF 1PA 63
FIGURE 4-31 VELOCITY WITH OUTERMOST BRANCH OUTLET PRESSURE OF 0.8PA 64
FIGURE 4-32 PRESSURE WITH OUTERMOST BRANCH OUTLET PRESSURE OF 0.8PA 64
FIGURE 4-33 VELOCITY WITH OUTERMOST BRANCH OUTLET PRESSURE OF 0.6PA 65
FIGURE 4-34 WITH OUTERMOST BRANCH OUTLET PRESSURE OF 0.6PA 65
FIGURE 4-35 VELOCITY WITH OUTERMOST BRANCH OUTLET PRESSURE OF 0.4PA 66
FIGURE 4-36 PRESSURE WITH OUTERMOST BRANCH OUTLET PRESSURE OF 0.4PA 66
FIGURE 4-37 WITH OUTERMOST BRANCH OUTLET PRESSURE OF 0.2PA 67
FIGURE 4-38 PRESSURE WITH OUTERMOST BRANCH OUTLET PRESSURE OF 0.2PA 67
FIGURE 4-39 VELOCITY WITH OUTERMOST BRANCH OUTLET PRESSURE OF 0.0PA 68
FIGURE 4-40 PRESSURE WITH OUTERMOST BRANCH OUTLET PRESSURE OF 0.0PA 68
FIGURE 4-41 SECONDARY FLOW ON A DAUGHTER TUBE 69
FIGURE 4-42 MAGNITUDES OF VELOCITY AT A REYNOLDS NUMBER OF 200 72
FIGURE 4-43 MAGNITUDES OF VELOCITY AT A REYNOLDS NUMBER OF 400 73
FIGURE 4-44 MAGNITUDES OF VELOCITY AT A REYNOLDS NUMBER OF 600 74
FIGURE 4-45 MAGNITUDES OF VELOCITY AT A REYNOLDS NUMBER OF 800 75
FIGURE 6-1 SHOWS THE LOCAL (ξ_0,η_0) AND GLOBAL (ξ,η) COORDINATE VARIABLES IN COMPUTATIONAL SPACE 87

FIGURE 6-2 ARBITRARY CV WITH NODES LOCATED AT A, B, C, AND D VERTICES. THE DIRECTION OF SIDES CAN BE IDENTIFIED AS NORTH (N), SOUTH (S), EAST (E), AND WEST (W) 89

FIGURE 6-3 PRE-DETERMINED MULTI-BLOCK ARRANGEMENT 92

FIGURE 6-4 UNSTRUCTURED ORDERING OF NODES FORMING STRUCTURED CELLS 94

FIGURE 6-5 COMPUTED PARTICLE TRACKS USING BI-LINEAR INTERPOLATION AND EXACT TRACKS WITH CONSTANT PARTICLE DISPLACEMENTS 97

FIGURE 6-6 COMPUTED PARTICLE TRACKS USING SHAPE FUNCTION INTERPOLATION AND EXACT TRACKS WITH CONSTANT TIME STEPS 97

FIGURE 6-7 ABSOLUTE ERROR BETWEEN THE EXACT AND THE COMPUTED RESULTANT DISPLACEMENT AT EACH TIME STEP 98

FIGURE 6-8 COMPUTED AND EXACT TRAJECTORIES IN THE POTENTIAL FLOW 99

FIGURE 6-9 THREE-DIMENSIONAL MULTI-BLOCK STRUCTURE 100

FIGURE 6-10 FOUR PARTICLE TRACKS IN THREE-DIMENSIONAL MULTI-BLOCK DOMAIN 101

FIGURE 6-11 PARTICLE TRACKS FOR STK=0.02. ONLY 10 PARTICLE TRACKS ARE SHOWN. 103

FIGURE 6-12 PARTICLE TRACKS FOR STK=0.12. ONLY 10 PARTICLE TRACKS ARE SHOWN. 104

FIGURE 6-13 NON-UNIFORM PARTICLE DISTRIBUTION WITH STARTING POSITIONS OF FIRST AND SECOND BIFURCATION DEPOSITION FOR STK=0.12. 104

FIGURE 6-14 NON-UNIFORM PARTICLE DISTRIBUTION WITH STARTING POSITIONS OF FIRST AND SECOND BIFURCATION DEPOSITION FOR STK=0.12. 105

FIGURE 6-15 PARTICLE DEPOSITION EFFICIENCY FOR A REYNOLDS NUMBER OF 600 105

FIGURE 11-1 CROSS-SECTIONS FROM WHICH DATA ARE EXTRACTED LABELLED FROM 1 TO 6 137

FIGURE 11-2 SIX SUB-FIGURES SHOWS SIX AXIAL VELOCITY PROFILES AT SIX CROSS-SECTIONS (SHOWN IN FIGURE 11-1) IN THE THREE DIMENSIONAL BIFURCATION 138

FIGURE 11-3 SIX SUB-FIGURES SHOWS THE VELOCITY PROFILES NORMAL TO THE HORIZONTAL BIFURCATION PLANE AT SIX CROSS-SECTIONS (SHOWN IN FIGURE 11-1) IN THE THREE DIMENSIONAL BIFURCATION 138

FIGURE 12-1 GEOMETRICAL SPECIFICATIONS OF THE THREE DIMENSIONAL BIFURCATION.. 139

List of Tables

TABLE 2-1 SOME AIRFLOW CHARACTERISTICS IN THE WEIBEL AIRWAY MODEL A UNDER NORMAL BREATHING CONDITIONS (PEDLEY AND KAMM 1991) 22

TABLE 3-1 SOME RULES TO CONSIDER IN A MULTI-BLOCK DESIGN 36

TABLE 4-1 MASS FLOW RATE AND PRESSURE CHANGES IN 5^{TH} TO 7^{TH} MODEL 59

TABLE 5-1 4^{TH} ORDER RUNGE-KUTTA METHOD TO SOLVE 2^{ND} ORDER PDE OF THE PARTICLE EQUATIONS OF MOTION 82

TABLE 5-2 SOME REQUIREMENTS OF THE INTERPOLATION FUNCTION. 83

TABLE 6-1 SUMMARY OF THE PARTICLE TRACKING PROCESS 95

TABLE 6-2 PARTICLE CHARACTERISTICS 102

Acknowledgement

I like to express my sincere gratitude to my internal examiner Dr. S. R. Turnock, and my project supervisors Dr. O. R. Tutty, and Dr. N. Bressloff for their guidance, support and patience in completing this project.

I also want to thank Prof. Leiber, Department of Mechanical and Aerospace Engineering at State University of New York (US) for allowing me to use his experimental data for validation purposes, and explaining the parametric geometrical equations describing an airway branch.

I am grateful to my colleagues at Computational Engineering Design Centre for their assistance on countless occasions. In particular I like to thank Monica Feng, William Battern, Alfred Tan, and Whui Liang.

1 Introduction

The human airway is composed of large number of bifurcating (dividing into two) tubes, all of which stem from the Trachea. The mouth and the nose open up the airways to atmospheric gases at atmospheric pressures. These gases are transported through the airways into the lower parts of the lung by the means of a pressure difference. Expansion of the lower airways is due to the expansion of the rib cage and downward movement of the diaphragm reduces the internal pressure of the lower airways with respect to the atmosphere. Air travels through the upper airways at moderate speeds (where compressible effects can be ignored) and is slowed down in lower airways until molecular diffusion becomes the means of air transport. Chemical reactions occur on the peripheral alveoli on the lower airway, where mostly oxygen from the gases is absorbed and carbon dioxide is released by the blood flowing in the capillaries.

Studying the process of respiration has assisted the diagnosis of various lung diseases and airway abnormalities that restrict airflow in the airways. It is realised that the airways offer a unique path for the administeration of drugs into various parts of the airways and then straight in to the blood stream. However currently only Asthmatic drugs are widely administered through the airways to help asthma sufferers to breathe normally. Other potential inhaled medicines, that are currently administered into the body by injections, include Insulin for diabetics and Antibiotics to treat chronic infections (Finlay, Lange et al. 2000). A study of respiration to deliver the drugs would require the knowledge of the flow dynamics within the airways and the dynamics of drug particles in the flow domain.

The pressure acting on the airway tubular structure may be different from one part of the lung to the other. In fact due to the airway asymmetry the air supplied to each lobe in the lung is different. The models describing the airways are investigated in chapter 2. The upper and the central airways under normal breathing conditions are considered to be rigid compared to the lower airways. Hence the pressure difference created at the lower airways is transmitted to the airway opening through cross-sectional pressure differences rather than pressure forces acting on the surface of the airways. The mass flow rate at of the central airways will be a function of the airway model and the airway branch outlet pressures.

In general, the airway tree is created by tubes that branch into two. The simplest model to study is an airway bifurcation, a single branch in the airways, where unique flow changes

are created. In chapter 2 the geometrical features of such a branch is investigated. The flow within a rigid branch is due to its local geometrical attributes, downstream flow conditions such as outlet pressures, and upstream flow conditions such as the velocity distribution. Hence a method has been developed to study the flow irregularities due to different outlet pressures for given inlet flow rate. The pressure acting on the flow at branch outlet is assumed to be constant across the outlet cross-section. The axial velocity profile through a single bifurcation seems to be skewed (Zhao Yao 1997). Hence double bifurcation geometry is needed to investigate the cascade effect of one bifurcation on the other. A symmetrical double branch is chosen at 5^{th} to 7^{th} generation of Weibel airway Model 'A', which lies in the central human airways. The choice of this airway branch will be made apparent in chapter 2, section 2.4.1. A structured multi-block grid, which is needed for Computational Fluid Dynamics (CFD) model used, is created on the model geometry and the detail grid generation process is given in chapter 3. The flow is greatly influenced by the exact complex shape of the model used. Hence a detail description of the airway models and bifurcations are given in chapter 2.

The lower airways are located in different lung lobes, which are labelled as either lower or upper. The path length from Trachea to the alveoli on the upper lobes is smaller than the path length to alveoli on lower lobes, which is evident from the asymmetrical structure of the upper airway. This implies that mass flow rate to central airways, which are located on each lobe, will be different. The central airways contain more symmetrical bifurcations. For a given symmetrical inlet flow profile the outlet branch axial velocity profiles are skewed. It is expected, when outlet pressures are relatively the same cascade of symmetrical branches would not have equal mass flow rates at their outlets (Andrade, Alencar et al. 1998). In order to provide near-equal mass flow rates to the lower airways different outlet pressures may have to be applied on the symmetrical branch outlets. This may be justified by the fact that the pressure exerted on the lung is different in different regions of the lung (Cohen 1993). In the absence of the mass flow rate data in central airways a range of pressure differences has to be investigated for realistic flow simulation, as well as, how the range of pressure differences affects the velocity profile and mass flow rate.

The airway flow dynamics are affected by multitude of factors including the wall friction from a large surface area and cartilage in upper airways, viscosity changes due to presence of water vapour, change in density due to local volume and pressure changes, and local pressure changes due to external forces acting on the airway wall. The various forces

applied to the airflow will invariably create unique flow patterns within the airway branches. In chapter 2 efforts are made to reduce the complexity of the fluid flow problem by creating assumptions that at the same time would retain the unique airflow patterns in inspiratory flow.

The CFD can be used to solve the detailed flow field around many complex shapes such as aircrafts and ships. In chapter 4, the methods used to solve the flow field within airway branches will be explained. Fluent, a commercial CFD flow solver (Fluent 1998) will be used to solve the flow field. In order to model and simulate the flow the pre-processor of Fluent "Gambit" is used to create the geometry as well as the grid. Using Fluent a converged solution on three different grids, each finer than the previous, is used to carry out a grid independence study. A solution is sought whose residual that does not to change within some tolerance limit from one grid to another. This result is used to carry out fluid flow analysis. The validity of the CFD results are confirmed by referring to experimental results given on literature of similar flow conditions and geometry, or by referring to results from already validated CFD results under the similar conditions.

Assuming that all drugs are administered in droplet or particle form the airway flow patterns must be known *a priori* to deduce the drug particle motion, and thus to make inferences on the effect of flow patterns on the drug delivery. Current drug delivery systems such as the inhalers used by asthma patients deliver the drug into the inspiratory air stream as fine particle sprays that go into the airway through the mouth. Drug studies have shown that significant amount of these drugs are deposited within the first few branches of upper airway, and not further down the airways where they are needed for effective treatment. Simulating particle tracks can provide general information on particle paths and particle deposition sites.

The administered drug is assumed to be composed of solid particles and these solid particle motions will be simulated using deterministic models, i.e. individual particles will be tracked. In chapter 5 the fluid forces that are exerted on the particle motion are explained, and the largest force, the Stokes' drag term, is used in the particle equations of motion. The numerical scheme used to solve the particle equations of motion is described with relevant interpolation schemes. A particle tracking code was written in C++ programming language. The executable program will take the grid independent Eulerian flow field data, cell centred grid, and the particle initial positions. Large number of particles can be tracked in sequence. In chapter 6 the particle tracking process is designed. The quality of particle tracks depends on the detailed flow patterns and how well this information is used to solve

the particle equations of motion. When a particle enters a wall cell (on a cell centred grid) it is considered to be deposited. The relevant program files are given in Appendix B (multi-block three-dimensional structured grid) and Appendix C (two-dimensional structured grid). The ratio between the deposited particles and those entering the bifurcation is calculated, which is the drug delivery efficiency that provide some information about the effectiveness of the inhaled drugs in the airways.

Considering the time limitation for designing a particle tracking routine in three-dimensions, and the complexity of the CFD procedure in three-dimensions, a two-dimensional CFD procedure was chosen. The complexity of the three-dimensional structured CFD procedure and generating particle tracking procedure on that data will be made evident in chapters 3 (grid generation) and chapter 6 (particle tracking). However most of the essential parts of the three-dimensional solution process and particle tracking will be highlighted without significant validation of the results. The two-dimensional CFD solution process and particle tracking procedure will be completely validated and tested. Finally in chapter 7 main conclusions from the project are drawn together with recommendations for future studies.

1.1 Summary

1. In an effort to simulate varying degree of asymmetrical flow, which is known to occur in the human airways, a double bifurcation two-dimensional model is used in the CFD study. The inflow boundary conditions are steady and the flow rate is fixed, and the outlet static pressure is varied to simulate the asymmetrical flow conditions.

2. Particle equations of motion are solved numerically using designed particle tracking procedure implemented in C++ program. The multiple particles are tracked as a post process using Eulerian CFD data from Fluent CFD software.

1.2 References

Andrade, J. S., A. M. Alencar, et al. (1998). "Asymmetrical Flow in Symmetric Branched Structures." Physical Review Letters **81**(4): 926-929.

Cohen, B. S. (1993). "Factors affecting distribution of airflow in a human tracheobronchial cast." Respiration Physiology **93**: 261-278.

Finlay, W. H., C. F. Lange, et al. (2000). "Lung delivery of aerosolized dextran." Americal Journal of Critical Care Medicine **161**: 91-97.

Fluent (1998). Pre-processor Gambit, and flow solver Fluent 5.5.

Zhao Yao, L. B. B., Brunskill C T (1997). "Inspiratory and expiratory steady flow analysis in a model symmetrically bifurcating airway." Journal of Biomechanical Engineering, (TASME) **119**: 52-58.

2 Flow through the airways

2.1 Introduction

This chapter present the model flow problem. To model the airway flows, the factors governing the flow, the model of the airway geometry, and the equation that best describe the flow field are essential. There are many factors that are known to affect the flow within airways, including the flow through complex cross-sections, periodic volume change and associated pressure change, and gas mixing and gaseous exchange. The approximations will be made to simplify the factors affecting the airflow. In section 2.2 airway branch models will be presented, and the airway bifurcation geometry is described. In the section 2.3 the general flow trends within the airway models and actual airways are presented. A description of the chosen flow model will be outlined in section 2.4.

2.2 Description and models of the airways

The complexity of the airway flow may be attributed to its complex structure of tubes that bifurcate out to form cascade of branches that leads to many hundreds of thousands of tubes. The cross-section of the airways has complex cross-sectional shapes. The complexity of the airway branching has been modelled by making some approximation as to how branching takes place in rivers and in vegetation (Horsfield 1991). Airway models were formed by some researchers aiming to understand the airflow distribution in lung, which could be used to identify diseased airways.

2.2.1 Airway models

The lungs are known to control various biological and chemical processes that are essential to the function of a healthy body. The air movement, and thus the oxygen release and carbon dioxide intake, is controlled. Air is moved in to the lung by the contracting of the diaphragm forcing it to move down and forward of the body, thereby increasing the thoracic volume. This is assisted by the muscles on the rib cage, which pull the ribcage up and forward. In general the majority of the volume change in the airways occurs in the lower airways and upper airways retain geometrical similarities during volume changes.

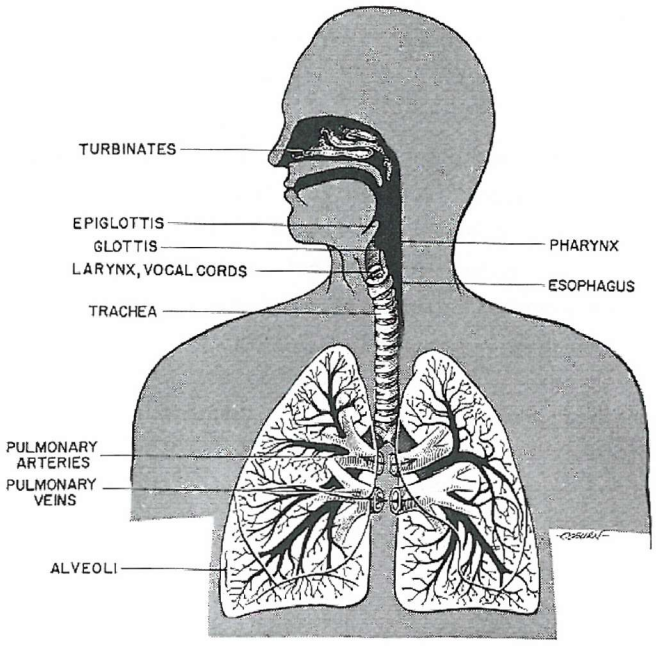


Figure 2-1 Main segments of the human airways (Weibel 1991)

Figure 2-1 shows a schematic of the whole airways starting at nose and mouth and ending at alveoli where air interfaces with the blood through a membrane. A more detailed picture of the airways from a cast is shown in Figure 2-2. It shows a complex system of branches, which needs to be represented by a simplified branching model so it can be studied.

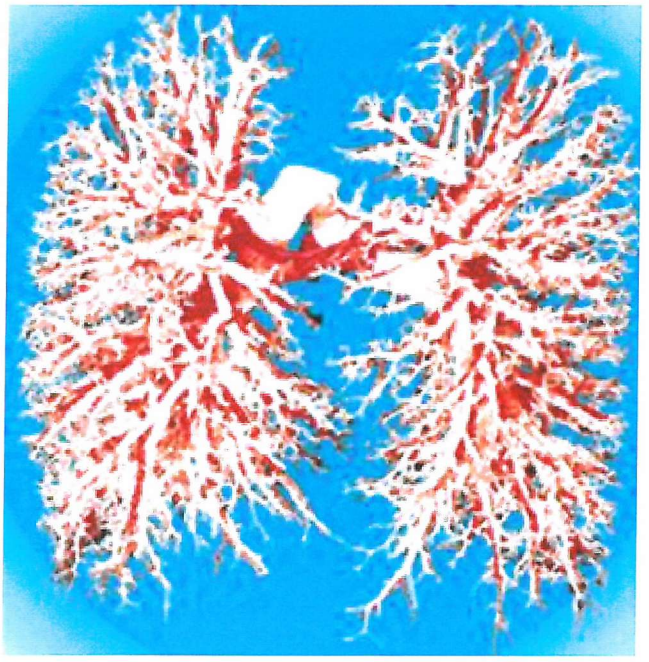


Figure 2-2 Cast of the human airways showing the multitude of the branching (Web Ref: 01)

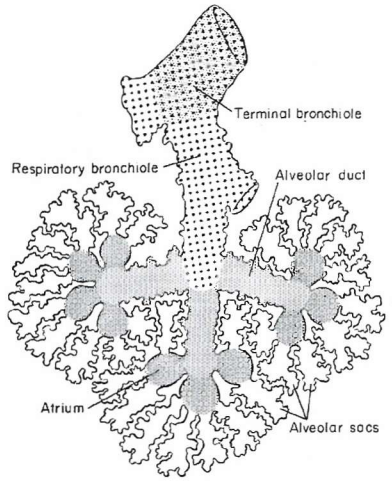


Figure 2-3 A peripheral segment of the lower human airway (Weibel 1991)

The central airways ends at the terminal bronchiole and from there the lower airways begin, as shown in Figure 2-3. The lower airways end at the alveoli.

The standard physiologcal model of the airways represents branching by dichotomy, with the parent branch dividing only into two branches, Figure 2-4. The stem of a bifurcation where branching takes place is called the parent and the tubes that branches out are called the daughters. However within the bronchial tree, as can be seen by Figure 2-4, branching by trichotomy is apparent where a parent branches out to produce three daughters (Weibel 1963). Such a branch could be described as two branches of dichotomy with very small parent length on the second branch. When dichotomy is used to model airways all the airway branches are represented as some combination of a unit branch, a bifurcation, which has some key geometrical parameters including lengths, diameters, branch angles, and outside curvatures (section 2.2.3). In a dichotomous system one may compare the changes in branch geometry from one branch to another to understand and develop a general airway model.

Airway measurements published by (Weibel 1963) and (Horsfield, Dart et al. 1971) are extensively used to describe airways. In both these investigations the airways have been separated into three parts: upper, central and lower airways. In (Horsfield, Dart et al. 1971) the upper airway ends at branches with 0.7mm diameters (terminal bronchioles). (Weibel 1963) named the upper airway as the conductive zone and the lower airways as the transition zone, as shown in Figure 2-5, where the conducting airways end at generation 16.

An airway system can be further analysed by a numbering system to identify a branch within the airways. It is here that these researchers differ, i.e. they used two different numbering systems to identify a branch within the airways, and also on the extent of

measurements they carried out. Weibel looked at the airway branches with respect to the Trachea. If the number of dichotomous sub-divisions or nodes, N_n are known then the total number of resulting branches, N_e can be given by Equation 2-1.

Equation 2-1

$$N_e = 2N_n + 1$$

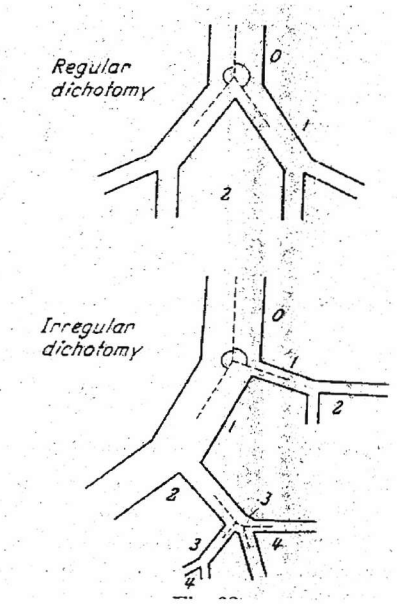


Figure 2-4 Regular and irregular branching models

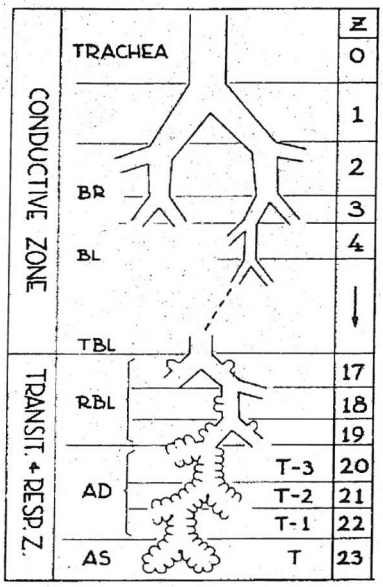


Figure 2-5 Weibel's symmetrical model of the airways

Equation 2-2

$$N(z) = 2^z$$

Weibel's model "A" of the transitory zone (upper airways) constitutes a symmetrical regular dichotomous airway system (measurement data given in Appendix A). The trachea was given an order of zero and each following branch is allocated a generation number, z. Then the total number of branches in a given generation, z can be presented by the Equation 2-2. All branches of a given generation have the same dimensions and subsequent branches from one generation to the next, some geometrical parameters are allowed to be varied, like the branch angle, branch diameter and the branch length. Since the real airways are not symmetrical Weibel also had "Model B" with irregular dichotomy, described with respect to the branch angle, as opposed to diameter difference of daughter branches. In this model the respiratory bronchioles are within the generations 17 to 19, with generation 20 to 23 being alveoli (belong to the transitory zone as shown in Figure 2-5). In general the smaller daughter tube would be at a larger angle (on average 30° to 65°) to the parent and the larger daughter branch at smaller angle (on average around 20°), (Weibel 1991).

Irregular dichotomy with equal daughter branch diameters has been noted to occur in the upper airways. In such a case the termination of the Weibel model would occur with different generation of branches.

It has been reported that the degree of irregularity in the branching in upper airways (generations 1-7) is primarily caused by lengths of daughter tubes and to a lesser extent by the diameter or the cross-sectional area of the daughter branches. The length to diameter ratio has been used to determine the degree of irregularity among the airway segments (Weibel 1991). However it must be noted that degree of variability also arises from the technique of casting procedure used, mainly due to the degree to which the lung was "blown", usually 75% of TRC (Total Residual Capacity), and also the inter-subject variability (i.e. due to lung differences due to body height, sex, age, etc.). So a model usually represents some "averaged" airways.

Weibels measurements of the upper airways were not complete. (Horsfield, Dart et al. 1971) claims to have measured all branches up to branch of diameters of 0.7mm from one lung. Horsfield numbered all the branches with 0.7mm diameter as one (order one) and increase number to branches towards the trachea. Then the daughter branches will have an order less than that of the parent branch. If a parent branch is of order **w** then the parent branch number as a sum of daughter branch number is given by Equation 2-3.

Equation 2-3

$$N_w = N_{w-1} + N_{w-n}$$

where **n** is the difference in numbering between the daughter branches. In a symmetrical model two of the same order meets to create the next highest order, i.e. $N_w = N_{w-1} + N_{w-1}$. The greater value of **n** means greater asymmetry at a branch. Horsfield found **n** to vary between 1 and 5.

Essentially describing the airways with a numbering system where branches with the same number share common geometrical properties, Horsfield manages to capture the essence of asymmetry of the airways. Using various values of **n** the measured data were fitted to each branch of the airway model. Over the years many lung models have been developed that either use the generation and dichotomous modelling of Weibel, or asymmetrical quantification technique of Horsfield. However the asymmetry models differ from one researcher to the next. Hence most fluid dynamical studies are conducted on a symmetrical airway model.

Note that bifurcation orientation within the lung, or relative to other branches, have not being discussed, since they are difficult to quantify using older methods of measurements.

2.2.2 Latest Models

Prior to 1980's researchers directly measured bifurcation geometry (length, diameter, branch angle and so on) and used statistical techniques to obtain the approximate dimensions of the airways. More recently, computer tomography, which extracts data based on radiopacity, have being used to obtain three-dimensional spatial data on airway bifurcations (Woods, Zerhouni et al. 1995). The surface of the airways and the surrounding spaces were identified by two colours after some manipulation of the scanned data. A voxel, V, (a three-dimensional pixel) was used to represent this data and the total volume of the data set in three dimensions may be $\mathbf{VX}\times\mathbf{VY}\times\mathbf{VZ}$, where number of voxels in (x,y,z) directions are (VX,VY,VZ). There is software, which could break down the three-dimensional image file in to two-dimensional sections. Then using the voxel size of say $\mathbf{x}\times\mathbf{y}\times\mathbf{z}$ (mm) two-colour intensity two-dimensional slices can be used to obtain parameter estimation of airway cross-sections (this can be done in Matlab software). Then coordinates from all the slices are combined to produce the surface geometry of the airways. The surface geometry can be used immediately in a CFD pre-processor.

However such a model, describing the upper airways and providing more quantitative information about the changes in upper airway cross-sections, is difficult to validate and the smaller airway cross-sections are difficult to capture. Therefore older models are still used to describe the airways.

2.2.3 Typical airway bifurcation

It may be assumed that an airway branch is a hollow cylinder. Then, obviously the parent branch will not interface perfectly with its daughter branches due to the circular cross-sections. Therefore the parent branch has to change its cross-sectional area and split into two before daughter tubes are formed. The complex cross-sectional shapes of an airway branch, a bifurcation, have been approximated by conic sections.

Figure 2-6 shows a geometrical model of the airway bifurcation with the geometrical parameters, as defined by (Horsfield, Dart et al. 1971). It can be seen that there are five distinct cross-sectional changes. The flow divider is formed when the top walls of the tube move in (section **d**) and ends just before daughter tubes are produced (section **e**). The flow divider may also be called the transition zone, **T**. The inlet cross-section of the flow divider is circular and it gradually changes into elliptical shape while maintaining the same cross-

sectional area. The elliptical shape deviates after the minor curvatures of the cross-section equate to the circular curvature of the daughter tube cross-section. The cross-sectional shape becomes more flattened at the top and bottom, the smaller axis of cross-section becomes more equal to the diameter of the daughter tube (in a symmetrical bifurcation) and the cross-sectional area increases. The change in area then occurs uniformly. The length of flow divider is where the changes in cross-section occur. The two daughter tube cross-sections are approximately circular. The outer curvature of the bifurcation from section **a** to section **e**, and at some distance thereafter was assumed to be constant.

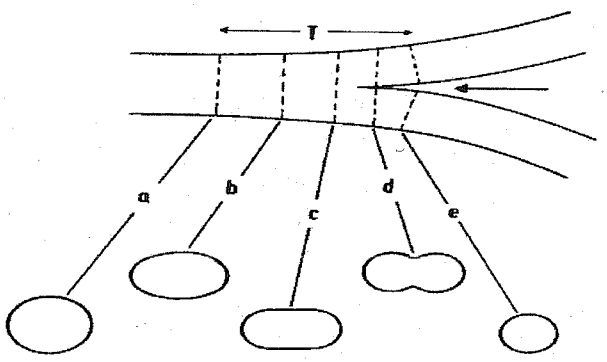


Figure 2-6 Cross-sectional changes 'a' to 'e' as seen on the bifurcation plane and the length of the flow divider or the transition zone, T

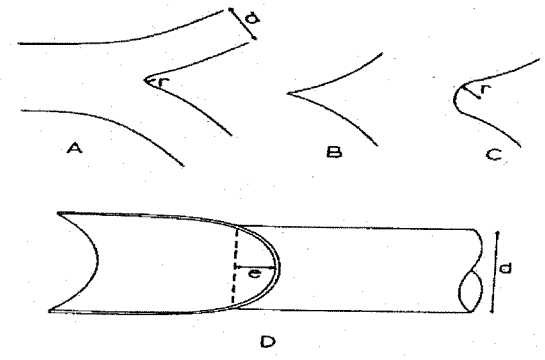


Figure 2-7 Different shapes of carina A, B and C, and D is the cross-section perpendicular to the bifurcation plane or the vertical plane, (Horsfield, Dart et al. 1971)

The shape of the apex of the bifurcation depends on the curvature of the apex, **r**, as shown in Figure 2-7. In general the ratio (**r/d**) is approximately 0.1, showing a sharp flow divider. At right angles to the plane of the bifurcation in the direction of the arrow in Figure 2-6, the length **e**, in Figure 2-7 is approximately equal to or less than the diameter of the daughter tube (Horsfield, Dart et al. 1971).

2.3 Flow in the Airways

As the air flows through the nose it gets filtered, and flow resistance increases reducing the air velocity. The temperature of the air increases. As the air passes through larynx it passes through the narrowest section of the upper airways. A jet-like flow expands in to the trachea, reaching the trachea diameter within a small distance. As a result, under normal breathing conditions, the airflow in the trachea is considered to be turbulent (Dekker 1961).

It is apparent from breathing exercises that convective flow dominates in the upper airways, and that the magnitude of convection reduces down the airways. This is apparent when speed of gases in each airway generation is estimated, as shown in Table 2-1. The Womersley parameter describes if the flow changes in time are significant. A value below 1 means most of the flow conditions can be approximated to be steady state (section 2.4.2). The larger branches like the trachea, primary bronchus, and lobar bronchus supply to many generations of branches that follow. The flow in the lower airways near gas exchange membranes is mostly by diffusion where the contact area for the gases is large. During inspiration there exists certain residual air volume inside the airways and when new air is taken in gas mixing is know to take place in the lower airways.

Table 2-1 Some airflow characteristics in the Weibel airway model A under normal breathing conditions (Pedley and Kamm 1991).

			Normal breathing at 0.5litre.sec^{-1}, or at a frequency of 0.25 Hz	
Generation	**Diameter/ cm**	**Length/ cm**	**Reynolds Number, Re**	**Womersley Parameter, α**
Trachea	1.8	12.0	2325	2.9
Primary bronchus	1.22	4.76	1719	2.0
Lobar bronchus	0.83	1.90	1281	1.3
3	0.56	0.76	921	0.91
4	0.45	1.27	594	0.73
5	0.35	1.07	369	0.57
10	0.13	0.46	32	0.21
15	0.066	0.20	1.9	0.11
20	0.045	0.083	0.09	0.07

Figure 2-8 shows the branch names organised in the lungs. Airways seem to have static and dynamic qualities to hold air and distribute it in a certain way. The branches that supply the air to the lobes have different mass flow rates depending on the number of branches followed and also the distance to the alveoli sacs. For example the lower lobes are better ventilated than the upper lobes (Cohen 1993) since the lower lobes have greater distance to

reach the alveoli sacs than the upper lobes. The airway model study from section 2.2.1 indicates that those airways that branch outwards from a branch will have shortest path to the alveoli. In general branches will decrease in diameter towards distal airways and the length of the airways decrease more on those branches that branch outwards with higher branch angle.

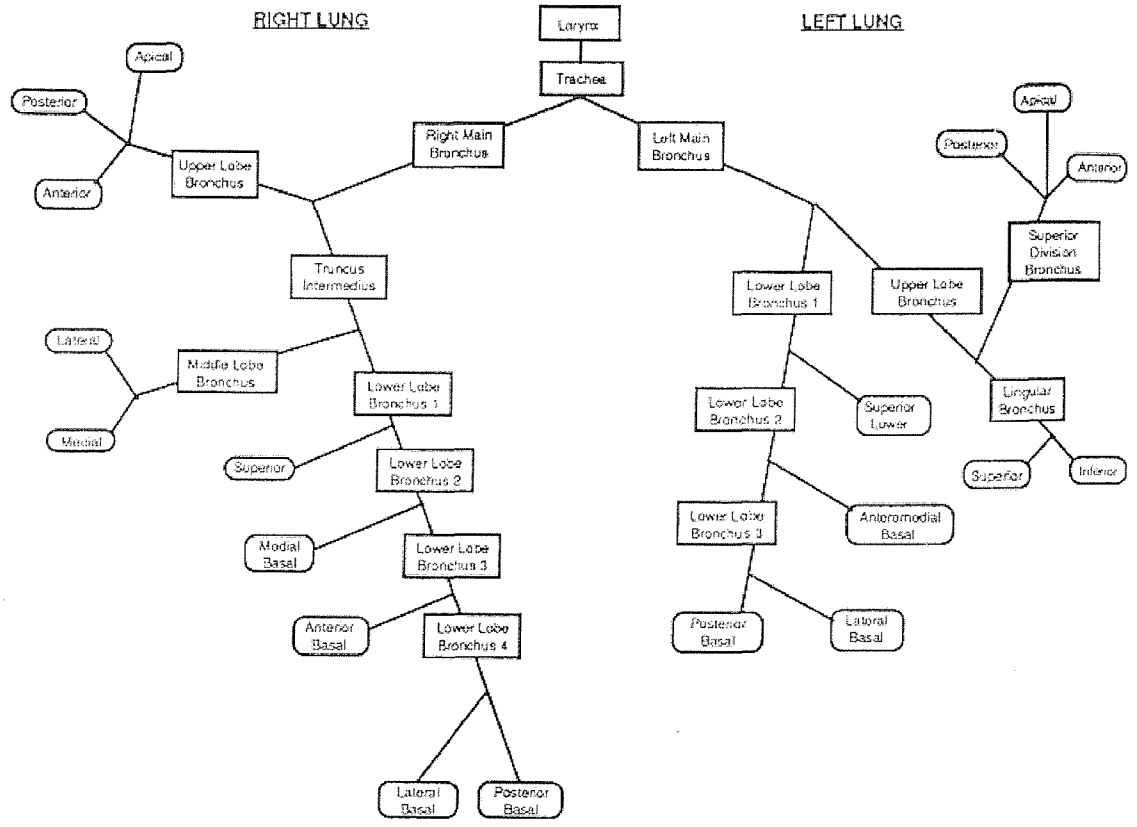


Figure 2-8 Identifying key branches of the upper airways, (Cohen 1993).

The pressure drop necessary to create a certain inflow rate is expected to depend on the energy loss due to viscous and inertial losses within the airways (for laminar flow). Using boundary layer theory (Pedley, Schroter et al. 1970a) and (Pedley, Schroter et al. 1970b) predicted the pressure drop on a fixed angle symmetrical tree. They claimed that in airway branch flows never come to a fully developed pipe flow, but always remain like a developing pipe entrance flow whenever a new daughter tube is reached. The branch length is not long enough for fully developed flow to be initiated under normal breathing flow Reynolds numbers. They have assumed that certain entry profile exists at the beginning of all the daughter branches but with different flow rates. Equation 2-4 gives their pressure drop relationship.

Equation 2-4

$$\Delta P = K\rho^{0.5}\mu^{0.5}\dot{V}^{1.5}$$

where Δp is the pressure drop across the tracheobronchial tree, $\dot{V}$ is the mean inspiratory flow rate, μ and ρ are the viscosity and the density of air, and K is a constant relating to the airway geometry. Pedley's equation and similar equations of others give only an approximate value for the pressure drop within airway branches and they assume similar flow features on each bifurcation of the airways.

Airflow distribution experiments can provide general information about the mass flow rates into branches. However specific flow rates into specific bifurcations in the airways, and outlet pressures of branches are difficult to quantify (Slutsky, Berdie et al. 1980) and (Patra and Afify 1983).

Any effort to obtain detailed flow properties would involve detailed flow modelling, and solving a flow model within a complex geometry by numerical means, i.e. using CFD. However as the number of branches increases CFD become computationally expensive and the mesh generation is expected to increase in complexity. Another major difficulty in simulating the exact flow features is the unknown boundary conditions at the model inlets and outlets. Hence the boundary conditions must be varied to realise the actual flow conditions.

2.4 Flow Model

2.4.1 Airway model

A double bifurcation model was chosen to study the flow field, with constant inflow rate and varying degrees of outlet pressure differences. Also, this model will be used to study the particle tracks and deposition. A 5^{th} to 7^{th} generation Weibel model 'A' airway branch was chosen. This airway branch lies within a lobe of the central airways, and not on the upper airways where major asymmetries lie. Hence a symmetrical geometrical model approximation is valid.

2.4.2 Flow conditions

Flow in an airway branch is investigated to determine the type of flow change that occurs, which would provide valuable information about flow resistance and pressure loss. The majority of the flow resistance seems to occur in the upper and central airways where convective flow is high and flow surface area is beginning to increase. Also as the flow rate is increased in the trachea the upper lobe resistance increases more than the lower lobe

resistance. (Slutsky, Berdie et al. 1980) suggested that this is due to the acute branch angle and shorter branch length, leading to increased flow resistance. Also different pressure differences may be created on different lobes at corresponding same flow rates to minimise the flow resistance and maximise regional ventilation. The airway resistance also depends on same global properties of the whole airways including the state of inflation of the lungs, on the direction and instantaneous magnitude of flow rate, on the density and viscosity of the gas, and to a lesser extent on the frequency of breathing, (Jeffrin and Kesic 1974). For a given lung inflation and inlet flow condition the energy losses in the airways may originate from boundary layer development, separation, turbulence or laminar pipe friction. Older resistive models assume that the total flow resistance in airways is a sum of individual branch resistive components due to pipe entrance flow development at each branch. A CFD study will investigate the flow resistance by simulating the pressure changes within the bifurcation domain.

Respiratory flow cycle is due to the periodic pressure variations within the airways. The flow resistance and the airway wall elasticity will respond to the pressure variations. Most of the upper airways do not expand as much as the lower airways, so we can assume that airway walls are rigid. The Womersley number, α, has been used to find when unsteady effects on the flow in a straight pipe are important. The Womersley number represents ratio of unsteady forces to viscous forces and it is given by Equation 2-5.

Equation 2-5

$$\alpha = 0.5\mathrm{d}\left(\frac{\omega}{\nu}\right)^{\frac{1}{2}}$$

where $\omega=2\pi f$ is the angular frequency of oscillation (for a normal breathing of 0.5 litre.s^{-1} breathing frequency, f=0.25Hz) and υ is the kinematic viscosity. If the Womersley number, $\alpha > 1$ then unsteady effects are important (Pedley and Kamm 1991). Table 2-1 provides some general Womersley parameters for the airways. It can be seen that from 3rd generation $\alpha < 1$, which signifies unsteady effects are not as important. Under steady state conditions the viscous forces, are larger and respond instantaneously to any pressure variations. Note that the above theory was defined for fully developed laminar flow on a straight pipe. However in the absence of a better theory it presents an adequate theory to suggest that most of the central airway flows under normal breathing conditions may be studied using a steady model.

During inspiration the flow rate increases rapidly from zero flow and the volume in the airways increases slowly. As the flow rate become constant the volume increase is also constant and this occur over majority of the inspiration time, as shown in Figure 2-9. Then as the flow rate is reduced to a minimum the air volume increases to a maximum and the flow rate reverses and expiration begins. In natural breathing conditions a deviation from steady flow is only observed near the initial and later part of the inspiration (Isabey and Chang 1981). So a major part of the inspiration can be considered to satisfy a steady state.

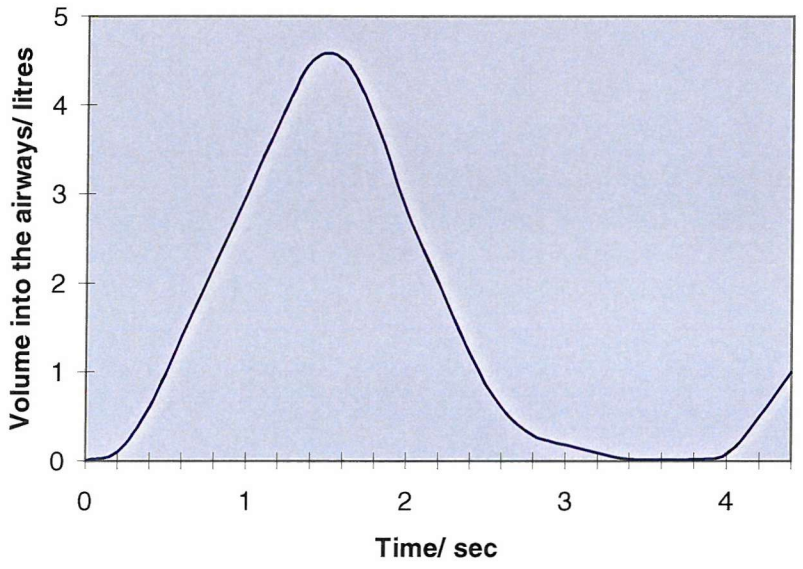


Figure 2-9 Approximate flow rate pattern for quiet breathing.

2.4.3 Flow equations

It has been identified that laminar flow dominates under normal breathing conditions in the central airways. It is assumed that there will be no heat transfer or internal energy change in the fluid. The variation of viscosity may occur if there is a marked temperature difference within the fluid, however in this case the viscosity is constant. The fluid is assumed to be Newtonian.

The flow model will be represented by the Navier-Stokes equations, Equation 2-6 and Equation 2-7, which describe the conservation of momentum and mass of an incompressible Newtonian fluid.

Equation 2-6

$$\rho\frac{Dv}{Dt}=\rho\left(\frac{\partial v}{\partial t}+v.\nabla v\right)=-\nabla P+\mu\nabla^2 v+\rho F$$

Equation 2-7

$$\nabla.\mathbf{v} = 0$$

At the wall of the geometry **u** velocity vector in **x** the no-slip boundary condition is assumed. The outlets of the flow domain will have pressure boundary conditions while the inlet will have constant flow rate. Equation 2-7 basically ensures mass conservation and Equation 2-6 shows the balance of convective, pressure and viscous forces. The body force term is ignored, F=0, since buoyancy effects are ignored. It is expected that the viscous term in the Equation 2-6 will play an important role as much as the inertial terms under the flow conditions described above 2.4.2, i.e. under laminar low Reynolds number flows.

2.5 Conclusion

Airways are complex geometrical structures. It appears that the best airway models use a branch that divides into two (dichotomy). The branch itself is modelled using conical cross-sectional shapes. From airway models some geometrical similarities of the airways were obtained, such as how the airway diameters reduce with the airway generation number, and how the airway diameter length ratio change for one branch to another. These relationships can provide useful information about the airflow development if airflow within one such branch is found accurately.

The flow is not expected to be the same from one branch to the next, since each bifurcation receives altered flow from the upstream branch. During inspiration on central airways the flow behaviour can be approximated to be steady state and laminar. It is expected that in a single bifurcation unique changes in flow field occur for certain inlet-outlet driving pressure differences and fixed inflow mass flow rate. Flow simulation using an outlet pressure difference is expected to provide useful information about typical flow changes within a bifurcation and the mechanism responsible for pressure loss or resistance. Since exact pressure differences are difficult to measure and quantify accurately in flow in model airways outlet pressure should be varied within some range.

2.6 References

Cohen, B. S. (1993). "Factors affecting distribution of airflow in a human tracheobronchial cast." Respiration Physiology **93**: 261-278.

Dekker, K. (1961). "Transition between laminar and turbulent flow in human trachea." Journal of Applied Physiology **16**(6): 1060-1064.

Horsfield, K. (1991). Morphometry of airways.

Horsfield, K., G. Dart, et al. (1971). "Models of the human bronchial tree." Journal of Applied Physiology **31**(2).

Isabey, D. and H. K. Chang (1981). "Steady and unsteady pressure-flow relationships in central airways." Journal of Applied Physiology **51**(5): 1338-1348.

Jeffrin, M. Y. and P. Kesic (1974). "Airway resistance: a fluid mechanical approach." Journal of Applied Physiology **36**(3): 354-361.

Patra, A. L. and E. M. Afify (1983). "An experimental study of velocity distribution in a human lung cast." Journal of Biomechanical Engineering, (TASME) **105**: 381-388.

Pedley, T. J. and R. D. Kamm (1991). Dynamics of gas flow and pressure-flow relationships. The Lung: Scientific Foundations. W. J. B. Crystal R G. New York, Raven Press: 995-1010.

Pedley, T. J., R. C. Schroter, et al. (1970a). "Energy losses and pressure drop in models of human airways." Respiration Physiology **9**: 371-386.

Pedley, T. J., R. C. Schroter, et al. (1970b). "The prediction of pressure drop and variation of resistance within the human bronchial airways." Respiration Physiology **9**: 387-405.

Slutsky, A. S., G. G. Berdie, et al. (1980). "Steady flow in a model of human central airways." Journal of Applied Physiology **49**(3): 417-423.

Web Ref: 01. http://depts.washington.edu/envh/lung.html.

Weibel, E. R. (1963). Morphometry of the human lung, Springer-Verlag, Berling Gohingen-Heidelberg.

Weibel, E. R. (1991). The Lung: Scientific Foundation.

Woods, S. A., E. A. Zerhouni, et al. (1995). "Measurement of three-dimensional lung tree structures using computer tomography." Journal of Applied Physiology **79**(5): 1687-1697.

3 Geometry and the grid

3.1 Introduction

A Computational Fluid Dynamic (CFD) model needs a grid on the geometry enclosing the flow domain in order to solve the discretised equations. This chapter discusses the structured grid generation for the bifurcation geometry, which was described in general in chapter two. More information about the rest of the CFD procedure will be provided in chapter four. The model geometry will be a three-dimensional single bifurcation and two-dimensional double bifurcation whose general geometrical parameters were described in chapter two. In section 3.2 a description of a typical bifurcation is given, where its shape will be described by parametric equations. There are various issues associated with structured grid generation in complex geometry, which are discussed in section 3.3. In order to have the best structured grid, a multi-block design procedure was followed and the final design approach is outlined in section 3.4. The resulting grid will be used in a CFD solver defined in chapter four. The three-dimensional single bifurcation CFD results could be validated with the experimental results of (Zhao and Lieber 1994).

3.2 Bifurcation geometries

Throughout the literature, human airway bifurcations chosen for flow studies vary. Most often Weibel's (Weibel 1963) and Horsfield's airway models (Horsfield, Dart et al. 1971) and their parameters describing bifurcation diameter and length are used to describe a typical branch for fluid dynamic investigations. (Heistracher and Hofmann 1995) investigated shapes of various bifurcation model geometries used in fluid flow studies. They described the general shape of the bifurcation geometry with some mathematical expressions. They have also categorised bifurcation models into symmetrical branch models, asymmetrical models, and branches with sharp or blunt bifurcation apexes. The extent of geometrical detail given in the Weibel and Horsfield on each bifurcation is limited, especially near the bifurcation region. Weibel and Horsfield only provide main branch diameter, daughter branch diameter, and angle of bifurcation. Therefore user-approximated parts of the geometry have to be created to form the bifurcation shape to match the description made by them. Mathematical expressions of the geometrical shapes are needed to approximate the shape of the bifurcation accurately.

Some researchers like (Zhao and Lieber 1994) chose a bifurcation geometry to make the flow vary slowly along the flow divider so flow parameters can be measured easily, while (Balashazy and Hofmann 1993) chose a "square" bifurcation, which differs considerably from any airway bifurcation, to make the computation and grid generation processes easier.

Hence in experiments and in CFD approximations to bifurcation geometry were made that deviate from the actual airway branch measurements. Nevertheless the CFD must be done on the geometry, where the flow can be validated. In order to do this, geometry that matches exactly with (Zhao and Lieber 1994) can be chosen, since they provide equations that describe the bifurcation, and also provide experimental results that can be used to validate, quantitatively, a CFD model. After validating the CFD model the parametric equations of the exact airway branch geometry described by (Heistracher and Hofmann 1995) can be used in another CFD study (not done in this project) to calculate more realistic flow about the actual airway branches. Appendix E provides detail geometric equations used to describe the geometry used by (Zhao and Lieber 1994).

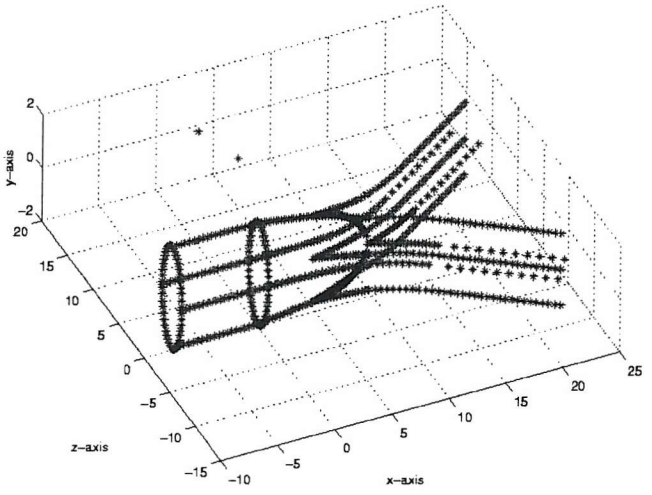


Figure 3-1 Three-dimensional single bifurcation geometry, drawn using the parametric equations described in Appendix E in Matlab software.

The mathematical equations given in Appendix E were evaluated in Matlab, and some coordinates of the geometry were calculated that describe the perimeter of the geometry and relevant centre lines, as shown in Figure 3-1. The geometrical details given are sufficient to realise the overall shape of the geometry. However it was realised that more geometrical detail, especially about the surface of the geometry, are needed to generate the surface grid accurately. The curves that appear to show the perimeter of the surface in Figure 3-1 are not sufficient to describe the exact surface of the geometry. Another set of curves has to be generated based on the information that the area change from parent to daughter branches is constant, i.e. $2d^2/D^2=1$, where d is the daughter tube diameter and D is the parent tube diameter. Once an accurate surface description is available then a surface for the geometry can be constructed using a grid generator.

3.3 Grid generation

CFD applied to complex geometry requires good solver-grid interactions to resolve the cell normal fluxes and interpolation of these fluxes within the stencil of control volumes. Then the solution produced through iteration process should be stable, and converge to an accurate solution. When the solver algorithms and the solver-grid interaction are fixed, it is the grid that needs to be altered to obtain the best solution. A grid in this case is basically a collection of points in space where small sets of those points can be used to form finite control volumes. Within each control volume the cell averaged flow quantities (such as the velocities and pressures) are approximated. There are a number of different grid types. The most effective and efficient of these are categorised into the body conforming curvilinear structured and unstructured categories.

The implicit connectivity within the grid is used to describe a structured grid. Structured grids have the advantage that the evaluation of the discretisation (of the flow equations) over the control volumes is more efficient during computation. In an unstructured grid the computation of each cell averaged property uses a number of adjacent cells falling on a certain stencil. Within this stencil the cell connectivity is not obvious, and must be stored prior to the computation. Hence the solution process takes up much more time in computation through data management. However an unstructured grid is flexible and it easily represents the complex nature of the geometry. It also has the ability to adapt to the solution. In this project structured grids are used for ease of computation and ease of particle tracking process. It will be seen that structured grids can be made to conform to the geometry using multi-block grid generation procedure.

When structured grids are used the solution is highly dependent on the grid shape and size. The best-structured grid would be those on an orthogonal face in two-dimensions. A solver based on a structured grid would perform at its best if the angles between the adjacent vectors of a control volume connected at a grid node are orthogonal and that the length scales of these vectors do not change dramatically between control volumes. It seems a difficult task to produce a grid on the complex geometry with a structured grid, since it is evident that at least some cells will be skewed.

In general the grid generation is performed over the physical domain where each grid point is described in terms of Cartesian coordinates (x, y, z). The CFD solver uses the map of this in the computational space, where the same grid point is described by curvilinear coordinates (ξ, η, ζ). Figure 3-2 shows how a physical space corresponds to an annulus,

where point 1 and 2 is the inner wall and 3 and 4 is the outer wall been transformed to the rectangular shape in computational space (Thompson, Warsi et al. 1985).

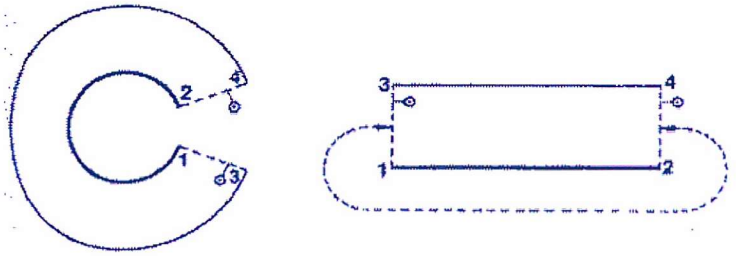


Figure 3-2 Physical and computational representation of an annulus, (Thompson, Warsi et al. 1985)

The grid generator used during this project is called Gambit, which is the Fluent pre-processor (Fluent 1998). The grid generation procedure places a number of grid points on an "edge", such as a curve describing the outer curvature of the bifurcation. Four such edges are joined to form a "face". Then the grid generator creates the interior grid nodes on the face. The grid generator may use an algebraic, elliptic or hyperbolic scheme to generate these interior grid nodes (more details are given in section 3.3.1). The user is allowed to choose the distribution and the number of cells along these edges. The grid generation scheme will then use this information together with the geometrical properties of the edges to create the face grid. The grid generator uses an interpolation scheme to interpolate the boundary grid nodes on to the interior domain of the face (more details are given in section 3.3.2). After that the generator will create a grid on the volume, a "Block" surrounded by six faces.

The user needs to define set of edges on a block (3D) or on a face (2D) such that the grid generator produces the best-shaped cells. Such a process can be executed by a multi-block strategy, where block edges are chosen carefully such that the surface grids on block faces form the best structured cells possible (more details are given in section 3.4), and in turn the face grids produce the best volume cells within the block.

3.3.1 Algebraic and elliptic grid generation

In algebraic grid generation the propagation of the grid nodes is calculated by algebraic means, and in elliptic grid generation the grid propagations is done through the solution of the partial differential equations. In the latter the generated grid is much more smooth, i.e. the gradient of grid lines vary smoothly. The effect of the edge distribution is felt more on the interior grid in the algebraic case than in the elliptic case. The curvature effects at the boundary edges can be taken into consideration in the elliptic case to produce quality-structured cells. The elliptical partial differential equations are solved iteratively. Therefore

the user must specify tolerance limits (magnitude of change in grid point locations between two successive iterative steps) and maximum number of iterations for a smoother grid. The algebraic scheme is much faster than elliptic one. It is apparent that a higher order scheme is required to generate grids on the complex geometry and elliptical or hyperbolic schemes are better suited for the structured grid generation in complex geometry. In Gambit software elliptic grid generation is used to produce the multi-block structured grid.

3.3.2 Geometrical surfaces and edges

The grid generator must be capable of generating complex geometrical surfaces using the information from the edges, i.e. the grid generator must interpolate accurately the grid nodes from the edges to the interior domain of the face. It was found that the grid generator software Gambit did not successfully generate an accurate surface using the four edges shown in Figure 3-3 as dark blue curves, since the surface enclosing the four edges requires more information to constrain its variability. Many ribs had to be drawn on the surface, which are seen as yellow coloured curves in Figure 3-3 to reduce the degree of variability on the surface geometry. Then Gambit produced a good quality surface grid.

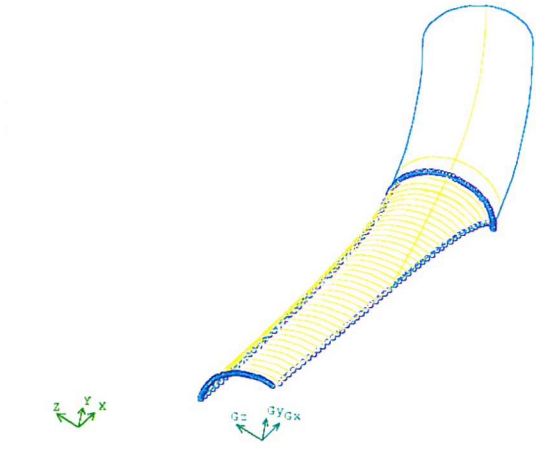


Figure 3-3 Complex surface shape of the transition zone of a bifurcation. The surface is made more apparent by the yellow curves, or ribs on the surface.

As a result a better surface grid was produced as can be seen in Figure 3-4. The detailed geometrical information made available through mathematical description of the bifurcation and the cross-sectional area changes reported by (Zhao and Lieber 1994)

facilitated in describing the bifurcation geometry to a good estimate of the original geometry. The Figure 3-5 show the error in the cross-sectional area of the geometry drawn compared to expected constant area of $2d^2/D^2$, along the flow divider starting from when parent tube cross-section is circular up to the start of daughter tube cross-section. The maximum error is just below 10%, and it occurs when the parent tube start to split. The errors could be due to rounding off when solving the equations, or due to the equations themelseves.

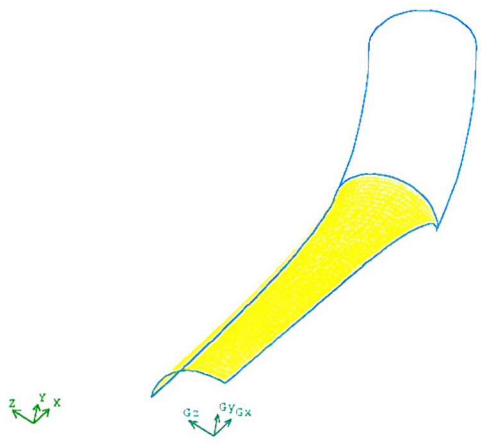


Figure 3-4 A good surface grid produced as a result of the ribs in Figure 3-3.

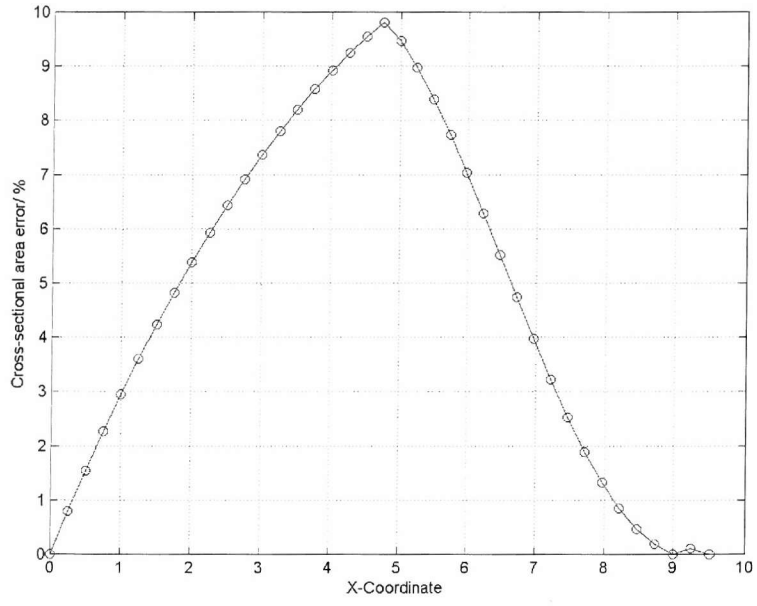


Figure 3-5 The percentage of cross-sectional error between calculated and exact, $2d^2/D^2$.

3.4 Multi-block design

Before attempting to the design of multi-blocks, the 'best' structured grid could be visualised by sketching the blocks. The logic in the design states is that for best structured grids the angle between two adjacent grids lines and thus between two adjacent face edges should be as close to 90° as possible. Also the size between cells should not vary significantly, and the edges describing a face must lie on a surface with minimum degree of spatial variation such that the grid generator is able to create a smooth surface grid successfully.

For internal grids on pipes, O-type grids seem to produce the best possible structured grids. This provides a basis for designing a multi-block design for the bifurcation. Some favourable properties of the O-type grid include the orthogonal intercept of the wall of the geometry, and orthogonal grids at the centre of the tube, which can be seen from Figure 3-6. The edge design can be thought of as a design that creates orthogonal grids at the centre of the tube meeting the constraint of orthogonal grids at the wall. Cells at the centre expand and shift in shape to produce near-orthogonal grid lines at the wall.

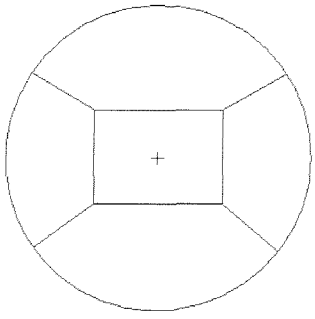


Figure 3-6 O-type edge design on a cross-section of a pipe.

To start off a number of rules were assembled, and these are given in Table 3-1. First, a face grid was produced using just four edges by considering one face at a time. A multi-block design for a three dimensional bifurcation and two dimensional bifurcation are shown in Figure 3-7 and Figure 3-8 respectively. The complete block design in two dimensions is shown in Figure 3-9, and a sample grid at the first branch of two dimensional bifurcation is shown in Figure 3-10 .

Table 3-1 Some rules to consider in a multi-block design

Rule No.	Multi-block design rules
1	Consider a face not a block when designing edges.
2	The grid cells must be small perpendicular to the wall boundary. This suggests that the edges must be perpendicular to the surfaces of the geometry.
3	Two opposing edges of a face must lie on a plane with minimum degree of variation. Otherwise the cell face could have two normal vectors from adjoining cells. Such a case could lead to flux errors in the computation.
4	The angle between pair of adjacent edges of a face must lie within some tolerance angle limit say >= 60° and <= 120° (limit set by the O-Type grid).
5	Inspect the mathematical description of the geometry to create more edges (curves) on the surface of the geometry as needed by the multi-block design and for better surface grid.
6	Note that solver may only handle certain block arrangements due to the way it has been designed to communicate information across block boundaries.

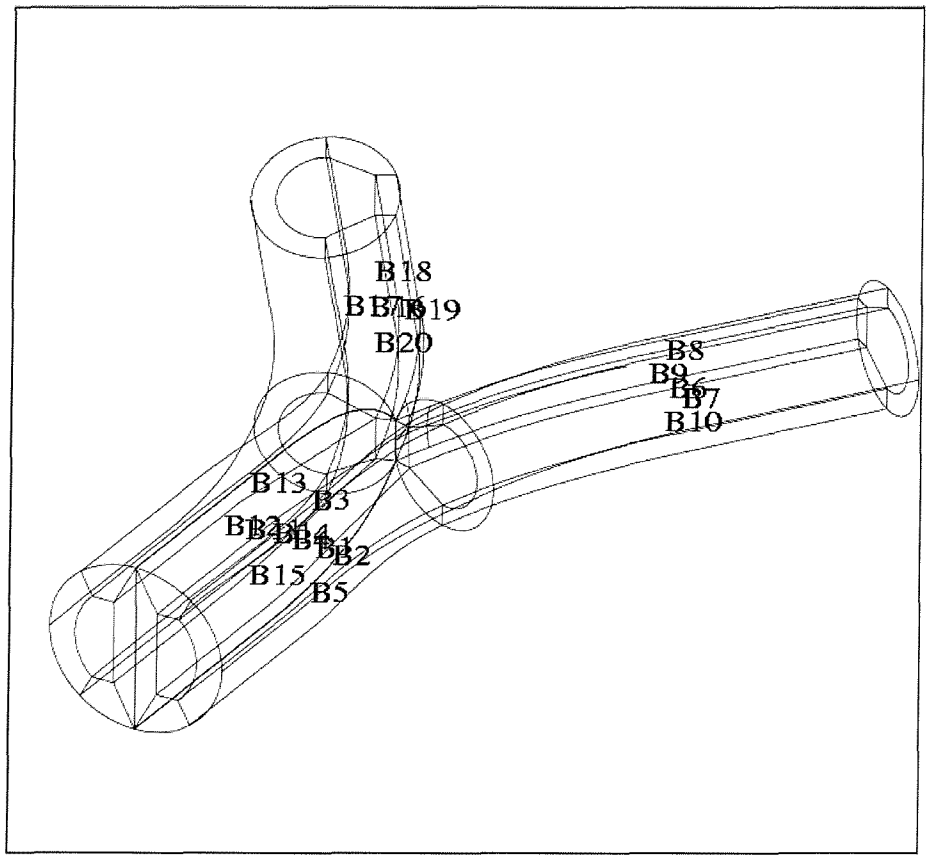


Figure 3-7 Multi-block design of the three-dimensional bifurcation

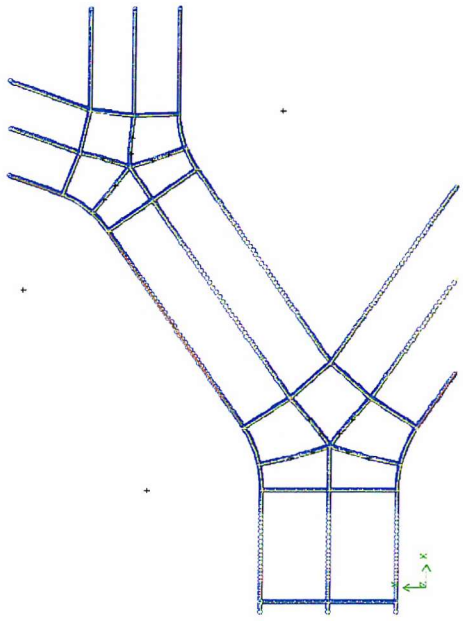


Figure 3-8 Multi-block design of two dimensional double bifurcation facing up

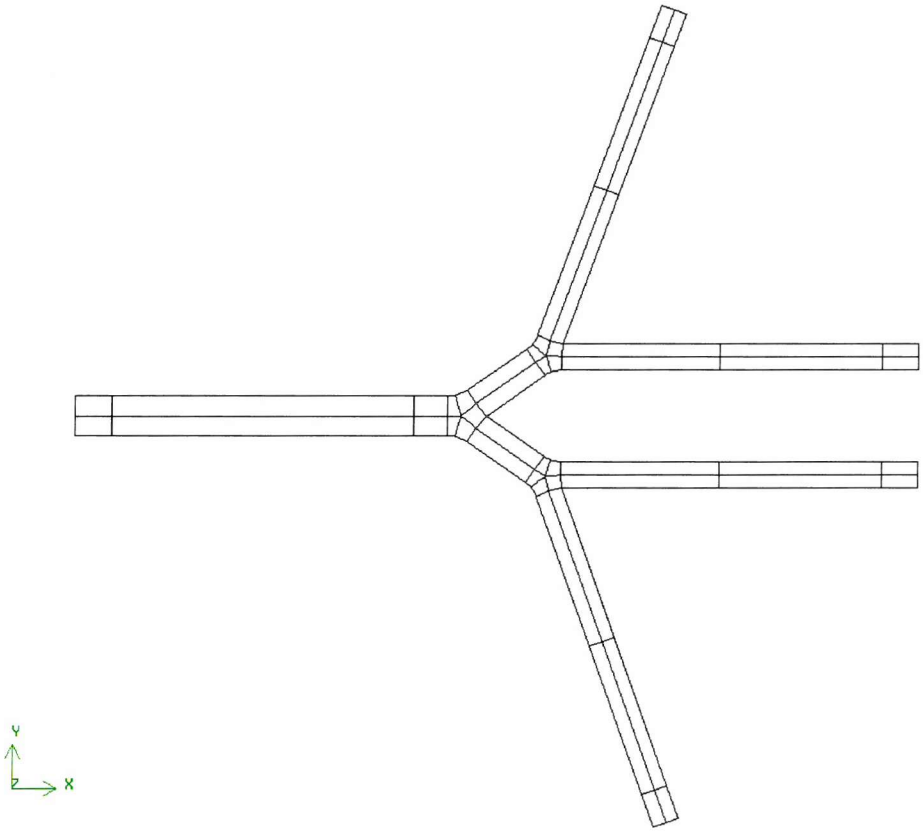


Figure 3-9 The complete two dimensional block design

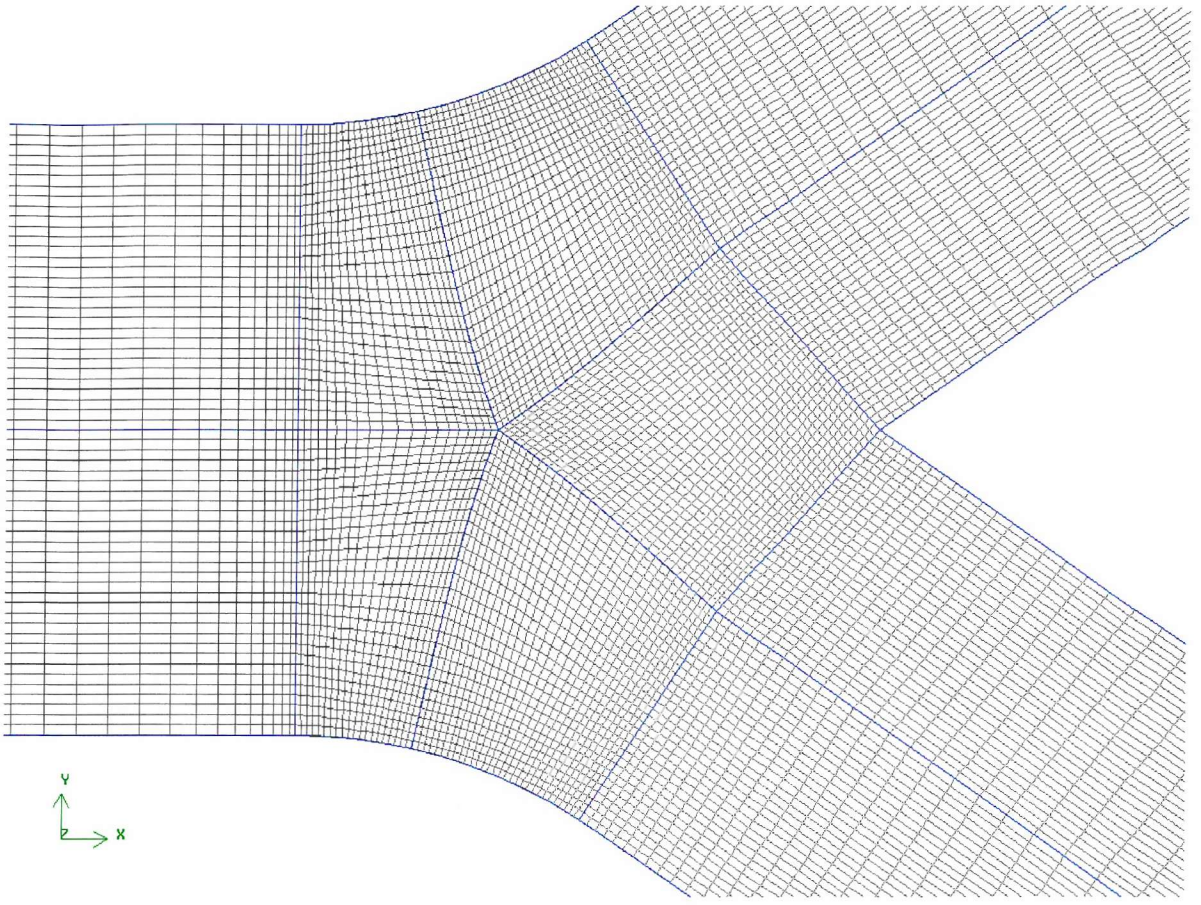


Figure 3-10 The structured grid at the first bifurcation region of the two dimensional geometry

3.4.1 Controlling the grid

The structured grid will be generated in each block in isolation and connected with adjacent blocks. There were some difficulties experienced with Gambit due to lack of smoothness in the grid at block boundaries. Ideally the global grid can be smoothed rather than smoothing the grid on one block at a time. The need to control the grid comes with the requirements to produce a smoother solution. After the block design a grid may be manipulated by nodal distribution on the edges and using smoothing functions to smooth the propagation of the grid cells towards the interior of the domain, as was done in Figure 3-10.

3.5 Conclusion

It is essential that the geometrical modelling and grid generation are done by the same package or that the grid generator must have links to a CAD package, in which case the grid generator could obtain the geometrical attributes of the model. It is always useful to have a mathematical description of the geometry and as much detail of the geometry as possible, especially about the surface geometry. A grid generator must be able to generate grids on complex surfaces, i.e. must have an elliptic or hyperbolic generation scheme. The user may need to add more edges to reduce unnatural variation in the surface grid produced.

It is not always possible to produce say, cells with angle range of 60-120° between adjacent cell edges. Hence the solver may need to be improved to tackle very skewed cells. It is expected that the computational effort of the solver-grid interactions and the effort made on the grid quality must be in balance to create an efficient way of producing a grid on bifurcation geometries. In this case an unstructured grid may be the most efficient, and which can provide a faster turnaround time.

Considering the time taken by the three dimensional multi-block design, using it to validate the simulated flow, and then designing a three dimensional particle tracking routine, a two-dimensional grid on the 5^{th} to 7^{th} generate branch from Weibel model 'A' was chosen. It is expected that once two dimensional particle tracking is tested, it can be extended to track particles in three dimensions. It was decided to use a two-dimensional grid to compute the flow field variables, and validate this result using grid independence.

If more than one bifurcation is to be used to resolve the complex flow structures, then a parallel CFD solver may be the most efficient way to proceed. For a parallel solver good interpolation routines and communication routines may have to be developed at block boundaries. Higher order solution technique may be essential to interpolate flow variables across block boundaries, since the flux vector from a cell face at a block boundary may not be in parallel to that of the adjacent cell face flux vector. In parallel CFD a multi-block design must be created using one additional constraint. Ideally the multi-blocks in parallel would have equal number of grid control volumes to make parallel processing efficient.

3.6 References

Balashazy, I. and W. Hofmann (1993). "Particle deposition in airway bifurcations 1: Inspiratory flow." Journal of Aerosol Science **24**(6): 745-772.

Fluent (1998). Pre-processor Gambit, and solver Fluent 5.5.

Heistracher, T. and W. Hofmann (1995). "Physiologically realistic models of bronchial airway bifurcations." Journal of Aerosol Science **26**(3): 497-509.

Horsfield, K., G. Dart, et al. (1971). "Models of the human bronchial tree." Journal of Applied Physiology **31**(2).

Thompson, J. F., Z. U. A. Warsi, et al. (1985). Numerical Grid Generation: Foundation and Apllication.

Weibel, E. R. (1963). Morphometry of the human lung, Springer-Verlag, Berling Gohingen-Heidelberg.

Zhao, Y. and B. B. Lieber (1994). "Steady inspiratory flow in a model symmetric bifurcation." Journal of Biomechanical Engineering, (TASME) **116**: 488-496.

4 CFD results

4.1 Introduction

The physical laws of conservation of mass and momentum were used to define the flow problem in chapter two. The steady Navier Stokes equations are a coupled system of non-linear elliptic equations. The details of the overall process used to solve the non-linear partial differential equations using the Finite Volume Method (FVM) is explained in section 4.2. The fluxes in the integral terms of the governing equations are discretised over the two-dimensional 5^{th} to 7^{th} generation branches of the airways, as meshed in chapter three. In the past the flow development within each cross-sectional change of the airways was investigated separately, experimentally and theoretically. However in recent years researchers succeeded in calculating the complex flow behaviours within single and double bifurcation airways using numerical techniques. Some key flow characteristics with airway branch geometry is described in section 4.4. The numerical accuracy of the solution is assessed in section 4.3.

4.2 Numerical Solution process

The Navier Stokes equations, Equation 2-6 and Equation 2-7, written in general form, appropriate for the application of the finite volume method, is given in Equation 4-1.

Equation 4-1

$$\frac{\partial U}{\partial t}+\frac{\partial f}{\partial x}+\frac{\partial g}{\partial y}=q$$

U is the solution vector, flux vectors **f** and **g** contain column flux expressions, and **q** contains possible source terms (in this case **q**=0). The finite volume method takes the volume integral of Equation 4-1, and converts it to surface flux integral form using Gauss Theorem. By integrating over the cell, local conservation of the variable is maintained and summing each integral equation for all cells in the domain, global conservation is maintained. In two dimensions, the FVM produces Equation 4-2, where Δx and Δy are the grid spacing in x and y directions respectively in a Cartesian coordinate system. The fluxes are evaluated at cell surfaces located at (i +1/2,j), (i -1/2,j), (i ,j+1/2), and (i ,j-1/2), where (i, j) indices apply to the centroid of the cell.

Equation 4-2

$$\frac{\partial \overline{U}}{\partial t} = -\frac{f_{i+\frac{1}{2},j} - f_{i-\frac{1}{2},j}}{\Delta x} - \frac{g_{i,j+\frac{1}{2}} - g_{i,j-\frac{1}{2}}}{\Delta y} + \overline{q}$$

The semi-discretised Equation 4-2 is a single equation representing semi-discrete versions of the continuity, x-momentum, and y-momentum equations. The right hand side fluxes are approximated by various schemes described in next sub-sections. In Fluent fluxes are evaluated to second order accuracy.

4.2.1 Evaluating the fluxes

FLUENT uses a co-located scheme, which stores both velocity and pressure at the cell centre. The cell face values of the dependent variables are used to evaluate the convective fluxes. The second order accurate upwind scheme called QUICK (Quadratic Upstream Interpolation for Convective Kinetics) (Leonard 1979) scheme is used for this purpose. In this scheme the variable at the desired cell face is obtained using three-point upstream-weighted quadratic interpolation. Fluent uses a procedure similar to Rhie and Chow (Rhie and Chow 1983) to interpolate the cell center velocity to the cell faces. This procedure meant to prevent the well-known checkerboard effect of pressure when cell center velocity is interpolated at cell faces using linear interpolation.

The pressure is also required at cell faces and an interpolation scheme is required to interpolate cell centre pressure values to cell face pressures. A scheme similar to Power Law (Patankar 1980) called PRESTO! (PREssure STaggering Option) is used. For quadrilateral meshes with curved boundaries FLUENT recommends this scheme (Fluent Inc. 1998). For the laminar bifurcation flow problem (flow problem in complex geometry), significant pressure gradients are expected, especially around the apex (stagnation region) of the bifurcation. Hence a high order pressure interpolation in the solution scheme is a necessity to minimize cell velocity overshoot and undershoot. To assist in resolving the pressure gradient in the apex region of the bifurcation a dense grid will be created using the multi-block grid generation and non-uniform grid nodal placement.

The diffusion terms are evaluated from cell centred values using central differencing to second order accuracy.

4.2.2 Solving the pressure-velocity coupling

The segregated solution method solves the momentum and Continuity equations sequentially. One of the families of the SIMPLE (Semi-implicit Method for Pressure-

Linked Equations), namely SIMPLEC (SIMPLE-Consistent) is used to evaluate the pressure-velocity coupling. First the momentum equation is solved using guessed value of pressure. Since this may not lead to correct value of flux through the cell faces to satisfy mass continuity a correction to the face flow rate is added. The corrected face flow rate is such that it forces satisfaction of the continuity equation. The corrected flow rate equation contains cell pressure correcting terms. SIMPLEC substitutes the flux correction equation in the continuity equation to obtain a discrete equation for pressure correction.

The pressure-correction equation is solved to update the pressure and face mass flow rate. A convergence criterion is then tested and if not satisfied above process is repeated with the updated values of dependent variables.

4.2.3 Solving the discrete system of equations

In the segregated method the discretised equations can be linearised either implicitly or explicitly. The updated solution is obtained when the linear system of equations are solved. The segregated solution method of FLUENT uses implicit linearisation with respect to that equation's dependent variable. Since the solution method is segregated only one equation for each cell (one equation for each dependent variable for each cell) is obtained. A point implicit (Gauss-Seidel) method with algebraic multigrid (AMG) method is used to solve the linear system of equations for the dependent variable in each cell. The AMG method with w-cycle is used to accelerate the rate of convergence, where the solution is interpolated to a sequence of coarser grids in an effort to eliminate low frequency errors. The higher frequency errors were treated by iterating two times after each w-cycle. Hence the segregated approach solves for each dependent variable in turn by considering all cells at the same time.

4.3 Computational runs

The residuals of the dependent variables could oscillate during the iteration process and could lead to instability and slower convergence. Hence there is a need to control the change in the dependent variables. This is achieved by under-relaxing the change in dependent variable during the iteration process. Under-relaxation factors of 0.7 for momentum and 0.3 for pressure are used.

Numerical schemes are tested for their usefulness by calculating some criterions for grid-convergence on the resulting flow field, and the stability of the solution process. The iteration process must be stable such that after successive iterations the solution residual

will eventually decrease, with the oscillations in the residual minimised for faster convergence. The grid convergence results are explained in section 4.3.1.

CFD results were produced for the 5^{th} to 7^{th} generation airway bifurcation model. Three meshes of increasing grid density were used to carry out a grid independence study. The meshes each with increasing grid cells in two orders of magnitude are labelled as g1, g2, and g3. The number of cells on each edge of the geometry is shown in Figure 4-1, where the top number to bottom is for the finest to coarsest grid respectively, i.e. in the order of g1, g2, and g3.

As a test case the u-velocity, v-velocity, and pressure were initialised to 1, 0.5 and 1 respectively on each grid. The discretisation schemes and the solution method used to solve the linear set of equations were given in section 4.2. The residual histories for g1, g2, and g3 are given in Figure 4-2, Figure 4-3, and Figure 4-4 respectively for up to 5000 global iterations. For the validation study explained in the next section more than 5000 iterations were used for grid labels g2 and g3. In fact the iterations were carries out until the residual change from one iteration to next reached a minimum and produced a "flat line" as shown in Figure 4-2 for coarsest grid, i.e. until the discretisation error limit is reached. The grid convergence occurs when the residual drops three to four order of magnitude less than the starting value. For a particular grid the residual magnitude at the "flat line" indicates the best possible numerical accuracy. It may be inferred from Figure 4-2 and Figure 4-3 that the discretisation error decreases as the number of cells within the domain increases, i.e. the flat line will occur at lower residual magnitude and at higher global iteration numbers. The time taken by each run span from 2 hours to up to 13 hours for the coarsest to the finest grid respectively on a 700MHz Pentium III PC.

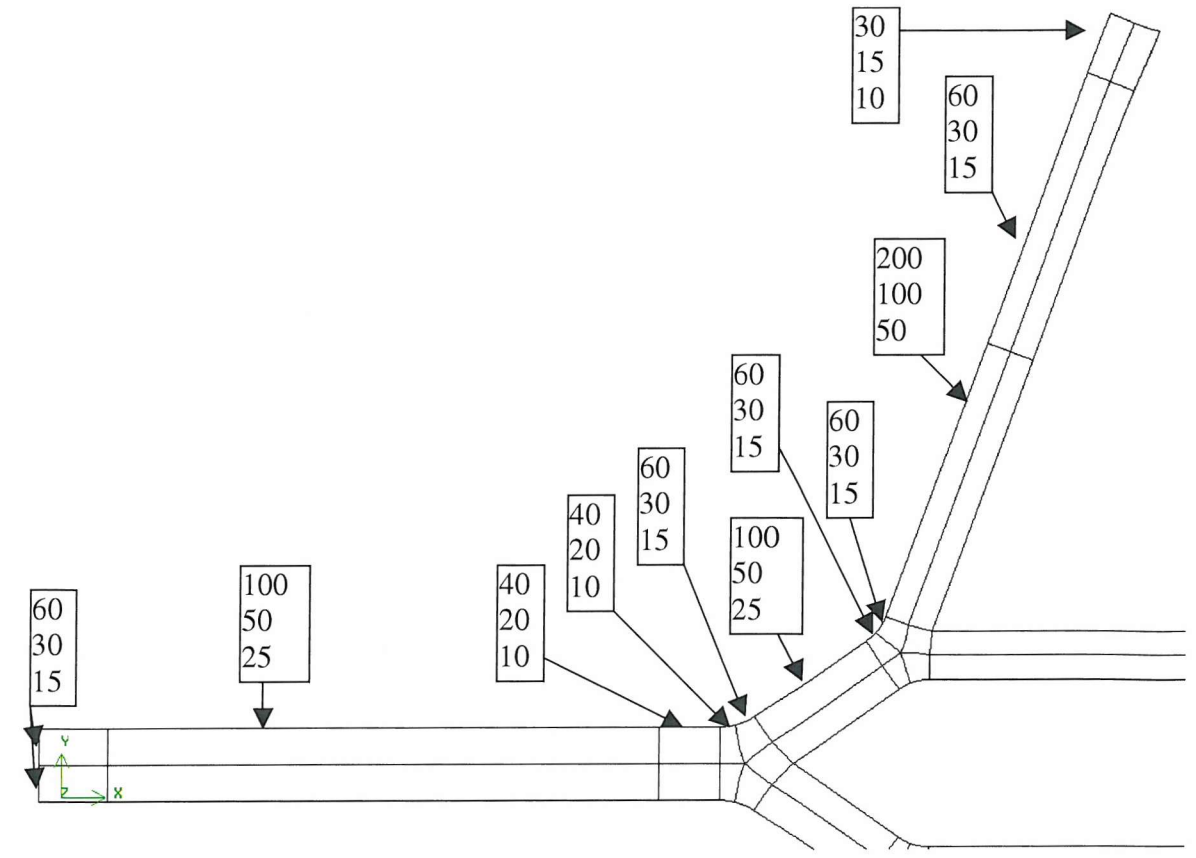


Figure 4-1 Cell numbers per edge for three grid sizes

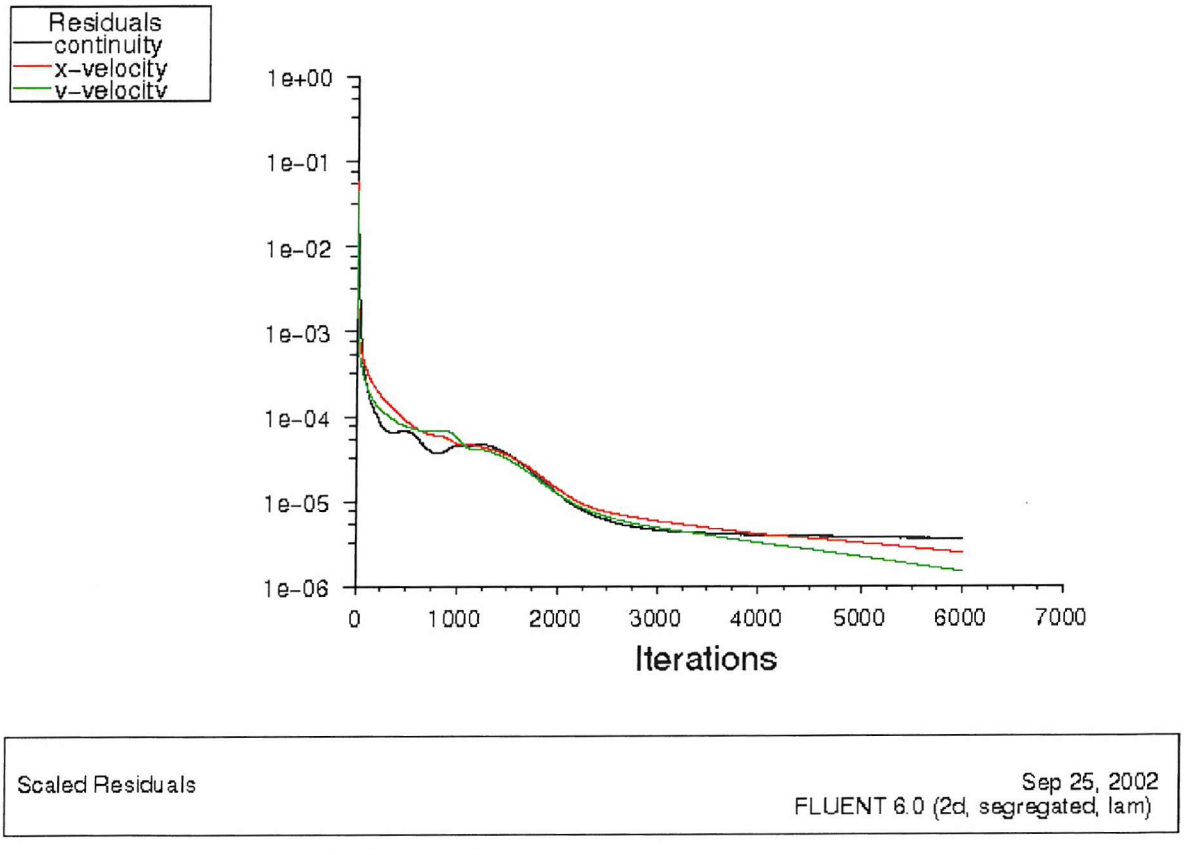


Figure 4-2 Residual plot for the finest grid, g1

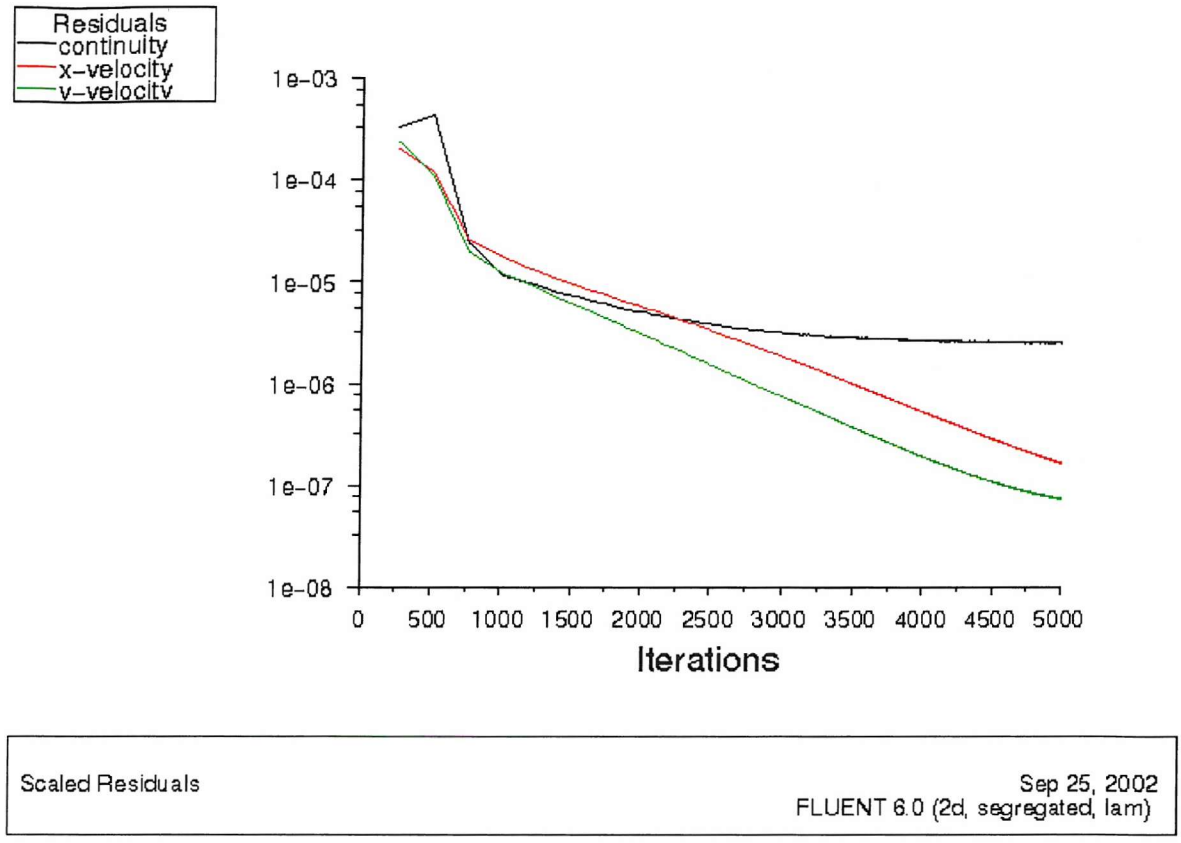


Figure 4-3 Residual plot for the medium size grid, g2

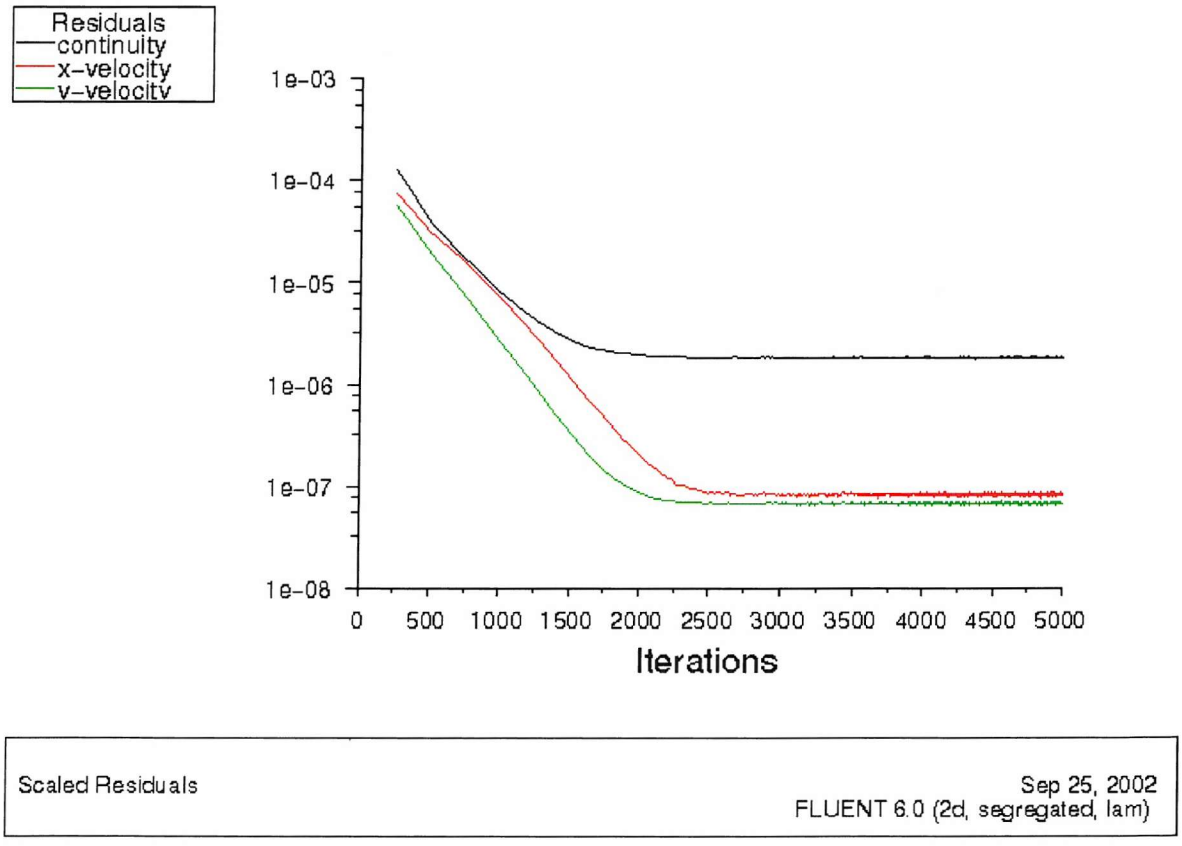


Figure 4-4 Residual plot for the coarsest grid, g3

4.3.1 Validation

In order to find an accurate numerical solution a grid independent solution is required. Usually a given grid size is halved and doubled to find a significant variation in solution. Two different studies were conducted, where in one case the fluid speed is calculated at a set of cross-sectional lines for each mesh and the other case the wall pressure is calculated for each mesh. The entrance velocity profile at a Reynolds number of 600 is parabolic and all outlet static pressures are set to 1 Pascal.

Fluid velocity is extracted using the Fluent post-processor on twenty points along each line segment. The line segments used are shown in Figure 4-5. The magnitude of velocity is calculated on each line of different meshes and they are shown in Figure 4-7 to Figure 4-19 (pages 48 to 54). The values for these figures were extracted from Fluent post-processor. Up to 20 points were allocated for each line. The end data points do not necessarily fall on the boundary of the domain, hence in some figures curves are incomplete. However the velocity profiles are clearly defined in all plots. For Figure 4-16 to Figure 4-19 perhaps more points should have been selected for data extraction.

In all plots the y-axis is normalised with respect to the inlet maximum velocity of $2.6218 ms^{-1}$, which corresponds to a Reynolds number of 600 (based on maximum inlet velocity and inlet diameter of 0.0035m). The x-axis is normalised with respect to the line perimeter length, where apart from line number 9, 10 and 12 x-axis magnitude of 1 means the top end of the line, and 0 or –1 means bottom end of the line shown in Figure 4-5. For the line numbers 9, 10, and 12 it is the other way around, where x-axis magnitude of 1 is at the outside wall and zero is within the geometry or at the inside wall.

It can be seen that parabolic inflow come on to the first bifurcation, and on lines 1,2 and 3 (Figure 4-7 to Figure 4-9), the flow does not change by a noticeable amount. In all other lines grid g3 shows noticeable deviation from g1 and g2. However for lines 1 to 6 g2 data are closer to g1 than to g3, and for lines 7-12 g2 is closer to g1 than to g3. This means the resolution at the second bifurcation needs to be increased a little so that data from g2 can be used as the grid independent results. Magnitude analysis of the difference in values for g2 and g3 were obtained by approximating each curve by a fourth order polynomial and calculating the mean of the difference between the two grids at some set of points. In general the difference is less than 2% of the entry velocity.

The second numerical validation case is to check the wall pressure values from different grids. The walls are labelled as in Figure 4-6. The pressure along these walls was extracted.

The static wall pressures for walls 1 to 5 for different grids are shown in Figure 4-20 to Figure 4-24, and a close up of wall 1 at the first bifurcation is shown in Figure 4-25 (pages 54 to 57). The red dots of g2 are in between black dots of g3 and blue dots of g1, but data from g3 does show the increased grid resolution on the finest grid. Hence g2 can be considered as a suitable grid to produce a good solution at a reasonable cost.

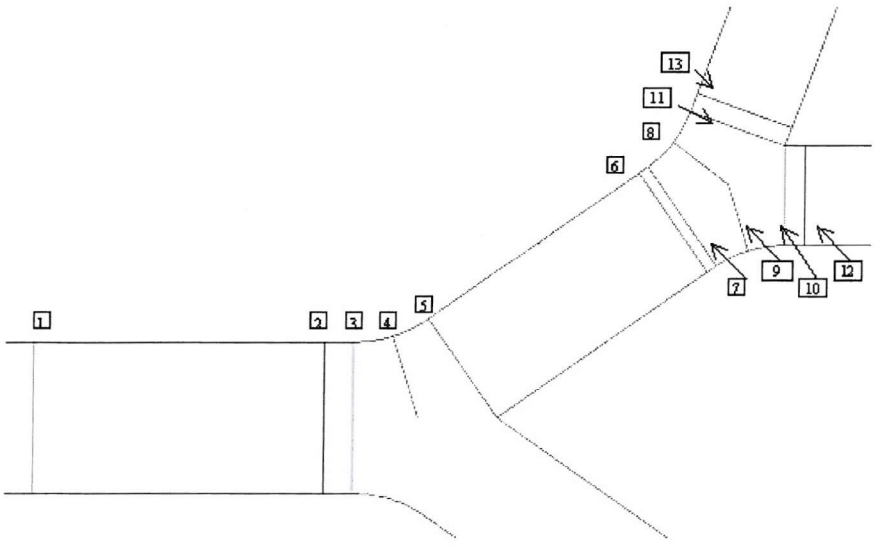


Figure 4-5 Cross-sectional velocity data extraction lines for grid independence.

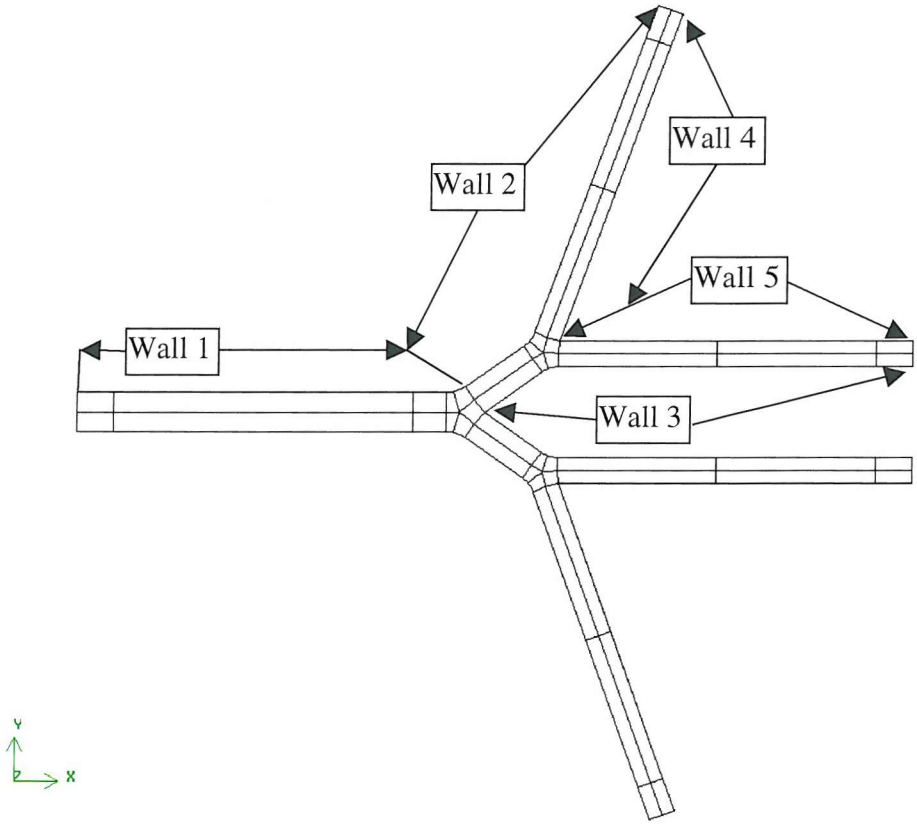


Figure 4-6 Wall labels use to extract static pressures

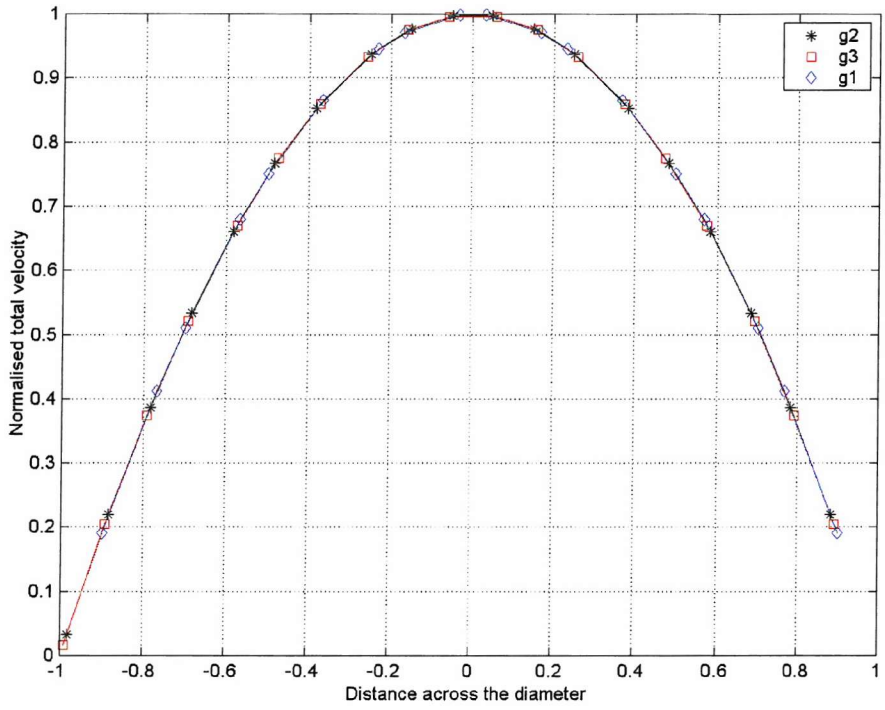


Figure 4-7 Magnitude of velocity on line-1

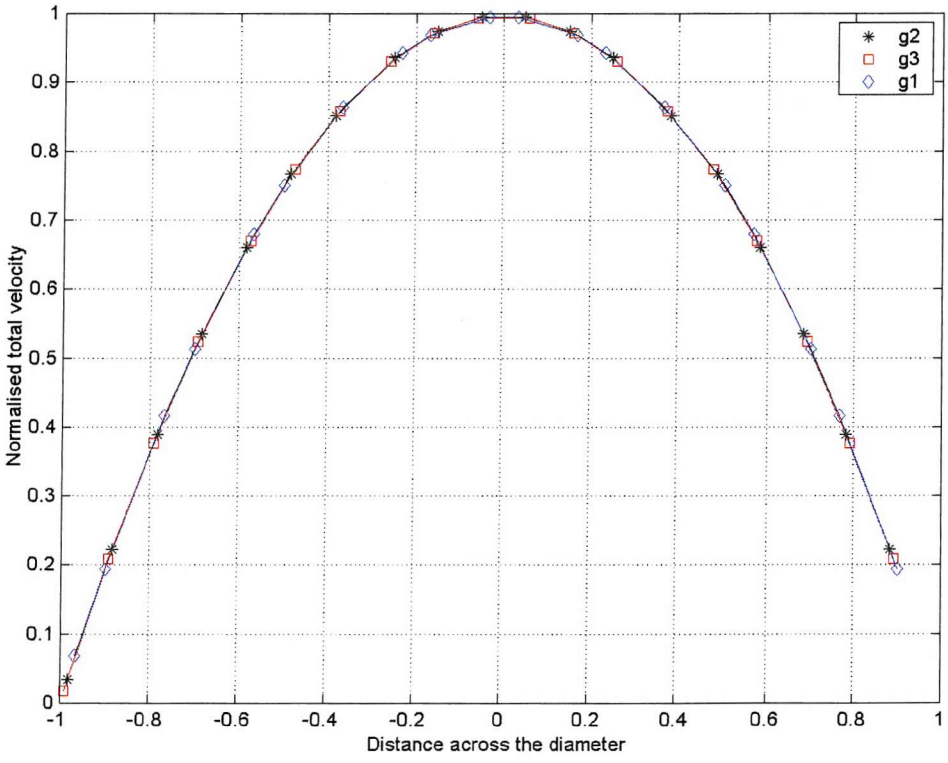


Figure 4-8 Magnitude of velocity on line-2

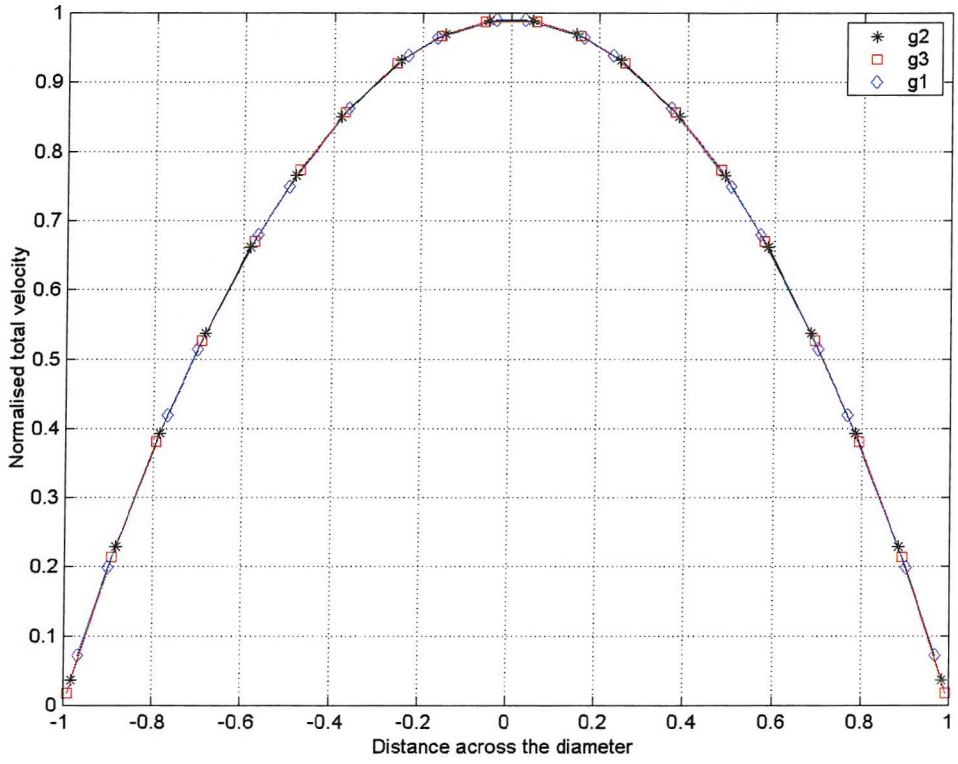


Figure 4-9 Magnitude of velocity on line-3

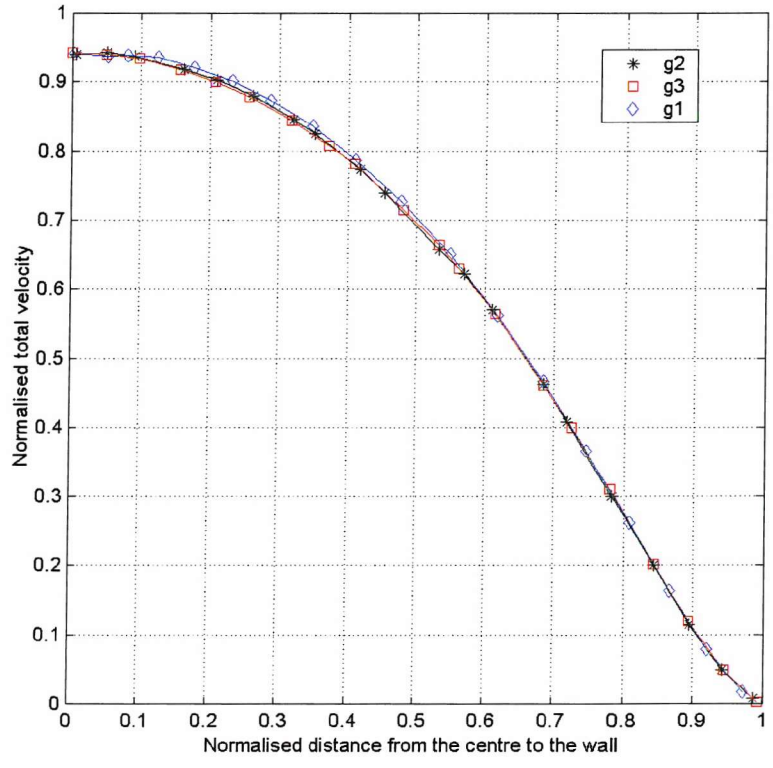


Figure 4-10 Magnitude of velocity on line-4

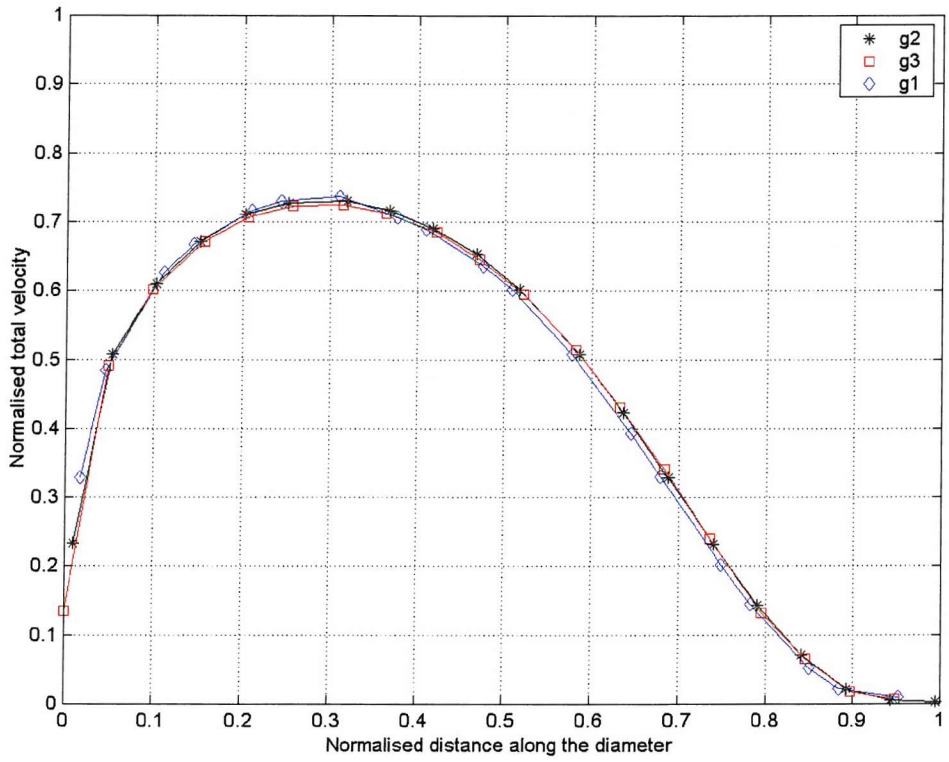


Figure 4-11 Magnitude of velocity on line-5

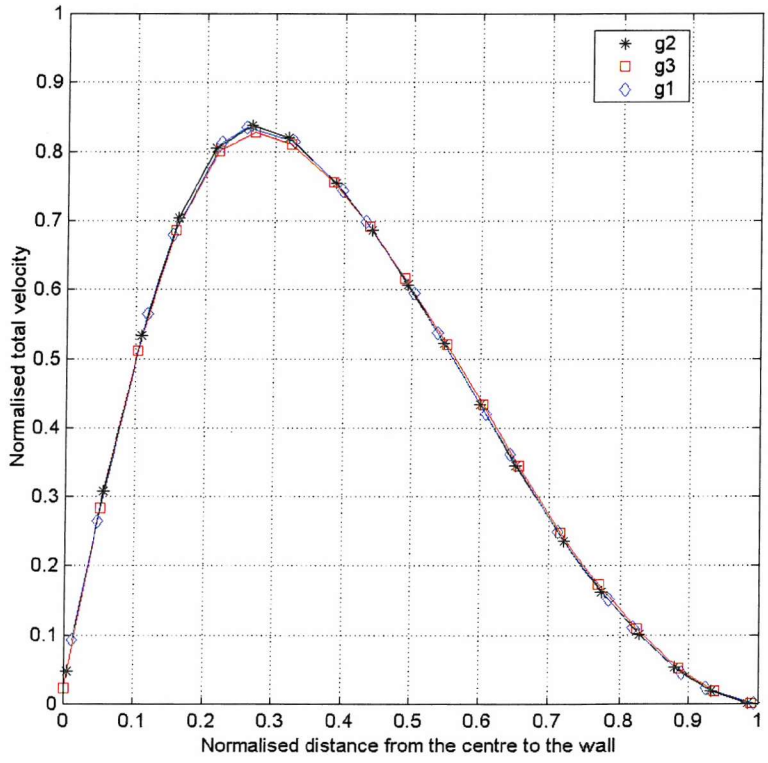


Figure 4-12 Magnitude of velocity on line-6

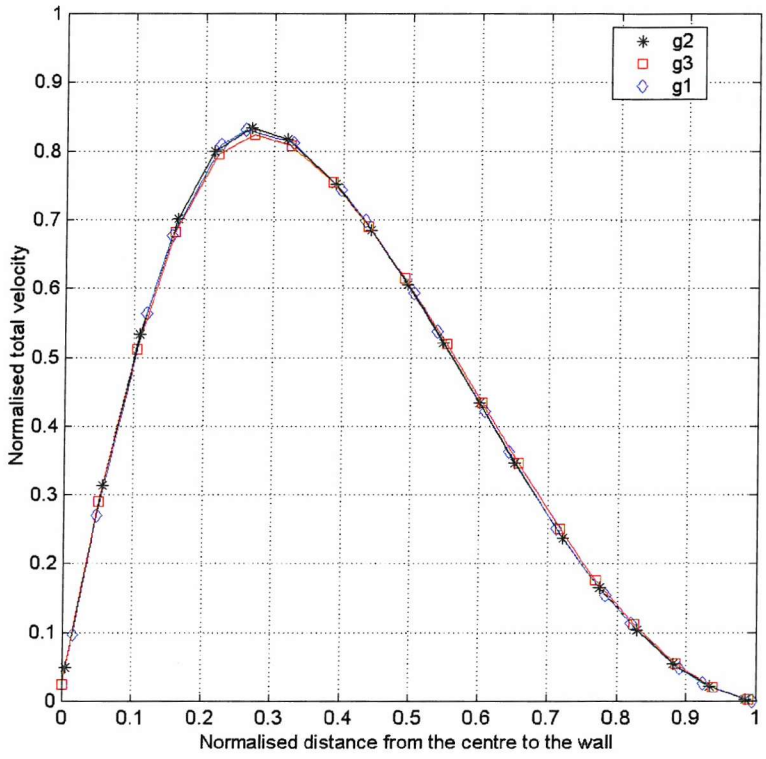


Figure 4-13 Magnitude of velocity on line-7

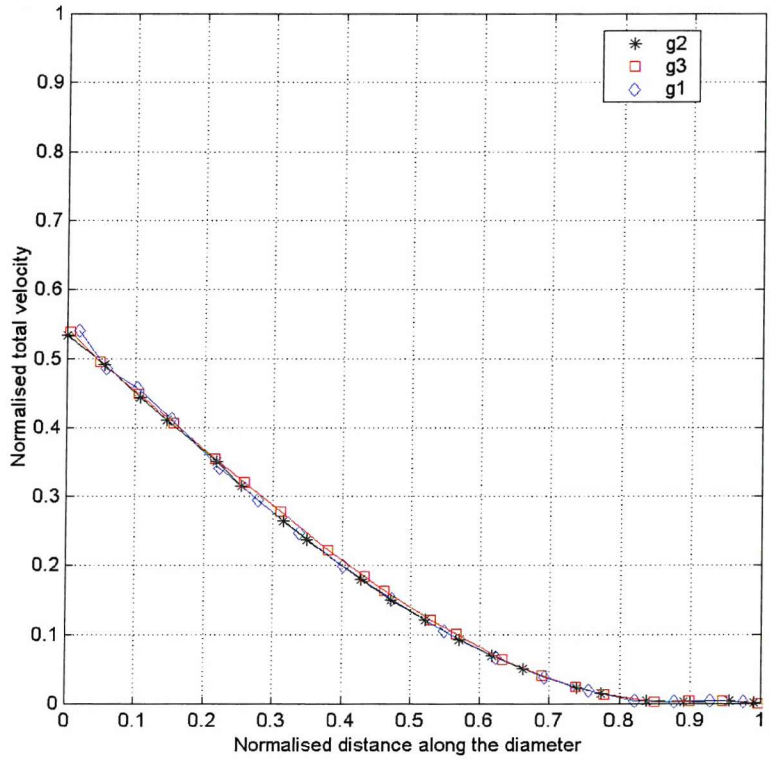


Figure 4-14 Magnitude of velocity on line-8

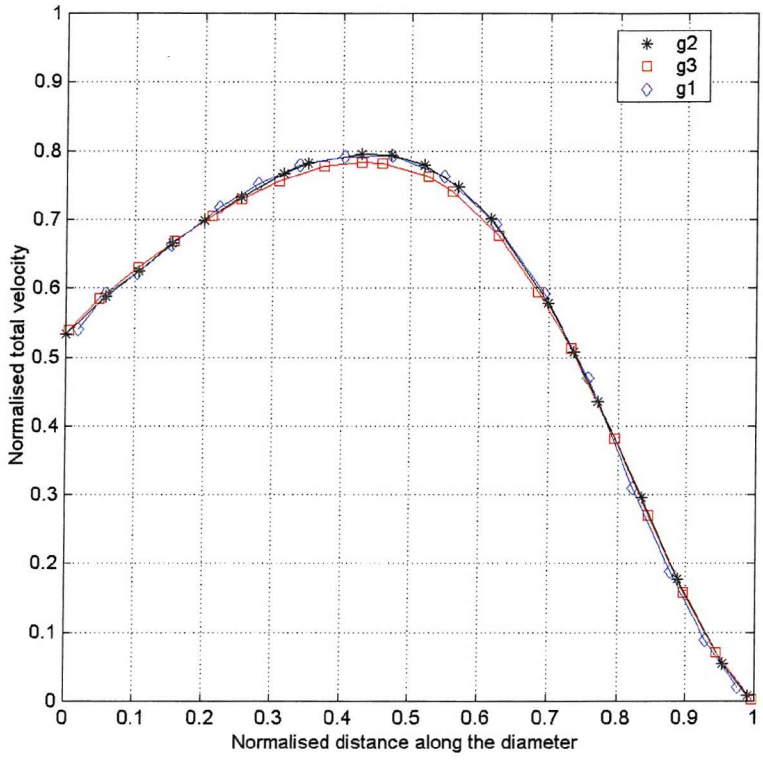


Figure 4-15 Magnitude of velocity on line-9

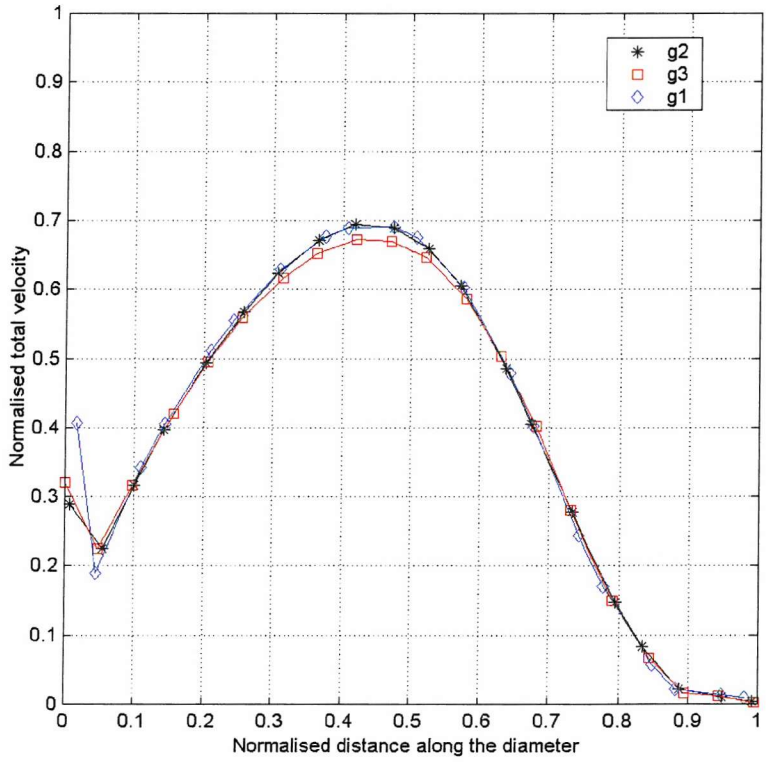


Figure 4-16 Magnitude of velocity on line-10

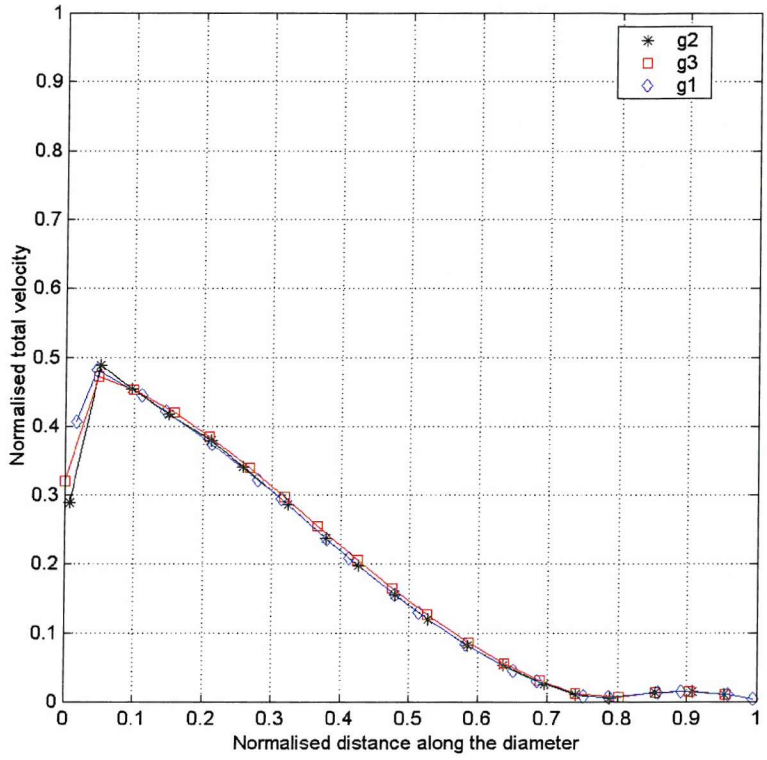


Figure 4-17 Magnitude of velocity on line-11

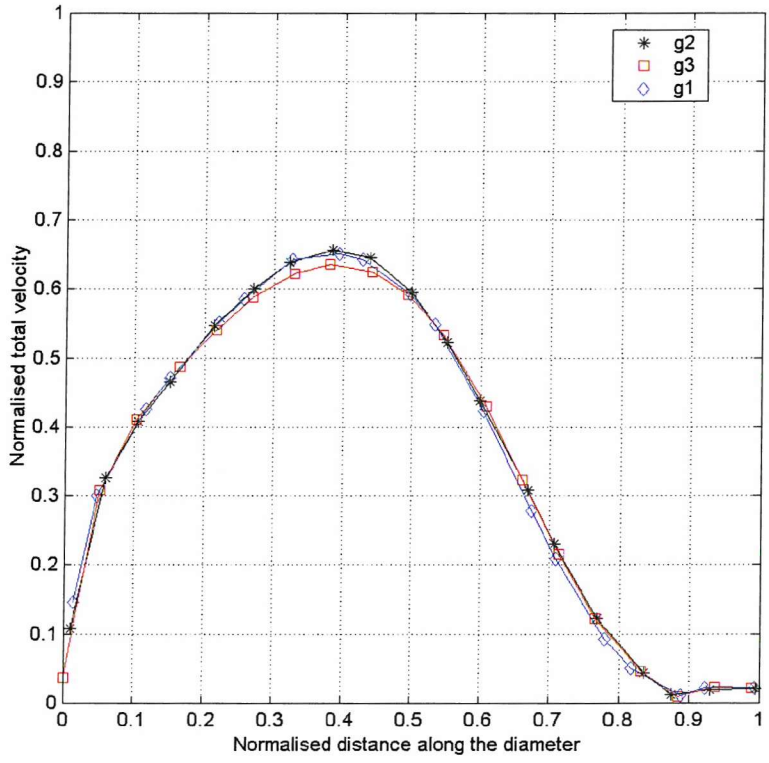


Figure 4-18 Magnitude of velocity on line-12

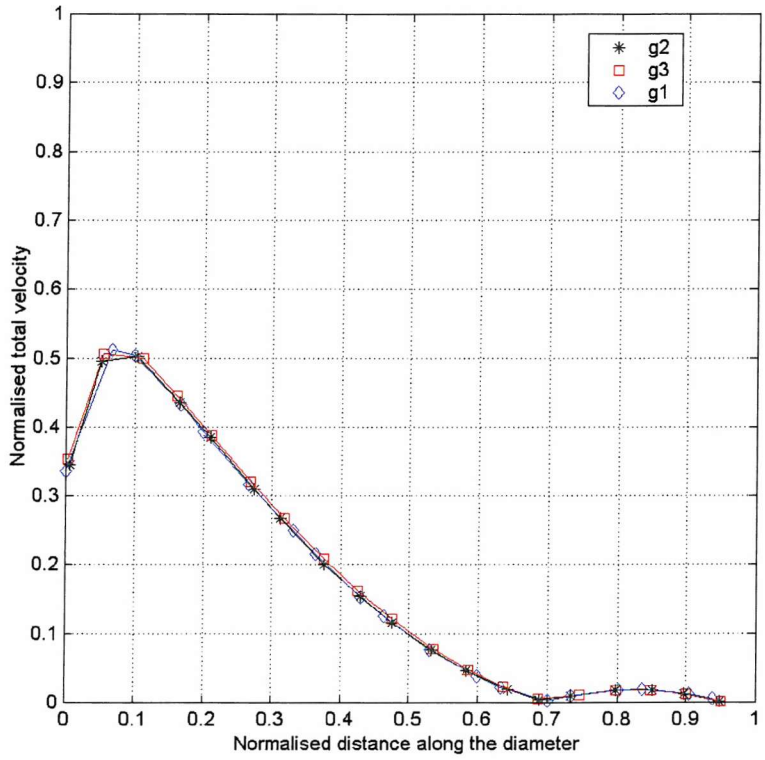


Figure 4-19 Magnitude of velocity on line-13

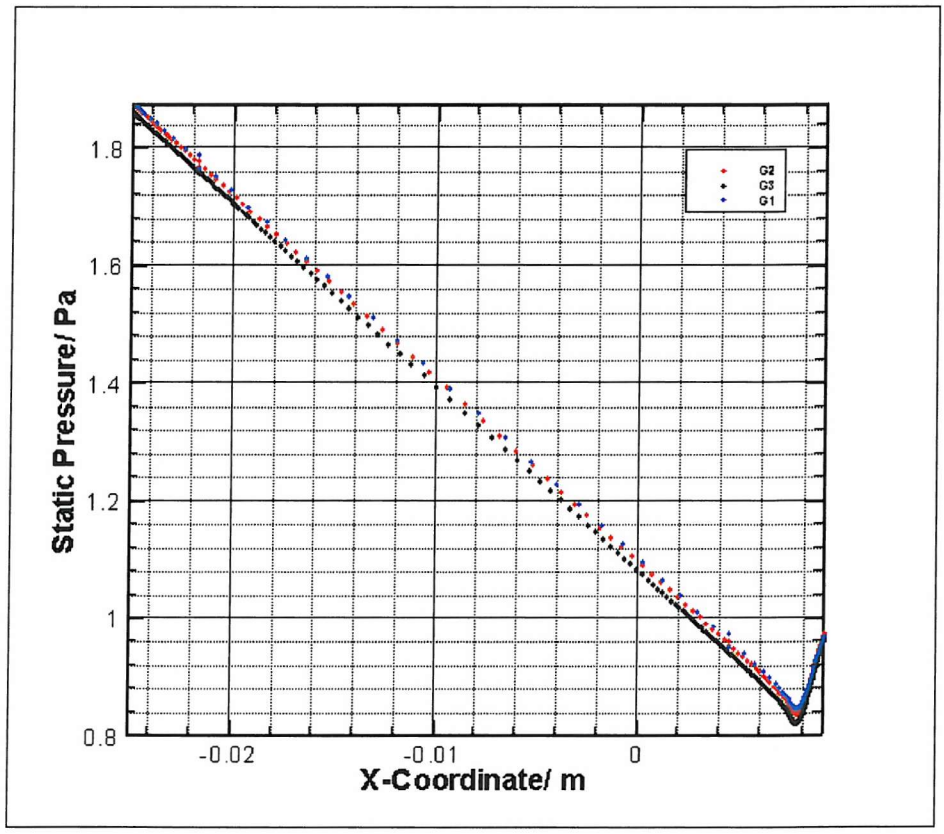


Figure 4-20 Static pressure along wall 1

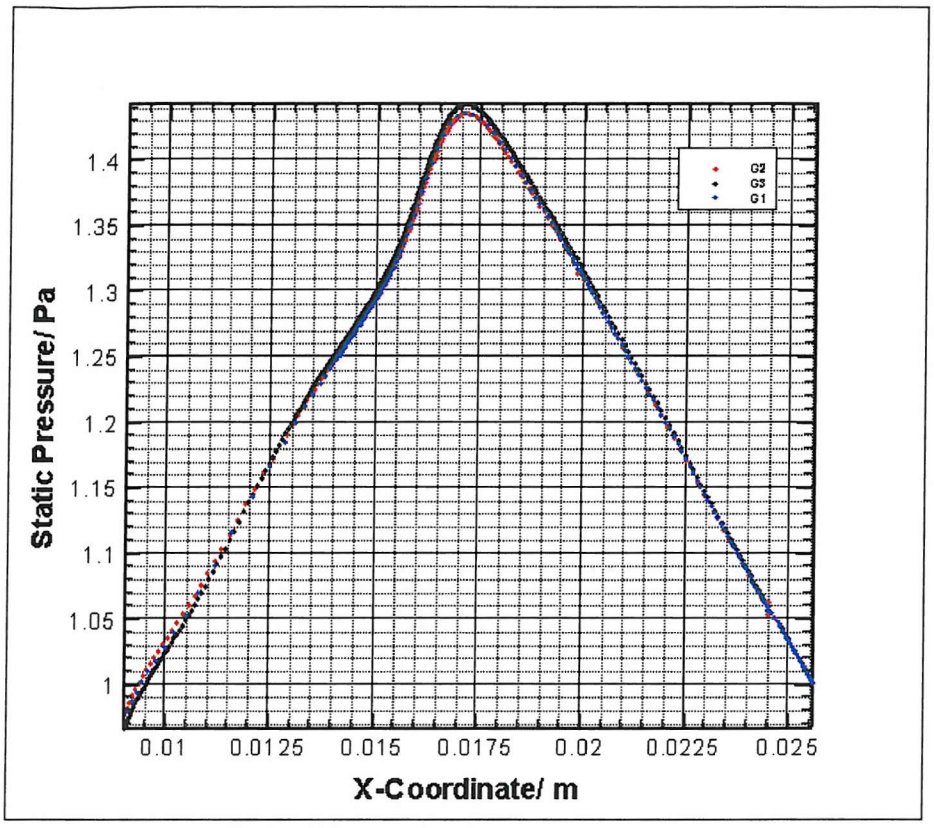


Figure 4-21 Static pressure along wall 2

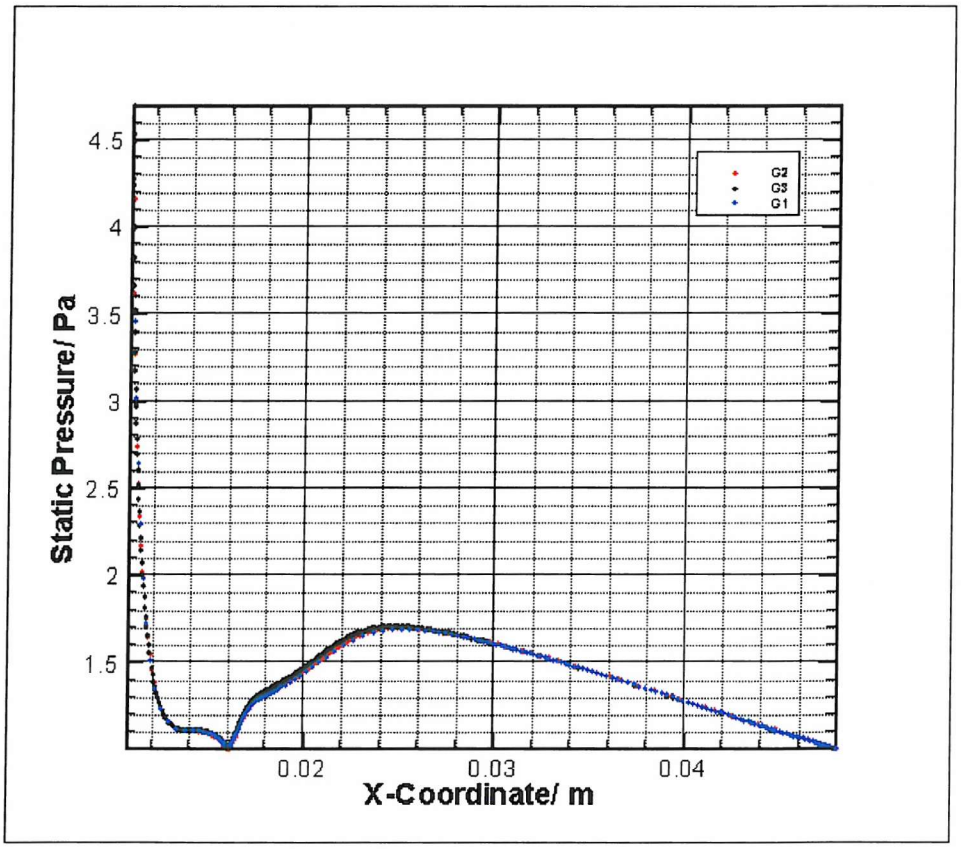


Figure 4-22 Static pressure along wall 3

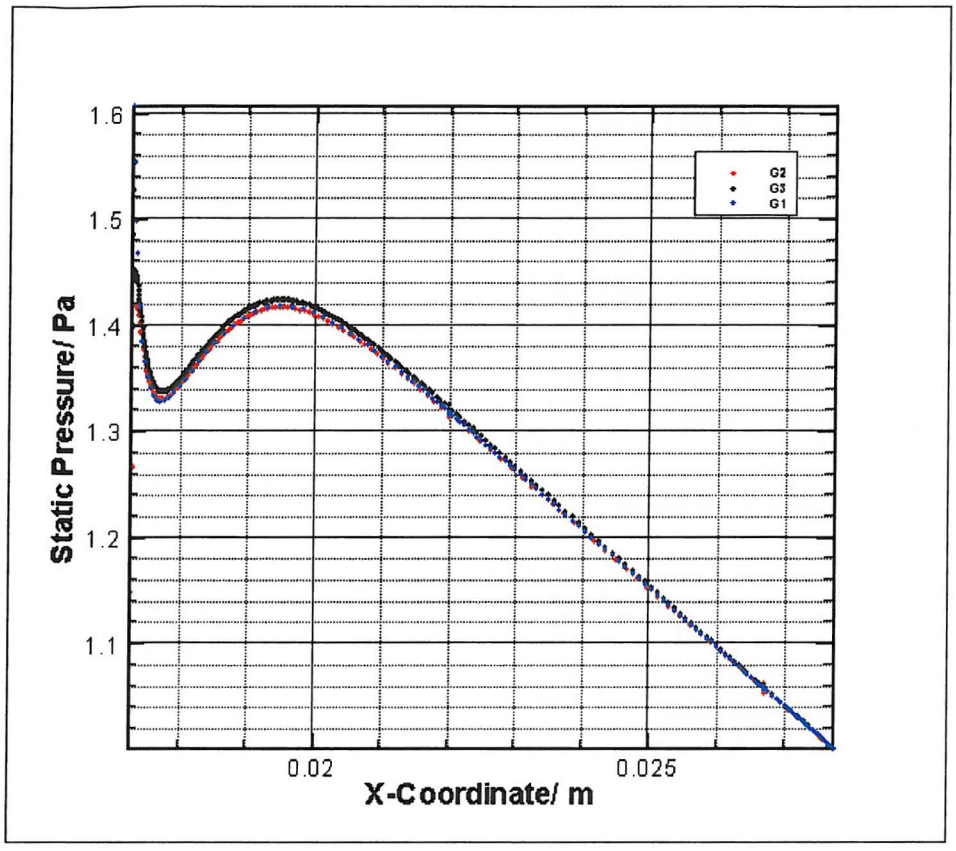


Figure 4-23 Static pressure along wall 4

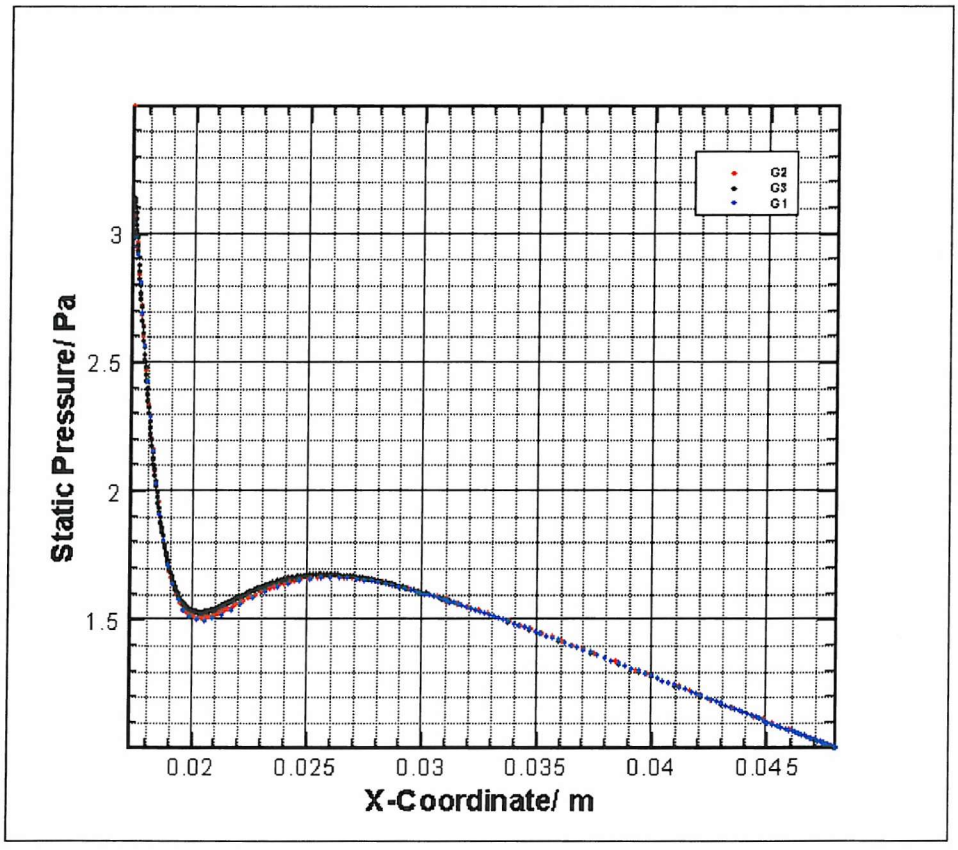


Figure 4-24 Static pressure along wall 5

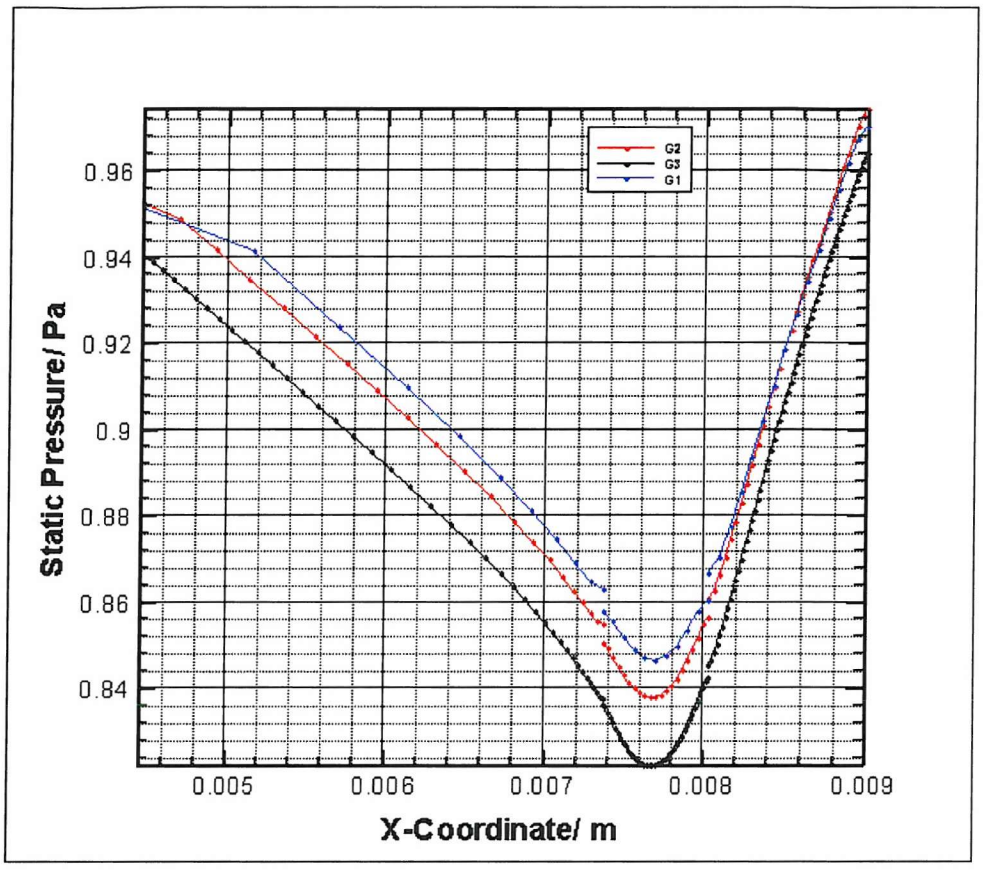


Figure 4-25 Close up at wall 1, outer wall of first bifurcation

4.4 General flow features within bifurcations

The data for run g2 is used for the following flow analysis (Note, since different outlet pressures will be selected it was thought best, at the time, to use full two-dimensional model without symmetry plane at the first parent branch). The bifurcation geometry was separated into several geometrical sections, where on each section certain flow changes take place (explained in section 4.4.1 to section 4.4.4). The entrance velocity profile was set to a parabolic profile but with undefined pressure. The outlet static pressure on the outermost branch is progressively reduced from 1 Pascal to 0 and the innermost branch outlet pressure was fixed at 1 Pascal. When the solution is converged the pressure at the inlet and the outlet would give the pressure drop across the system. The Reynolds number of the flow at the inlet was fixed to 600 based on maximum inlet velocity. The main results are shown in Table 4-1, and the velocity and static contour plots for different outer daughter outlet conditions are shown in Figure 4-29 to Figure 4-40 (pages 63 and 68). The details of the figures will be explained later.

The outlet mass flow rates due to variation in pressure in the model are given in Table 4-1, and in Figure 4-27. It can be seen that the mass flow rate varies linearly with the outlet pressure changes and equal mass flows in both outlets when the outermost daughter outlet

pressure is set to 0.5. However the pressure difference between the inlet and the outlets does not vary linearly when one outlet pressure is altered linearly, which can be seen in Figure 4-26. In Figure 4-26 the static pressure at outermost branch and the inner branch was subtracted from the inlet pressure to provide the pressure loss information. Two out of four outlets were taken since the double bifurcation is symmetrical and, for this case, the pressure was varied only in one outer daughter branch with respect the corresponding inner daughter branch.

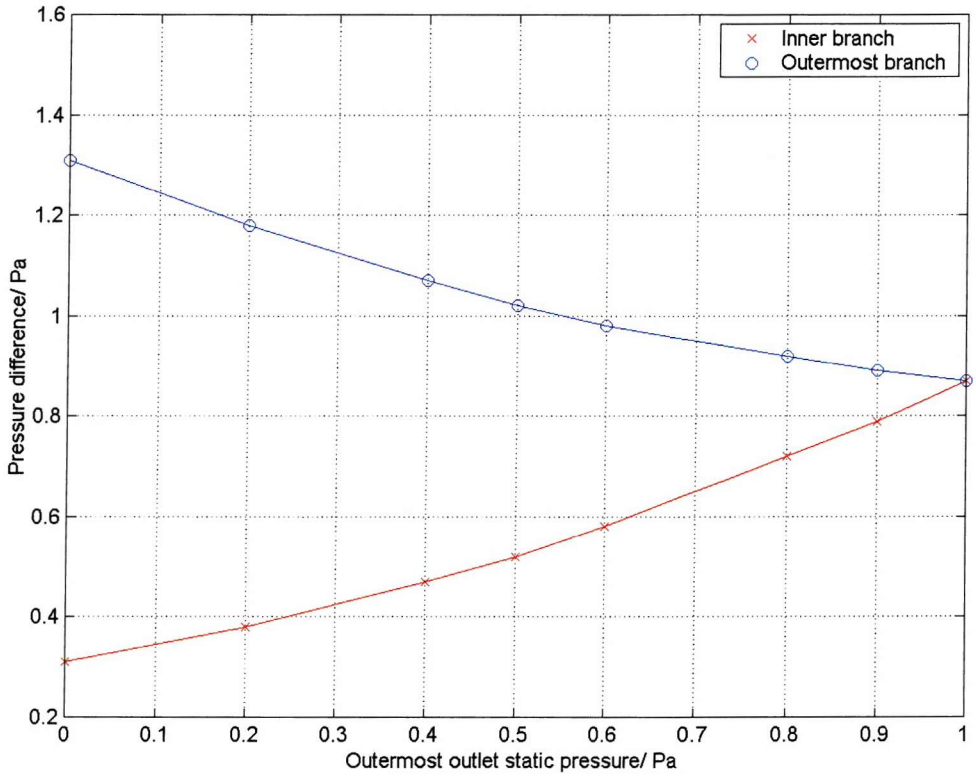


Figure 4-26 The pressure difference between each branch outlet and the inlet

When the pressure in one branch outlet is varied the pressure drop between that branch and the inlet is different from the pressure drop between fixed outlet branch pressure and the inlet pressure. In order to give an overall performance measure of an airway branch it may be better to consider the sum of the pressure differences between the inlet and all outlets. For a symmetrical double bifurcation it would be between the two outlets and the inlet. The difference in pressures between the inlet and the both outlets added together is shown in Figure 4-28. An additional run was done with 0.5 Pa pressure in one outlet. It can be seen from Figure 4-28 that sum of minimum pressure difference is when outermost daughter tube pressure is set to a value between 0.5 and 0.4.

Table 4-1 Mass flow rate and pressure changes in 5th to 7th model

Outlet pressure/Pa	Inlet pressure/Pa	Max. pressure/Pa	Min. Pressure/Pa	Mass outer/ Kg s^{-1}	Mass inner/ Kg s^{-1}
1.0	1.87	4.495721	0.8377708	0.001252684	0.002388619
0.8	1.72	4.343751	0.6857026	0.001466957	0.00217443
0.6	1.58	4.206892	0.5487187	0.0017078429	0.0019336694
0.4	1.47	4.091633	0.4	0.0019723729	0.0016692505
0.2	1.38	4.002254	0.2	0.0022477766	0.0013938793
0.0	1.31	3.936529	0.0	0.0025136778	0.0011281014

The velocity changes due to Reynolds number change of 200 to 800 is shown in Figure 4-42 to Figure 4-45, when all outer branch relative static pressures are the same. Further increase in Reynolds number after 800 was difficult to simulate as the separation zone on the second bifurcation along the outer branch grew to such a large size that flow could not pass into the outermost branch. A zero or negative mass flow rate is experienced at the outermost branch outlet. Hence the calculation diverged. However using unequal pressure difference applied to the outlets, Reynolds number up to 1200 could be simulated. This goes to show that during heavy breathing where large mass flow rates are created the lung has to expand in such a way as to create small pressure difference across airway branches. During heavy breathing the expansion of the lung lobes could possibly be different owing to this flow behaviour.

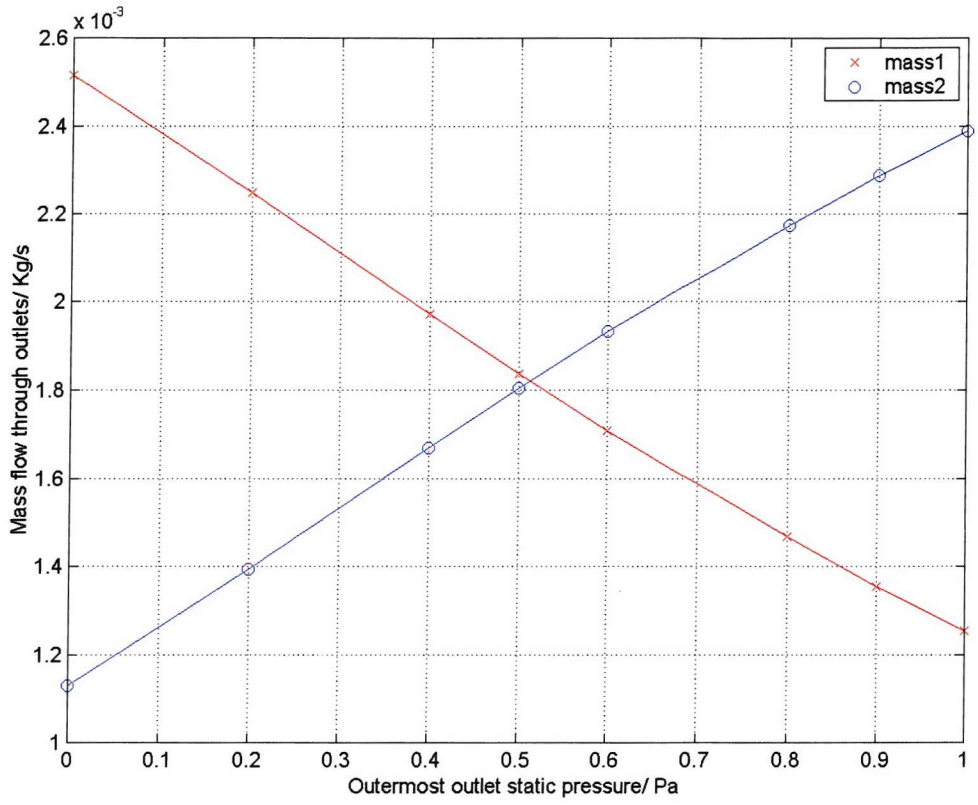


Figure 4-27 Mass flow rate through the outlets, where mass1 is the mass flow rate out of the outermost branch, while mass2 is the mass flow rate out of the innermost branch.

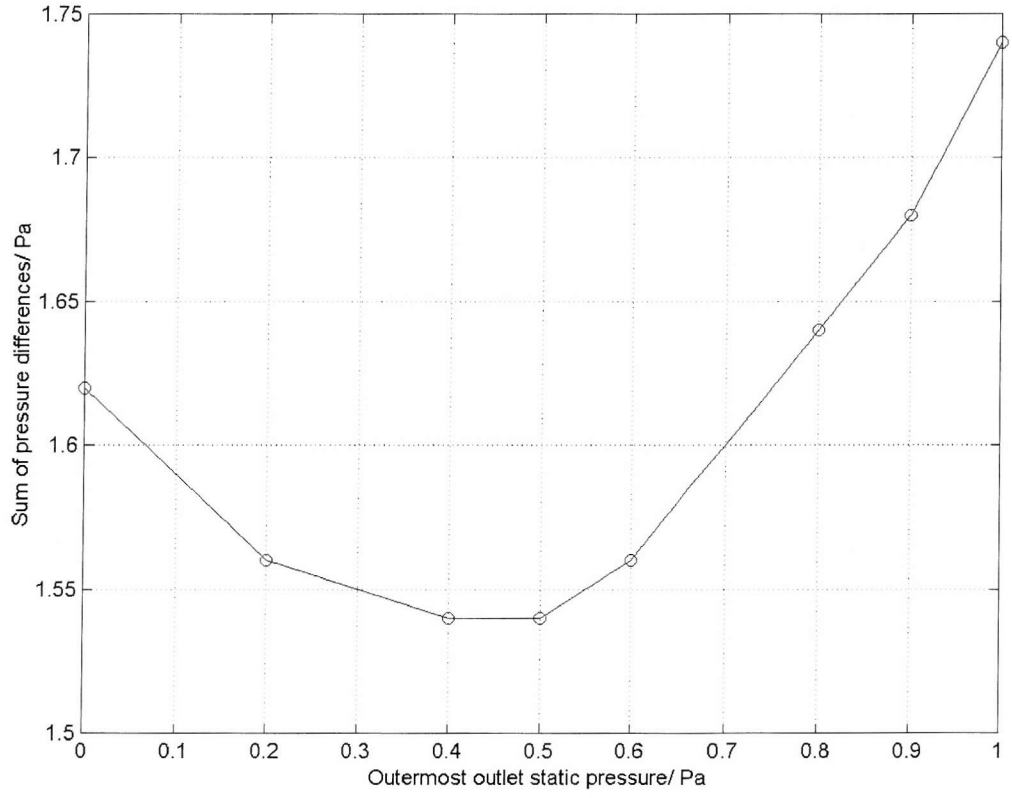


Figure 4-28 Sum of pressure differences between two outlets and the inlet

4.4.1 Entrance flow

Since the velocity profile was parabolic at the entrance the flow remains fully developed all the way to the bifurcation zone entrance. A long entrance length is used so that when particles released at some distance from the bifurcation entrance they will have time to adjust to the entrance flow conditions. It can be seen for equal outlet pressures the wall pressure reduces at a constant rate on the fully developed region as shown in Figure 4-20, as expected from fully developed flow.

4.4.2 First bifurcation

The static pressure gradually decreases before the entrance to the first bifurcation. Then it increases at the diffuser like entrance to the first bifurcation. In a diffuser type of flow the axial flow decelerates and this deceleration gradually propagates to the diffuser entrance. In order to maintain continuity at the diffuser entrance the flow accelerates at the diffuser wall region. Hence the pressure and the wall shear stress rises at the diffuser inlet wall. Momentum loss at the bifurcation is mainly due to wall shear stress increase, and inertial loss.

Flow separation is delayed at the transition zone (flow divider) of the first bifurcation primarily due to the presence of stagnation point. The pressure reduces all around the stagnation region and it retards the fluid upstream of the stagnation point. The fluid is pushed to either side of the stagnation point.

4.4.3 Entrance to first set of daughter tubes

The maximum pressure point is at the stagnation point. Near the stagnation point the pressure reduces fast radially. In fact at the entrance to the first daughter tubes the pressure reduction is in the axial direction of the daughter tubes. The axial flow therefore accelerates at the daughter tube entrance and reaches a maximum axial velocity just aft of the entrance. The maximum velocity in the daughter branches reduces due to viscous dissipation in the new boundary layer on the inner wall. Due to the rapid axial flow acceleration at the daughter tube entrance, flow separates on the outside wall. The Reynolds number is high enough for the separation zones on the first and second bifurcation to combine on the outermost wall. However as the outermost daughter branch pressure is reduced the pressure close to the outer wall is reduced and the separation region is reduced, but not removed.

The entrance length of the first daughter tubes is not long enough for the velocity profile to become fully developed. Note that the distance and the curvature of the flow divider seem to be important parameters, which influence the formation of boundary layer and boundary

layer development on the outer walls. For example having a shorter flow divider length (length from the diffuser-like entrance to the apex point of a given bifurcation) means that the outer wall boundary layer will be developed quickly, but with a larger length the boundary layer development will be suppressed, since the flow will have sufficient time to adjust to a slowly enlarging flow divider area before the stagnation point. Larger curvature would mean larger separation zones and higher inertial loss. The displacement thickness may be used to quantify the amount by which the streamlines are shifted owing to the formation of the new boundary layer on daughter branches and changes to it brought about by the outlet pressure changes.

4.4.4 Second bifurcation

The entrance velocity profile is skewed, hence allowing greater mass flow rate towards the inner branch. Here the stagnation pressure is not symmetrical about the second parent branch, and a large separation zone is developed at the second bifurcation zone. The stagnation point on the second bifurcation moves towards the outermost branch as the outlet pressure on the outermost branch is reduced, and more of the flow is directed along the outer branch. The separation bubble in the inner branch of the second bifurcation increases in width and length. When the pressure around the stagnation point becomes less asymmetrical the mass flow rates do not become symmetrical. In this case the approaching velocity profile shows that more mass is directed to the outer branch than to the inner branch.

(Andrade, Alencar et al. 1998) have considered a two dimensional, symmetrical, three generation airway model. They have shown that at low Reynolds numbers the flow seems to divide equally among the distal daughter branches. This is due to flow having enough time to reach fully developed flow at each daughter branch. But as the Reynolds number is increased the axial flow becomes biased towards the innermost branches.

One major piece of the flow physics that cannot be modelled by considering two-dimensional flow is the secondary flow created by the centripetal force acting on the axial velocity. This issue is considered briefly in next section.

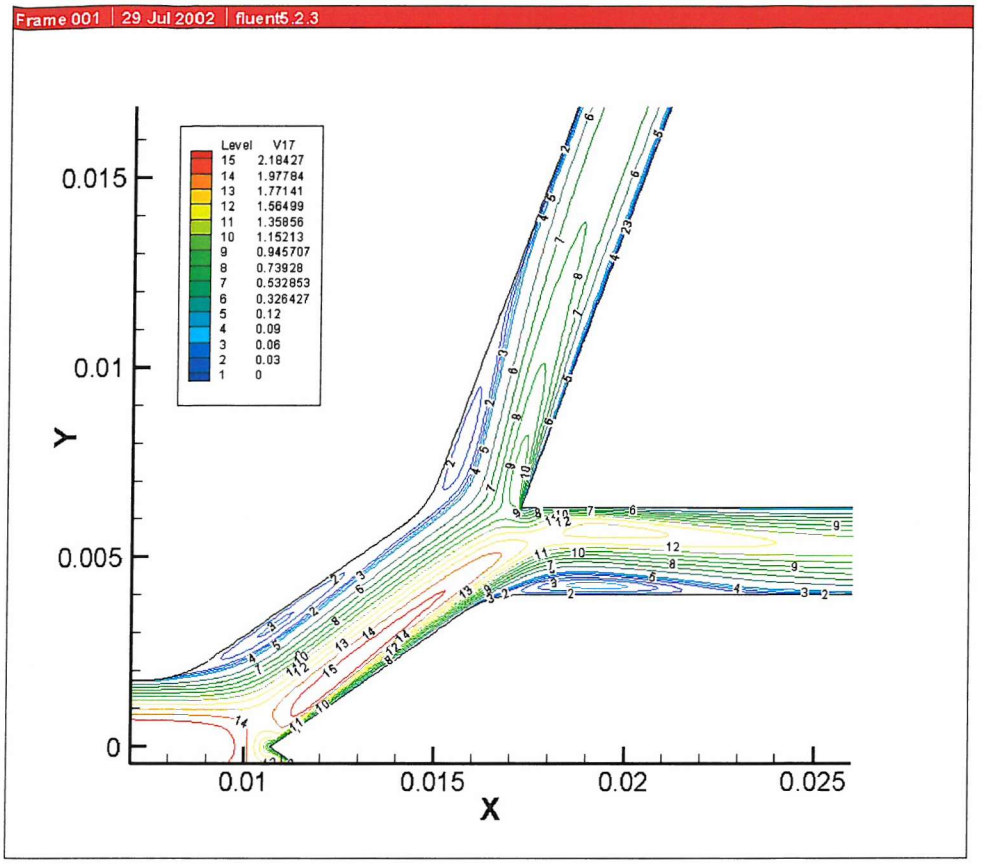


Figure 4-29 Velocity with outermost branch outlet pressure of 1Pa

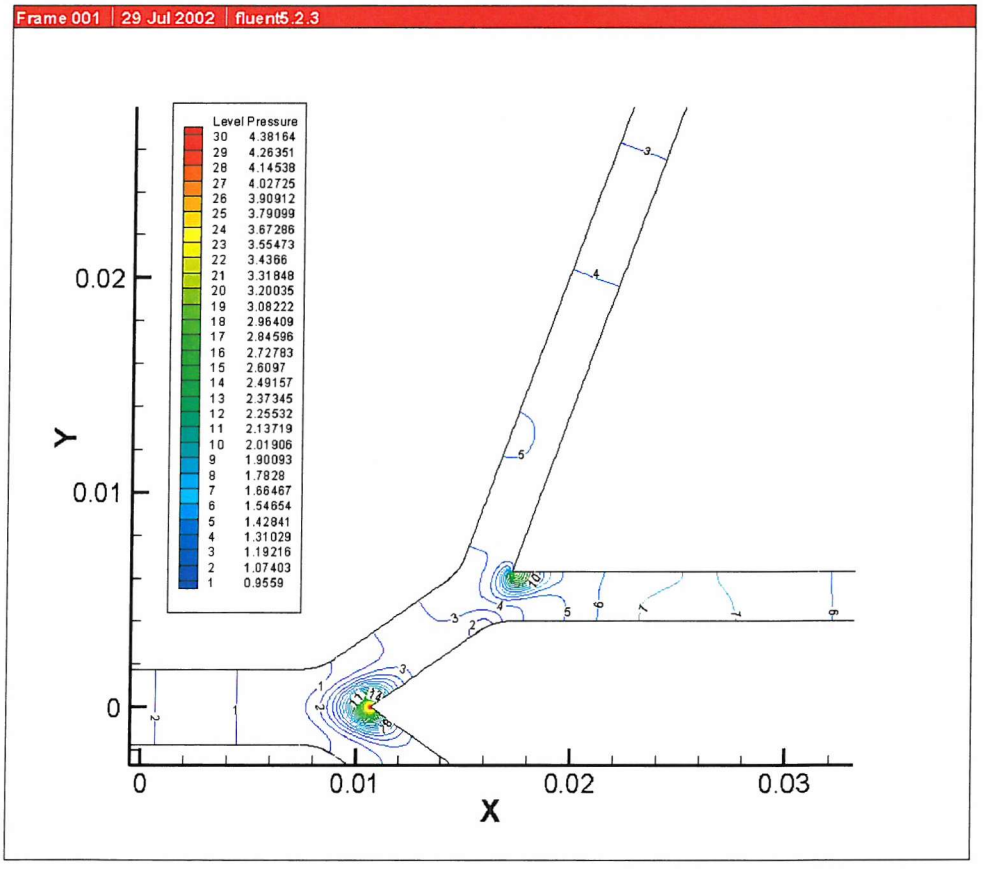


Figure 4-30 Pressure with outermost branch outlet pressure of 1Pa

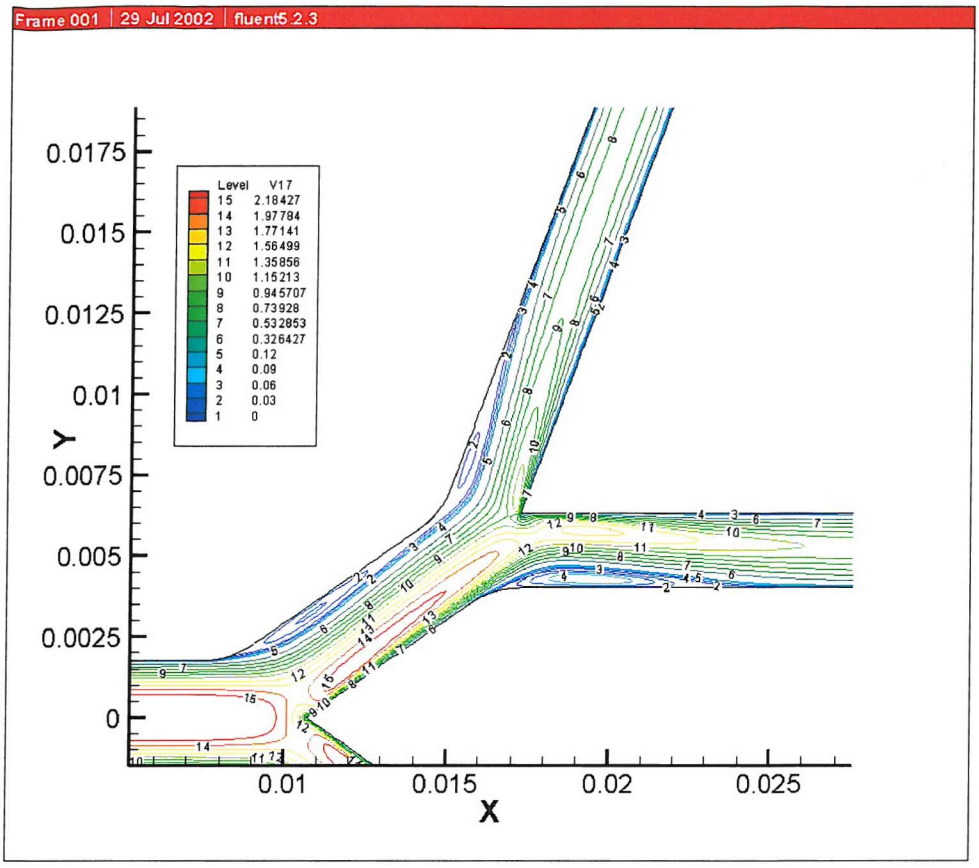


Figure 4-31 Velocity with outermost branch outlet pressure of 0.8Pa

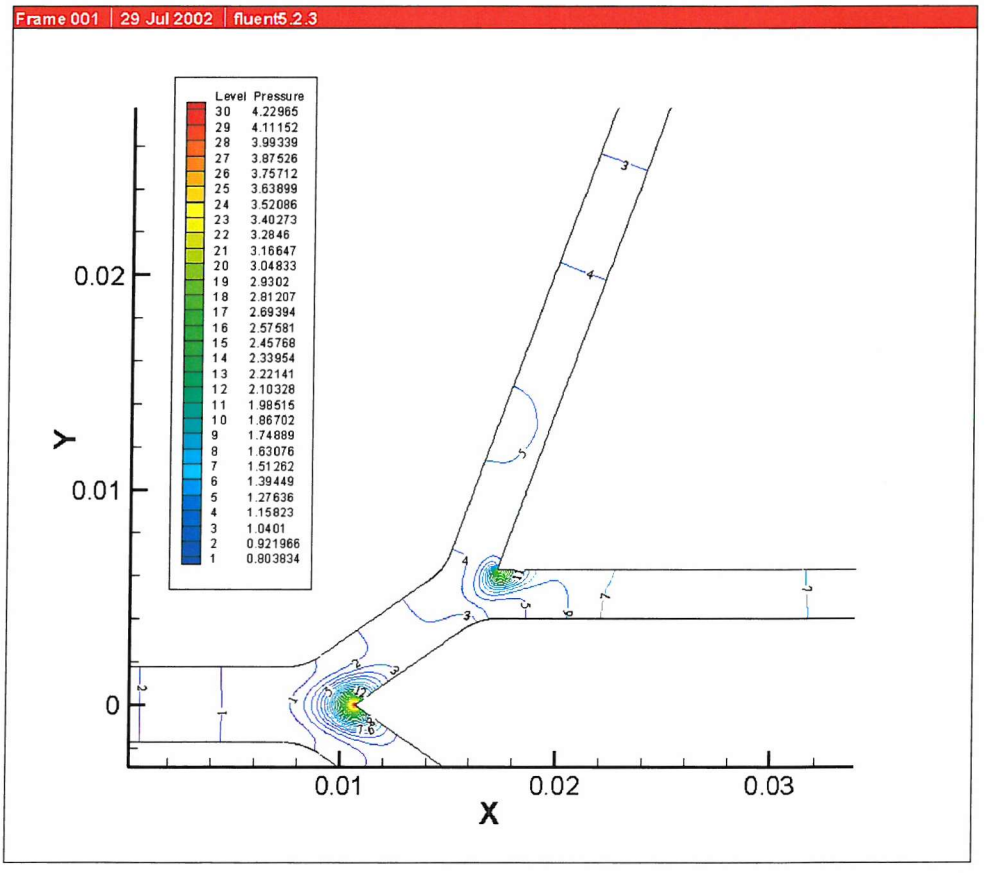


Figure 4-32 Pressure with outermost branch outlet pressure of 0.8Pa

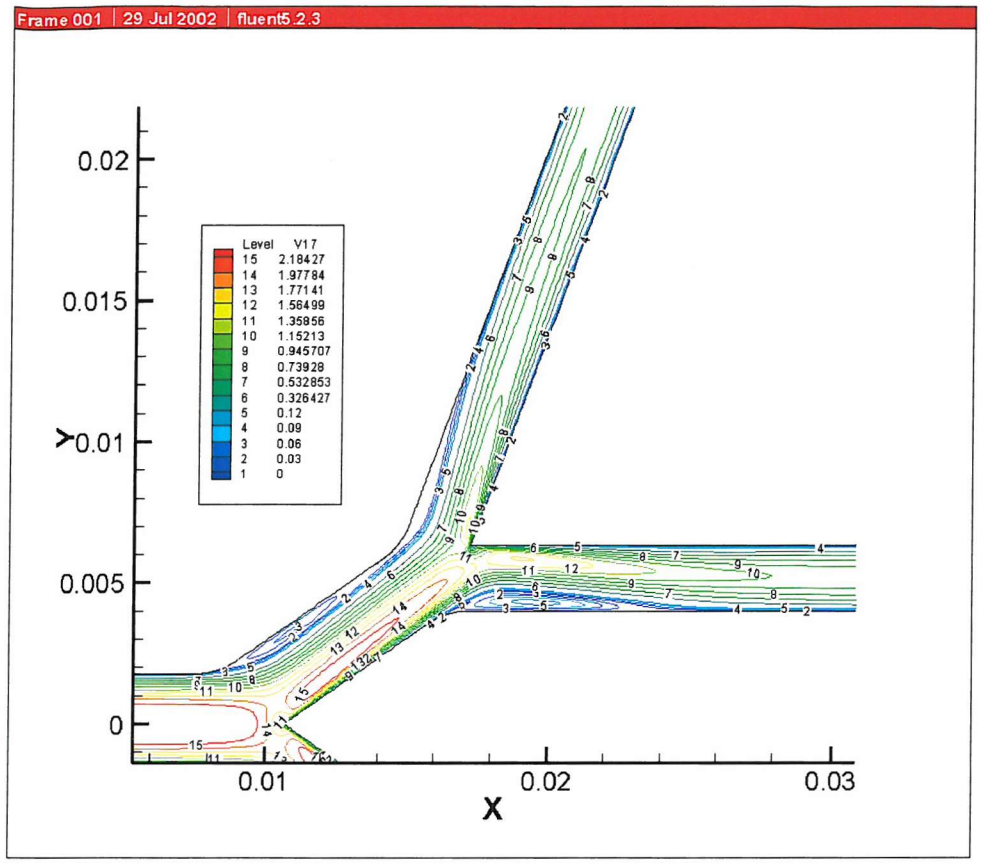


Figure 4-33 Velocity with outermost branch outlet pressure of 0.6Pa

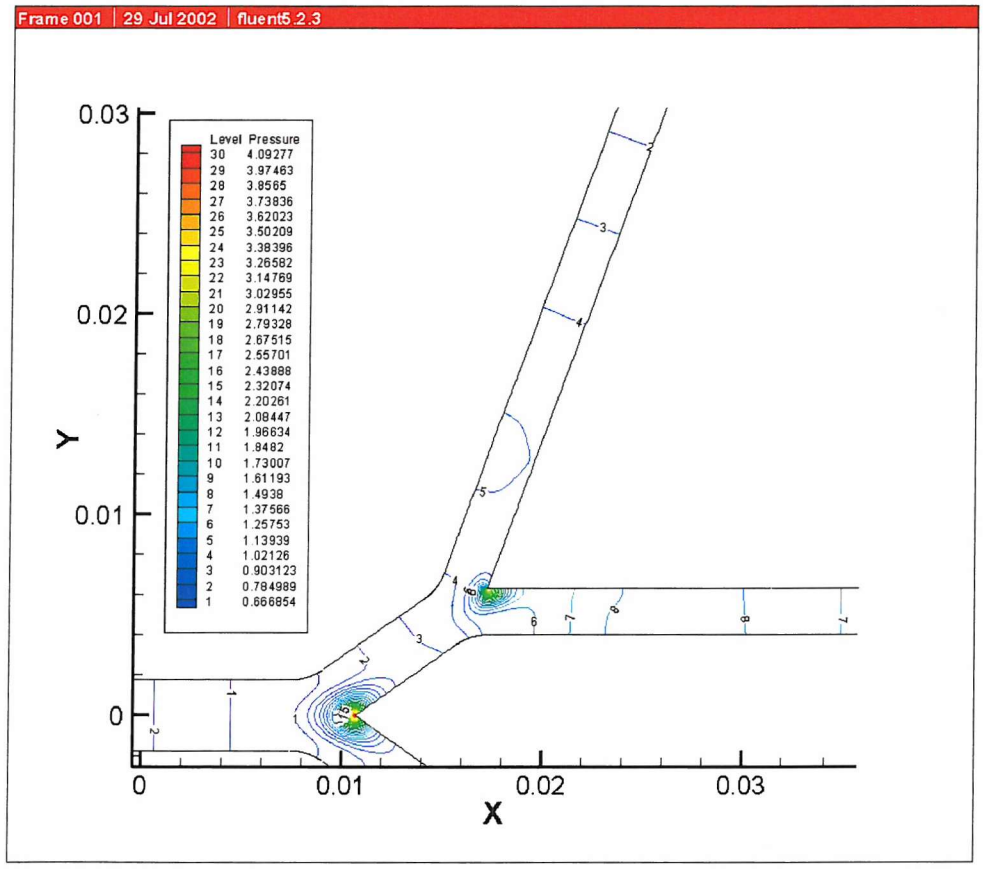


Figure 4-34 with outermost branch outlet pressure of 0.6Pa

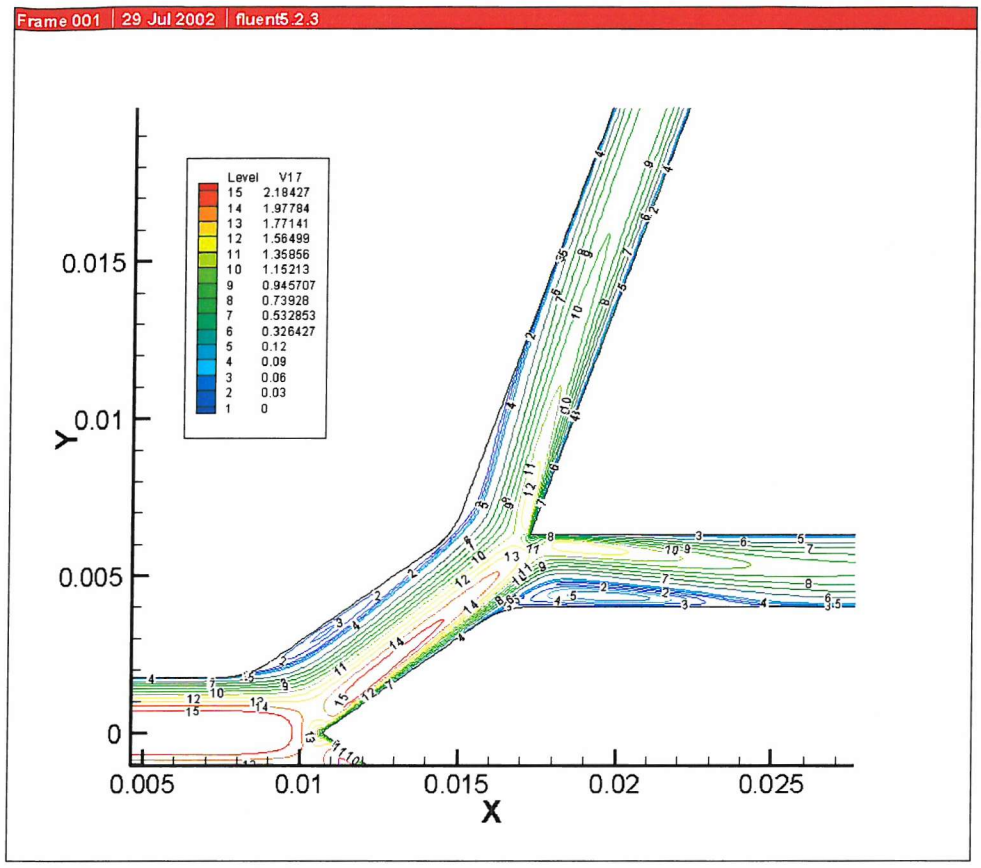


Figure 4-35 Velocity with outermost branch outlet pressure of 0.4Pa

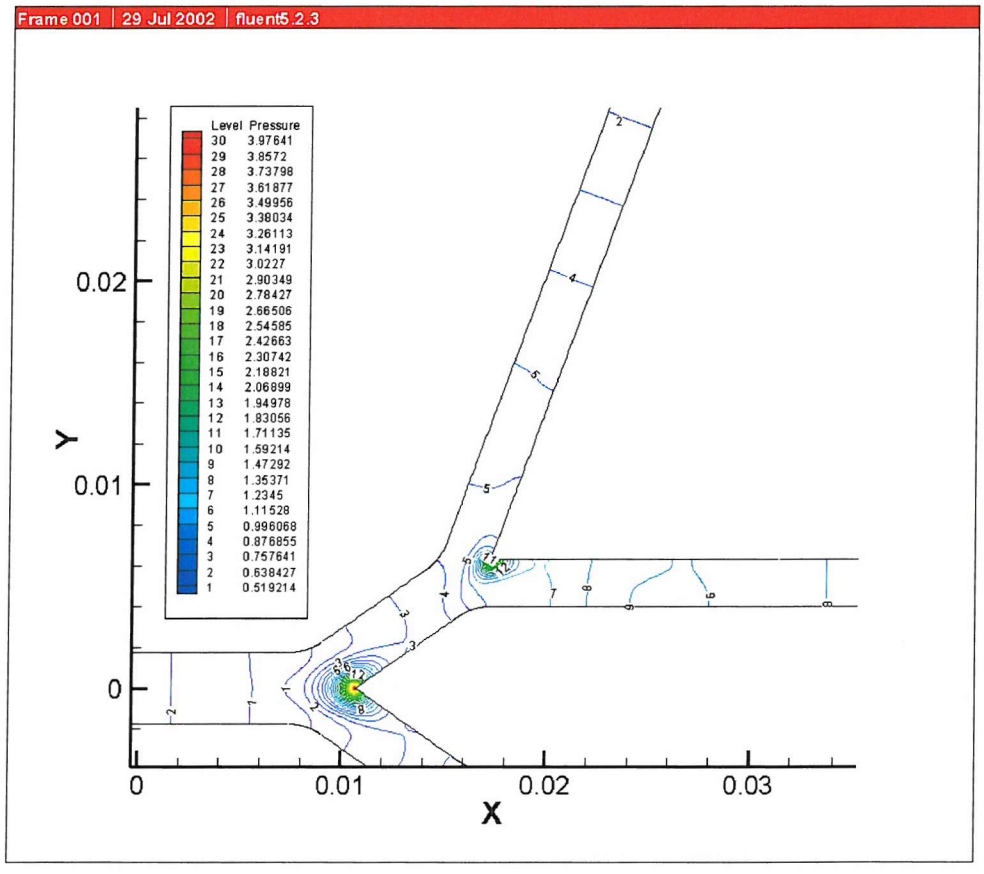


Figure 4-36 Pressure with outermost branch outlet pressure of 0.4Pa

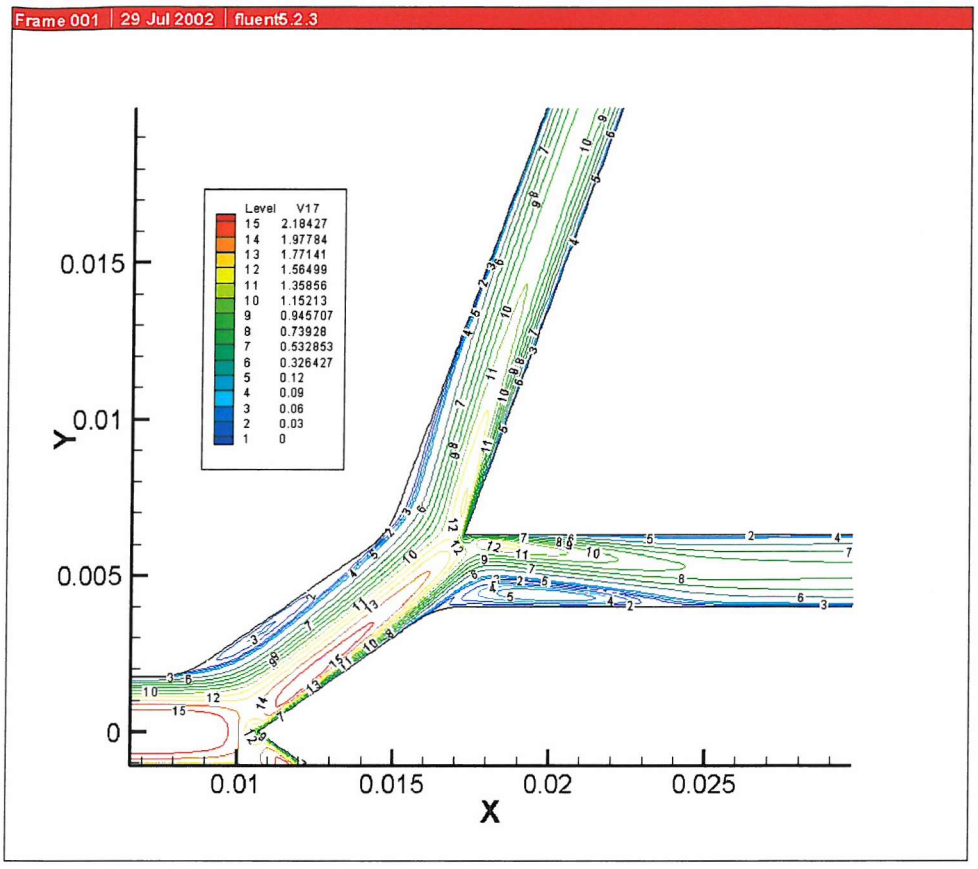


Figure 4-37 with outermost branch outlet pressure of 0.2Pa

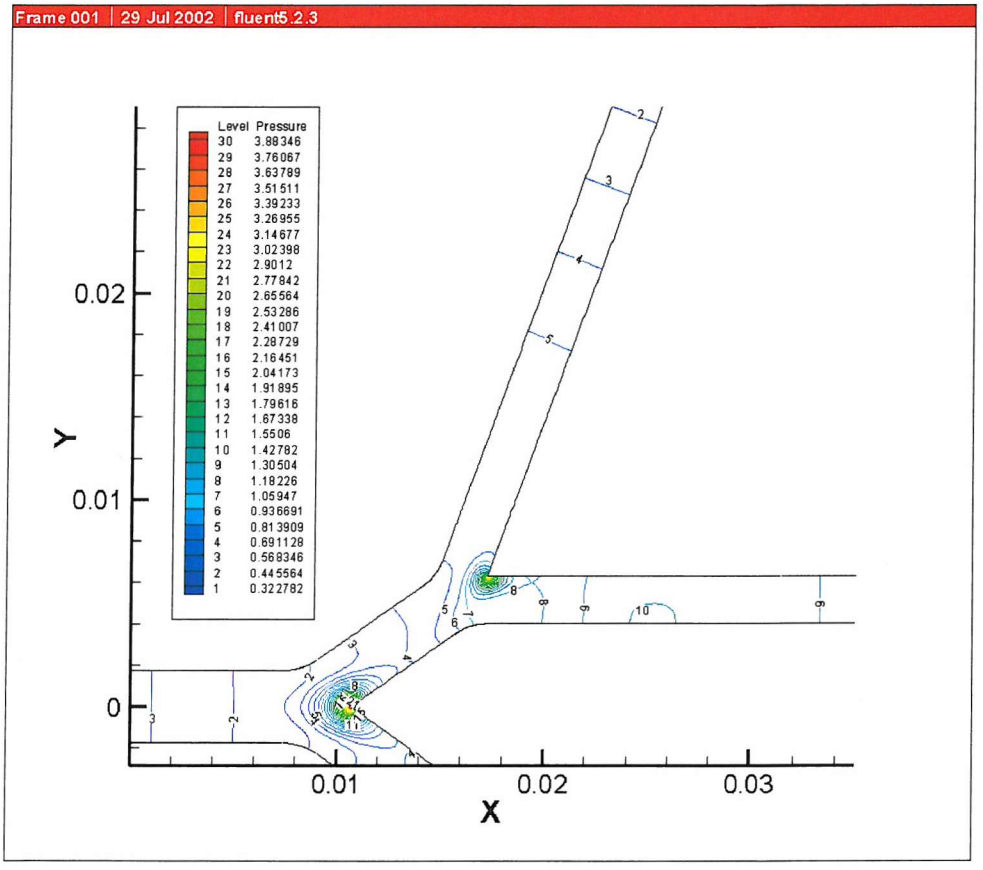


Figure 4-38 Pressure with outermost branch outlet pressure of 0.2Pa

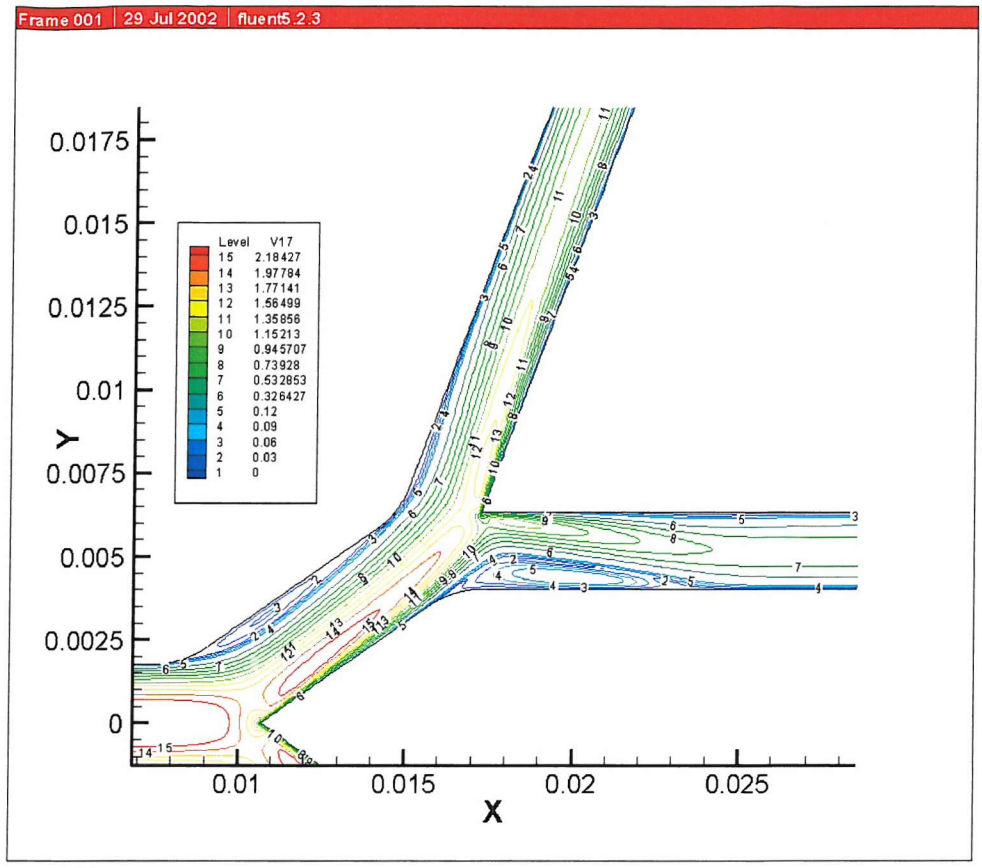


Figure 4-39 Velocity with outermost branch outlet pressure of 0.0Pa

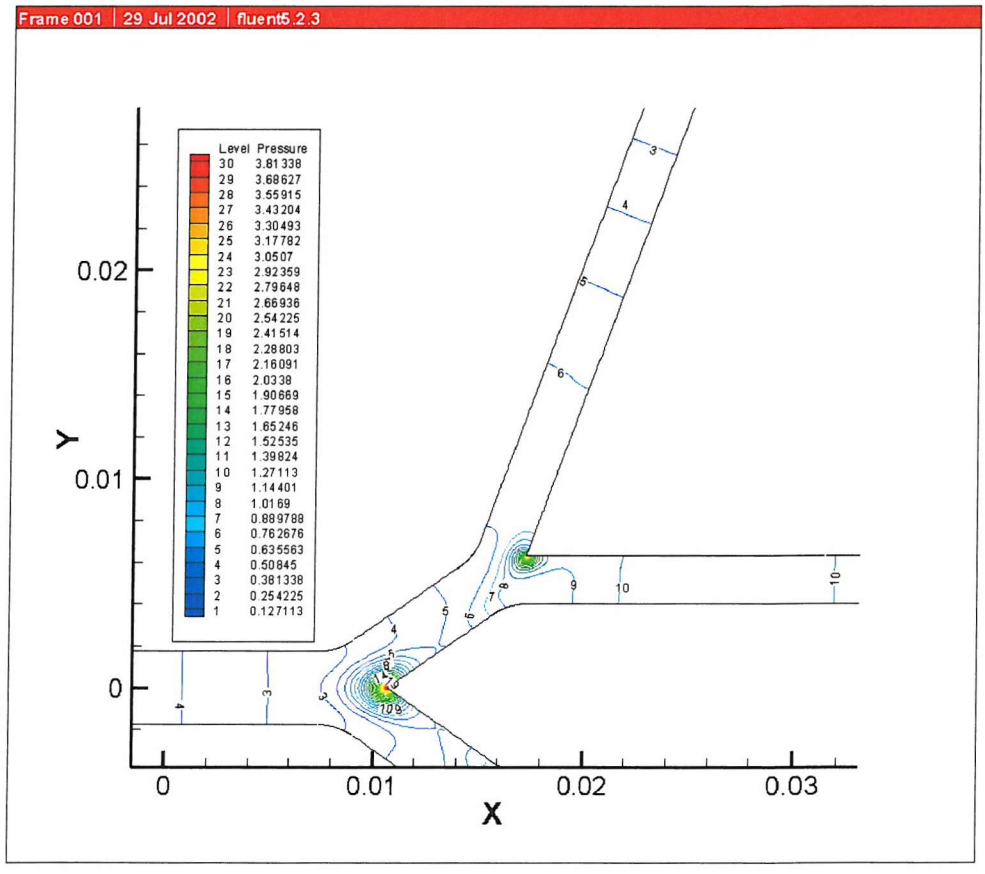


Figure 4-40 Pressure with outermost branch outlet pressure of 0.0Pa

4.5 Three dimensional flow model

The general flow patterns were observed and measured for quasi-steady cases in three-dimensions for single bifurcations in detail by (Comer, Kleinstreuer et al. 2001a), (Zhao and Lieber 1994) and (Chang and Masry 1982).

The outside curvature of the flow divider geometrically altered by the transition length (flow divider length) and the daughter tube curvature. The curvature present at the transition region creates centripetal acceleration on the axial flow, which leads to creating secondary vortices. The action of these vortices on the axial flow is to transfer the axial momentum to outer wall of the bifurcation where the centripetal acceleration is smallest. The secondary velocities tend to reduce the onset of separation and stabilise the flow. The Dean number, provides a measure of the flow development in curved tubes as a function of Reynolds number and the curvature ratio of the pipe. It is an important parameter that must be calculated to measure the magnitude of the secondary flow.

Some preliminary three-dimensional simulations were conducted in this project, using the multi-block structure shown in Figure 3-6. The secondary vortices produced as expense of the axial velocity in three-dimensional simulation would mean that the axial velocities in two-dimensional simulation are greater, but in two-dimensional simulation the axial velocities are altered by existence of the separation zones. A vector plot of secondary flow on a daughter tube of a three-dimensional single bifurcation, is shown in Figure 4-41. Perpendicular to the bifurcation plane in the daughter branches the parabolic axial entry flow is transformed to a M-shape axial velocity profile downstream as a result of the action of secondary vortices. More cross-sectional velocity profiles in three-dimensions are shown in Appendix D.

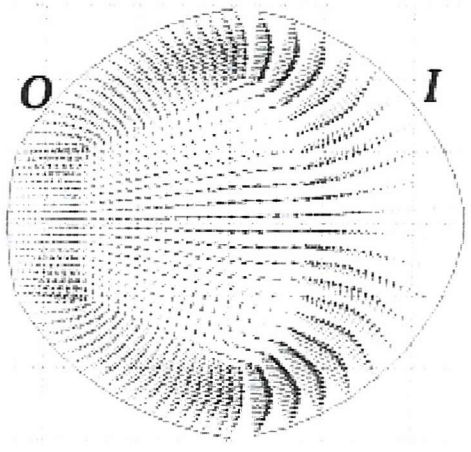


Figure 4-41 Secondary flow on a daughter tube

4.6 Conclusion

The earliest flow resistance studies assumed that fully developed flow existed in all branches of the airways. Then later on flow resistance is attributed to developing pipe-entrance flow (Pedley, Schroter et al. 1970b). However this study has showed that flow resistance is mainly due to various energy dissipating flow phenomena that occurs in the complex flow fields in bifurcating tubes. The formation of separation regions and stagnation region in each bifurcation increases momentum loss. The diffuser type entrance to each bifurcation induces wall shear at the diffuser-like entrance. The viscous dissipation significantly reduces the axial velocity at each daughter branch due to the formation of the new boundary layer on inner walls. The results showed that the minimum pressure loss occurs in the double bifurcation two-dimensional model when the pressure on the outermost branches is around 60% less than that of innermost branch. At this flow condition the magnitudes of all separation regions are small.

When the outlet pressures are the same relative to each other the mass flow rate is biased towards the innermost branch. It was identified that in central airways mechanisms exists to create pressure changes, this was observed by noting that upper lobes are expanded to a lesser degree than the lower lobes. But there also could be other mechanisms that help to create very small pressure changes at branch outlets in such a way to maximise the flow rate and to minimise the pressure loss.

Grid independence study based on velocity was obtained convincingly, but when wall pressure is used it was found at places higher grid density was required, especially in the bifurcation regions and separation regions. The pressure changes across the bifurcation were very small but created significant changes in mass flow rate.

Flow behaviour under steady flow conditions has been investigated, including flow dynamics under variation in outlet pressure, and various Reynolds numbers with the same relative outlet pressures. A better understanding of the pressure losses could be made if the flow is studied under different outlet pressure conditions and under different inflow profiles.

4.7 References

Andrade, J. S., A. M. Alencar, et al. (1998). "Asymmetrical Flow in Symmetric Branched Structures." Physical Review Letters **81**(4): 926-929.

Chang, H. K. and O. A. E. Masry (1982). "A model study of flow dynamics in human central airways: Part 1 axial velocity profiles." Respiration Physiology **49**: 75-95.

Comer, J. K., C. Kleinstreuer, et al. (2001a). "Flow structures and particle deposition patterns in double-bifurcation airay models: Part 1 Air flow fields." Journal of Fluid Mehanics **435**: 25-54.

Fluent Inc. (1998). Fluent Manual.

Leonard, B. P. (1979). "A stable and accurate convective modelling procedure based on quadratic upstream interpolation." Computer Methods Applied in Mechanical Engineering **19**: 59-98.

Patankar, S. V. (1980). Numerical heat transfer and fluid flow. Washington, Hemisphere Pub. Co.

Pedley, T. J., R. C. Schroter, et al. (1970b). "The prediction of pressure drop and variation of resistance within the human bronchial airways." Respiration Physiology **9**: 387-405.

Rhie, C. M. and W. L. Chow (1983). "Numerical study of the turbulent flow past an airfoil with trailling edge separation." AIAA Journal **21**(11): 1525-1532.

Zhao, Y. and B. B. Lieber (1994). "Steady inspiratory flow in a model symmetric bifurcation." Journal of Biomechanical Engineering, (TASME) **116**: 488-496.

4.8 Velocity magnitude plots for various Reynolds numbers

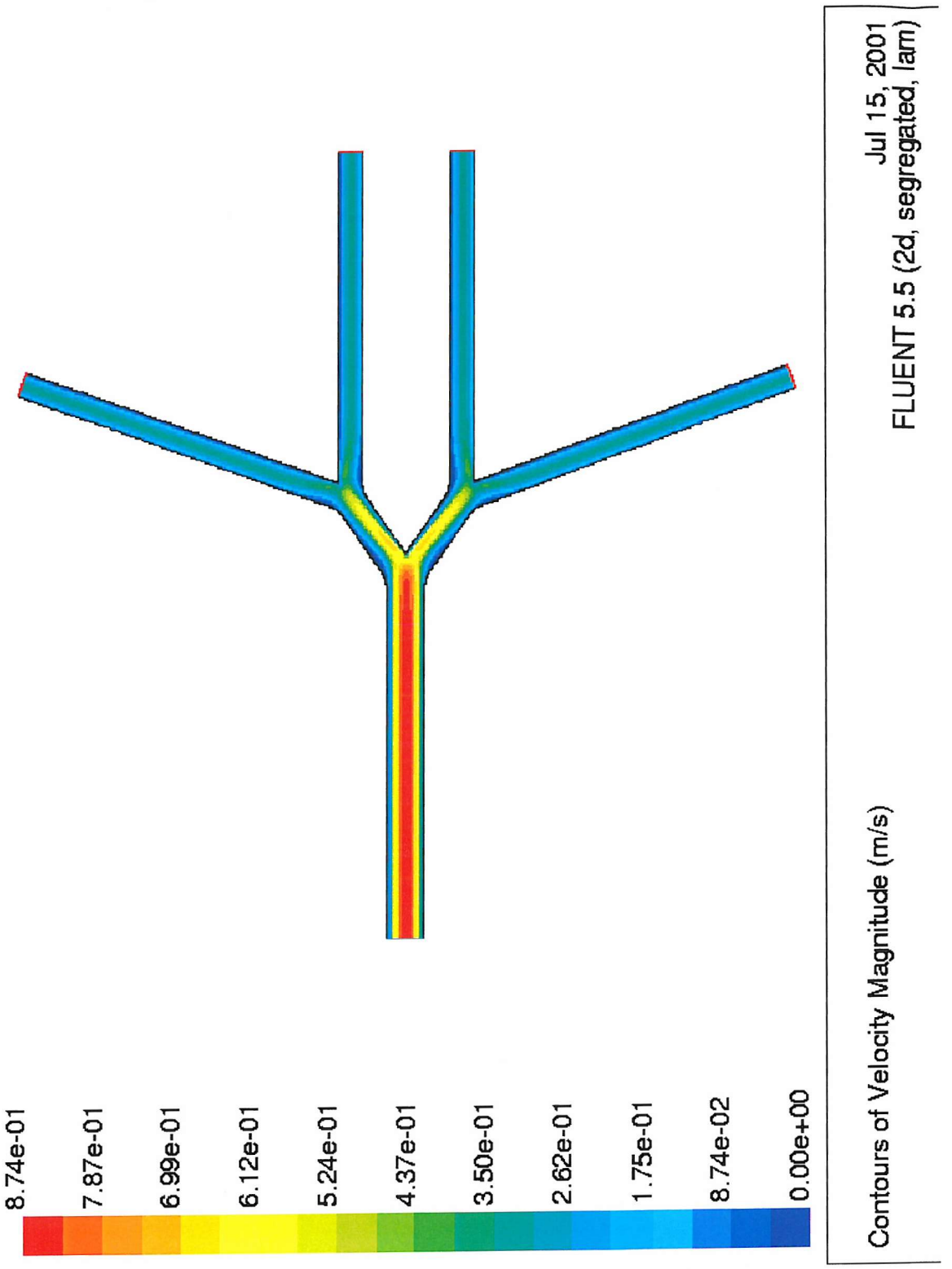


Figure 4-42 Magnitudes of velocity at a Reynolds number of 200

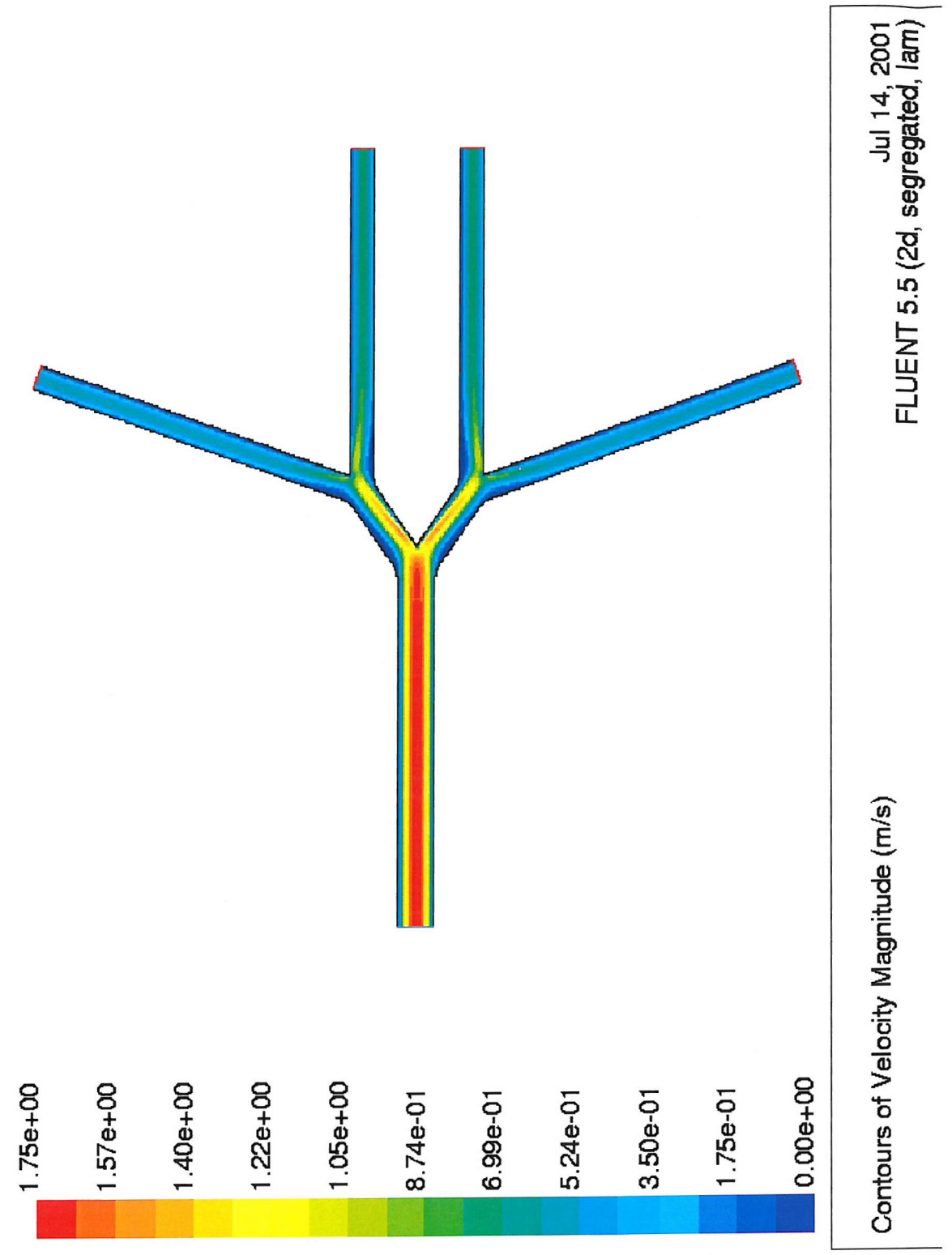


Figure 4-43 Magnitudes of velocity at a Reynolds number of 400

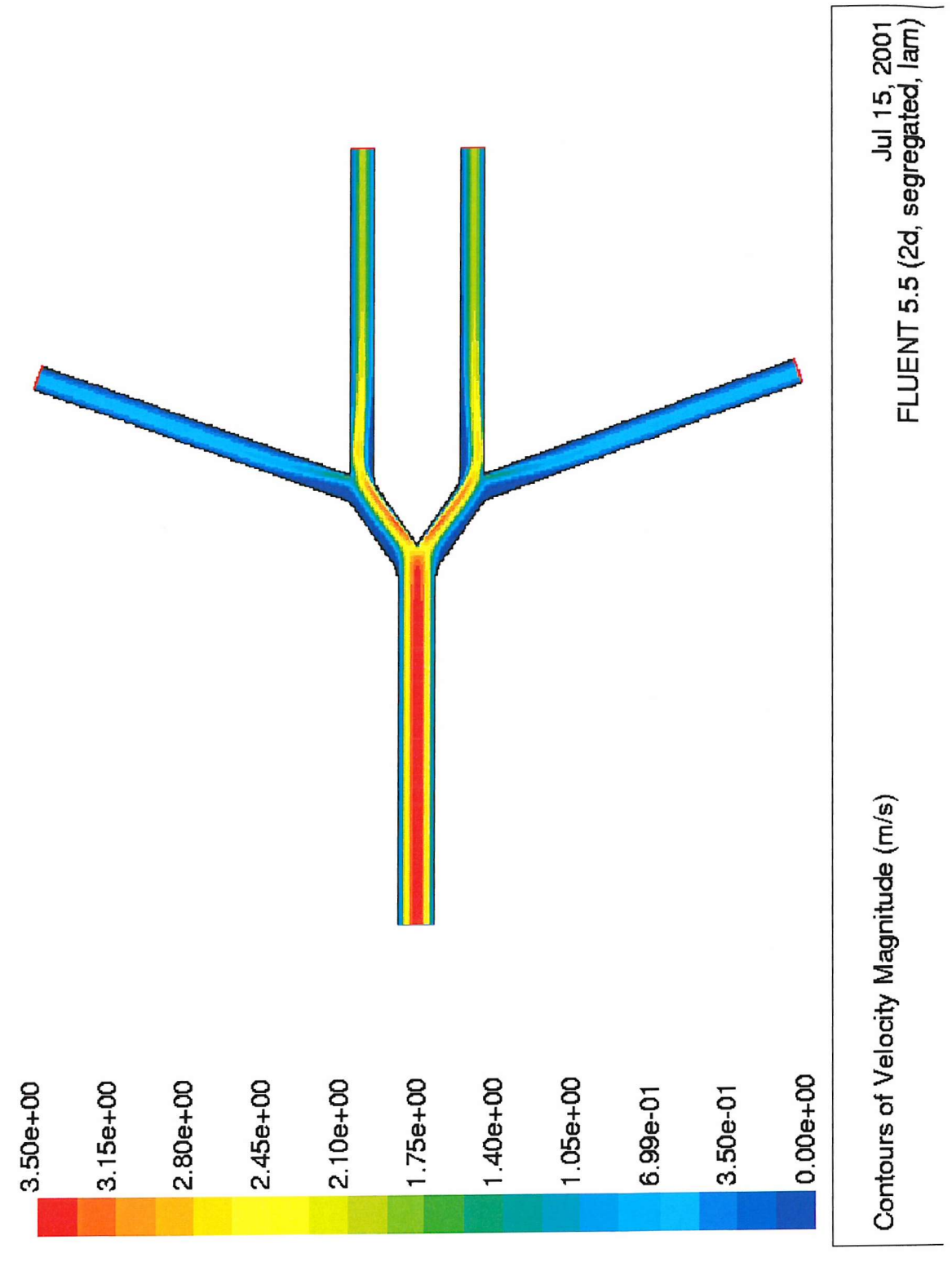


Figure 4-44 Magnitudes of velocity at a Reynolds number of 600

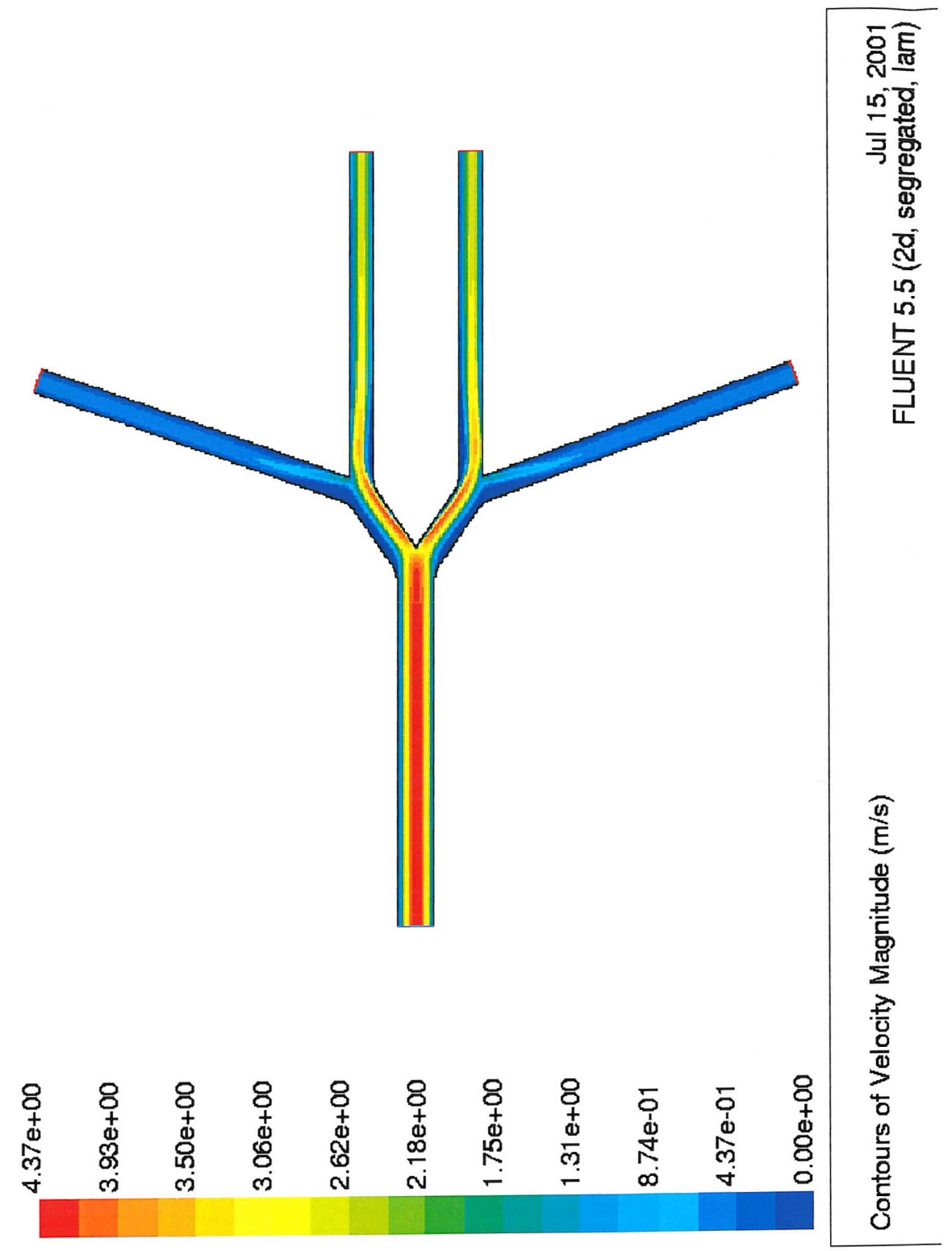


Figure 4-45 Magnitudes of velocity at a Reynolds number of 800

5 Particle Equations of motion and Interpolation methods

5.1 Introduction

The particle tracking process, which will be explained in next chapter, requires an equation of particle motion, a numerical solution process to solve it and fluid velocity interpolation at the particle locations. The aim is to model the motion of aerosol particles coated with drugs, which are introduced into the airway flow. The properties of aerosol particles and the assumptions made in describing the particle properties are given in section 5.2. Preliminary considerations used to derive the particle equations of motion for a particle carried by the fluid and influenced by various fluid forces, will be clarified in section 5.3. The exact derivation of particle equations of motion from first principles is given by (Maxey and Riley 1983). From this equation of motion an assessment will be made in section 5.3 as to which forces that depend on the relative velocity between the particle and the fluid might be of importance to the aerosol particle tracking in two-dimensional bifurcation flow. In section 5.4 an appropriate numerical scheme to solve the particle equations of motion will be presented. Then in section 5.5 various interpolation schemes to calculate the fluid phase properties at particle position will be introduced.

5.2 Characteristics of aerosol particles

Aerosols are solid or liquid particles suspended in a gas. They are particles that can be dispersed in air. It is solid aerosols that are of interest since they behave distinctly to the surrounding forces. Their size and density, for a particular shape, dictate where they are most likely to be deposited.

Aerosol particles used in the particle deposition experiments on airway bifurcation have a radius range of 0.5 to 10μm. When deriving the particle equations of motion the shape of the aerosol particle is considered to be spherical. However aerosol particles come in various shapes like ovals, squares, spikes and so on, and in general they are far from spherical (Fuchs 1964). The density of an aerosol particle is much greater than that of air.

It is on the surface of aerosol particles that drugs may be coated. It has been noted that aerosol particles could absorb water and increase their mass and change their chemical properties. Aerosol particles easily coagulate with each other when they are in close proximity to each other. Some particles are volatile and could evaporate under small temperature variations. Thus mass transfer is likely to take place when they are in motion.

However it is assumed that the aerosol particle considered here have only the density and size of an aerosol particle and none of the other properties.

Other than aerosol particle depositions, small particle-fibre depositions in the airways are also of interest, (Zhang, Bahman et al. 1996), since some industrial fibres are responsible for lung disease and decease in lung function. Knowing the fibre deposition sites would enable medical professionals to apply more effective treatment. Even though the fibre particle equations of motion is different to that of aerosol particle the main methodology of the tracking process is the same.

5.3 Particle motion

The particle dynamics are controlled by three groups of forces.

- Forces that act through the interface between fluid and particle
- Forces due to the interaction between particles
- Forces imposed by external fields

It is assumed that the particle concentration is dilute and forces on a particle due to particle collision are negligible. When the particle concentration increases the fluid phase properties such as the effective viscosity in the vicinity of the particles may also change (Soo 1990). Then a more complicated multi-phase flow system results. The orientation of the bifurcation with respect to the gravitational force is unknown. Hence, even though the particle mass is larger than the fluid particle the influence of gravitational force is ignored, so that no external forces are assumed to act on the particle. The dominant forces that directly influence the motion of particle will be the forces from the fluid-particle interactions.

In the literature there are many theoretical basis on which single particle motion has been derived. Their derivations depend on following characteristic fluid-particle interactions.

- A particle moving with constant velocity in a uniform flow field
- A particle accelerating in a uniform flow field
- A particle moving with constant velocity in a non-uniform flow field
- A particle rotating with constant angular velocity in a uniform flow field

Studies have shown four types of dominant forces: the drag force, Basset force, Saffman force, and Magnus force.

5.3.1 Drag force

When the particle velocity, $\mathbf{U}_p$, differs from the fluid velocity, $\mathbf{U}$ an unbalanced pressure distribution and viscous stresses are generated on the particle surface. (Note that bold face letters represent vector quantities). A resultant drag force per unit mass of particle (particle is treated as point masses since there are no coupling involved), $\mathbf{F}_D$, is imposed by the fluid on the particle called the, which is given by Equation 5-1.

Equation 5-1

$$\mathbf{F}_D = \frac{18\mu}{d_p^2 \rho_p}(\mathbf{U} - \mathbf{U}_p) C_D \frac{Re_p}{24}$$

Where ρ_p and d_p are the particle density and particle diameter respectively. C_D is the drag coefficient of the particle and it is a function of the particle Reynolds number Re_p. (Cliff, Grace et al. 1978) gave the following empirical relations for C_D.

Equation 5-2

$$C_D = \begin{cases} \dfrac{24}{Re_p} & \text{for} \quad 0 < Re_p \le 1.0 \\ \dfrac{24}{Re_p^{0.646}} & \text{for} \quad 1.0 < Re_p \le 400 \end{cases}$$

The particle Reynolds number is given by Equation 5-3.

Equation 5-3

$$Re_p = \rho d_p \frac{|\mathbf{U} - \mathbf{U}_p|}{\mu}$$

Where μ and ρ are the fluid viscosity fluid density. In Stokes flow, $C_D=24/Re_p$, and it is assumed that particle Reynolds number, $Re_p << 1$, so that Stokes formulae can be used. This implies that viscous forces dominate around the particle.

5.3.2 Basset force

The Basset force is important when the particle is accelerating or decelerating in a fluid. It adjusts the particle acceleration by taking into account the past acceleration on the particle motion. The force could be in the direction of motion or added to the drag force direction. In a simple model with constant particle acceleration the ratio of Basset force to the Stokes drag, R_{BS} (a non-dimensional quantity) is given in (Rudinger 1980) as follows.

Equation 5-4

$$R_{BS} = \sqrt{\frac{18}{\pi}\frac{\rho}{\rho_p}\frac{\tau_S}{\tau}}$$

It is expected that time change of particle is much longer than the Stokes relaxation time, τ_S, given in Equation 5-5 and hence the acceleration rate of the particle is expected to be small.

Equation 5-5

$$\tau_S = \frac{\rho_p d_p^2}{18\mu}$$

5.3.3 Saffman force

When a particle moves at a constant velocity in a flow where a velocity gradient exists around the particle additional force a lift force, acting perpendicular to particle motion, is imposed on the particle called the Saffman's lift force (Saffman 1965). A lift force could also be produced not just by velocity gradient but also by a pressure gradient or a temperature gradient. The Saffman to the Stokes drag is given by Equation 5-6 (Fan and Zhu 1998).

Equation 5-6

$$R_{SS} = \frac{Kd_p}{12\pi}\sqrt{\frac{1}{\upsilon}\left|\frac{\partial(\mathbf{U}-\mathbf{U}_p)}{\partial y}\right|}$$

At low Re_p the Saffman's force is negligible, but near the wall this force could contribute significantly to the particle resistance force. In this study Staffman force is ignored.

5.3.4 Magnus force

A particle could rotate due to a non-uniform velocity region around the particle. In low Reynolds number flows particle rotation leads to fluid entrainment, resulting in an increase in the velocity on one side of the particle and decrease on the other side. Thus a lift force, acting normal to the particle motion results, which moves the particle towards the region of higher velocity. This phenomenon is known as the Magnus effect. In fact it was claimed by (Soo 1990) that the Magnus effect caused particles to move towards the centre of tubes in Poiseuille flow. The lift force due to the particle spin is negligibly small compared to the drag force when the particle size is small or the spin velocity is low. The Magnus effect to the Stokes drag is expressed by Equation 5-7 (Fan and Zhu 1998).

Equation 5-7

$$R_{MS} = \frac{d_p^2 \rho}{24\mu} \Omega$$

where Ω is the angular velocity of particle. In this study the Magnus effect is ignored.

5.3.5 Particle equations of motion

Equation 5-8

$$\frac{dU_p}{dt} = \sum F_p$$

The equation of particle motion is shown in Equation 5-8, where ΣF_p is the linear additives of the above-mentioned forces per unit mass. However, it was discussed that most of these forces in some simple flow field can be ignored with respect to most dominant particle force is the Stokes drag force. Then the particle equation of motion becomes Equation 5-9. Replacing the drag coefficient, C_D, with the expression given in section 5.3.1 for particle Reynolds number in the range of $0 < Re_p \leq 1.0$, and substituting the expression for Stokes relaxation time, τ, from Equation 5-5, the particle equations of motion Equation 5-9 finally reduces to Equation 5-10.

Equation 5-9

$$\frac{dU_p}{dt} = \frac{18\mu}{d_p^2 \rho_p}(U - U_p) C_D \frac{Re_p}{24}$$

Equation 5-10

$$\frac{dU_p}{dt} = \frac{1}{\tau}(U - U_p)$$

The particle path equation subject to the initial condition $\mathbf{U}_p(\mathbf{x},0) = \mathbf{U}(\mathbf{x})$ is given in Equation 5-11, where $\mathbf{x}$ is the particle position.

Equation 5-11

$$\frac{d^2\mathbf{x}}{dt^2} = \left(\frac{\mathbf{U} - \mathbf{U}_p}{\tau}\right)$$

The fluid velocity $\mathbf{U}$ at particle position is evaluated from the Eulerian flow field data and the Ordinary Differential Equation (ODE), Equation 5-11, will be integrated twice to

obtain the particle trajectories. These two issues will be investigated in section 5.4 and section 5.5 respectively.

The particle equation of motion has been derived by (Maxey and Riley, 1983) from first principles. The disturbed and undisturbed flow field due to a particle presence and without particle presence respectively were represented by the non-linear Navier Stokes equations centred on a coordinate system at the particle centre. Ignoring the inertial terms of the disturbed fluid due to the presence of the particle, the stress integral of the stress tensor was solved around the particle to obtain the particle equation of motion. The final form of the particle equation of motion contained the Stokes drag term, Basset term, terms due to fluid acceleration with the particle, and an external force term, whose effects were discussed in previous sesctions. Maxey and Riley showed that Stokes drag term was the biggest and all other forces are considerably smaller when the particle Reynolds number is less than one.

It is assumed that the particle is large enough to ignore rarefied gas effects, and also that the particle travels at such velocity that Brownian effects are negligible. The equation of motion is centred on the particle centre and treats the particle as a point mass, which in effect makes the slip velocity to be $|\mathbf{U}-\mathbf{U}_p|$. Also two-way coupling between particle forces and fluid forces is avoided.

The Lagrangian trajectory approach is the best modeling approach to study the motion of individual particles. The stochastic type of particle trajectory solvers is computationally costly and more suited to turbulent flow fields. In either case the fluid phase dynamics has to be pre-determined. The particle trajectories will be computed for a steady flow field and it will be a post-process.

5.4 Numerical schemes to solve the second order ODE

The equation of particle motion given by Equation 5-11 will be solved numerically. The second order ODE can be used to solve two first order ODEs.

Equation 5-12

$$\frac{d\mathbf{x}}{dt} = \mathbf{U}_p$$
$$\frac{d\mathbf{U}_p}{dt} = \frac{(\mathbf{U}-\mathbf{U}_p)}{\tau}$$

To solve the Equation 5-12 the particle position at zero time, $\mathbf{x}(t_0)=\mathbf{x}_0$, and particle velocity at this position is needed. The velocity will be set to the fluid velocity at particle

position, $\mathbf{U}_p(t_0) = \mathbf{x}'(t_0) = \mathbf{U}(x_0)$. A 4th order Runge-Kutta method can be applied to solve the equations using the given initial conditions. The algorithm used, given in Table 5-1, is the Runge-Kutta-Nystrom method for N time steps.

Table 5-1 4th order Runge-Kutta method to solve 2nd order PDE of the particle equations of motion

For n=0 to N-1
$F_1 = h\tau(\mathbf{U}(\mathbf{x}_n) - \mathbf{U}_p(\mathbf{x}_n))$ $F_2 = h\tau(\mathbf{U}(\mathbf{x}_n + K_1) - (\mathbf{U}_p(\mathbf{x}_n) + F_1))$ $K_1 = \frac{1}{2}h\left(\mathbf{U}_p(\mathbf{x}_n) + \frac{1}{2}F1\right)$ $F_3 = h\tau(\mathbf{U}(\mathbf{x}_n + K_1) - (\mathbf{U}_p(\mathbf{x}_n) + F_2))$ $F_4 = h\tau(\mathbf{U}(\mathbf{x}_n + K_2) - (\mathbf{U}_p(\mathbf{x}_n) + 2F_3))$
$\mathbf{x}_{n+1} = \mathbf{x}_n + h\left(\mathbf{U}_p + \frac{1}{3}(F_1 + F_2 + F_3)\right)$ $\mathbf{U}_p(\mathbf{x}_{n+1}) = \mathbf{U}_p(\mathbf{x}_n) + \frac{1}{3}(F_1 + 2F_2 + 2F_3 + F_4)$

Forth order Runge-Kutta method have the advantage that it reproduce the terms in the Taylor series up to and including h^4, i.e. it has local truncation error of $O(h^5)$. It is efficient in a sense that it only requires moderate number of operations to give high order accuracy levels. Nevertheless error of $O(h^5)$ is added to the solution at each time step.

5.5 Interpolation schemes

The fluid velocity must be known at the particle positions to solve the particle equations of motion. Particles are assumed to be located within four nodes (in two-dimensions) where the fluid properties at these nodes are known. Let $\mathbf{x}^{(i)}$ be the local coordinates of node 'i', and the functional value at node 'i' is denoted by $f_i=f(\mathbf{x}^{(i)})$. Let $a_i(\mathbf{x})$ be the interpolation weight at node 'i'. The interpolation function g(x) can be described in general form by the Equation 5-13.

Equation 5-13

$$\mathbf{g}(\mathbf{x}) = \sum_{i=1}^{P} \mathbf{a}_i(\mathbf{x})\mathbf{f}(\mathbf{x}^{(i)})$$

Various interpolation schemes have been tested by (Balachandar and Maxey 1989) and, (Yeung and Pope 1988) when studying particle motion with a spectral simulation of turbulence. Some requirements that all interpolation schemes must satisfy are given by Table 5-2. Their best choice was the fourth order accurate interpolation scheme called the

partial hermite interpolation. It requires the evaluation of second and third order derivatives, which does require more data storage and large computation time. The higher order schemes like Lagrangian interpolation, which is sixth order accurate, requires three nodes to be on either side of the particle. The flow calculations are only second order accurate, unlike the spectral methods, and hence a high order scheme, which requires expensive interpolation is not justified.

Table 5-2 Some requirements of the interpolation function.

1	The interpolation weights must equal the nodal values $g(x^{(i)})=f(x^{(i)})$.
2	The sum of the interpolation weights must equal to one.
3	It may also be desirable to have g(x) to be continuous across cell boundaries as well. This will ensure that calculated particle velocities change smoothly without abrupt jumps as the particle crosses an interface.

Bi-linear and shape function interpolation were chosen based on the level of information they use. The shape function interpolation requires extra information on the derivative of fluid velocity at nodal positions. In both cases the molecule of interpolation is fixed to a single cell. It has been found that increasing the molecule size does not necessarily increase the order of accuracy of interpolation (Yeung and Pope 1988).

Here bi-linear interpolation scheme is given by the Equation 5-14.

Equation 5-14

$$v(x,y) = \sum_{j}\sum_{i} u(x_i, y_j) L_i(x) L_j(y)$$

where the fluid velocity v(x,y) at particle position (x,y) is approximated using the fluid velocity $u(x_i,y_j)$ at nodes in Cartesian directions (i,j). The basis functions $L_i(x)$, and $L_j(y)$ are given by Equation 5-15.

Equation 5-15

$$L_1(x_i) = \xi$$
$$L_2(x_i) = 1 - \xi$$
$$L_1(y_i) = \eta$$
$$L_2(y_i) = 1 - \eta$$

where (ξ,η) are the local coordinates of a cell whose values are between 0 and 1 and at the cell origin they are (0,0). The linear interpolation scheme is second order accurate. In this case interpolation error decreases asymptotically as $(\Delta x)^2$. The linear interpolation scheme

also satisfies the requirement 1 and 2 on Table 5-2. However it completely ignores the non-linear variations on length scales smaller than one grid spacing.

Equation 5-16

$$v(x,y)=\sum_{j}\sum_{i}\left(\begin{array}{l} u(x_i,y_j)H_i(x)H_j(y)+ \\ \dfrac{\partial u(x_i,y_j)}{\partial x}G_i(x)H_j(y)+ \\ \dfrac{\partial u(x_i,y_j)}{\partial y}H_i(x)G_j(y) \end{array}\right)$$

The second interpolation scheme is the shape function interpolation scheme, given by Equation 5-16, which is used in the Finite Element Method (Zienkiewicz 1979), where the basis functions $H_i(x)$, $H_j(x)$, $G_i(x)$, and $G_j(x)$, are given by Equation 5-17 and Equation 5-18.

Equation 5-17

$$H_0(x_i)=(1-\xi)^2(1+2\xi)$$
$$H_1(x_i)=\xi^2(3-2\xi)$$

Equation 5-18

$$G_0(x_i)=h(1-\xi)^2\xi$$
$$G_1(x_i)=h\xi^2(\xi-1)$$

The local coordinates (ξ,η) are the same as in linear interpolation, and 'h' is the cell size in respective coordinate directions. The shape function interpolation scheme is 4^{th} order accurate. In one-dimension, Shape function interpolation provides hermite interpolation. Hermite interpolation satisfies the conditions given in Table 5-2 and also ensures that a polynomial of least degree interpolates its derivative at nodal points. Hence not only the velocity but also its derivatives are continuous at cell nodes. This should allow much smoother particle motion across cell boundaries. In shape function interpolation a cubic shape function is fitted to satisfy the necessary conditions given in Table 5-2 such as that interpolation formula should provide the fluid velocity and its derivative at nodal locations.

The accuracy of interpolation is expected to increase by using a dense grid.

5.6 Conclusion

In this chapter reason and theory have been provided to make the particle tracking equations of motion appropriate for the aerosol particle tracking. The particle is assumed to be a spherical aerosol particle, which will be treated as a point mass particle in the computations. The Stokes darg was shown to be the dominant force on the pacticle. The coupling effects between the particle and the fluid is ignored. Various models for aerosol particle deposition are further discussed by (Hofmann 1996).

The order of accuracy of the interpolation scheme must be similar to that of the integration scheme. The Shape function interpolation with 4^{th} order Runge-Kutta satisfies this condition, and such a condition is known to maximise the overall accuracy of the trajectory calculation (Balachandar and Maxey 1989), (Yeung and Pope 1988).

Factors such as storage requirements, available memory and paging input/output will also affect the choice of the interpolation scheme, as well as the information available from the solver. If staggered grid velocities are available then more accurate interpolation scheme could be devised, for example using a node located at the centre of the interpolating cell.

5.7 References

Balachandar, S. and M. R. Maxey (1989). "Methods for evaluating fluid velocities in spectral simulation of turbulence." Journal of computational physics **83**: 96-125.

Batchelor, G. K. (1993). An introduction to fluid dynamics, Cambridge University press.

Cliff, R., J. R. Grace, et al. (1978). Bubbles, Drops and Particles. New York, Academic Press.

Fan, L.-S. and C. Zhu (1998). Principles of gas-solid flows, Cambridge University press.

Fuchs, N. A. (1964). The mechanics of aerosols, Pergamon Press.

Hofmann, W. (1996). "Modelling techniques for inhaled particle deposition: The state of the art." Journal of Aerosol Science **9**(3).

Maxey, M. R. and J. J. Riley (1983). "Equation for a small rigid sphere in a non-uniform flow." Physics of fluid **26**(4): 883-889.

Rudinger, G. (1980). Fundamentals of gas-particle flow, Elsevier Scientific.

Saffman, f. (1965). "The lift on a small particle in a slow shear flow." Journal of Fluid Mehanics **22**: 385.

Soo, S. L. (1990). Multiphase fluid dynamics, Science Press, Beijing.

Yeung, P. K. and S. B. Pope (1988). "An algorithm for tracking fluid particles in numerical simulations of homogeneous turbulence." Journal of computational physics **79**: 373-416.

Zhang, L., A. Bahman, et al. (1996). "Inertial and intersecptal deposition of fibres in a bifurcatng airways." Journal of Aerosol Science **9**(3): 419-430.

Zhu, C. and S. L. Fan (1998). Principles of Gas-Solid Flows, Cambrudge University Press.

Zienkiewicz, O. C. (1979). The finite element method. New York, McGrw-Hill**:** 737.

6 Particle Tracking on a Structured Grid

6.1 Introduction

In order to calculate the particle deposition on the walls of a bifurcation a particle-tracking program was written. The particle tracking process is basically a procedure to track a particle within a cell centred grid. In such a process the particle is located within a cell and fluid properties at the particle position are interpolated from the surrounding nodes. The particle time advance scheme is used to solve particle equations of motion and increment the particle position. When the particle escapes the current control volume a search and locate algorithm was initiated to locate the particle containing cell. There are two means of locating a particle on a structured grid. One way is to carry out the search-and-locate procedure on the computational domain and the other is to carry out the procedure in the physical domain. Each method in turn will be investigated in section 6.2. The particle locating algorithm uses cell number to identity each cell and a data structure was created such that the cell number can be used to access all necessary information required by the tracking process. The data structure used in the CFD computation was consulted to create the new data structure for particle tracking. The data structure used by CFD, in general, are explained in section 6.3, and the chosen data structure used in the particle tracking will be derived from it. All requirements of the particle tracking algorithm will be explained in 6.4. Then in section 6.5 each part of the particle tracking process is validated and the validation results will be presented. A series of test cases will be run to test firstly the fluid velocity interpolation at particle position and integration of particle equation of motion by comparing the calculated trajectory with exact trajectory from analytical solutions. Then particles moving through a three-dimensional multi-block domain will be tested. Finally on section 6.6 particle tracking results on two dimensional double bifurcation will be presented, including particle deposition results.

6.2 Search and locate process

The idea of locating a particle within a Control Volume (CV) can be implemented either on computational space or on the physical space. In the physical space on an orthogonal Cartesian grid the CV containing particle may be found by comparing the particle position with the respective nodes of a CV's until a CV is found to contain the particle. Such a scheme would be inaccurate and inefficient on a non-orthogonal grid. But an efficient method can be created that uses the angle between the normal vectors to cell edges and particle relative vector with respect to cell edge origins. Detail description of the procedure will be given in section 6.2.2. The search procedure to locate a particle within a CV would

be iterative in nature where the particle is located on either sides of an edge for all the edges of the two dimensional CV. However in computational space, particle tracking will be carried out in a transformed space where it is necessary to convert the physical space variables into computational space variables. Computational space tracking eliminates the need for a search algorithm (thus iterations), as the grid is uniform in computational space. The choice of the two methods will depend on the efficiency of each scheme in terms of computer memory usage, accuracy of locating the particle within a cell and the speed at which a particle is located within a CV. Particle location procedure in computational space is explained next.

6.2.1 Solution procedure in Computational Domain

In computational space the global coordinate variables are (ξ,η) in two dimensions, and the local coordinates are the (ξ_0,η_0) as shown in Figure 6-1. Aside from these coordinate variables within each CV local parametric coordinate variables (m,n) are also defined. Considering just one parametric variable (equally applied to the other) it is assumed that m varies between –1 and +1 and related to the global variable ξ in the computational space by Equation 6-1.

Equation 6-1

$$m = 2(\xi - \xi_0) - 1$$

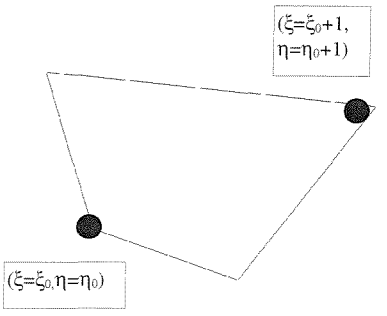


Figure 6-1 Shows the local (ξ_0,η_0) and global (ξ,η) coordinate variables in computational space

The global coordinates (ξ,η) are related to the index system (I,J) used to identify the nodes in domain, for example if $\xi_0 \equiv I$ then $\xi_0+1 \equiv I+1$. Hence there exists a convenient means of determining CV containing the particle. When a global coordinate ξ reach ξ_0+1, then particle is has moved to another CV. The global coordinate is calculated by solving the Equation 6-2.

Equation 6-2

$$\xi_{new} = \xi_{old} + \left(\frac{d\xi}{dt}\right) \times \Delta t$$

where Δt is the particle time step and $d\xi/dt$ is the contravariant velocity component of the particle. The transformed velocity components can be obtained by solving Equation 6-3.

Equation 6-3

$$\frac{dx}{dt} = \frac{\partial x}{\partial \xi}\frac{d\xi}{dt} + \frac{\partial x}{\partial \eta}\frac{d\eta}{dt}$$
$$\frac{dy}{dt} = \frac{\partial y}{\partial \xi}\frac{d\xi}{dt} + \frac{\partial y}{\partial \eta}\frac{d\eta}{dt}$$

where dx/dt and dy/dt are the velocity components of particle in physical space and $\partial x/\partial \xi$, $\partial x/\partial \eta$, $\partial y/\partial \xi$, and $\partial y/\partial \eta$ are the metric coefficients.

Equation 6-4 provides a means of relating the position of particle in physical space to the corresponding position in terms of local CV coordinates (m,n) (such equations are used in Finite Element studies for interpolating fluid properties).

Equation 6-4

$$\mathbf{x} = \frac{1}{4}\left[(1-\mathbf{m})(1-\mathbf{n})\mathbf{x}_1 + (1+\mathbf{m})(1-\mathbf{n})\mathbf{x}_2 + (1+\mathbf{m})(1+\mathbf{n})\mathbf{x}_3 + (1-\mathbf{m})(1+\mathbf{n})\mathbf{x}_4\right]$$

Since 'm' is related to ξ by Equation 6-1, computational space variables (ξ,η) can be substituted for (m,n) as in Equation 6-4. Then differentiating, (x,y), with respect to each global coordinate, (ξ,η), will give expressions for metric coefficients. To calculate the metric coefficients a non-linear system of equations has to be solved. The solution then can be substituted to Equation 6-3 together with velocity components of particle in physical space. A linear system of equations is formed, and by solving it, the particle contravarient velocity can be calculated. Finally Equation 6-2 can be solved to determine the new particle position. Note that once the particle position in physical space is available the local coordinates (m,n) with respect to each CV must be obtained by solving a set of non-linear equations (equations of form shown in Equation 6-4). Using the parametric coordinates the fluid velocity at particle positions can be interpolated.

A set of linear equations and a set of non-linear equations are solved to compute the contravariant velocity components of particle. Then Equation 6-2 is evaluated to obtain the

new position of particle in computational domain. Such a solution procedure for tracking particles in computational space was used by (Patankar and Karki 1999) and (Shirayama 1993). A major disadvantage of this procedure is that there are too many computations in solving sets of equations. Also computational error could creep into the final solution.

6.2.2 Solution procedure in Physical Domain

Consider a two dimensional CV, as shown in Figure 6-2, where the particle is located at some vector **P** from one of the nodes of, say at point A. A normal vector $\mathbf{N}_{e_n}$ is calculated on each edge vectors $\mathbf{e}_n$ where n=1,2,3,4. The normal vector is chosen such that it always points away from the CV.

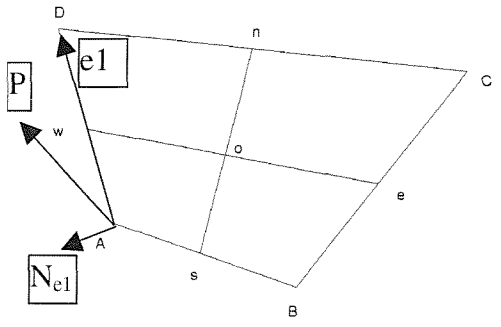


Figure 6-2 Arbitrary CV with nodes located at A, B, C, and D vertices. The direction of sides can be identified as north (n), south (s), east (e), and west (w)

Then the dot product of the particle vector **P** and $\mathbf{N}_{e_n}$ associated with an edge is given by Equation 6-5

Equation 6-5

$$\mathbf{N}_{e_n}.\mathbf{P} = |\mathbf{N}_{e_n}|.|\mathbf{P}|.\cos(\theta)$$

where θ is the angle between vectors. The cos(θ) is computed for each edge labelled w, e, s, and n (West, East, South, and North). A particle will be within the CV only if the angle between the vectors is equal to or greater than 90 degrees for all four edges (six faces in three dimensions).

The algorithm that computes cos(θ) to locate the particle may also needs to search for the particle among adjacent CV's after the particle position is incremented. If particle lies outside of an edge then a new CV is calculated. The property of the cos(θ) can be used to

calculate the new CV index. The sign of the cosine angle can be set to be either +1,or –1. A sign can be calculated as shown in Equation 6-6. When sign is positive when the particle is outside of an edge and the index in the edge normal direction must be increased or decreased by one.

Equation 6-6

$$\text{sign} = \frac{\cos\theta}{|\cos\theta|}$$

The particle displacement is made to be less than the grid spacing by adjusting the integration time step. Hence the particle will always move on to an adjacent CV. However the search and locate algorithm could iteratively find the particle containing CV eventually.

Note that once the particle is located within a CV in physical space the local coordinates of the particle must be obtained ((m,n) in section 6.2.1) to interpolate the nodal velocity to particle position, which means solving one set of non-linear equations (similar to Equation 6-4). A system of non-linear equations could be solved using the Newton's method, which linearizes the equations to second order accuracy. The resulting system of linear equations could be solved using matrix solution methods such as partial pivoting and Gaussian elimination.

(Chen 1997) came up with a similar search and locate algorithm where the sign of the normal vector was considered arbitrary (i.e. normal vector to an edge was calculated from arbitrary combination of edge vectors) and the decision to determine which side of the edge particle is located is determined by computing the sign of Equation 6-7.

Equation 6-7

$$\Omega_w = (\mathbf{r}_o.\mathbf{N}_w)(\mathbf{r}_p.\mathbf{N}_w)$$

where the $\mathbf{r}_o$ is a vector from CV origin to CV centre, and $\mathbf{r}_p$ is the particle relative vector from CV origin. It has been pointed out by (Zhou Q 1999) that Chen's scheme has too many computations and Zhou method only calculated one cross product and one dot product, as shown by Equation 6-8

Equation 6-8

$$\mathbf{L}_w = (-\mathbf{r}_w \times \mathbf{r}_p).\mathbf{k}$$

where **k** is the unit vector pointing out of the paper. In two dimensional quadrilateral cells, if four positive signs of $\mathbf{L}_d$, where d=w,e,n,s edges, are calculated then that means the particle is within the CV. To search for the particle within the four edges certain order, say anti-clockwise sense, was used. When one of the $\mathbf{L}_d$ is found to be negative the particle locating calculations took place on the cell adjacent to that edge. Hence further optimising the search and locate algorithm.

6.3 Data structures

The CFD software has its own data structure to represent their data. There are two main data groups; structured and unstructured. Fluent and CFX, two CFD software packages, uses unstructured and structured means of representing data respectively (determined by the data output files of each software). In section 6.3.1 and 6.3.2 structured single block, structured multi-block and unstructured data structures are looked in to in order to find a means of calculating a cell number and how to associate cell node fluid properties to the cell number. The ways of determining the cell number and how data is stored and accessed is explained.

When the particle reaches the boundary in two dimensions the intersection point along a boundary edge, must be calculated. The parametric coordinate 't' along an edge is calculated by the following equation.

Equation 6-9

$$t = \frac{\left|x_{p_2} - x_{p_1}\right|}{\left|x_{e_1} - x_{e_0}\right|} = \frac{\left|y_{p_1} - y_{p_2}\right|}{\left|y_{e_1} - y_{e_0}\right|}$$

where coordinate subscript 'e refer to an edge and 'p' refer to the particle. Two coordinate positions of the particle is required, one inside the flow domain and one outside, denoted by 1 and 2. The 't' would fall into the range of $\mathbf{0} \le |\mathbf{t}| \le \mathbf{1}$.

In three dimensions the particle wall disposition point is easily calculated by calculating the intersection point when the particle vector intersect the plane of the wall cell face.

6.3.1 Structured single block data structure

Data on a single block can be represented by the way it is swept in the program in I-direction first, J-direction second and K-direction last (in three dimensions) for a block with size (IN×JN×KN). The structured grid nodes can be numbered by the way these loops

work as given by Equation 6-10. The data on single block can be placed on to an array accessed by 'num'. Note here that (I,J,K) start with 0 and end with (IN-1,JN-1,KN-1).

Equation 6-10

$$num = I + (J \times IN) + (K \times IN \times JN)$$

The cell number adjacent to each cell can then be found with ease.

6.3.2 Structured multi-block data structure

Particle tracking on multi-block domain adds a little complexity to particle tracking algorithm. The complexity is due to the means of calculating the adjacent cell number to a cell at a block boundary. The arrangement of data on each block could start from different block origin. During the block design process it may be possible on some software packages to arrange the block origins such that they fall on some pre-determined pattern, as shown in Figure 6-3, where A is a cross-section of a pipe being blocked with an O-type block design, which is schematically shown in B. Notice that block origins shown by '+' in a circle are arranged in a pre-determined way. The arrangement of each block interface with respect to adjacent face must be identified in terms of data how the data are read in each block.

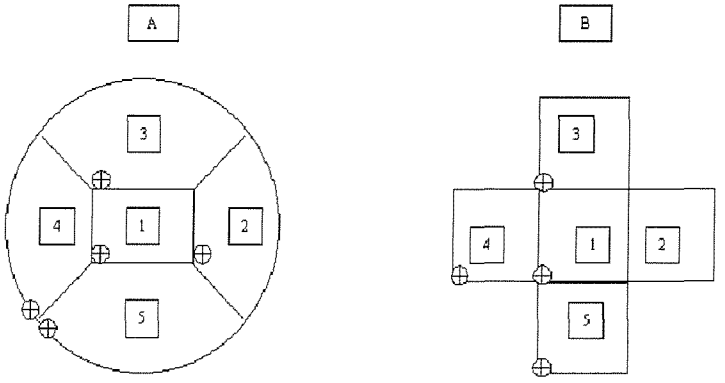


Figure 6-3 Pre-determined multi-block arrangement

The cells on each block is linked to the adjacent block by the way block faces are patched. Each block face could be numbered. If blocks are patched in some order, say blocks 1 and 5 in Figure 6-3, then each face will be given a unique sweep directions copied from the parent block sweep directions. However when joining blocks 4 and 3 the adjacent sweep directions may not be so obvious. Extra information may needs to be stored about the origin of the face with respect to the block origin can be used to find the sweep directions

on adjacent edges. Then using particle's (I,J,K) index and face sweep directions the particle index on the adjacent block could be determined.

Particle tracking on a cell centred grid poses problems near block boundaries, since the boundary is marked by grid nodes not cell centre nodes. For this reason all blocks are allocated ghost cells (obtained from dump files of CFD software which often contain ghost cell information). Once a particle is located within a ghost cell it may be within another block, it may be very close to the block boundary or it may be outside of the domain. In fact if the wall grid is fine enough a particle located within ghost cell on a cell centred grid can be considered to be outside of the domain. A particle entering a ghost cell will only travel back to interior domain if a strong force is exerted on the particle by the boundary layer. Of course the boundary cell could be very large and particle could move along the wall to other boundary cells. However to a first approximation it is assumed that cells near the wall are small and if a particle enters a ghost cell, the particle is considered to be deposited (assuming that there is no adjacent block to that face).

In multi-block domain the necessary information for particle tracking program are as follows.

- Size of each block
- Boundary conditions on each block face
- Adjacent block to each block face
- Orientation of the block face with respect to the block origin
- The sweep directions of the face

The programs that do particle tracking on multi-block domain are given in Appendix B. A three-dimensional test case tested is given in 6.5.3.

6.3.3 Unstructured data structure

The Fluent CFD software uses unstructured data. The dump file from this software writes data as per node and lists the node numbers separately as line format (i.e. when Tecplot data export option is used in Fluent). The line format basically lists two node numbers for all the edges found in the two-dimensional structured cell centred mesh where fluid property values are assigned. The idea is to re-construct the nodal arrangement into a structure where four nodes occupy a cell, which can be used for particle tracking on a structured grid. It was already discussed that particle tracking program need four vertices

surrounding it so that interpolation can be done and that it needs adjacent cell numbers to each cell edge. So the idea is to construct a data structure that can represent these data.

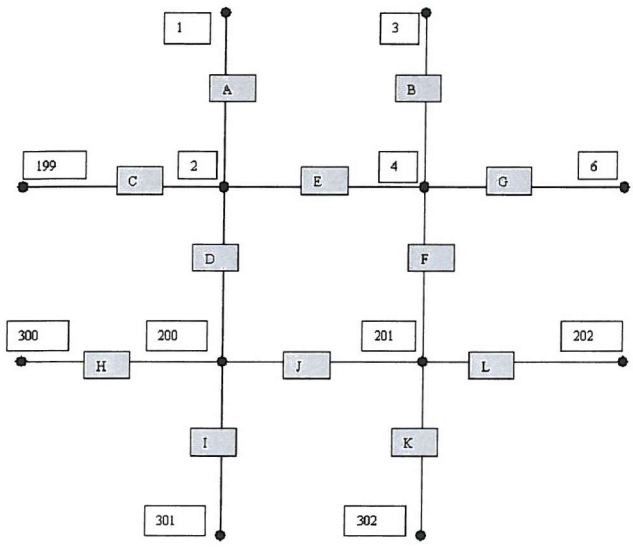


Figure 6-4 Unstructured ordering of nodes forming structured cells

The unstructured nodal numbers on structured grid are shown in Figure 6-4, written in line format. Fluent writes node numbers in two-dimensions such that the first two nodes connect an edge. Each node number can be used to access certain properties of the fluid. What is required here is to calculate cell numbers and the associated four nodes (in two-dimensions). The edge letters, shown in Figure 6-4, are in the order in which the Fluent dump the line format node numbers (connectivity). Using this information cell numbers can be calculated for a set of edges forming a closed control volume.

Each edge can be allocated a number (a letter in Figure 6-4) and the edge node numbers can be identified. Choose an edge, say 'D', and find all the edges that share one of its nodes, say node 2. Three edges are found, C, A and E. Place then into a group, G1. Choose the other node of edge 'D' i.e. node 200 and find the other three connecting edges, H, I and J. These go on group, G2. Now there will only be one edge that shares one node from the edges of G1 and the other node from edges of G2. This edge could easily be found from the edge database. Hence four edges enclosing a cell can be found and a cell number can be allocated to it. Note there will be some control volumes on the boundary of the domain where this algorithm to calculate cell number does not work. The CVs that cannot be formed on the boundary are the ghost CVs. It is assumed that particles entering such a control volume will deposit on the boundary.

The relevant programs to pre-compute cell number, adjacent cell number and multi-particle tracking are given in Appendix C.

6.4 Tracking process

The physical space tracking was chosen, because it is easier to implement. Only one set of non-linear equations has to be solved to obtain the local position of particle when global Cartesian location is given. The optimal scheme given by (Zhou Q 1999) was not used, instead the simple form of the tracking explained in section 6.2.2 was coded. Particle tracking in computational space could introduce additional errors in evaluating particle position since numerous transformations are used to transfer data back and forth and also in general it is more complex than physical space tracking.

The particle tracking on a single structured block is very convenient and it requires the least amount of pre-processing. However when the particle arrives at block boundaries in multi-block structured domain particle tracking process is slowed, since various decision loops are necessary to evaluate the particle containing cell. The code for particle tracking in single and multi-block was tested in three-dimensions and the results are shown in section 6.5.3. The particle tracking in two dimensional double bifurcation using post-processed Fluent data is given in section 6.6. Before giving the particle trajectories, it is important to validate the time Runge-Kutta time advance scheme used to solve the particle equations of motion and the Shape function interpolation scheme. In summary the particle search and locate scheme is conducted according to following steps given in Table 6-1.

Table 6-1 Summary of the particle tracking process

1	Particle is placed at some cell with known index (I,J) on a cell centred grid. Newly created data structure is used to access fluid properties and adjacnt cell number for a given particle containing cell number.
2	Starting with the edge $\mathbf{e}_w$ the cos(θ) is calculated, between edge normal and particle relative vector with respect to a given edge. In three dimension it would be the face normal and relative particle position vector with respect to the face origin that needs to be calculated.
3	The sign is calculated using Equation 6-6
4	If the sign is negative then step 1 and 2 is repeated until four negative signs are obtained, i.e. particle is located within a cell.
5	If the sign is positive the Index in the direction normal to the edge, is incremented. The adjacent CV is found and step 1 to 4 is repeated.
6	Initial velocity of particle is the interpolated fluid velocity from the surrounding known nodal data.
7	Time step is taken for each particle in turn and the time advance is done by forth order Runge-Kutta.
8	The tracking stops when the particle enters a ghost cell.

6.5 Validation process

The validation of the particle tracking process is actually to validate the interpolation and integration schemes. Two types of flow fields were taken to validate the particle tracks: a flow that spiral inwards (vortical flow) of (Murman and Powell 1989) and potential flow around a cylinder (Shirayama 1993). The size of the time step and the grid spacing could affect the local interpolation error, and this will be studied. The particle time steps were calculated using the cell size such that particle will land on a cell at least two times and do not jump over cells.

6.5.1 Fluid particle tracks in Vortical flow

The two-dimensional flow field is given by Equation 6-11, where a=-0.5 and b=3.0.

Equation 6-11

$$u(x,y) = ax - by$$
$$v(x,y) = ay + bx$$

A discrete flow field is computed by evaluating Equation 6-11 at each grid node whose size is 52×52. The domain dimensions are 1×1.

If the time step was fixed such that the magnitude of the particle trajectory vector will always be less than the magnitude of cell size, the particle tracking process would have higher efficiency. However it was found that in this test case as the particle trajectory vector is fixed the trajectory never seems to reach the centre, since along the trajectory the velocity reaches zero hence fixed particle trajectory will produce limit cycles around the centre of domain. The limit cycle size around the centre seems to be comparable to the size of the time step selected in terms of the cell size.

Using a small constant time step the error from the integration can be reduced. The accuracy of the Shape function interpolation and bi-linear interpolation could then be assessed. In fact when constant time step was used the trajectories seems to converge around the centre of the domain. The exact particle tracks were computed at each time step using Equation 6-11. The computed and the exact trajectories are plotted in Figure 6-5 and Figure 6-6 for bi-linear and shape function interpolations respectively.

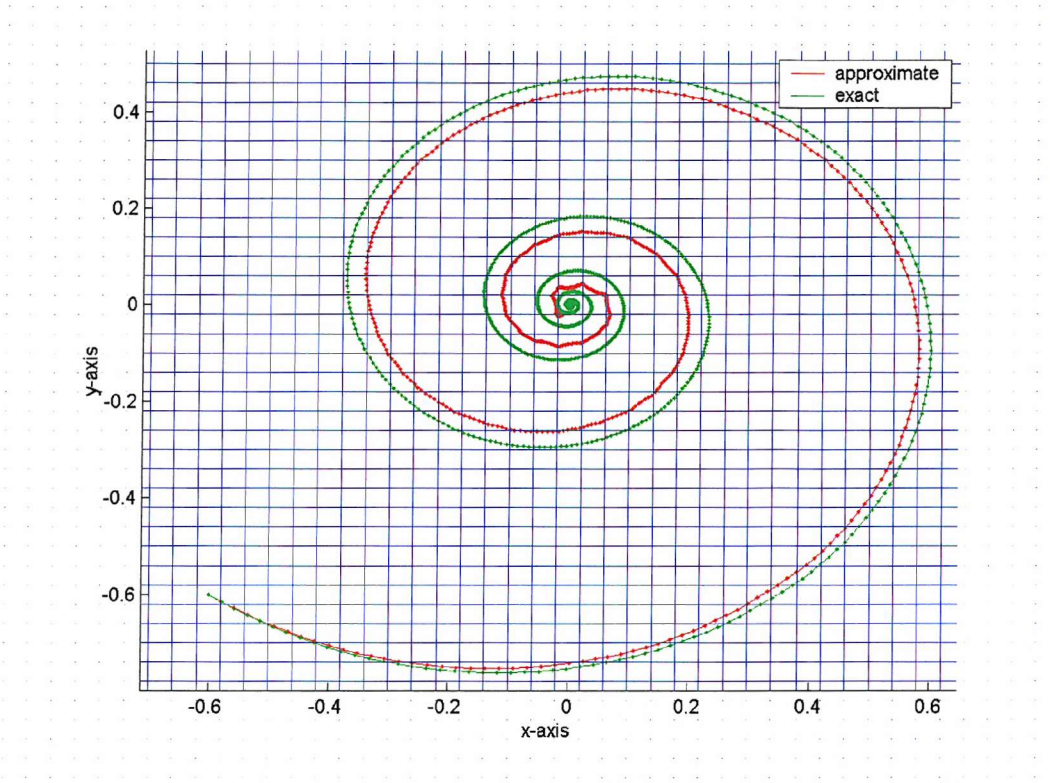


Figure 6-5 Computed particle tracks using bi-linear interpolation and exact tracks with constant particle displacements

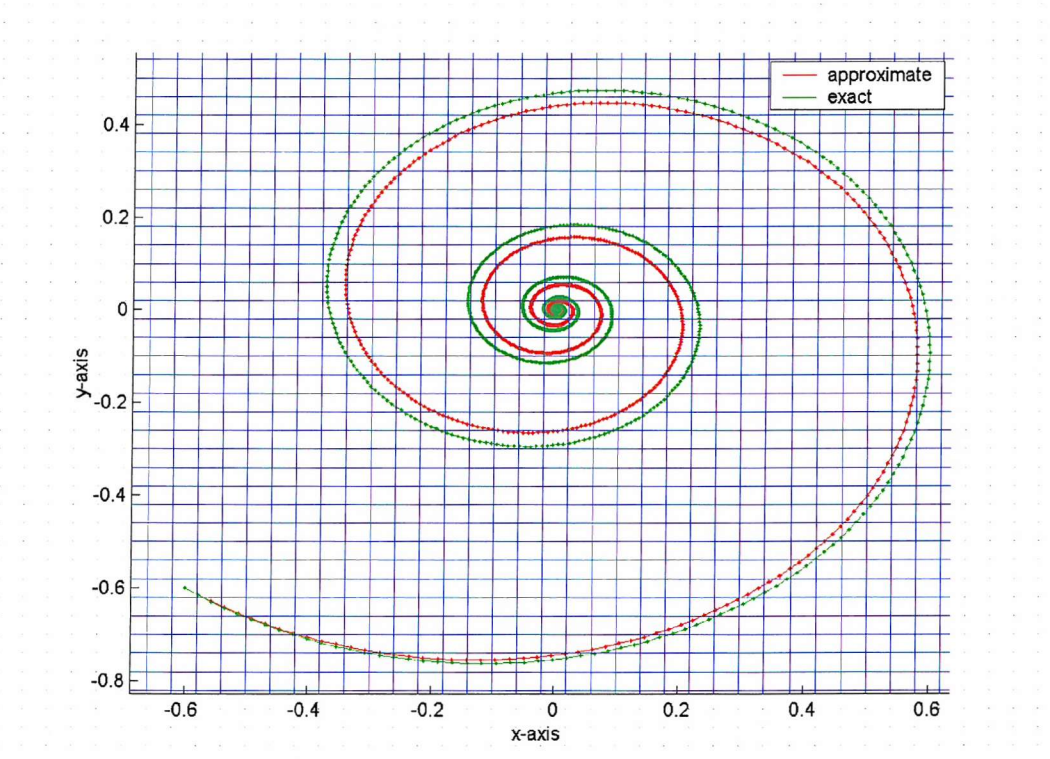


Figure 6-6 Computed particle tracks using Shape function interpolation and exact tracks with constant time steps

When bi-linear interpolation was used the maximum difference in calculated particle position and the exact particle position was 0.0367units and the maximum difference in velocity was 0.2579units, while for shape function interpolation it was 0.0290units and 0.0951units respectively. A central differencing scheme was used to calculate the velocity derivatives at nodal points, which is necessary for the shape function interpolation. Figure 6-7 shows that even though the overall trajectories seem quite similar, when shape function interpolation is used the trajectory become much smoother. This is made even more evident from Figure 6-7, where displacement error verses time step is shown for two interpolation schemes (i.e. the difference is particle position between computed and the exact at each time step). The time stepping error here is expected to be much smaller than the interpolation error.

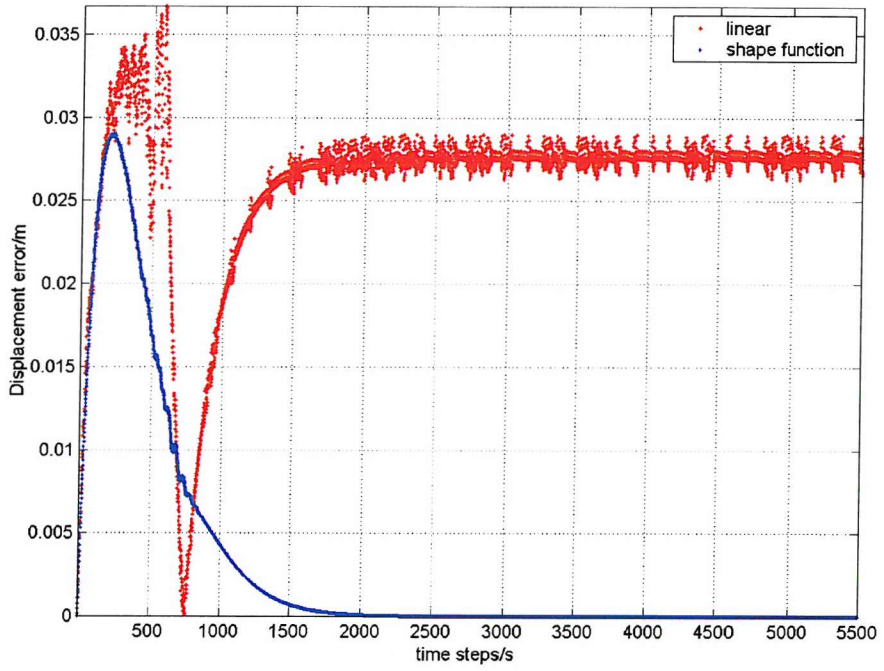


Figure 6-7 Absolute error between the exact and the computed resultant displacement at each time step

The Figure 6-7 shows that bi-linear interpolation has very oscillatory displacement errors, while for Shape function interpolation it is much smoother, which may be attributed to its use of velocity derivatives. Using an interpolation scheme whose interpolated velocity satisfies C^1 continuous condition, i.e. the velocity derivative is continuous across cell boundaries, allow the particle trajectory to be less oscillatory.

6.5.2 Fluid particle tracks in potential flow

The particle path integration scheme can be tested on a flow field that varies the curvature of the fluid particle trajectory. A potential flow field around a cylinder with some

circulation fits this criteria. In this case shape function interpolation was used throughout. The grid used to evaluate the fluid velocity and its derivatives are the same as in the previous test case. The flow field is described by the following equation.

$$u = U_0 + U_0 \frac{a^4}{r^4}\left(-(x-c_x)^2 + (y-c_y)^2\right) - \frac{\Gamma}{2\pi}\frac{(y-c_x)}{r^2}$$

$$v = -2U_0 \frac{a^4}{r^4}(x-c_x) + (y-c_y) + \frac{\Gamma}{2\pi}\frac{(y-c_x)}{r^2}$$

where $r = \sqrt{(x-c_x)^2 + (y-c_y)^2}$, U_0 is the uniform flow velocity, Γ represents the circulation, a is the radius of the cylinder and (c_x,c_y) is the centre of the cylinder. The parameters chosen were as follows: U_0=1.0, a=0.5, c_x=c_y=0.0, and Γ=6.0. The exact and computed trajectories are shown in Figure 6-8. It can be seen that as the particle path is highly curved the particle path deviates from the exact path. Even for very small time step the cumulative error does not change much, however when the grid density was increased the error was reduced.

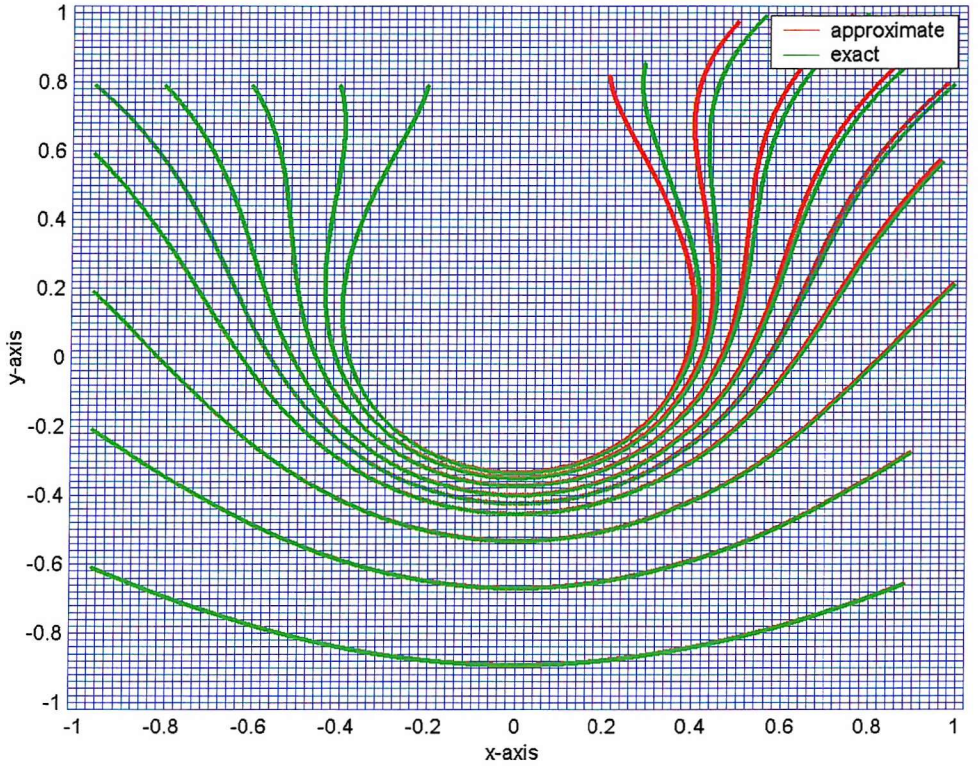


Figure 6-8 Computed and exact trajectories in the potential flow

(Shirayama 1993) used particle tracking in computational space rather than the physical space and the particle paths presented show similar deviation from the exact path lines.

6.5.3 Three dimensional fluid particle tracks

As a test case for three-dimensional particle tracking in multi-block structured data structure a 15 block 90° bend geometry was chosen, where the first five blocks forms an O-type grid at the entrance, and the second set of five blocks form the curved section of the pipe and the last set of five blocks form the exit straight section of the pipe. The block arrangement is shown in Figure 6-9. The uniform inlet flow was specified at a Reynolds number of 400 based on inlet diameter and inlet velocity. The outlet condition at the top was zero relative pressure. Three-dimensional flow field was computed using CFX commercial software with a laminar flow model.

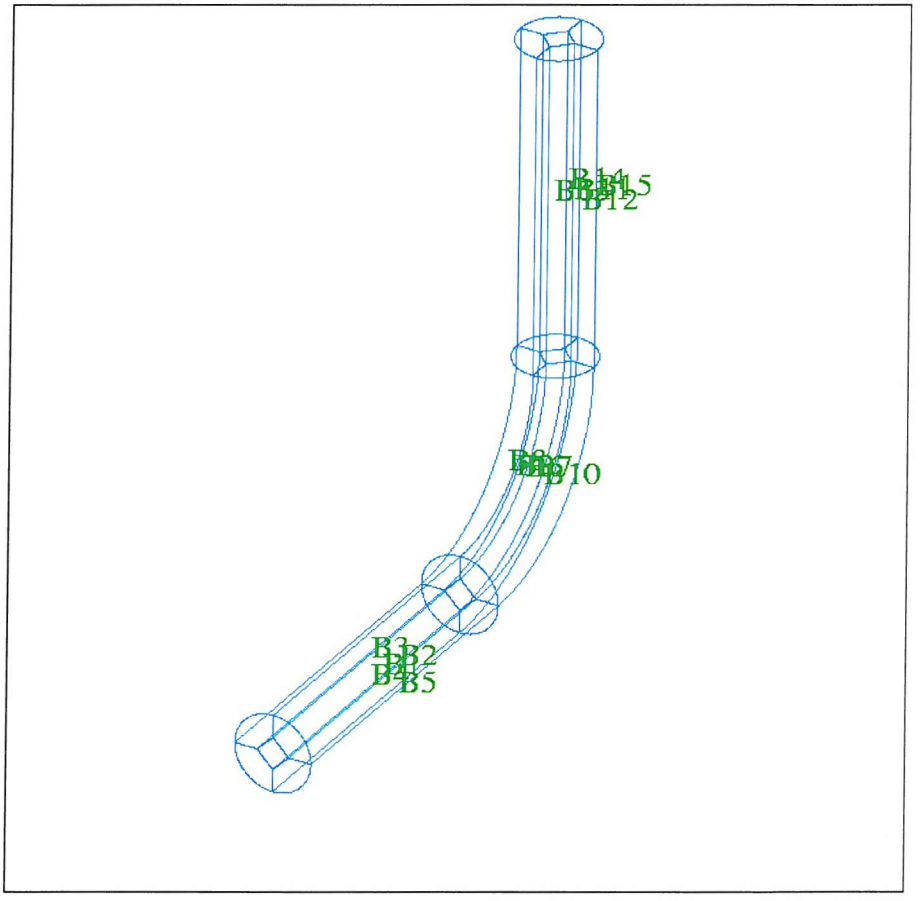


Figure 6-9 Three-dimensional multi-block structure

The four particles are introduced into the flow field and the resulting tracks are shown in Figure 6-10. It can be seen that the particles p4 and p3 (yellow and green) rotate with respect to each other as expected of the flow field in bent tube, where secondary flow field is generated. Note that in Figure 6-10 the inlet is at lower x-coordinate, and the black rings meant to show the circular cross-sections of the cylinder lying on the horizontal plane.

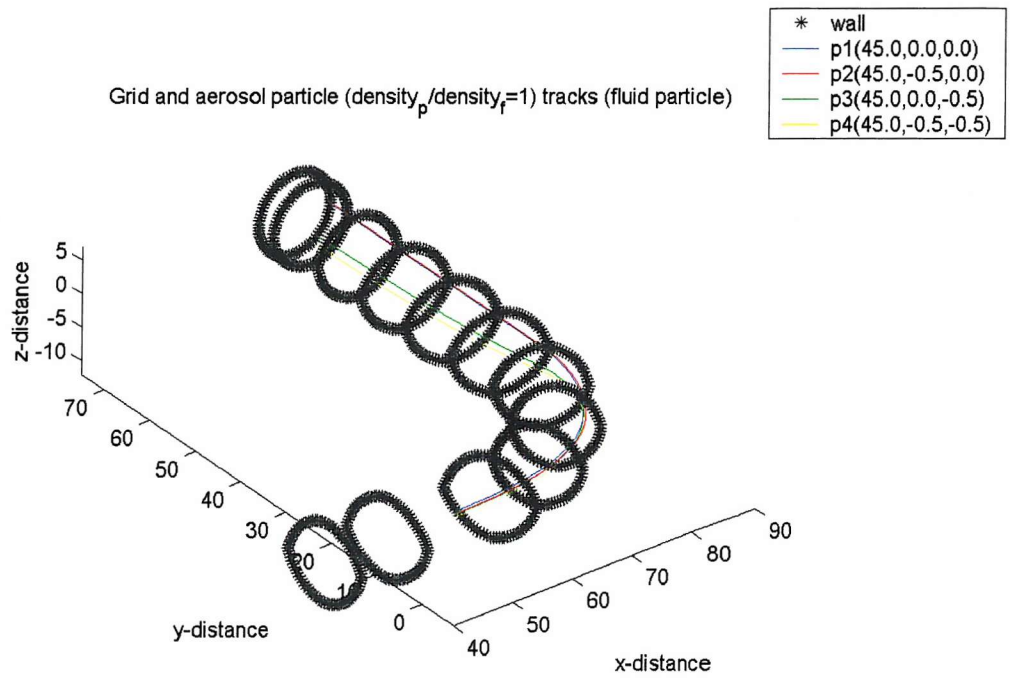


Figure 6-10 Four particle tracks in three-dimensional multi-block domain

6.6 Particle deposition

In section 6.5 the fluid particle tracks were tested for numerical integration and interpolation schemes. Using a smaller particle time step and shape function interpolation reasonably accurate particle tracks can be obtained. Now the particle equations of motion with Stokes' drag can be solved, and particles tracked on the two-dimensional 5^{th} to 7^{th} branch airway model. The Reynolds number of the flow is set to 600 and outlet pressures are all equal to 1 Pa. The Stokes number, Stk, given in Equation 6-12 is a non-dimensional quantity used to study particle deposition due to change in particle properties.

Equation 6-12

$$Stk = \frac{\rho_p d_p^2 U}{18 D \mu}$$

where ρ_p, d_p, U, D, and μ are particle density, particle diameter, inlet fluid velocity, diameter of the parent tube, and fluid viscosity. Stokes number range of between 0.02 to 0.12 was chosen, as shown in Table 6-2. The particle diameter was fixed at 3×10^{-6}m and the density of the particle was calculated for various Stokes numbers.

Table 6-2 Particle characteristics

Stk	0.02	0.04	0.06	0.08	0.1	0.12
Particle density	0.97183×10^{3}	1.9437×10^{3}	2.9155×10^{3}	3.8873×10^{3}	4.8592×10^{3}	5.831×10^{3}

A large number of particle tracks have to be simulated in order to compute the deposition efficiency, which is defined as the ratio of number of particles deposited on the wall of the geometry to the number that escaped through the outlet. Ten particle tracks for Stokes number of 0.02 and 0.12 are shown in Figure 6-11 and Figure 6-12 respectively. Initially they were positioned non-uniformly, where more particles occupy the core region. (Soo 1990) showed that in fully developed flow particles accumulate at the centre of the tube, hence the use of non-uniform distribution.

For the particle deposition study 91 particles were distributed non-uniformly as shown in Figure 6-13. The initial location of particles start at y=0 and end just below the outside wall at $y=1.75\times10^{-3}$. The simulated flow field is symmetrical about y=0. The deposition efficiency for the first bifurcation does not change significantly for any larger number of particles. The Figure 6-13 also shows the starting position of particles with Stokes number 0.12 that got deposited on the first and the second bifurcations in red colour. The particles deposited on the second bifurcation were originated from a position closer to the wall of the parent tube, while those got deposited on the first bifurcation came from the core region of the tube. Even with the highest Stokes number used, the area from which the released particles got deposited is very small. Figure 6-14 shows the deposition zones of particles in the first and the second bifurcation (small blue circles), which were released from the inlet at red locations.

The particle deposition efficiency was computed for the Stokes numbers shown in Table 6-2 and are graphed in Figure 6-15. The deposition efficiency for each bifurcation is calculated as a percentage by computing the total number of particles entering each bifurcation divided by the number that gets deposited in that branch. (Comer, Kleinstreuer et al. 2001b) used CFX, commercial CFD software, to carry out the flow simulations and particle tracking in a three dimensional double bifurcation model. (Comer, Kleinstreuer et al. 2001b) displayed results for flow Reynolds number of 500 with sharp carina. When comparing results the trend in deposition efficiency for the first bifurcation is quite similar for increasing Stokes number, and the exact values differs by about 1%. In three dimensional case the first bifurcation deposition values are greater, but the second bifurcation values are less. In three dimensional case the particle deposition seems to occur all around the second bifurcation apex region, while in two dimensional case the particle

deposition is seen only on one side of the second bifurcation Figure 6-14. In order to get a better deposition efficiency results for the second bifurcation in the two dimensional case more particles must be released, then the trend would be the similar as those given by (Comer, Kleinstreuer et al. 2001b).

When the flow condition is such that the mass flow rate out of all the outlets is similar, for example at the flow condition when the minimum pressure loss occurs across the bifurcations, the particle deposition would be highest. This is because at this condition the flow has the greatest overall curvature change. Large number of particle paths would have many curvature changes.

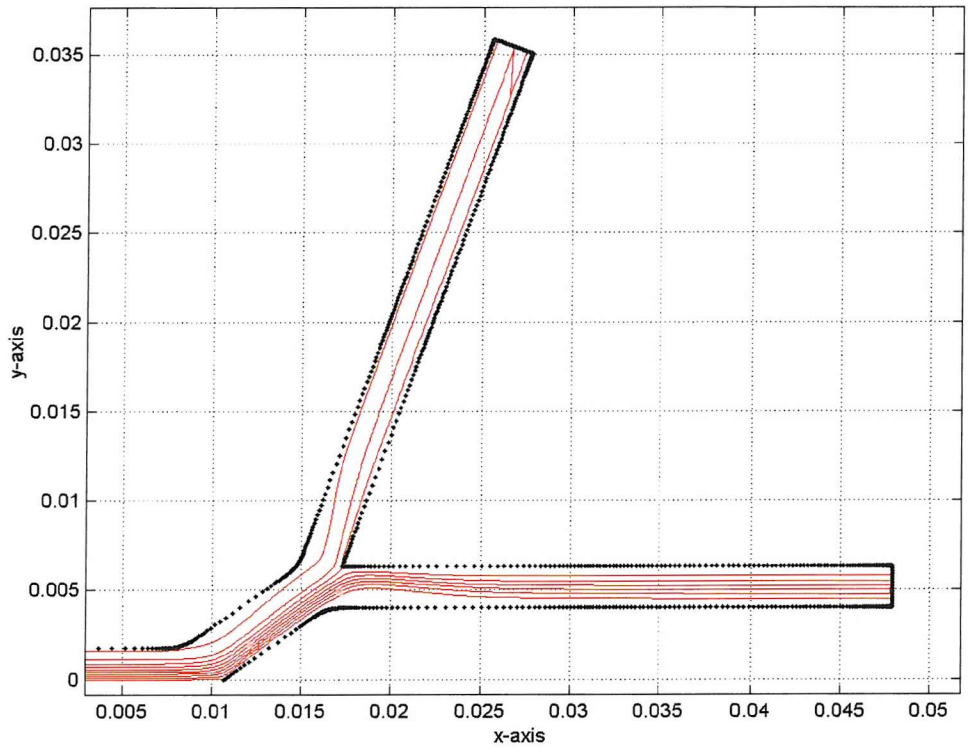


Figure 6-11 Particle tracks for Stk=0.02. Only 10 particle tracks are shown.

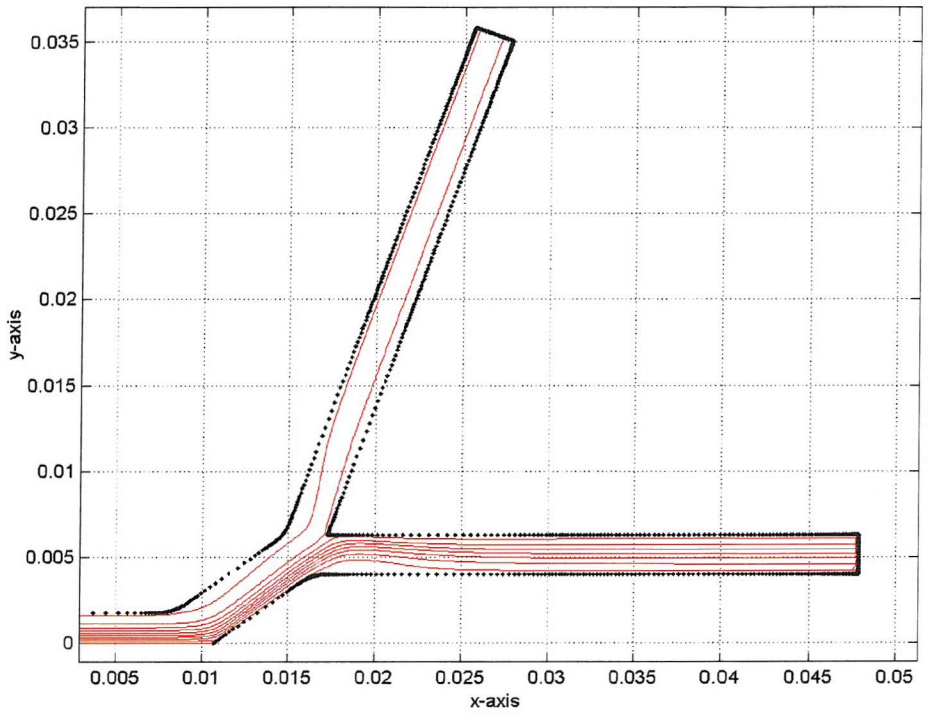


Figure 6-12 Particle tracks for Stk=0.12. Only 10 particle tracks are shown.

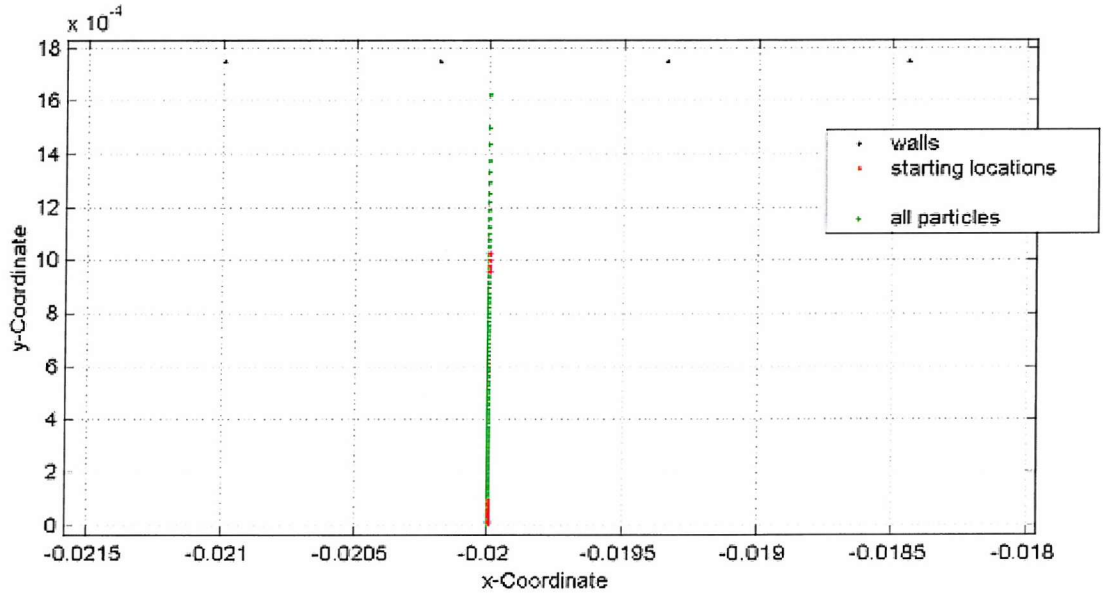


Figure 6-13 Non-uniform particle distribution with starting positions of first and second bifurcation deposition for Stk=0.12.

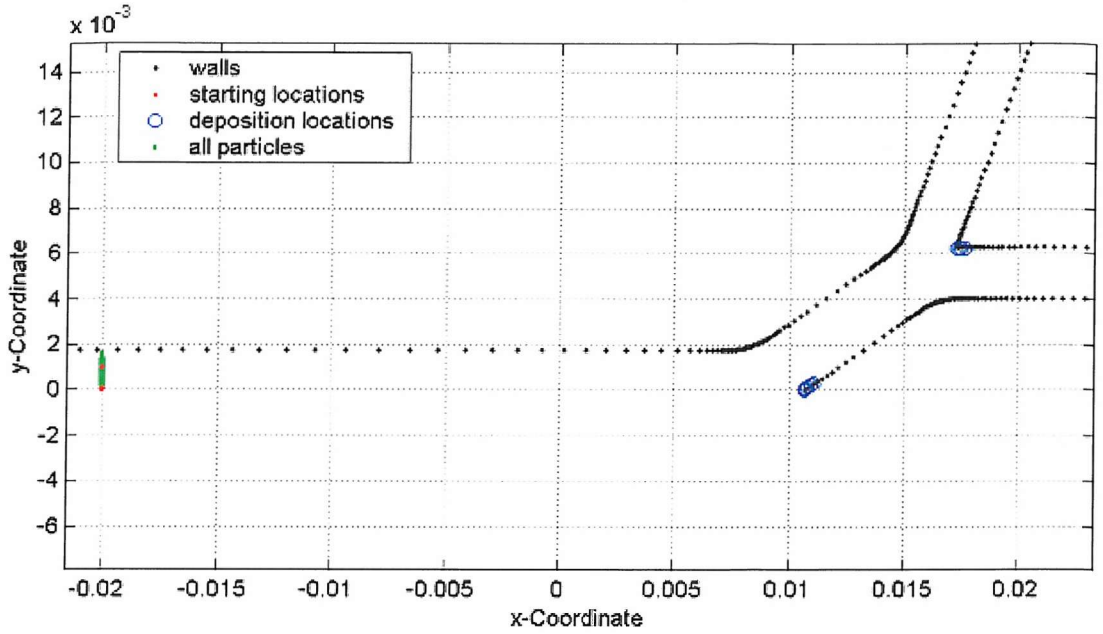


Figure 6-14 Non-uniform particle distribution with starting positions of first and second bifurcation deposition for Stk=0.12.

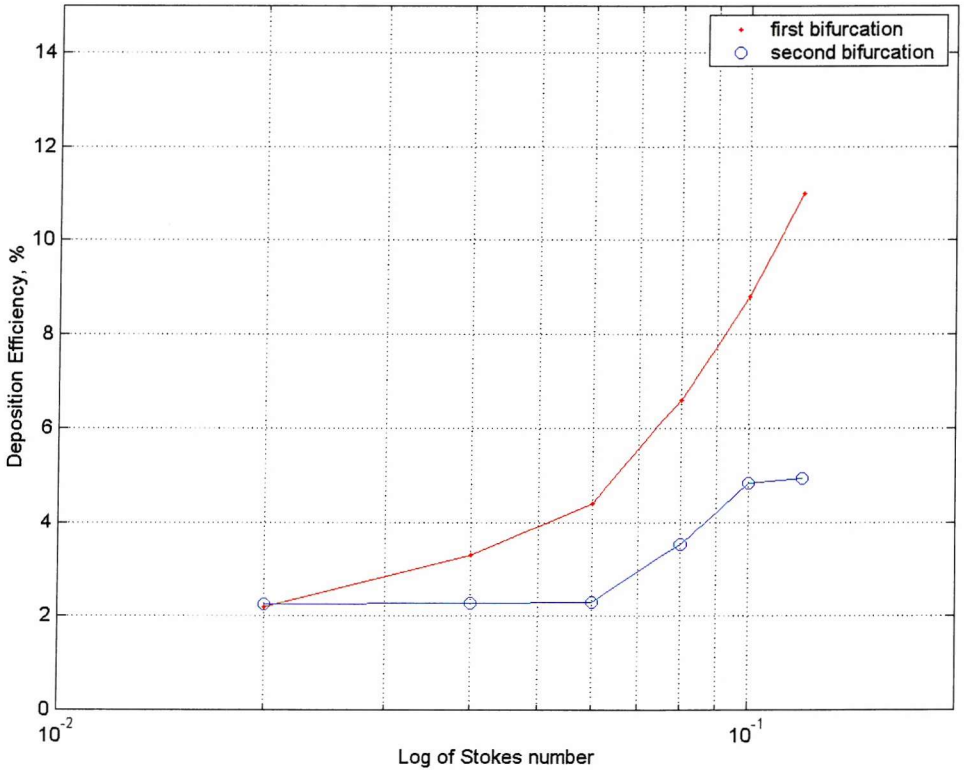


Figure 6-15 Particle deposition efficiency for a Reynolds number of 600

6.7 Conclusion

A particle search and locate procedure in physical space was designed to carry out particle tracking. It was realised that tracking in physical domain involved less computations,

simpler, and has less of an error in locating the particle. Also data structure was created to track particles using data represented by unstructured means in Fluent.

The tracking algorithm can easily be extended to three dimensions, which was proved by the three-dimensional test case. Particle tracking procedure is most efficient when its own data structure was used. The two dimensional test cases showed that linear interpolation introduced errors that included non-smooth particle tracks, and that smaller time steps with runge-kutta scheme was adequate for the particle tracking in laminar flow field. However higher order particle equation solvers, and finer grids may be necessary if more accurate particle paths are desired at the expense of computational cost. In all particle paths the relative velocity of particle with respect to the fluid velocity at particle position did not exceed a Reynolds number of 1, which implied that Stokes drag term was the most significant force experienced by the particle.

Recently there have been numerous publications on particle deposition in double- triple bifurcation three dimensional airway branches using commercial CFD software (Comer, Kleinstreuer et al. 2001b), (Zhang, Kleinstreuer et al. 2001) and (Comer, Kleinstreuer et al. 2000). They used in-build particle tracking routines within the CFD solver. No indications of the accuracy of the interpolation scheme or the integration scheme used were accessed. This study shows that interpolation scheme and particle equation solver play an important role in particle tracking.

Particle tracking in two dimensional flow field is only 1-2% different to that of the three dimensional case for double bifurcation flow. However the trend in particle deposition efficiency with Stokes number is the same. When mass flow rate out of the outlets are relatively equal the majority of the flow will have a large change in curvature, hence the deposition efficiency would be the highest.

Other application of the particle tracking could include gathering Lagangian statistics: two-particle velocity and distance correlation, mean square separations between particles; and influence of mean shear upon the downstream dispersion.

6.8 References

Chen, X. Q. (1997). "Efficient particle tracking algorithm for two-phase flows in geometries using curvilinear coordinates." Numerical Heat Transfer Part A **31**: 387-405.

Comer, J. K., C. Kleinstreuer, et al. (2000). "Aerosol transport and deposition in sequentially bifurcating airways." Journal of Biomechanical Engineering, (TASME) **122**: 152-158.

Comer, J. K., C. Kleinstreuer, et al. (2001b). "Flow structures and particle deposition patterns in double bifurcation airway models: Part 2 Aerosol transport and deposition." Journal of Fluid Mehanics **435**: 55-80.

Murman, E. M. and K. G. Powell (1989). "Trajectory integration in vortical flows." AIAA Journal **27**(7): 982-985.

Patankar, S. V. and K. C. Karki (1999). "Calculation of particle trajectories in complex meshes." Numerical Heat Transfer Part B **35**: 431-437.

Shirayama, S. (1993). "Processing of computed vector field for visualization." Journal of computational physics **106**: 30-41.

Soo, S. L. (1990). Multiphase fluid dynamics, Science Press, Beijing.

Zhang, Z., C. Kleinstreuer, et al. (2001). "Flow structure and particle transport in a triple bifurcation airway model." Journal of Fluids Engineering (Transactions of the ASME) **123**: 320-330.

Zhou Q, L. M. A. (1999). "An improved particle-locating algorithm for Eulerian-Lagrangian computations of two-phase flows in general coorinates." International journal of multi-phase flow **25**: 813-825.

7 Conclusions

Drug delivery through the human airways is a novel and effective means of drug delivery. The airway tree forms a very complex branching pattern, which can be approximated using the Weibel and Horsfield models. CFD is capable of using detailed geometrical descriptions of the geometry to carry out accurate simulations of the flow field (Comer, Kleinstreuer et al. 2001a), and (Comer, Kleinstreuer et al. 2000).

The most tedious part in the CFD process was the mesh generation on a complex geometry. It took a longer time than that required for the CFD solver in obtaiing a grid converged solution. Improving the turnaround time in mesh generation would increase the efficiency of the CFD process, since the solver performance was adequate. The performance of the solver was improved by the multi-block design technique, which improved the grid. However if faster turnaround time is needed then an unstructured grid must be used. A reliable mathematical description of the three-dimensional model is required to perform accurate mesh generation.

During grid independent study it was found that to resolve the pressure to an adequate accuracy (to check the pressure losses within the flow domain) a very fine grid was necessary. This was because the flow responded to very small pressure changes. As much as 1Pa was sufficient to alter the mass flow rate in one branch almost completely. The solution algorithms in most CFD software responds to large pressure changes by a compatible change in velocity, and in the same way for a small pressure change, the change in mass flow can be minute. However in the double bifurcation flow field a small change in pressure brought about a large change in mass flow rate. Therefore for accurate simulation the detailed pressure field has to be resolved using a fine grid, which in turn requires a large number of iterations to reach the residual limit. Multi-grid techniques in Fluent are useful to speed up the convergence process.

In this project a two dimensional 5^{th} to 7^{th} generation airway branch was used to carry out the flow simulation. CFD results showed complex flow structure within the bifurcation. There was new boundary layer formation, stagnation regions, separation regions, and expansion flow through diffuser sections. In one case the fixed inflow rate was varied using the same outlet pressures. The results showed as the fluid Reynolds number was increased the mass flow on distal branches became more and more biased. Under the normal breathing conditions the Reynolds number on the 5^{th} generation airway fell under a

Reynolds number of 369 (chapter 2 and (Pedley and Kamm 1991)). Hence the flow is only slightly biased in 7^{th} generation airways when it is completely unbiased in 5^{th} generation. In the second case when the inflow was fixed and outer most branch pressure was altered it was found that using very small pressure difference can be used to make the flow through all the branches equal. So during heavy breathing the lung has to perform work to create these pressure differences otherwise the lungs would not be well ventilated by the symmetrical branches in central airways.

There are large amount of publications on flow studies in airway bifurcations and also on blood flow in arterial bifurcations. At present there has beeng limited research relating to studying the pressure changes within airway bifurcation due to outlet pressure variations. Pressure variations in the airways must be studied in more detail experimentally first and then computationally in three dimensions. It is the small cascade effects of non-uniform pressure changes in airway cross-sections that force the air at correct proportions to ventilate the lung.

Particle tracking on a data structure of its own resolved many issues that arose due to using multi-block structured data structure. Search and locate algorithms in physical space was fast, and simpler than tracking in computational space. Multi-particle tracking in sequence, in general, using a small time step was slow. The particle path integration and interpolation of fluid velocity was thoroughly tested. The Runge-Kutta method and Shape function method gave good results. The deposition efficiency for the first bifurcation was only 1-2% off that predicted in three-dimensional simulations, which attributed to presence of secondary flow and lack of separation regions in three-dimensions.

In two-dimensional flow model the axial flow profiles are qualitatively accurate compared to the three-dimensional axial flow profiles. Hence particle tracking in two-dimensional flow field provided a means of testing and simulating actual three-dimensional behaviour.

It was reported that drug particles get deposited in upper airway branches and they have difficulty in reaching the lower airway branches. This has being verified by larger particle deposition zone in the first bifurcation than the second bifurcation shown in chapter 6. Those that got deposited came from certain entry zones at the parent branch. Therefore if particle were dispersed appropriately then there is more chances that these particles reaching outlets, hence enhancing the efficiency of drug delivery. The particles are less likely to be deposited when the flow through the branches are asymmetrical, when the overall flow curvature in all branches are small. Further study must now be done in three-

dimensions since the issues relating to geometric construction of airway bifurcation and mesh generation, solver options, particle search and locate algorithm, fluid particle interpolation, and particle path integration has being tested and known to work accurately and efficiently.

7.1 References

Comer, J. K., C. Kleinstreuer, et al. (2000). "Aerosol transport and deposition in sequentially bifurcating airways." Journal of Biomechanical Engineering, (TASME) **122**: 152-158.

Comer, J. K., C. Kleinstreuer, et al. (2001a). "Flow structures and particle deposition patterns in double-bifurcation airay models: Part 1 Air flow fields." Journal of Fluid Mehanics **435**: 25-54.

Finlay, W. H., C. F. Lange, et al. (2000). "Lung delivery of aerosolized dextran." Americal Journal of Critical Care Medicine **161**: 91-97.

Pedley, T. J. and R. D. Kamm (1991). Dynamics of gas flow and pressure-flow relationships. The Lung: Scientific Foundations. W. J. B. Crystal R G. New York, Raven Press: 995-1010.

8 Appendix A

8.1 Weibels Model A

Generation, z	Number per z, n(z)	Diameter, d(z)/ cm	Length, l(z)	Total cross-section, S(z)/cm^2	Total Volume, V(z)/cm^3	Accumulative Volume, /cm^3
0	1	1.81	12.0	2.54	30.50	30.5
1	2	1.22	4.76	2.33	11.25	41.8
2	4	0.83	1.90	2.13	3.97	45.8
3	8	0.56	0.76	2.00	1.52	47.2
4	16	0.45	1.27	2.48	3.46	50.7
5	32	0.35	1.07	3.11	3.30	54.0
6	64	0.28	0.90	3.96	3.53	57.5
7	128	0.23	0.76	5.10	3.85	61.4
8	256	0.186	0.64	6.95	4.45	65.8
9	512	0.154	0.54	9.56	5.17	71.0
10	1024	0.130	0.46	13.4	6.21	77.2
11	2048	0.109	0.39	19.6	7.56	84.8
12	4096	0.095	0.33	28.8	9.82	94.6
13	8192	0.082	0.27	44.5	12.45	106.0
14	16384	0.074	0.23	69.4	16.40	123.4
15	32768	0.066	0.20	113.0	21.70	145.1
16	65536	0.060	0.165	180.0	29.70	174.8
17	131072	0.054	0.141	300.0	41.80	216.6
18	262144	0.050	0.117	534.0	61.10	277.7
19	524288	0.047	0.099	944.0	93.20	370.9
20	1048576	0.045	0.083	1600.0	139.50	510.4
21	2097152	0.043	0.070	3220.0	224.30	734.7
22	4194304	0.041	0.059	5880.0	350.00	1084.7
23	8388608	0.041	0.050	11800.0	591.00	1675.0

9 Appendix B

9.1 Input parameters used in the particle tracking code

9.1.1 Multi-Block connectivity in file "mod.txt"

I. Global grid information

I. Number of different grid sizes.

If only one then use 1, do not use 0.

II. Number of grid sizes on the block in I, J, and K directions for each different grid size.

Note that grid size is cell number plus 3, i.e. ghost cells are included and one extra for the last grid point (as in Elmore restart ascii grid files).

Also the number of points in Elmore restart-variable files is cell numbers plus 2 (including ghost cells).

Note that Elmore ghost cells are just an algebraic displacement of the face cells, NOT therefore the adjacent cells. Hence it is assumed that the ghost cells in Elmore are close enough for the adjacent cells of the adjacent face to the current face.

For each block following information is necessary.

III. Boundary condition on each face

Boundary Condition	ID
Interface	0
Wall	1
Inlet	2
Outlet	3

IV. For each face interface

a. Adjacent block to each face

i. First block is 0.

ii. If no block in adjacent side then put –1.

Sweep direction of the face i j k	ID
0	No sweep
1	First direction
2	Second direction

If the face is not an interface then use a value of 6.

Sweep directions	ID
lo I	0
hi I	1
lo j	2
hi j	3
lo k	4
hi k	5
Non	6

9.2 Programs for three-dimensional particle tracking

9.2.1 Introduction

Following program were compiled in Visual C++ V6. The main program is **track.cpp**, it then calls the necessary functions. All the functions are separated out to files. The program produces output file, **part_proper.txt**.

9.2.2 bgather.cpp

```
//---------------Getting the block information-------------------
// Last updated 16/09/00

#include "tophead.h"

 extern int gridnum[BN+1][NVAR+1];
 extern int change[BN+1];
 extern int adj_block[TF];
 extern int bound[TF];
 extern int orien_plane[TF];

void block_info()
{
 char variable[5]="xyzu";
 int a,i,j,k,x;
 int diffblock;
 ifstream omod;
omod.open("mod.txt");
cout << "\n\t\tReading mod.txt for global grid properties...\n";
omod >> diffblock;
//cout << diffblock << endl;
for(i=0; i<diffblock; i++){
for(j=0; j<3; j++){
omod >> x;
gridnum[i][j]=x;
}
            }
            for(i=0; i<BN; i++){
                        omod >> x;
                        change[i]=x;
//cout << x << " ";     }
// collect the adjacent block to each face
for(i=0; i<TF; i++){
omod >> x;
adj_block[i]=x;
}
// collect the boundary condition on each face
for(i=0; i<TF; i++){
            omod >> x;
            bound[i]=x;
}
// collect i,j,k lo,hi or non values of each global face number
for(i=0; i<TF; i++){
            omod >> x;
            orien_plane[i]=x;
}
            omod.close();
/*
cout << "\nBoundary conditions are\n";
a=1;
for(i=0; i<BN; i++){
            for(j=0; j<6; j++){
cout << "Block=" << i << " face num=" << j << " and bound[" <<
a << "]=" << bound[a] << endl;
}
}
*/
}
```

9.2.3 calc_gridnum.cpp

```
// ----------------------------------------------------------------------
// Calculating the particle grid number
#include "tophead.h"
extern double part_position[NVAR];
//extern float * xgrid;
//extern float * ygrid;
//extern float * zgrid;
extern double xgrid[TEMP+1];
extern double ygrid[TEMP+1];
extern double zgrid[TEMP+1];
extern int gridnum[BN+1][NVAR+1];
extern int change[BN+1];
extern int grid_ijk[NVAR];
extern int cblock_num;

void calc_gridnum()
{
            int i,j,k,x;
            int
start_i=1,status_1=0,count_1=0,status_2=0,count_2=0;
            //initialise all to 0
            int collect1_i[YNUM*ZNUM]={0};
            int xnum,ynum,znum;
            x=change[cblock_num];
            xnum=gridnum[x][0];
            ynum=gridnum[x][1];
            znum=gridnum[x][2];
// set indices to interior cells
            cout << "\n In calc_gridnum\n";
            cout << "xnum=" << xnum << " ynum=" << ynum << "
znum=" << znum << endl;
            cout << "\ncblock_num=" << cblock_num << endl;
ofstream out1;
out1.open("i_input.txt");
//          use status to find if the sweeping direction is increasing
or decreasing
//          start_i = grid point before the particle
//          IN = total No. i-grid points
//          JN = total No. j-grid points
//          KN = total No. k-grid points
//          grid_x[] = x-coordinates of the grid points
//          part_posi[] = x,y,z position of the particle

//----------------------------------------------------------------------
--
//find jk-plane
//----------------------------------------------------------------------
--
for(k=1; k<znum-1; k++){
            for(j=1; j<ynum-1; j++){
// bottom : when j changes need to make sure grid point is behind
the particle position
if(count_2==1 && count_1!=1){
            // sweeping in increasing direction
// exit while loop if false statement
// below start_i is the last know sub-division where particle is in
front of grid vertex
            while(xgrid[start_i+(j*xnum)+(k*xnum*ynum)] >
part_position[0]){
                        // bottom : if particle not behind last
subdivion then reduce
            start_i=start_i-1;
            }
}
if(count_1==1 && count_2!=1){
                        // when sweeping in decreasing direction
            while(xgrid[start_i+(j*xnum)+(k*xnum*ynum)] <
part_position[0]){
            start_i=start_i-1;
            }
}
if(start_i<1){
            cout << "\n\n\t\tExit due to failure\n\n";
```

```
                out1.close();
 }
                // need to re-initialise the count and status
                status_1=0;
                count_1=0;
                status_2=0;
                count_2=0;
                        for(i=start_i; i<xnum-1; i++){
 if(xgrid[i+(j*xnum)+(k*xnum*ynum)] < part_position[0]){
 status_1=1;
 count_1++;
 }
 else{
 status_2=1;
 count_2++;
 }
 if(status_1+status_2 == 2){
                // top : particle trapped
                start_i = (i-1);
                // top : particle will be aft of start_i
                collect1_i[j+(ynum*k)]=start_i;
                // top : in 3D collect the jk-plane behind the particle

                // x-, y-, z-coordinates and i-subdivision behind he
particle
                out1 << xgrid[start_i+(j*xnum)+(k*xnum*ynum)] << "
";
                out1 << ygrid[start_i+(j*xnum)+(k*xnum*ynum)] << "
";
                out1 << zgrid[start_i+(j*xnum)+(k*xnum*ynum)] << "
";
                out1 << start_i << " " << j << " " << k;
                out1 << endl;

                i = xnum;
                // top : end the current i loop. This assumes that change
of sign
                // does not happen again in the same loop
 }
 else{
 // carry-on the i loop
 }
 } // end of i-for loop
 } // end of j-for loop
 } // end of k-for loop
 out1.close();
 cout << "\n\njk-plane found\n\n";
 // ---------------------------------------------------------------------------
 -------------
 // Use the jk-plane computed before to narrow down the position of
 the particle.
 // At the end in 3D find a line behind the particle
 //---------------------------------------------------------------------------
 ------------
 //find k line
 //---------------------------------------------------------------------------
 --------------
 int start_j=1,status_3=0,count_3=0,status_4=0,count_4=0;
 int temp_collect1;
 int collect2_i[ZNUM]={0};
 int collect2_j[ZNUM]={0};
 ofstream out2;
 out2.open("j_input.txt");
                k=1;
                j=1;
                temp_collect1=collect1_i[j+(k*ynum)];
                // top : temp_collect conatin the i_position of first grid
 point of the jk-plane grid points
 for(k=1; k<znum-1; k++){
                if(count_4==1 && count_3!=1){
                        // top : sweeping in increasing direction
                        // need to make sure that the i-poisiton is
 behind the particle for every change in j-position
                        // bottom : so use last j-division and new k to
 see

                temp_collect1=collect1_i[start_j+(k*ynum)];
 // bottom : exit while loop if false statement
                while(ygrid[temp_collect1+(start_j*xnum)+(k*xnum*y
 num)] > part_position[1]){
                start_j=start_j-1;
                temp_collect1=collect1_i[start_j+(k*ynum)];
                // top : assign new i-division of the jk-plane
                }
                }
                if(count_3==1 && count_4!=1){ // sweeping in
 decreasing direction

                temp_collect1=collect1_i[start_j+(k*ynum)];
                while(ygrid[temp_collect1+(start_j*xnum)+(k*xnum*y
 num)] < part_position[1]){
                start_j=start_j-1;
                temp_collect1=collect1_i[start_j+(k*ynum)];
                }
                }
                if(start_j<1){
                cout << "\n\n\t\tExit due to failure\n\n";
                out2.close();
                }
                status_3=0;
                count_3=0;
                status_4=0;
                count_4=0;
 for(j=start_j; j<ynum-1; j++){
                // bottom : collect first i-value on the jk-plane
                temp_collect1=collect1_i[j+(k*ynum)];
                if(ygrid[temp_collect1+(j*xnum)+(k*xnum*ynum)] <
 part_position[1]){
 status_3=1;
 count_3++;
                }
 else{          // ygrid[temp_collect+(j*xnum)] > part_position[1]
 status_4=1;
 count_4++;
 }
 if(status_3+status_4 == 2){            // particle trapped
                start_j = (j-1);        // particle will be aft of start_j
                collect2_j[k]=start_j; // in 3D this would be j points of a
 line in direction of k behind the particle
                temp_collect1=collect1_i[start_j+(k*ynum)];
                collect2_i[k]=temp_collect1;
                out2 <<
 xgrid[temp_collect1+(start_j*xnum)+(k*xnum*ynum)] << " ";
                out2 <<
 ygrid[temp_collect1+(start_j*xnum)+(k*xnum*ynum)] << " ";
                out2 <<
 zgrid[temp_collect1+(start_j*xnum)+(k*xnum*ynum)] << " ";
                out2 << temp_collect1 << " " << start_j << " " << k;
                out2 << endl;
                j = ynum; // end the current j loop. Assume only one
 change of sign per j-loop
 }
 else{
 // carry-on the j loop
 }
 } // end of j-for loop
 } // end of k-for loop
 out2.close();
 cout << "End of jk-plane\n\n";
 //---------------------------------------------------------------------------
 // find grid vertex point f the cv that particle reside on
 // ---------------------------------------------------------------------------
 int start_k=1,status_5=0,count_5=0,status_6=0,count_6=0;
 int collect3_k=0,temp_collect2, collect_z;
 ofstream out3;
 out3.open("k_input.txt");
                k=1;
 for(k=start_k; k<znum-1; k++){
                temp_collect1=collect2_i[k]; // temp_collect1 contain
 the i_positions of the k-curve
                temp_collect2=collect2_j[k]; // temp_collect2 conatin
 the j_positions of the k-curve

                if(zgrid[temp_collect1+(temp_collect2*xnum)+(k*xnu
 m*ynum)] < part_position[2]){
                status_5=1;
                count_5++;
                }
                else{
                status_6=1;
```

```
                count_6++;
                }
                if(status_5+status_6 == 2){         // particle trapped
                start_k = (k-1);          // particle will be aft of start_k
                collect_z=start_k; // k component behind the particle
                temp_collect1=collect2_i[start_k];
                temp_collect2=collect2_j[start_k];
                out3 <<
xgrid[temp_collect1+(temp_collect2*xnum)+(collect_z*xnum*ynu
m)] << " ";
                out3 <<
ygrid[temp_collect1+(temp_collect2*xnum)+(collect_z*xnum*ynu
m)] << " ";
                out3 <<
zgrid[temp_collect1+(temp_collect2*xnum)+(collect_z*xnum*ynu
m)] << " ";
                out3 << temp_collect1 << " " << temp_collect2 << " "
<< start_k;
                out3 << endl;
                k = znum; // end the current k loop. Assume only one
change of sign per k-loop
}
else{
// carry-on the k loop
}

} // end of k-for loop
out3.close();
cout << "End of k-curve\n\n";
//int grid_num=start_i+(start_j*(xnum-1))+(start_k*(xnum-
1)*(ynum-1));
grid_ijk[0]=temp_collect1;
grid_ijk[1]=temp_collect2;
grid_ijk[2]=start_k;
}
```

9.2.4 direction.cpp

```
// ----------------------------------------------------------------------------
// Determine which cell particle is at

#include "tophead.h"

                extern double relative[7][NVAR];
                extern int grid_ijk[NVAR];
                extern double normal[7][NVAR];
                extern int normal_num[7];
                extern int direct_num[7];
                extern int exit_cond[6][BN];
                extern int cblock_num;

int direction()
{
                int i,j,state_one=0,access_1,access_2;
                int within[7];
                double product=0.0;
                double cos_theta,mod_normal,mod_part;
                for(j=0; j<6; j++){ // for each face
                                product=0.0;
                                for(i=0; i<3; i++){ // for each variable

                product=product+(relative[j][i]*normal[j][i]);
                }
                // calculate modulus
mod_part=sqrt(pow(relative[j][0],2)+pow(relative[j][1],2)+pow(rel
ative[j][2],2));
mod_normal=sqrt(pow(normal[j][0],2)+pow(normal[j][1],2)+pow(
normal[j][2],2));
                // calculate dot product
                cos_theta=(product/(mod_part*mod_normal));
//              cout << "\nThe cos_theta=" << cos_theta << endl;
if(cos_theta > 0){
                                // cos_theta > 90 degrees, particle is out
within[j]=0;
}
else { // cos_theta <= 0 particle is within
                                within[j]=1;
}
//-----------------------------------------
} // end of j-loop
for(j=0; j<6; j++){ // for each face direction
                i=within[j];
                if(i==0){ // particle is out of face
                                switch (j){
                case 0:
                    grid_ijk[0]=grid_ijk[0]-1;
                                if(grid_ijk[0] < exit_cond[0][cblock_num]){
                                cout << "\n\nGone wrong West\n\n";
                                                grid_ijk[0]=grid_ijk[0]+1;
                                                // do not break
                                                }
                                                else{
                                                cout << "\nWest\n";
                                                }
                                                break;
                case 1:
                    grid_ijk[0]=grid_ijk[0]+1;
                                                if(grid_ijk[0] >
exit_cond[1][cblock_num]){
                cout <<"\n grid_ijk[0]=" << grid_ijk[0];
                cout << "\n\nGone wrong East\n\n";
                                                grid_ijk[0]=grid_ijk[0]-1;
                                                return 7;
                                                // do not break
                                                }
                                                else{
                                                cout << "\nEast\n";
                                                }
                                                break;
                case 2:
                    grid_ijk[1]=grid_ijk[1]-1;
if(grid_ijk[1] < exit_cond[2][cblock_num]){
cout << "\n\nGone wrong South\n\n";
                                                grid_ijk[1]=grid_ijk[1]+1;
                                                return 7;
                                                // do not break
                                                }
                                                else{
                                                cout << "\nSouth\n";
                                                }
                    break;
                case 3:
                    grid_ijk[1]=grid_ijk[1]+1;
if(grid_ijk[1] > exit_cond[3][cblock_num]){
cout << "\n\nGone wrong North\n\n";
                                                grid_ijk[1]=grid_ijk[1]-1;
                                                return 7;
                                                // do not break
                                                }
                                                else{
                                                cout << "\nNorth\n";
                                                }
                    break;
                case 4:
                    grid_ijk[2]=grid_ijk[2]-1;
                if(grid_ijk[2] < exit_cond[4][cblock_num]){
cout << "\n\nGone wrong Bottom\n\n";
                                                grid_ijk[2]=grid_ijk[2]+1;
                                                return 7;
                                                // do not break
                                                }
                                                else{
                                                cout << "\nBottom\n";
                                                }
                    break;
                case 5:
                    grid_ijk[2]=grid_ijk[2]+1;
if(grid_ijk[2] > exit_cond[5][cblock_num]){
cout << "\n\nGone wrong Top\n\n";
                                                grid_ijk[2]=grid_ijk[2]-1;
                                                return 7;
                                                // do not break
                                                }
                                                else{
                                                cout << "\nTop\n";
                                                }
                    break;
                default:
                    cout << "Unknown operator " << i << endl;
                    break;
                                }// end of switch
                                j=6;
```

```
                    state_one=0;
            }// end of if loop
            else{
//                  cout << "\n\nParticle is within\n";
                    state_one=9;
            }
}// end of j-loop
for(j=0; j<3; j++){
//          cout << "grid_ijk[" << j <<"]=" << grid_ijk[j] << " ";
}
return state_one;
}
```

9.2.5 face.cpp

```
// ---------------------------------------------------------------------------
// Determine which cell particle is at

#include "tophead.h"

        extern int grid_ijk[NVAR];
        extern int gfn[2];
        extern int bound[TF];
        extern int adj_block[TF];
        extern int change[BN+1];
        extern int gridnum[BN+1][NVAR+1];
        extern int orien_plane[TF];

int face()
{

        int i,j,temp;
        int xnum;
        int ynum;
        int znum;
        temp=gfn[0]; // current face
                switch (bound[temp]){
        case 0:
                                        // interface
            break;
                case 1:
             cout << "\nHit a wall! "<< endl;
             return 0;
        case 2:
                        cout << "\nAt inlet! "<< endl;
                        return 0;
        case 3:
                        cout << "\nAt outlet! "<< endl;
                        return 0;
        default:
             cout << "Unknown operator " << i << endl;
             return 0;
                }
// has confirmed that the face is an interface
// adjacent global face number
temp=adj_block[temp];
temp=change[temp];
xnum=gridnum[temp][0];
ynum=gridnum[temp][1];
znum=gridnum[temp][2];
cout << "\n The new xnum=" << xnum << " ynum=" << ynum <<
" znum" << znum;
        cout << endl;
//
temp=gfn[1];
//cout << temp << endl;
//cout << orien_plane[temp] << endl;
cout << "\nBefore";
cout << "\ngrid_ijk[0]=" << grid_ijk[0]<< " grid_ijk[1]=" <<
grid_ijk[1] << " grid_ijk[2]=" << grid_ijk[2] <<endl;
// find the hi/low of the adjacent face, make sure new grid_ijk
// points to interior cells not to ghost cells
// Note: index start from 0 and end at *num-1
switch(orien_plane[temp]){
        case 0:
                grid_ijk[0]=1;
                break;
        case 1:
                grid_ijk[0]=xnum-3;
                break;
        case 2:
                grid_ijk[1]=1;
                break;
        case 3:
                grid_ijk[1]=ynum-3;
                break;
        case 4:
                grid_ijk[2]=1;
                break;
        case 5:
                grid_ijk[2]=znum-3;
                break;
        case 6:
                cout << "\n Previous steps interface has not
been confirmed\n\n";
                return 0;
        default:
    cout << "Unknown operator "<< endl;
    return 0;
}
cout <<"\nAfter";
cout << "\ngrid_ijk[0]=" << grid_ijk[0]<< " grid_ijk[1]=" <<
grid_ijk[1] << " grid_ijk[2]=" << grid_ijk[2] << endl;
return 1;
}
```

9.2.6 grid_vertex.cpp

```
// ---------------------------------------------------------------------------
// Acquire CV vertices

#include "tophead.h"
        extern double point_loca[9][NVAR];
        extern double xgrid[TEMP+1];
        extern double ygrid[TEMP+1];
        extern double zgrid[TEMP+1];
        extern int grid_ijk[NVAR];
        extern int gridnum[BN+1][NVAR+1];
        extern int change[BN+1];
        extern int cblock_num;
        extern double mag_cell_cen;

void grid_vertex()
{
        int xnum,ynum,znum;
        int j,x;
        double cell_cen_x;
        double cell_cen_y;
        double cell_cen_z;
        double rel_cell_x;
        double rel_cell_y;
        double rel_cell_z;

        x=change[cblock_num];
        xnum=gridnum[x][0];
        ynum=gridnum[x][1];
        znum=gridnum[x][2];
        int x_divi, y_divi, z_divi;
x_divi=grid_ijk[0];
y_divi=grid_ijk[1];
z_divi=grid_ijk[2];
        cout << "\ni=" << grid_ijk[0] << " ";
        cout << "j=" << grid_ijk[1] << " ";
        cout << "k=" << grid_ijk[2] << endl;
        int zero, one, two, three, four, five, six, seven;
        zero=x_divi+(y_divi*xnum)+(z_divi*ynum*xnum);
        one=(x_divi+1)+(y_divi*xnum)+(z_divi*ynum*xnum);
two=(x_divi+1+xnum)+(y_divi*xnum)+(z_divi*ynum*xnum);
three=(x_divi+xnum)+(y_divi*xnum)+(z_divi*ynum*xnum);
four=x_divi+(y_divi*xnum)+((z_divi+1)*ynum*xnum);
five=(x_divi+1)+(y_divi*xnum)+((z_divi+1)*ynum*xnum);
six=(x_divi+1+xnum)+(y_divi*xnum)+((z_divi+1)*ynum*xnum);
seven=(x_divi+xnum)+(y_divi*xnum)+((z_divi+1)*ynum*xnum);
        // grid vertex information
        point_loca[0][0]=xgrid[zero];  // current
        point_loca[1][0]=xgrid[one];  // after
        point_loca[2][0]=xgrid[two];  // top after
        point_loca[3][0]=xgrid[three];  // top
        point_loca[4][0]=xgrid[four]; // current-next
        point_loca[5][0]=xgrid[five]; // current-next after
        point_loca[6][0]=xgrid[six]; // current-next top after
        point_loca[7][0]=xgrid[seven]; // current-next top
        point_loca[0][1]=ygrid[zero];  // current
```

```
            point_loca[1][1]=ygrid[one];  // after
            point_loca[2][1]=ygrid[two];  // top after
            point_loca[3][1]=ygrid[three];  // top
            point_loca[4][1]=ygrid[four]; // current-next
            point_loca[5][1]=ygrid[five]; // current-next after
            point_loca[6][1]=ygrid[six]; // current-next top after
            point_loca[7][1]=ygrid[seven]; // current-next top
            point_loca[0][2]=zgrid[zero];  // current
            point_loca[1][2]=zgrid[one];  // after
            point_loca[2][2]=zgrid[two];  // top after
            point_loca[3][2]=zgrid[three];  // top
            point_loca[4][2]=zgrid[four]; // current-next
            point_loca[5][2]=zgrid[five]; // current-next after
            point_loca[6][2]=zgrid[six]; // current-next top after
            point_loca[7][2]=zgrid[seven]; // current-next top
// cell centers
cell_cen_x=(1.0/8.0)*(point_loca[0][0]+point_loca[1][0]+point_lo
ca[2][0]+point_loca[3][0]+point_loca[4][0]+point_loca[5][0]+poin
t_loca[6][0]+point_loca[7][0]);
cell_cen_y=(1.0/8.0)*(point_loca[0][1]+point_loca[1][1]+point_lo
ca[2][1]

            +point_loca[3][1]+point_loca[4][1]+point_loca[5][1]+p
oint_loca[6][1]+point_loca[7][1]);
cell_cen_z=(1.0/8.0)*(point_loca[0][2]+point_loca[1][2]+point_lo
ca[2][2]+point_loca[3][2]+point_loca[4][2]+point_loca[5][2]+poin
t_loca[6][2]+point_loca[7][2]);
rel_cell_x=cell_cen_x-point_loca[0][0];
rel_cell_y=cell_cen_y-point_loca[0][1];
rel_cell_z=cell_cen_z-point_loca[0][2];
mag_cell_cen=sqrt(pow(rel_cell_x,2)+pow(rel_cell_y,2)+pow(rel_
cell_z,2));
}
```

9.2.7 makesure.cpp

```
// ------------------------------------------------------------------------
// Make sure one time which direction the particle might be and
then
// move the particle until it has cross the CV faces

#include "tophead.h"

void grid_vertex();
void side_vector();
int direction();

void makesure()
{
int state_one=0, check_one=0;
grid_vertex();
side_vector();
check_one=0;
do{
state_one=direction();
grid_vertex();
side_vector();
check_one++;
cout <<"\nMaking sure CV number " << check_one << " times\n";
}while(state_one==0);
}
```

9.2.8 move_check.cpp

```
// ------------------------------------------------------------------------
// Make sure one time which direction the particle might be and
then
// move the particle until it has cross the CV faces

#include "tophead.h"

extern int grid_ijk[NVAR];
extern double uVel[TEMP];
extern double vVel[TEMP];
extern double wVel[TEMP];
extern double part_velo[7];
extern double vel_f[NVAR];
extern double part_position[NVAR];
extern int gridnum[BN+1][NVAR+1];
extern int change[BN+1];
extern int cblock_num;
extern ofstream out;
extern int storage[5][7];

void runge();
void part_rel();
int direction();

void move_check()
{
            int check_one=0,state_one=0, num_cell;
            int x;
            int xnum,ynum,znum;
            x=change[cblock_num];
            xnum=gridnum[x][0];
            ynum=gridnum[x][1];
            znum=gridnum[x][2];
do{
            if(check_one>0){
// move particle using 4th order Runge-Kutta time
steppingnum_cell=grid_ijk[0]+grid_ijk[1]*(xnum-
1)+grid_ijk[2]*(xnum-1)*(ynum-1);
cout << "\nnum_cell=" << num_cell << endl;
cout << "\ni=" << grid_ijk[0] << " ";
cout << "j=" << grid_ijk[1] << " ";
cout << "k=" << grid_ijk[2] << " " << endl;
                                    vel_f[0]=uVel[num_cell];
                                    vel_f[1]=vVel[num_cell];
                                    vel_f[2]=wVel[num_cell];
                                    part_velo[0]=vel_f[0];
                                    part_velo[1]=vel_f[1];
                                    part_velo[2]=vel_f[2];
//                          cout << "\nvel_f[0]=" << vel_f[0]<< "
vel_f[1]=" << vel_f[1]<< " vel_f[2]=" << vel_f[2];
//                          cout << "\n In do loop\n\n";
                                    runge();
                                    part_rel();
                                    out << "\nWriting\n";
out << part_velo[0] << " " << part_velo[1] << " "<< part_velo[2]
<< " ";
out << part_velo[3] << " " << part_velo[4] << " "<< part_velo[5]
<< " ";
out << part_position[0] << " "<< part_position[1] << " "<<
part_position[2] << endl;
            }
            else{}
state_one=direction();
check_one++;
}while(state_one==9); // do 'till particle is not within CV
}
```

9.2.9 nameblock.cpp

```
// Define names of data files per block

#include "tophead.h"

            extern char block_name1[LEN];
            extern char block_name2[LEN];
            extern char block_name3[LEN];
            extern int cblock_num;

void nameblock()
{
            int a=0,b=0,i=0;
block_name1[0]='B';
block_name1[1]='l';
block_name1[2]='o';
block_name1[3]='c';
block_name1[4]='k';
a=0;
b=1;
block_name1[5]=48+a;
block_name1[6]=48+b;
            for(i=0; i<cblock_num; i++){
                        b++;
                        if(b==10){
                        b=0;
                        a++;
                        }
block_name1[5]=48+a;
block_name1[6]=48+b;
```

```
            }
 block_name1[7]='u';
 block_name1[8]='.';
 block_name1[9]='d';
 block_name1[10]='a';
 block_name1[11]='t';
 block_name1[12]='\0';
//-------------------------------------
 block_name2[0]='B';
 block_name2[1]='l';
 block_name2[2]='o';
 block_name2[3]='c';
 block_name2[4]='k';

 a=0;
 b=1;
 block_name2[5]=48+a;
 block_name2[6]=48+b;
          for(i=0; i<cblock_num; i++){
                    b++;
                    if(b==10){
                    b=0;
                    a++;
                    }
 block_name2[5]=48+a;
 block_name2[6]=48+b;
          }
 block_name2[7]='v';
 block_name2[8]='.';
 block_name2[9]='d';
 block_name2[10]='a';
 block_name2[11]='t';
 block_name2[12]='\0';
//-------------------------------------
 block_name3[0]='B';
 block_name3[1]='l';
 block_name3[2]='o';
 block_name3[3]='c';
 block_name3[4]='k';
 a=0;
 b=1;
 block_name3[5]=48+a;
 block_name3[6]=48+b;
          for(i=0; i<cblock_num; i++){
                    b++;
                    if(b==10){
                    b=0;
                    a++;
                    }
 block_name3[5]=48+a;
 block_name3[6]=48+b;
          }
 block_name3[7]='w';
 block_name3[8]='.';
 block_name3[9]='d';
 block_name3[10]='a';
 block_name3[11]='t';
 block_name3[12]='\0';
}
```

9.2.10 namegrid.cpp

```
// Define namea of data filea per block
// Laat updated 16/09/00

#include "tophead.h"

extern char grid_name1[LEN];
extern char grid_name2[LEN];
extern char grid_name3[LEN];
extern int cblock_num;

void namegrid()
{
int a=0,b=0,i=0;
 grid_name1[0]='g';
 grid_name1[1]='g';
 grid_name1[2]='r';
 grid_name1[3]='i';
 grid_name1[4]='d';
          a=0;
          b=1;
 grid_name1[5]=48+a;
 grid_name1[6]=48+b;
          for(i=0; i<cblock_num; i++){
                    b++;
                    if(b==10){
                    b=0;
                    a++;
                    }
          grid_name1[5]=48+a;
          grid_name1[6]=48+b;
          }
 grid_name1[7]='x';
 grid_name1[8]='.';
 grid_name1[9]='d';
 grid_name1[10]='a';
 grid_name1[11]='t';
 grid_name1[12]='\0';
//-------------------------------------
 grid_name2[0]='g';
 grid_name2[1]='g';
 grid_name2[2]='r';
 grid_name2[3]='i';
 grid_name2[4]='d';
          a=0;
          b=1;
 grid_name2[5]=48+a;
 grid_name2[6]=48+b;
          for(i=0; i<cblock_num; i++){
                    b++;
                    if(b==10){
                    b=0;
                    a++;
                    }
          grid_name2[5]=48+a;
          grid_name2[6]=48+b;
          }
 grid_name2[7]='y';
 grid_name2[8]='.';
 grid_name2[9]='d';
 grid_name2[10]='a';
 grid_name2[11]='t';
 grid_name2[12]='\0';
//-------------------------------------
 grid_name3[0]='g';
 grid_name3[1]='g';
 grid_name3[2]='r';
 grid_name3[3]='i';
 grid_name3[4]='d';
          a=0;
          b=1;
 grid_name3[5]=48+a;
 grid_name3[6]=48+b;
          for(i=0; i<cblock_num; i++){
                    b++;
                    if(b==10){
                    b=0;
                    a++;
                    }
          grid_name3[5]=48+a;
          grid_name3[6]=48+b;
          }
 grid_name3[7]='z';
 grid_name3[8]='.';
 grid_name3[9]='d';
 grid_name3[10]='a';
 grid_name3[11]='t';
 grid_name3[12]='\0';
}
```

9.2.11 part_rel.cpp

```
// ------------------------------------------------------------------------
// Calculating the direction vectors

#include "tophead.h"
//#include "variables.h"
          extern double point_loca[9][NVAR];
          extern double relative[7][NVAR];
          extern double part_position[NVAR];
 extern int storage[6][4];
```

```
// void storage_num();

void part_rel()
{
        int two;
        int i,j;
for(j=0; j<6; j++){ // for each face direction
                two=storage[j][1];
        for(i=0; i<3; i++){ // for each variable
relative[j][i]=part_position[i]-point_loca[two][i];
        }
}
}
```

9.2.12 reading.cpp

```
//-------------------------------------------------------------------------
//initialise flow variables
//-------------------------------------------------------------------------

#include "tophead.h"

extern int cblock_num;
extern int change[BN+1];
extern int gridnum[BN+1][NVAR+1];
extern int exit_cond[6][BN];
extern int bound[TF];
extern char grid_name1[LEN];
extern char grid_name2[LEN];
extern char grid_name3[LEN];
extern char block_name1[LEN];
extern char block_name2[LEN];
extern char block_name3[LEN];
//extern float * xgrid;
//extern float * ygrid;
//extern float * zgrid;
extern double xgrid[TEMP+1];
extern double ygrid[TEMP+1];
extern double zgrid[TEMP+1];
extern double uVel[TEMP];
extern double vVel[TEMP];
extern double wVel[TEMP];

void reading()
{
        int x,i,j,k,a;
        int xnum,ynum,znum;
        ifstream in1,in2,in3,in4,in5,in6;
        x=change[cblock_num];
        xnum=gridnum[x][0];
        ynum=gridnum[x][1];
        znum=gridnum[x][2];
cout << "\nxnum=" << xnum << " ynum=" << ynum << " znum="
<< znum << endl;
cout << "\nReading the grid vertex points ...";
in1.open(grid_name1);
        x=0;
 for(k=0; k<znum; k++){
        for(j=0; j<ynum; j++){
                for(i=0; i<xnum; i++){
                        in1 >> xgrid[x];
                        x++;
                }
        }
 }
in1.close();
in2.open(grid_name2);
        x=0;
 for(k=0; k<znum; k++){
        for(j=0; j<ynum; j++){
                for(i=0; i<xnum; i++){
                        in2 >> ygrid[x];
                        x++;
                }
        }
}
in2.close();
in3.open(grid_name3);
        x=0;
 for(k=0; k<znum; k++){
        for(j=0; j<ynum; j++){
                for(i=0; i<xnum; i++){
                        in1 >> xgrid[x];
                        in2 >> ygrid[x];
                        in3 >> zgrid[x];
                        x++;
                }
        }
 }
in3.close();
cout << endl << "... OK.\n";
//-------------------------------------------------------------------------
cout << "Reading in flow field variables ... ";

  in4.open(block_name1);
        x=0;
for(k=0; k<(znum-1); k++){
        for(j=0; j<(ynum-1); j++){
                for(i=0; i<(xnum-1); i++){
                in4 >> uVel[x];
                x++;
                }
        }
}
in4.close();
in5.open(block_name2);
  x=0;
for(k=0; k<(znum-1); k++){
        for(j=0; j<(ynum-1); j++){
                for(i=0; i<(xnum-1); i++){
                in5 >> vVel[x];
                x++;
                }
        }
}
in5.close();
in6.open(block_name3);
        x=0;
for(k=0; k<(znum-1); k++){
        for(j=0; j<(ynum-1); j++){
                for(i=0; i<(xnum-1); i++){
                in6 >> wVel[x];
                x++;
                }
        }
}
in6.close();
cout << "... OK\n";
// defining block boundaries
a=0;
for(i=0; i<BN; i++){
        for(j=0; j<6; j++){
if(bound[a]==0){ // if the face is interface
        if(j==0){
        exit_cond[0][i]=0;
        }
        if(j==1){
        exit_cond[1][i]=xnum-2;
        }
        if(j==2){
        exit_cond[2][i]=0;
        }
        if(j==3){
        exit_cond[3][i]=ynum-2;
        }
        if(j==4){
        exit_cond[4][i]=0;
        }
        if(j==5){
        exit_cond[5][i]=znum-2;
        }
}
else{// if the face is not interface
        if(j==0){
        exit_cond[0][i]=1;
        }
        if(j==1){
        exit_cond[1][i]=xnum-3;
        }
        if(j==2){
        exit_cond[2][i]=1;
```

```
            }
            if(j==3){
            exit_cond[3][i]=ynum-3;
            }
            if(j==4){
            exit_cond[4][i]=1;
            }
            if(j==5){
            exit_cond[5][i]=znum-3;
            }
}
a=a+1;
            }
} //end of first for loop
}
```

9.2.13 runge.cpp

```
// ------------------------------------------------------------------------
// Calculating the direction vectors

#include "tophead.h"
//#include "variables.h"

// Attempting to solve the equation
// dr/dt=v;
// dv/dt=(u-v)/tau;
//
// with boundary conditions of y(0)=1

extern double vel_f[NVAR];
extern double part_velo[7];
extern double part_position[NVAR];
extern double relax_factor;
extern double mag_cell_cen;

void runge()
{
double k1=0.0,k2=0.0,k3=0.0,k4=0.0,dep_var_dev=0.0,
dep_var=0.0;
double time_step=5.0/1.0,g=0.0; // time stepping and gravity
            int i;
            double delta_t;


            // for x-direction
//cout << "\n\nParticle x-position is =" << part_position[0]<< endl;
dep_var_dev=sqrt(pow(part_velo[0],2)+pow(part_velo[1],2)+pow(
part_velo[2],2));
delta_t=(1.0/30.0)*(mag_cell_cen/dep_var_dev);
            for(i=0; i<3; i++){
                        dep_var_dev=part_velo[i];
                        dep_var=part_position[i];
            dep_var=dep_var+delta_t*dep_var_dev;
            part_position[i]=dep_var;
            part_velo[i]=dep_var_dev;

part_velo[3+i]=(vel_f[i]-dep_var_dev); // diferences in velocity
            }// end of i loop
}
```

9.2.14 side_vector.cpp

```
// ------------------------------------------------------------------------
// Calculating the direction vectors

#include "tophead.h"

extern double OA[NVAR];
            extern double OB[NVAR];
            extern double point_loca[9][NVAR];
            extern double relative[7][NVAR];
            extern double part_position[NVAR];
            extern double normal[7][NVAR];
            extern int storage [6][4];

void side_vector()
{
            int i,j;
            int one,two,three,four;
// store some number to access CV points from respective arrays
            for(j=0; j<6; j++){ // for each face direction
                        one=storage[j][0];
                        two=storage[j][1];
                        three=storage[j][2];
                        four=storage[j][3];     // note that two and
four are all the same see storage_num.cc
            for(i=0; i<3; i++){ // for each variable
                        OA[i]=point_loca[one][i]-
point_loca[two][i];
OB[i]=point_loca[three][i]-point_loca[four][i];
relative[j][i]=part_position[i]-point_loca[two][i];
            }

                        // calculate the normal components in i,j,k
direction
            normal[j][0]=OA[1]*OB[2]-OA[2]*OB[1];
            normal[j][1]=OA[2]*OB[0]-OA[0]*OB[2];
            normal[j][2]=OA[0]*OB[1]-OA[1]*OB[0];
            } // end of j-loop
}
```

9.2.15 track.cpp

```
//  Particle position defined manually and then the CV number is
found
// Date 02/01/01

#include "tophead.h"
#include "functions.h"
#include "variables.h"

int
main()
{
            cout << "Which block does the particle occupy > ";
            cin >> cblock_num;
// collect grid domain information
            block_info();
// define the name of the grid file
            namegrid();
// define the name of the variable files for u and v
            nameblock();
            cout << grid_name1 << endl;
            cout << grid_name2 << endl;
            cout << grid_name3 << endl;
            cout << block_name1 << endl;
            cout << block_name2 << endl;
            cout << block_name3 << endl;
// initialise the grid variables
// ------------------------------------------------------------------------
            int x,i=0,j=0,k=0;
            int xnum,ynum,znum;
            int access_1;
            x=change[cblock_num];
            xnum=gridnum[x][0];
            ynum=gridnum[x][1];
            znum=gridnum[x][2];
            // grid sub-divisions of the chosen block
            cout << endl << "xnum=" << xnum << " ynum=" <<
ynum << " znum=" << znum << endl;
            // Read in grid data i.e. cell vertices
            // Note that Block* conatain cell vertex information
//-------------------------------------------------------------------
// Reading in the grid vertices and the flow variables
reading();
//-------------------------------------------------------------------
//-------------------------------------------------------------------
// Calculate the grid number number based on user input particle
position
calc_gridnum();
//-------------------------------------------------------------------
cout << "The particle containing CV orgin grid number is\n";
cout << "i=" << grid_ijk[0] << endl;
cout << "j=" << grid_ijk[1] << endl;
cout << "k=" << grid_ijk[2] << endl;
// if calc_gridnum.cc has predicted incorrectly guess
// which cv index particle might be
cout << "\nReset grid number\n";
grid_ijk[0]=46;
grid_ijk[1]=11;
grid_ijk[2]=11;
```

```
cout << "The particle containing CV orgin grid number is\n";
cout << "i=" << grid_ijk[0] << endl;
cout << "j=" << grid_ijk[1] << endl;
cout << "k=" << grid_ijk[2] << endl;
//-------------------------------------------------------------------------
int num_cell=grid_ijk[0]+grid_ijk[1]*(xnum-
1)+grid_ijk[2]*(xnum-1)*(ynum-1);
int
num_grid=grid_ijk[0]+grid_ijk[1]*xnum+grid_ijk[2]*xnum*ynum
;
cout << "The starting grid number is " << num_grid << endl;
cout << "The starting CV number is " << num_cell << endl;
cout << "The origin of CV x=" << xgrid[num_grid] << " y=" <<
ygrid[num_grid] << " z=" << zgrid[num_grid] << endl;
//-------------------------------------------------------------------------
//determine direction of particle
int state_one=0,state_two=0, check_one, check_three=0;
int count_one=0, count_two=0,xn,yn,zn;
// initialise particle velocity
part_velo[0]=uVel[num_cell];
part_velo[1]=vVel[num_cell];
part_velo[2]=wVel[num_cell];
cout << "\npart_velo[0]=" << part_velo[0]<< " part_velo[1]=";
cout << part_velo[1]<< " part_velo[2]=" << part_velo[2] << endl;
cout << "\npart_posi[0]=" << part_position[0];
cout << " part_position[1]=" << part_position[1];
cout << " part_position[2]=" << part_position[2] << endl;
double density_p=1*1.19, density_f=1.19, diameter=10*1e-5,
visco=1.82e-5, kine_visco;
double power;
        //assume that diameter of aerosol is 10 micro meters
        kine_visco=visco/density_f;
        power=pow(diameter,2);
relax_factor=(2.0/9.0)*(density_p/density_f)*(power/kine_visco);
 cout << "\nrelax_factor=" << relax_factor << endl;
grid_vertex();
side_vector();
check_one=0;
do{
state_one=direction();
grid_vertex();
side_vector();
check_one++;
cout <<"\nMaking sure CV number " << check_one << " times\n";
}while(state_one==0);
num_grid=grid_ijk[0]+grid_ijk[1]*xnum+grid_ijk[2]*xnum*ynum
;
cout << "The particle containing CV orgin grid number is\n";
cout << "i=" << grid_ijk[0] << " ";
cout << "j=" << grid_ijk[1] << " ";
cout << "k=" << grid_ijk[2] << " ";
cout << "num_grid=" << num_grid << endl;
        cout << "\n-------------------------------------------------";
        cout <<"\n Taking time steps\n";
int z=0;    // number of time steps
for(z=0; z<40000; z++){
cout <<"Step=" << z << endl;
//          do{
// make use of the CV origin. Determine all grid vertices of the
current cell,
// determine the side vectors corresponding to a particular face
side_vector();
// see if you can call storage from side_vector
check_one=0;
do{
                cout << "\n In do loop\n\n";
        if(check_one>0){
cout << "\nIn IF LOOP\n";
// exit conditions
cout << "West exit " <<exit_cond[0][cblock_num]<< endl;
if(grid_ijk[0]==exit_cond[0][cblock_num] && state_one==9){
        cout << "\n\nReached a boundary x - West Face\n";
        gfn[0]=0+6*cblock_num; // current
        access_1=gfn[0];
        ablock_num=adj_block[access_1];
        gfn[1]=1+6*ablock_num; // adjacent
                access_1=face();
                if(access_1==0){
                        return 0;
                }
                cblock_num=ablock_num;
                namegrid();
                nameblock();
                cout << grid_name1 << endl;
                cout << grid_name2 << endl;
                cout << grid_name3 << endl;
                cout << block_name1 << endl;
                cout << block_name2 << endl;
                cout << block_name3 << endl;
                reading(); // initilise new flow variables
        x=change[cblock_num];
        xnum=gridnum[x][0];
        ynum=gridnum[x][1];
        znum=gridnum[x][2];

}// first if statement
xn=xnum-2; // ghost cell CV origin
//cout << "\nxn=" <<xn << endl;
cout << "East exit " <<exit_cond[1][cblock_num]<< endl;
if(grid_ijk[0]==exit_cond[1][cblock_num] && state_one==9){
        cout << "\n\nReached a boundary x - East Face\n";
        gfn[0]=1+6*cblock_num; // current
        access_1=gfn[0];
        ablock_num=adj_block[access_1];
        gfn[1]=0+6*ablock_num; // adjacent
                access_1=face();
                if(access_1==0){
                        return 0;
                }
                cblock_num=ablock_num;
        cout << "\nCurrent block=" << cblock_num << endl;
                namegrid();
                nameblock();
                cout << grid_name1 << endl;
                cout << grid_name2 << endl;
                cout << grid_name3 << endl;
                cout << block_name1 << endl;
                cout << block_name2 << endl;
                cout << block_name3 << endl;
                reading(); // initilise new flow variables
        x=change[cblock_num];
        xnum=gridnum[x][0];
        ynum=gridnum[x][1];
        znum=gridnum[x][2];
}// end of first if
cout << "South exit " <<exit_cond[2][cblock_num]<< endl;
if(grid_ijk[1]==exit_cond[2][cblock_num] && state_one==9){
        cout << "\n\nReached a boundary y - South Face\n";
        gfn[0]=2+6*cblock_num; // current
        access_1=gfn[0];
        ablock_num=adj_block[access_1];
        gfn[1]=3+6*ablock_num; // adjacent
                access_1=face();
                if(access_1==0){
                        return 0;
                }
                cblock_num=ablock_num;
                namegrid();
                nameblock();
                cout << grid_name1 << endl;
                cout << grid_name2 << endl;
                cout << grid_name3 << endl;
                cout << block_name1 << endl;
                cout << block_name2 << endl;
                cout << block_name3 << endl;
                reading(); // initilise new flow variables
        x=change[cblock_num];
        xnum=gridnum[x][0];
        ynum=gridnum[x][1];
        znum=gridnum[x][2];
} // end of first if
yn=ynum-2; // ghost cell CV origin
cout << "North exit " <<exit_cond[3][cblock_num]<< endl;
if(grid_ijk[1]==exit_cond[3][cblock_num] && state_one==9){
        cout << "\n\nReached a boundary y - North Face\n";
        gfn[0]=3+6*cblock_num; // current
        access_1=gfn[0];
        ablock_num=adj_block[access_1];
        gfn[1]=2+6*ablock_num; // adjacent
                access_1=face();
```

```
                        if(access_1==0){
                                return 0;
                        }
                        cblock_num=ablock_num;
                        namegrid();
                        nameblock();
                        cout << grid_name1 << endl;
                        cout << grid_name2 << endl;
                        cout << grid_name3 << endl;
                        cout << block_name1 << endl;
                        cout << block_name2 << endl;
                                cout << block_name3 << endl;
                        reading(); // initilise new flow variables
                x=change[cblock_num];
                xnum=gridnum[x][0];
                ynum=gridnum[x][1];
                znum=gridnum[x][2];
        } // end of first if loop
        cout << "Bottom exit " <<exit_cond[4][cblock_num]<< endl;
        if(grid_ijk[2]==exit_cond[4][cblock_num] && state_one==9){
                cout << "\n\nReached a boundary z - Bottom Face\n";
                gfn[0]=4+6*cblock_num; // current
                access_1=gfn[0];
                ablock_num=adj_block[access_1];
                gfn[1]=5+6*ablock_num; // adjacent
                        access_1=face();
                        if(access_1==0){
                                return 0;
                        }
                        cblock_num=ablock_num;
                        namegrid();
                        nameblock();
                        cout << grid_name1 << endl;
                        cout << grid_name2 << endl;
                        cout << grid_name3 << endl;
                        cout << block_name1 << endl;
                        cout << block_name2 << endl;
                        cout << block_name3 << endl;
                        reading(); // initilise new flow variables
                x=change[cblock_num];
                xnum=gridnum[x][0];
                ynum=gridnum[x][1];
                znum=gridnum[x][2];
        } // first if statement
        zn=znum-2; // ghost cell CV origin
        cout << "Top exit " <<exit_cond[4][cblock_num]<< endl;
        if(grid_ijk[2]==exit_cond[5][cblock_num] && state_one==9){
                cout << "\n\nReached a boundary z - Top Face\n";
                gfn[0]=5+6*cblock_num; // current
                access_1=gfn[0];
                ablock_num=adj_block[access_1];
                gfn[1]=4+6*ablock_num; // adjacent
                        access_1=face();
                        if(access_1==0){
                                return 0;
                        }
                        cblock_num=ablock_num;
                        namegrid();
                        nameblock();
                        cout << grid_name1 << endl;
                        cout << grid_name2 << endl;
                        cout << grid_name3 << endl;
                        cout << block_name1 << endl;
                        cout << block_name2 << endl;
                        cout << block_name3 << endl;
                        reading(); // initilise new flow variables
                x=change[cblock_num];
                xnum=gridnum[x][0];
                ynum=gridnum[x][1];
                znum=gridnum[x][2];
        }// first if loop
        // move particle using 4th order Runge-Kutta time stepping
                num_cell=grid_ijk[0]+grid_ijk[1]*(xnum-
1)+grid_ijk[2]*(xnum-1)*(ynum-1);
        cout << "\nnum_cell=" << num_cell << endl;
        cout << "\ni=" << grid_ijk[0] << " ";
        cout << "j=" << grid_ijk[1] << " ";
        cout << "k=" << grid_ijk[2] << " " << endl;
                vel_f[0]=uVel[num_cell];
                vel_f[1]=vVel[num_cell];
                vel_f[2]=wVel[num_cell];
                part_velo[0]=vel_f[0];
                part_velo[1]=vel_f[1];
                part_velo[2]=vel_f[2];
                runge();
                part_rel();
        out << part_velo[0] << " " << part_velo[1] << " "<< part_velo[2]
        << " ";
        out << part_velo[3] << " " << part_velo[4] << " "<< part_velo[5]
        << " ";
        out << part_position[0] << " "<< part_position[1] << " "<<
        part_position[2] << endl;

                                cout << "\nPart_position ";
                                cout << part_position[0] << " "<<
        part_position[1] << " "<< part_position[2] << endl;
        cout << "\nIn IF LOOP\n";
        } // end of if( particle within cell
                                else{ }
        // check to see if the particle is within, if not new particle CV origin
        is found
        state_one=direction();
        // /*
        num_cell=grid_ijk[0]+grid_ijk[1]*(xnum-1)+grid_ijk[2]*(xnum-
        1)*(ynum-1);
        cout << "\nnum_cell=" << num_cell << endl;
        cout << "\ni=" << grid_ijk[0] << " ";
        cout << "j=" << grid_ijk[1] << " ";
        cout << "k=" << grid_ijk[2] << " " << endl;
                vel_f[0]=uVel[num_cell];
                vel_f[1]=vVel[num_cell];
                vel_f[2]=wVel[num_cell];
        //      cout << "\nvel_f[0]=" << vel_f[0]<< " vel_f[1]=" <<
        vel_f[1]<< " vel_f[2]=" << vel_f[2] << endl;
        // */
                if(state_one==7){
        return 0;
                                }
        //-------------------------------------------------------------------------
        check_one++;
        }while(state_one==9); // exit when particle is not within CV
                                cout << "\n Out do loop\n\n";
        num_cell=grid_ijk[0]+grid_ijk[1]*(xnum-1)+grid_ijk[2]*(xnum-
        1)*(ynum-1);
        cout << "\nnum_cell=" << num_cell << endl;
        cout << "\ni=" << grid_ijk[0] << " ";
        cout << "j=" << grid_ijk[1] << " ";
        cout << "k=" << grid_ijk[2] << " " << endl;
        }
        out.close();
        cout << "\n\nProgram END\n\n\n";
        return 0;
        }
```

9.2.16 functions.h

```
void block_info();
void namegrid();
void nameblock();
void calc_gridnum();
void grid_vertex();
void side_vector();
void runge();
void part_rel();
void reading();
void move_check();

int direction();
int face();

//void storage_num();
```

9.2.17 tophead.h

```
// include all the header files used by all programs


#include <iostream.h>
#include <math.h>
#include <fstream.h>
#include <iomanip.h>
#define LEN 13
```

```
#define BN 15
#define NVAR 3
#define XNUM 53
#define YNUM 23
#define ZNUM 23
#define TEMP XNUM*YNUM*ZNUM
#define TF BN*6
```

9.2.18 variables.h

```
#include "tophead.h"

// current and adjacent block number
 int cblock_num=0;
 int ablock_num=0;
 int gridnum[BN+1][NVAR+1];
 int change[BN+1];
 int grid_ijk[NVAR];
 int bound[TF]; // boundary condition type
 int gfn[2]; // global face number current and adjacent
 int adj_block[TF];
 int exit_cond[6][BN];

 // collect i,j,k, lo,hi or non locations of interface planes w.r.t. block
origin
 int orien_plane[TF];

 char grid_name1[LEN];
 char grid_name2[LEN];
 char grid_name3[LEN];
 char block_name1[LEN];
 char block_name2[LEN];
 char block_name3[LEN];

 double point_loca[9][NVAR];
// float * xgrid = new float [TEMP+1];
// float * ygrid = new float [TEMP+1];
// float * zgrid = new float [TEMP+1];
 double xgrid[TEMP+1];
 double ygrid[TEMP+1];
 double zgrid[TEMP+1];
 double vel_f[NVAR];
 double relax_factor;
 double OA[NVAR];
 double OB[NVAR];
 double relative[7][NVAR];
 double normal[7][NVAR];
 double uVel[TEMP];
 double vVel[TEMP];
 double wVel[TEMP];
 double mag_cell_cen;


 double part_velo[7];
// double part_position[NVAR]={3.4,0.5,0.2};

// * double part_position[NVAR]={40.4,4.0,0.0};
// double part_position[NVAR]={40.4,4.0,0.0};
double part_position[NVAR]={45.0,0.0,0.0};

// double part_position[NVAR]={3.47,-3.0700e-06,0.7480};
// double part_position[NVAR]={-0.0299,1.75e-3,0.00};
// double part_position[NVAR]={-0.01,9e-4,0.00};
// double part_position[NVAR]={-0.0299,9.1875e-4,0.00};

 // for ELMORE type coordinate system
 //int
storage[7][5]={{4,0,3,0},{2,1,5,1},{1,0,4,0},{7,3,2,3},{3,0,1,0},{5
,4,7,4}};
 int
storage[6][4]={{4,0,3,0},{2,1,5,1},{1,0,4,0},{7,3,2,3},{3,0,1,0},{5
,4,7,4}};

// for right handed coordinate system
// int
storage[7][5]={{7,4,0,4},{1,5,6,5},{0,4,5,4},{6,7,3,7},{3,0,1,0},{5
,4,7,4}};

/* // to determine +ve or -ve normal
 int comp_num[7][3]={{1,0},{0,1},{3,0},{0,3},{4,0},{0,4}};
 double compare[7];
 */
 int normal_num[7]={0,0,1,1,2,2};
 int direct_num[7]={-1,1,-1,1,-1,1};

ofstream out("part_proper.txt);
```

10 Appendix C

10.1 Pre-processing Fluent unstructured data – I

The first pre-processing step is to calculate the cell numbers. Following programs starting with **flu_man.cpp** as the main program output a file with the cell number and the associated vertex numbers.

10.1.1 find_six.cpp

```
// finds the six edges connected to a known edge

#include "things.h"
#include "flu_struct.h"

extern int count[2];
extern int ori_edge_num;
extern int ori_edge_vert[2];

void six_edges(int e0)
{
int i=0,j=0;
int v0=0,v1=0,v3=0,v2=0,k=0,q=0;
int numer[3]={0,1,0};

ori_edge_num=e0;
ori_edge_vert[0]=edges_all[e0].vert_num[0];
ori_edge_vert[1]=edges_all[e0].vert_num[1];

count[0]=0;
count[1]=0;
for(i=0; i<E; i++){
        for(j=0; j<2; j++){
        if(i==e0){
        // do nothing
        }
        else{
        if(edges_all[i].vert_num[j]==ori_edge_vert[0]){
        //cout << i << endl;
        v2=numer[j+1];
        v3=edges_all[i].vert_num[v2];
        edge01[k].number=i;
        edge01[k].vert_num[0]=ori_edge_vert[0];
        edge01[k].vert_num[1]=v3;
        k++;
        count[0]=count[0]+1;
        }
        if(edges_all[i].vert_num[j]==ori_edge_vert[1]){
        v2=numer[j+1];
        v3=edges_all[i].vert_num[v2];
        edge02[q].number=i;
        edge02[q].vert_num[0]=ori_edge_vert[1];
        edge02[q].vert_num[1]=v3;
        q++;
        count[1]=count[1]+1;
        }
        } // end of else
        } // end of vertices
        } // end of edges
}
```

10.1.2 Flu_man.cpp

```
// Read Tecplot files into ijk format

#include "things.h"
#include "struct.h"

int count[2]={0,0};
int ori_edge_num=0;
int ori_edge_vert[2]={0,0};
int last_edge_num=0;
int last_edge_vert[2]={0,0};
int cell_num=0;
void six_edges(int e0);
void print_fun();
void ini_read();
void store_cell();

int
main()
{

int i=0;
int count_cell=0;
// read connectivity integers
ini_read();
// find edges sharing chosen vertex, and their respective vertices
six_edges(i);

for(i=1; i<E; i++){
cout << "Trial edge number " << i << endl;
// find six edges attached to a given edge
if(count[0]!=3 || count[1]!=3){
        //cout << i << endl;
        six_edges(i);
        if(count[0]==3 && count[1]==3){
        store_cell();
        count[0]=0;
        count[1]=0;
        }
}
}
print_fun();
return 0;
}
```

10.1.3 ini_read.cpp

```
// Read file information

#include "things.h"
#include "flu_struct.h"

void ini_read()
{
int i=0,j=0;
ifstream in1;
in1.open("connect.dat");

cout << "Reading information\n";
for(i=0; i<E; i++){
in1 >> j;
edges_all[i].vert_num[0]=j;
in1 >> j;
```

```
edges_all[i].vert_num[1]=j;
// new fluent only has two entries
//in1 >> j;
}
cout << "DONE\n";
in1.close(); }
```

10.1.4 print_fun.cpp

```
// print information

#include "things.h"
#include "flu_struct.h"

extern int ori_edge_num;
extern int ori_edge_vert[2];
extern int cell_num;

void print_fun()
{
int i=0,j=0;
ofstream out1;
out1.open("cell_xy.dat");

for(i=0; i<cell_num; i++){
        out1 << i << "\t";
        for(j=0; j<4; j++){
        out1 << cells[i].nodes[j] << "\t";
        }
out1 << endl;
}
out1.close();
}
```

10.1.5 store_cell.cpp

```
// finds the six edges connected to a known edge

#include "things.h"
#include "flu_struct.h"

extern int ori_edge_num;
extern int ori_edge_vert[2];
extern int last_edge_num;
extern int last_edge_vert[2];
extern int cell_num;

void store_cell()
{
int i=0,k=0,j=0,v2=0,v3=0,q=0;
int numer[3]={0,1,0};
int f=0,g=0,h=0,p=0,l=0;

// find edges sharing edge01 vertex and edge02 vertex
for(k=0; k<3; k++){
for(i=0; i<E; i++){
for(j=0; j<2; j++){
        if(edges_all[i].vert_num[j]==edge01[k].vert_num[1]){
        v2=numer[j+1];
        v3=edges_all[i].vert_num[v2];
        for(q=0; q<3; q++){
        if(edge02[q].vert_num[1]==v3){
        last_edge_num=i;
        last_edge_vert[0]=edge01[k].vert_num[1];
        last_edge_vert[1]=edge02[q].vert_num[1];


        cells[cell_num].edges[0].number=0;
        cells[cell_num].edges[0].vert_num[0]=ori_edge_vert[
0];
        cells[cell_num].edges[0].vert_num[1]=ori_edge_vert[
1];

        cells[cell_num].edges[1].number=1;
        cells[cell_num].edges[1].vert_num[0]=edge01[k].vert
_num[0];
        cells[cell_num].edges[1].vert_num[1]=edge01[k].vert
_num[1];

        cells[cell_num].edges[2].number=2;
        cells[cell_num].edges[2].vert_num[0]=last_edge_vert[
0];
        cells[cell_num].edges[2].vert_num[1]=last_edge_vert[
1];

        cells[cell_num].edges[3].number=3;
        cells[cell_num].edges[3].vert_num[0]=edge02[q].vert
_num[1];
        cells[cell_num].edges[3].vert_num[1]=edge02[q].vert
_num[0];

        // store cell vertices in order
        cells[cell_num].nodes[0]=ori_edge_vert[0];
        cells[cell_num].nodes[1]=edge01[k].vert_num[1];
        cells[cell_num].nodes[2]=last_edge_vert[1];
        cells[cell_num].nodes[3]=edge02[q].vert_num[0];

if(cell_num==0){
cell_num++;
}
else{
l=cell_num;
for(f=0; f<l; f++){
p=0;
for(h=0; h<4; h++){
for(g=0; g<4; g++){
if(cells[cell_num].nodes[h]==cells[f].nodes[g]){
p++;
}
}
}
if(p==4){
f=E;
}
}// end of f loop


if(p!=4){
cell_num++;
}
}// end of else
}
}
}// if to find edge
}
} // loop all edges
}
}
```

10.1.6 flu_struct.h

```
// A header file of structures

#include "things.h"

struct ed{
        int vert_num[2];
        int number;
};
extern struct ed edge01[3];
extern struct ed edge02[3];
extern struct ed edges_all[E+1];
```

```
struct ce{
        struct ed edges[EDGE];
        int nodes[4];
};
extern struct ce cells[C+1];
```

10.1.7 struct.h

```
// A header file of structures

#include "things.h"

struct ed{
        int vert_num[2];
        int number;
};
struct ed edge01[3];
struct ed edge02[3];
struct ed edges_all[E+1];
struct ce{
        struct ed edges[EDGE];
        int nodes[4];
};
struct ce cells[C+1];
```

10.1.8 things.h

```
// the header file

#include <iostream.h>
#include <fstream.h>

#define NUM 60181
#define E 117110
#define C E
#define EDGE 4
```

10.2 Pre-processing Fluent unstructured data – II

The second pre-processing step is to calculate the adjacent cell numbers to each cell. Following programs starting with **flu_man.cpp** as the main program output a file with the adjacent cell numbers to each cell edge. If there is no adjacent cell to an edge of a current cell numerical value of –1 is used.

10.2.1 find_six.cpp

```
// finds the six edges connected to a known edge

#include "things.h"
#include "flu_struct.h"

// take in cell number and the edge number
void share_edges(int c0, int e0)
{
int i=0,j=0;
int v0=0,v1=0,v3=0,v2=0,k=0,q=0;
int numer[3]={0,1,0};
int ori_edge_vert[2];
int count=99;

ori_edge_vert[0]=cells[c0].edges[e0].vert_num[0];
ori_edge_vert[1]=cells[c0].edges[e0].vert_num[1];

for(i=0; i<C; i++){
for(j=0; j<EDGE; j++){
for(k=0; k<2; k++){
// if current cell number then do nothing
        if(i==c0){
        // do nothing
        }
        else{
        if(cells[i].edges[j].vert_num[k]==ori_edge_vert[0]){
        v2=numer[k+1];
        if(cells[i].edges[j].vert_num[v2]==ori_edge_vert[1]){
        cells[c0].edges[e0].adj=i;
        count=0;
        }
        }
        } // end of else
} // end of vertices
} // end of edges
} // end of cells

if(count!=0){
cells[c0].edges[e0].adj=-1;
}
}
```

10.2.2 flu_man.cpp

```
// Read Tecplot files into ijk format

#include "things.h"
#include "struct.h"

void share_edges(int i, int j);
void print_fun();
void ini_read();
//void store_cell();

int
main()
{
int i=0,j=0;
int count_cell=0;
// read cell nodes
ini_read();
//share_edges(i,j);
//        return 0;

for(i=0; i<C; i++){
cout << "Trial cell number " << i << endl;
for(j=0; j<EDGE; j++){
// find sharing cells
share_edges(i,j);
}
}
print_fun();
return 0;
}
```

10.2.3 ini_read.cpp

```
// Read file information

#include "things.h"
#include "flu_struct.h"

void ini_read()
{
int i=0,j=0,a=0,b=0,c=0,d=0,e=0;
ifstream in1;
in1.open("cell_xy.dat");
```

```
cout << "Reading information\n";
for(i=0; i<C; i++){
in1 >> a;
in1 >> b;
in1 >> c;
in1 >> d;
in1 >> e;

cells[i].edges[0].vert_num[0]=b;
cells[i].edges[0].vert_num[1]=c;
cells[i].edges[1].vert_num[0]=c;
cells[i].edges[1].vert_num[1]=d;
cells[i].edges[2].vert_num[0]=d;
cells[i].edges[2].vert_num[1]=e;
cells[i].edges[3].vert_num[0]=e;
cells[i].edges[3].vert_num[1]=b;
}
cout << "DONE\n";
in1.close();
}
```

10.2.4 print_fun.cpp

```
// print information

#include "things.h"
#include "flu_struct.h"

void print_fun()
{
int i=0,j=0;
ofstream out1;
out1.open("cell_adj.dat");
for(i=0; i<C; i++){
out1 << i << "\t";
for(j=0; j<EDGE; j++){
out1 << cells[i].edges[j].adj << "\t";
}
out1 << endl;
}
out1.close();
}
```

10.2.5 flu_struct.cpp

```
// A header file of structures

#include "things.h"

struct ed{
        int vert_num[2];
        int number;
        int adj;
};
struct ce{
        struct ed edges[EDGE];
        int nodes[4];
};
extern struct ce cells[C+1];
```

10.2.6 struct.cpp

```
// A header file of structures

#include "things.h"

struct ed{
        int vert_num[2];
        int number;
        int adj;
};
struct ce{
        struct ed edges[EDGE];
        int nodes[4];
};
struct ce cells[C+1];
```

10.2.7 things.h

```
// the header file

#include <iostream.h>
#include <fstream.h>

#define NUM 60181
#define E 117110
#define C 56930
//#define C 110
#define EDGE 4
```

10.3 Solver

The main program is **main_prog.cpp**.

10.3.1 cart_curvi.cpp

```
// interpolation routine
#include "tophead.h"
#include "ext_struct.h"

extern int cparticle;
extern int ccell;
extern double x[MAX_N+1];
extern double y[MAX_N+1];
extern double ach[MN];

void cart_curvi()
{
int i=0,j=0;
int n1=0,n2=0,n3=0,n4=0;
double a1=0.0,b1=0.0,c1=0.0,d1=0.0;
double a2=0.0,b2=0.0,c2=0.0,d2=0.0;
double a11=0.0,a12=0,a21=0.0,a22=0.0;
double b11=0.0, b22=0.0;
double h0=0.0,h1=0.0;
double eta1=0.0, eta2=0.0;
double diff=0.0,tol1=0.0,tol2=0.0;
int t=0;

n1=cells[ccell].nodes[0];
n2=cells[ccell].nodes[1];
n3=cells[ccell].nodes[2];
n4=cells[ccell].nodes[3];
n1=n1-1;
n2=n2-1;
n3=n3-1;
n4=n4-1;
t=all_particles[cparticle].last_t;
a1=x[n1]-all_particles[cparticle].px[t];
b1=x[n2]-x[n1];
c1=x[n4]-x[n1];
d1=x[n1]-x[n2]+x[n3]-x[n4];

a2=y[n1]-all_particles[cparticle].py[t];
```

```
b2=y[n2]-y[n1];
c2=y[n4]-y[n1];
d2=y[n1]-y[n2]+y[n3]-y[n4];

// let the guessed value of eta1 eta2 is cell centre
eta1=0.25*(x[n1]+x[n2]+x[n3]+x[n4]);
eta2=0.25*(y[n1]+y[n2]+y[n3]+y[n4]);

// set error limit to 1/1000 th of mid length
tol1=0.001*fabs(eta1-x[n1]);
tol2=0.001*fabs(eta2-y[n1]);

for(j=0; j<MAX_CURV2; j++){
for(i=0; i<MAX_CURV1; i++){
a11=b1+d1*eta2;
a12=c1+d1*eta1;
a21=b2+d2*eta2;
a22=c2+d2*eta1;

b11=a1+b1*eta1+c1*eta2+d1*eta1*eta2;
b22=a2+b2*eta1+c2*eta2+d2*eta1*eta2;

if(a21==0.0){
h1=-b22/a22;
h0=(-b11-a12*h1)/a11;
}
else{
h1=(a11*(b22/a21)-b11)/(a12-a11*(a22/a21));
h0=-(h1*(a22/a21)+b22/a21);
}
eta1=eta1+h0;
eta2=eta2+h1;
}
diff=fabs(h0);
if(diff<=tol1){
j=MAX_CURV2+1;  // exit loop
}
diff=fabs(h1);
if(diff<=tol2){
j=MAX_CURV2+1;  // exit loop
}
}
ach[0]=eta1;
ach[1]=eta2;
}
```

10.3.2 initial_part.cp

```
//  Main of the particle tracking code
// Converting in to 2D form by hiding the 3D works
// Date 30/05/01

#include "tophead.h"
#include "ext_struct.h"

extern int cparticle;
extern int particle_num;

void initial_part()
{
int x=0,i=0,j=0;
double ini_x=0,ini_y=0;
ifstream par_xy;
par_xy.open("particle_xy.txt");
//  reading the total number of particles
par_xy >> particle_num;
all_particles[i].tot_par=particle_num;
cout << "\n Number of particles are " << particle_num << endl;
for(i=0; i<particle_num; i++){
par_xy >> x;              // particle number
par_xy >> ini_x;
all_particles[i].px[0]=ini_x;
par_xy >> ini_y;
all_particles[i].py[0]=ini_y;
all_particles[i].last_t=0;
}
par_xy.close();
}
```

10.3.3 initial_search.cpp

```
// ------------------------------------------------------------------------
-
// Search for the particle cell and locate it

#include "tophead.h"
#include "ext_struct.h"

extern int cparticle;
extern int ccell;
extern double x[MAX_N+1];
extern double y[MAX_N+1];

int initial_search()
{
double edge_vec[2];
double rela_vec[2];
double norm_vec[2];
double origin[2];
double rel_pos[2];
double dot=0.0;
double mod_rel=0.0,mod_norm=0.0,cos_theta=0.0;
int a=0,b=0,c=0,d=0;
int i=0,j=0,t=0;
int count=0;

cout <<"\nIn search and locate\n";
//              find SEARCH_LOOP number of cells
for(j=0; j<C; j++){
cout << endl << "loop =" << j << endl;
count=0;
for(i=0; i<EDGE; i++){
//                              cout << i << endl;
a=cells[ccell].edges[i].vert_num[0];
b=cells[ccell].edges[i].vert_num[1];
a=a-1;
b=b-1;
origin[0]=x[a];
origin[1]=y[a];
edge_vec[0]=x[b]-x[a];
edge_vec[1]=y[b]-y[a];
a=cells[ccell].nodes[0];
b=cells[ccell].nodes[1];
c=cells[ccell].nodes[2];
d=cells[ccell].nodes[3];

a=a-1;
b=b-1;
c=c-1;
d=d-1;
rela_vec[0]=0.25*(x[a]+x[b]+x[c]+x[d])-origin[0];
rela_vec[1]=0.25*(y[a]+y[b]+y[c]+y[d])-origin[1];

norm_vec[0]=edge_vec[0]*edge_vec[1]*rela_vec[1]-
rela_vec[0]*pow(edge_vec[1],2);
norm_vec[1]=edge_vec[0]*edge_vec[1]*rela_vec[0]-
rela_vec[1]*pow(edge_vec[0],2);
t=all_particles[cparticle].last_t;
rel_pos[0]=all_particles[cparticle].px[t]-origin[0];
rel_pos[1]=all_particles[cparticle].py[t]-origin[1];
```

```
dot=0.0;
dot=dot+rel_pos[0]*norm_vec[0];
dot=dot+rel_pos[1]*norm_vec[1];
mod_rel=sqrt(pow(rel_pos[0],2)+pow(rel_pos[1],2));
mod_norm=sqrt(pow(norm_vec[0],2)+pow(norm_vec[1],2));
cos_theta=dot/(mod_rel*mod_norm);

if(cos_theta>0.0){
cout << "\nParticle not within=" << ccell << endl;
// particle is in the adjacent edge
i=EDGE+1;
ccell++;
}
else{
count++;
}
} // end of edges
if(count==4){
cout << endl << "particle located in search and locate in> " <<
ccell << endl;
return 0;
j=C+1;
}
} // end of trials
if(count !=4){
cout << endl << "Particle not found within the domain" << endl;
return 99;
}
return 0;
}

//
*****************************************************
***********
//          NOTE INSTEAD OF DO LOOP COULD HAVE
FOR LOOP WITH
//          MAXIMUM NUMBER OF ITERATIONS. THEN
ERROR SIGNAL
//          CAN BE USED TO DISCOVER THE PROBLEM
//
*****************************************************
***********

//
*****************************************************
***********
//          KNOW THAT IN 3D ONLY 26 CV'S
SURROUNDING A GIVEN
//          CV. THE PARTICLE MUST GO TO ONE OF
THESE CELLS
//          HENCE THE FOR LOOP MUST ONLY SEARCH
WITHIN TWO OTHER CVs.
//          OTHERWISE THE PARTICLE MUST HAVE
JUMPED A CELL OR THAT PARTICLE
//          HAS GONE BEYOND THE COMPUTATIONAL
BOUNDARIES
//
*****************************************************
***********
```

10.3.4 Interpol.cpp

```
// performs linear interpolation

#include "tophead.h"
#include "ext_struct.h"

extern int ccell;
extern int cparticle;
extern double x[MAX_N+1];
extern double y[MAX_N+1];
extern double u[MAX_N+1];
extern double v[MAX_N+1];
extern double dudx[MAX_N+1];
extern double dudy[MAX_N+1];
extern double dvdx[MAX_N+1];
extern double dvdy[MAX_N+1];

double ach[MN]={0.0,0.0};
void cart_curvi();

void interpol()
{
int n1=0,n2=0,n3=0,n4=0;
int ci=0,cj=0,ck=0,i=0,temp1=0,temp2=0,t=0;
double zerou=0.0,oneu=0.0,twou=0.0,threeu=0.0;
double zerov=0.0,onev=0.0,twov=0.0,threev=0.0;
double zerodudx=0.0,zerodudy=0.0,zerodvdx=0.0,zerodvdy=0.0;
double onedudx=0.0,onedudy=0.0,onedvdx=0.0,onedvdy=0.0;
double twodudx=0.0,twodudy=0.0,twodvdx=0.0,twodvdy=0.0;
double
threedudx=0.0,threedudy=0.0,threedvdx=0.0,threedvdy=0.0;
double up=0.0,vp=0.0;

// velocity at 8 vertices
n1=cells[ccell].nodes[0];
n2=cells[ccell].nodes[1];
n3=cells[ccell].nodes[2];
n4=cells[ccell].nodes[3];

n1=n1-1;
n2=n2-1;
n3=n3-1;
n4=n4-1;

zerou=u[n1];
zerov=v[n1];
zerodudx=dudx[n1];
zerodudy=dudy[n1];
zerodvdx=dvdx[n1];
zerodvdy=dvdy[n1];

oneu=u[n2];
onev=v[n2];
onedudx=dudx[n2];
onedudy=dudy[n2];
onedvdx=dvdx[n2];
onedvdy=dvdy[n2];

twou=u[n3];
twov=v[n3];
twodudx=dudx[n3];
twodudy=dudy[n3];
twodvdx=dvdx[n3];
twodvdy=dvdy[n3];

threeu=u[n4];
threev=v[n4];
threedudx=dudx[n4];
threedudy=dudy[n4];
threedvdx=dvdx[n4];
threedvdy=dvdy[n4];

// obtain the local coordinate of particle position
// i.e. curvilinear coordinates bounded by CV
cart_curvi();

/*
```

```
// ****************** linear interpolation
*************************
// customised for 2D
            up=0.0;
            up=zerou*ach[0]*ach[1];
            up=up+oneu*(1-ach[0])*ach[1];
            up=up+twou*(1-ach[0])*(1-ach[1]);
            up=up+threeu*ach[0]*(1-ach[1]);

            vp=0.0;
            vp=zerov*ach[0]*ach[1];
            vp=vp+onev*(1-ach[0])*ach[1];
            vp=vp+twov*(1-ach[0])*(1-ach[1]);
            vp=vp+threev*ach[0]*(1-ach[1]);

*/

///*
// ****************** shape function interpolation
*************************

double step_x=0.0,step_y=0.0;

step_x=fabs(x[n1]-x[n2]);
step_y=fabs(y[n3]-y[n2]);

up=0.0;
up=up+zerou*(pow((1-ach[0]),2))*(1+2*ach[0])*(pow((1-
ach[1]),2))*(1+2*ach[1]);
up=up+zerodudx*ach[0]*step_x*(pow((1-ach[0]),2))*(pow((1-
ach[1]),2))*(1+2*ach[1]);
up=up+zerodudy*(pow((1-
ach[0]),2))*(1+2*ach[0])*step_y*ach[1]*(pow((1-ach[1]),2));

up=up+oneu*(pow(ach[0],2))*(3-2*ach[0])*(pow((1-
ach[1]),2))*(1+2*ach[1]);
up=up+onedudx*step_x*(pow(ach[0],2))*(ach[0]-1)*(pow((1-
ach[1]),2))*(1+2*ach[1]);
up=up+onedudy*(pow(ach[0],2))*(3-
2*ach[0])*step_y*ach[1]*(pow((1-ach[1]),2));

up=up+threeu*(pow((1-
ach[0]),2))*(1+2*ach[0])*(pow(ach[1],2))*(3-2*ach[1]);
up=up+threedudx*step_x*ach[0]*(pow((1-
ach[0]),2))*(pow(ach[1],2))*(3-2*ach[1]);
up=up+threedudy*(pow((1-
ach[0]),2))*(1+2*ach[0])*step_y*(pow(ach[1],2))*(ach[1]-1);

up=up+twou*(pow(ach[0],2))*(3-2*ach[0])*(pow(ach[1],2))*(3-
2*ach[1]);
up=up+twodudx*step_x*(pow(ach[0],2))*(ach[0]-
1)*(pow(ach[1],2))*(3-2*ach[1]);
up=up+twodudy*(pow(ach[0],2))*(3-
2*ach[0])*step_y*(pow(ach[1],2))*(ach[1]-1);

vp=0.0;
vp=vp+zerov*(pow((1-ach[0]),2))*(1+2*ach[0])*(pow((1-
ach[1]),2))*(1+2*ach[1]);
vp=vp+zerodvdx*ach[0]*step_x*(pow((1-ach[0]),2))*(pow((1-
ach[1]),2))*(1+2*ach[1]);
vp=vp+zerodvdy*(pow((1-
ach[0]),2))*(1+2*ach[0])*step_y*ach[1]*(pow((1-ach[1]),2));

vp=vp+onev*(pow(ach[0],2))*(3-2*ach[0])*(pow((1-
ach[1]),2))*(1+2*ach[1]);
vp=vp+onedvdx*step_x*(pow(ach[0],2))*(ach[0]-1)*(pow((1-
ach[1]),2))*(1+2*ach[1]);
vp=vp+onedvdy*(pow(ach[0],2))*(3-
2*ach[0])*step_y*ach[1]*(pow((1-ach[1]),2));

vp=vp+threev*(pow((1-
ach[0]),2))*(1+2*ach[0])*(pow(ach[1],2))*(3-2*ach[1]);
vp=vp+threedvdx*step_x*ach[0]*(pow((1-
ach[0]),2))*(pow(ach[1],2))*(3-2*ach[1]);
vp=vp+threedvdy*(pow((1-
ach[0]),2))*(1+2*ach[0])*step_y*(pow(ach[1],2))*(ach[1]-1);

vp=vp+twov*(pow(ach[0],2))*(3-2*ach[0])*(pow(ach[1],2))*(3-
2*ach[1]);
vp=vp+twodvdx*step_x*(pow(ach[0],2))*(ach[0]-
1)*(pow(ach[1],2))*(3-2*ach[1]);
vp=vp+twodvdy*(pow(ach[0],2))*(3-
2*ach[0])*step_y*(pow(ach[1],2))*(ach[1]-1);

//*/

t=all_particles[cparticle].last_t;
all_particles[cparticle].fu[t]=up;
all_particles[cparticle].fv[t]=vp;
}
```

10.3.5 main_prog.cpp

```
// The main program

#include "tophead.h"
#include "structures.h"

void reading();
void initial_part();
int initial_search();
void tstep();
//void write_dump(int ii);
void write_dump(int ii, int count,int dis);
void write_dump2(int ii);
int search_locate();
void interpol();

int cparticle=0;
int ccell=0;
int particle_num;
double time_step=0.0;
double x[MAX_N+1];
double y[MAX_N+1];
double u[MAX_N+1];
double v[MAX_N+1];
double dudx[MAX_N+1];
double dudy[MAX_N+1];
double dvdx[MAX_N+1];
double dvdy[MAX_N+1];

int
main()
{
int i=0;
int dis=0;

// read in the data files
// grid nodes,
// cell centered,
// velocity components
reading();

// initialise particle position
initial_part();

//              ccell=15075;

int part=0,t=0,count=0;
```

```
double x0=0.0,y0=0.0;
double cu1=0.0,cu2=0.0,cu3=0.0,cu4=0.0;
double cv1=0.0,cv2=0.0,cv3=0.0,cv4=0.0;
double store_T[T_STEP+1];

double k=0.0,ku1=0.0,ku2=0.0,ku3=0.0,ku4=0.0;
double kv1=0.0,kv2=0.0,kv3=0.0,kv4=0.0;
double f1=0.0,xp=0.0,yp=0.0,up=0.0,vp=0.0,uf=0.0,vf=0.0;
double kxp=0.0,kyp=0.0,kuf=0.0,kvf=0.0;

double tau=0.0;
double viscosity=1.82e-5;
double velo=2.6218487395, dia=3.5e-3;
double part_dia=3.0e-6;
//double stk=0.02;
//double part_den=9.7183333e2;
//double stk=0.04;
//double part_den=1.9437e3;
//double stk=0.06;
//double part_den=2.9155e3;
//double stk=0.08;
//double part_den=3.8873e3;
//double stk=0.1;
//double part_den=4.85917e3;
double stk=0.12;
double part_den=5.831e3;

for(part=0;part<PAR; ++part){
cparticle=part;
//ccell=16675;
//ccell=15375;
//ccell=15375;
//ccell=15425;
//ccell=16225;

//              /*
// find the particle containing CV numver
ccell=0;
i=initial_search();
if(i==99){
cout << "\nParticle not found within the domain\n";
return 0;
}
else{
cout << "\nParticle is located within the domain\n";
}

//              return 0;
//*/
// find fluid velocities at particle position
interpol();
cout << endl << ccell << endl;
//return 0;
// set initial conditions
t=all_particles[cparticle].last_t;
all_particles[cparticle].pu[t]=all_particles[cparticle].fu[t];
all_particles[cparticle].pv[t]=all_particles[cparticle].fv[t];
//              tstep();
count=0;
int var9=0;

store_T[0]=0.0;
double tt=0.0;

for(i=0; i<T_STEP; i++){
cout << "time step=" << i << " ccell=" << ccell;

// ********** step04***************
// calculate the time step
tstep();
store_T[i]=tt+time_step;
tt=store_T[i];
cout << " time=" << store_T[i] << endl;

//tau=(18*viscosity)/(dp*dp*rhop);
tau=(velo/dia)/stk;
//tau=5000;
tau=(18*viscosity)/(part_dia*part_dia*part_den);

// calculate particle position from know particle velocity
t=all_particles[cparticle].last_t;
xp=all_particles[cparticle].px[t];
yp=all_particles[cparticle].py[t];
up=all_particles[cparticle].pu[t];
vp=all_particles[cparticle].pv[t];
uf=all_particles[cparticle].fu[t];
vf=all_particles[cparticle].fv[t];

///*
f1=tau*(uf-up);
ku1=0.5*time_step*f1;
f1=tau*(vf-vp);
kv1=0.5*time_step*f1;
//*/

/*
f1=tau*(uf-uf);
ku1=0.5*time_step*f1;
f1=tau*(vf-vf);
kv1=0.5*time_step*f1;
*/
///*
k=0.5*time_step*(up+0.5*ku1);
kxp=xp+k;
k=0.5*time_step*(vp+0.5*kv1);
kyp=yp+k;
//*/
/*
k=0.5*time_step*(uf+0.5*ku1);
kxp=xp+k;
k=0.5*time_step*(vf+0.5*kv1);
kyp=yp+k;
*/
t=all_particles[cparticle].last_t;
all_particles[cparticle].px[t]=kxp;
all_particles[cparticle].py[t]=kyp;

dis=search_locate();
if(dis==99){
cout << endl << "Particle is outside (1)" << endl;
//write_dump2(var9+1);
write_dump2(i);
write_dump(i,count,dis);
i=T_STEP+5;
count=0;
cout << "\ntau=" << tau << endl;
//return 0;
}
else{
interpol();
t=all_particles[cparticle].last_t;
kuf=all_particles[cparticle].fu[t];
kvf=all_particles[cparticle].fv[t];
///*
f1=tau*(kuf-up-ku1);
ku2=0.5*time_step*f1;
f1=tau*(kvf-vp-kv1);
kv2=0.5*time_step*f1;
f1=tau*(kuf-up-ku2);
ku3=0.5*time_step*f1;
```

```
f1=tau*(kvf-vp-kv2);
kv3=0.5*time_step*f1;

k=time_step*(up+ku3);
kxp=xp+k;
k=time_step*(vp+kv3);
kyp=yp+k;
//*/

/*
f1=tau*(kuf-uf-ku1);
ku2=0.5*time_step*f1;
f1=tau*(kvf-vf-kv1);
kv2=0.5*time_step*f1;
f1=tau*(kuf-uf-ku2);
ku3=0.5*time_step*f1;
f1=tau*(kvf-vf-kv2);
kv3=0.5*time_step*f1;

k=time_step*(uf+ku3);
kxp=xp+k;
k=time_step*(vf+kv3);
kyp=yp+k;
*/

t=all_particles[cparticle].last_t;
all_particles[cparticle].px[t]=kxp;
all_particles[cparticle].py[t]=kyp;
dis=search_locate();
if(dis==99){
cout << endl << "Particle is outside (2)" << endl;
//write_dump2(var9+1);
write_dump2(i);
write_dump(i,count,dis);
i=T_STEP+5;
count=0;
cout << "\ntau=" << tau << endl;
//return 0;
}
else{
interpol();
t=all_particles[cparticle].last_t;
kuf=all_particles[cparticle].fu[t];
kvf=all_particles[cparticle].fv[t];
///*
f1=tau*(kuf-up-2.0*ku3);
ku4=0.5*time_step*f1;
f1=tau*(kvf-vp-2.0*kv3);
kv4=0.5*time_step*f1;
//*/
/*
f1=tau*(kuf-uf-2.0*ku3);
ku4=0.5*time_step*f1;
f1=tau*(kvf-vf-2.0*kv3);
kv4=0.5*time_step*f1;
*/
t=all_particles[cparticle].last_t;
all_particles[cparticle].px[t]=xp;
all_particles[cparticle].py[t]=yp;
all_particles[cparticle].fu[t]=uf;
all_particles[cparticle].fv[t]=vf;
all_particles[cparticle].pu[t]=up;
all_particles[cparticle].pv[t]=vp;

all_particles[cparticle].last_t++;
t=all_particles[cparticle].last_t;
/*
cout << "ku1=" << ku1 << endl;
cout << "ku2=" << ku2 << endl;
cout << "ku3=" << ku3 << endl;
cout << "ku4=" << ku4 << endl;
*/
///*
all_particles[cparticle].px[t]=xp+time_step*(up+(1.0/3.0)*(ku1+k
u2+ku3));
all_particles[cparticle].py[t]=yp+time_step*(vp+(1.0/3.0)*(kv1+k
v2+kv3));
all_particles[cparticle].pu[t]=up+(1.0/3.0)*(ku1+2.0*ku2+2.0*ku
3+ku4);
all_particles[cparticle].pv[t]=vp+(1.0/3.0)*(kv1+2.0*kv2+2.0*kv
3+kv4);
//*/
/*
all_particles[cparticle].px[t]=xp+time_step*(uf+(1.0/3.0)*(ku1+k
u2+ku3));
all_particles[cparticle].py[t]=yp+time_step*(vf+(1.0/3.0)*(kv1+k
v2+kv3));
all_particles[cparticle].fu[t]=uf+(1.0/3.0)*(ku1+2.0*ku2+2.0*ku3
+ku4);
all_particles[cparticle].fv[t]=vf+(1.0/3.0)*(kv1+2.0*kv2+2.0*kv3
+kv4);
all_particles[cparticle].pu[t]=uf+(1.0/3.0)*(ku1+2.0*ku2+2.0*ku
3+ku4);
all_particles[cparticle].pv[t]=vf+(1.0/3.0)*(kv1+2.0*kv2+2.0*kv
3+kv4);
*/
//count++;

// ********** step05***************
dis=search_locate();
if(dis==99){
cout << endl << "Particle is outside (3)" << endl;
//write_dump2(var9+1);
write_dump2(i);
write_dump(i,count,dis);
i=T_STEP+5;
count=0;
cout << "\ntau=" << tau << endl;
//return 0;
}
else{
// interpolation
interpol();
//t=all_particles[cparticle].last_t;
//cout << "before fu=" << all_particles[cparticle].fu[t] <<  "\t" <<
t << endl;
count++;
if(count==DUMP_T){
var9=i;
write_dump(i,count,dis);
count=0;
//t=all_particles[cparticle].last_t;
//cout << "after fu=" << all_particles[cparticle].fu[t] << "\t" << t
<< endl;
}
}
}//2
}//1
}// end of time stepping
}// end of all particles

cout << "\ntau=" << tau << endl;
return 0;
}
```

10.3.6 reading.cpp

```
//-------------------------------------------------------------------------
//initialise flow variables
//-------------------------------------------------------------------------
```

```
#include "tophead.h"
#include "ext_struct.h"

extern double x[MAX_N+1];
extern double y[MAX_N+1];
extern double u[MAX_N+1];
extern double v[MAX_N+1];
extern double dudx[MAX_N+1];
extern double dudy[MAX_N+1];
extern double dvdx[MAX_N+1];
extern double dvdy[MAX_N+1];

void reading()
{
int i=0,j=0,a=0,b=0,c=0,d=0,e=0;
ifstream in1;
in1.open("cell_xy.dat");
// Reading cell node numbers
cout << "Reading information\n";
for(i=0; i<C; i++){
in1 >> a;
in1 >> b;
in1 >> c;
in1 >> d;
in1 >> e;
cells[i].edges[0].vert_num[0]=b;
cells[i].edges[0].vert_num[1]=c;
cells[i].edges[1].vert_num[0]=c;
cells[i].edges[1].vert_num[1]=d;
cells[i].edges[2].vert_num[0]=d;
cells[i].edges[2].vert_num[1]=e;
cells[i].edges[3].vert_num[0]=e;
cells[i].edges[3].vert_num[1]=b;

cells[i].nodes[0]=b;
cells[i].nodes[1]=c;
cells[i].nodes[2]=d;
cells[i].nodes[3]=e;

}
cout << "DONE\n";
in1.close();
// Reading adjacent cell numbers
in1.open("cell_adj.dat");
for(i=0; i<C; i++){
in1 >> a;
for(j=0; j<EDGE; j++){
in1 >> cells[i].edges[j].adj;
}
}
in1.close();

// reading the cell centered x-coordinate
in1.open("x_cen.dat");
for(i=0; i<MAX_N; i++){
in1 >> x[i];
}
in1.close();
// reading the cell centered y-coordinate
in1.open("y_cen.dat");
for(i=0; i<MAX_N; i++){
in1 >> y[i];
}
in1.close();

// reading the cell centered u-velocity
in1.open("u_cen.dat");
for(i=0; i<MAX_N; i++){
in1 >> u[i];
}
in1.close();
// reading the cell centered v-velocity
in1.open("v_cen.dat");
for(i=0; i<MAX_N; i++){
in1 >> v[i];
}
in1.close();

///*
// reading the cell centered dudx
in1.open("dudx_cen.dat");
for(i=0; i<MAX_N; i++){
in1 >> dudx[i];
}
in1.close();

// reading the cell centered dudy
in1.open("dudy_cen.dat");
for(i=0; i<MAX_N; i++){
in1 >> dudy[i];
}
in1.close();

// reading the cell centered dvdx
in1.open("dvdx_cen.dat");
for(i=0; i<MAX_N; i++){
in1 >> dvdx[i];
}
in1.close();

// reading the cell centered dvdy
in1.open("dvdy_cen.dat");
for(i=0; i<MAX_N; i++){
in1 >> dvdy[i];
}
in1.close();
//*/
}
```

10.3.7 search_locate.cpp

```
// ------------------------------------------------------------------------
-
// Search for the particle cell and locate it

#include "tophead.h"
#include "ext_struct.h"

extern int cparticle;
extern int ccell;
extern double x[MAX_N+1];
extern double y[MAX_N+1];

int search_locate()
{
double edge_vec[2];
double rela_vec[2];
double norm_vec[2];
double origin[2];
double rel_pos[2];
double dot=0.0;
double mod_rel=0.0,mod_norm=0.0,cos_theta=0.0;

int a=0,b=0,c=0,d=0;
int i=0,j=0,t=0;
int count=0;

//          cout <<"\nIn search and locate\n";
//          find SEARCH_LOOP number of cells
```

```
for(j=0; j<SEARCH_LOOP; j++){
//                      cout << endl << "loop =" << j << endl;
count=0;
for(i=0; i<EDGE; i++){
//                      cout << i << endl;
a=cells[ccell].edges[i].vert_num[0];
b=cells[ccell].edges[i].vert_num[1];

a=a-1;
b=b-1;

origin[0]=x[a];
origin[1]=y[a];

edge_vec[0]=x[b]-x[a];
edge_vec[1]=y[b]-y[a];

a=cells[ccell].nodes[0];
b=cells[ccell].nodes[1];
c=cells[ccell].nodes[2];
d=cells[ccell].nodes[3];

a=a-1;
b=b-1;
c=c-1;
d=d-1;

rela_vec[0]=0.25*(x[a]+x[b]+x[c]+x[d])-origin[0];
rela_vec[1]=0.25*(y[a]+y[b]+y[c]+y[d])-origin[1];

norm_vec[0]=edge_vec[0]*edge_vec[1]*rela_vec[1]-
rela_vec[0]*pow(edge_vec[1],2);
norm_vec[1]=edge_vec[0]*edge_vec[1]*rela_vec[0]-
rela_vec[1]*pow(edge_vec[0],2);

t=all_particles[cparticle].last_t;
rel_pos[0]=all_particles[cparticle].px[t]-origin[0];
rel_pos[1]=all_particles[cparticle].py[t]-origin[1];

dot=0.0;
dot=dot+rel_pos[0]*norm_vec[0];
dot=dot+rel_pos[1]*norm_vec[1];

mod_rel=sqrt(pow(rel_pos[0],2)+pow(rel_pos[1],2));
mod_norm=sqrt(pow(norm_vec[0],2)+pow(norm_vec[1],2));

cos_theta=dot/(mod_rel*mod_norm);
if(cos_theta>0.0){
ccell=cells[ccell].edges[i].adj;
if(ccell==-1){
// reached a wall/boundary
return 99;
}
else{
// particle is in the adjacent edge
i=EDGE+1;
}
}
else{
count++;
}
} // end of edges
if(count==4){
//                      cout << endl << "particle located in search
and locate" << endl;
return 0;
j=SEARCH_LOOP+1;
}
} // end of trials
return 0;
}

//
*****************************************************
**********
//          NOTE INSTEAD OF DO LOOP COULD HAVE
FOR LOOP WITH
//          MAXIMUM NUMBER OF ITERATIONS. THEN
ERROR SIGNAL
//          CAN BE USED TO DISCOVER THE PROBLEM
//
*****************************************************
**********

//
*****************************************************
**********
//          KNOW THAT IN 3D ONLY 26 CV'S
SURROUNDING A GIVEN
//          CV. THE PARTICLE MUST GO TO ONE OF
THESE CELLS
//          HENCE THE FOR LOOP MUST ONLY SEARCH
WITHIN TWO OTHER CVs.
//          OTHERWISE THE PARTICLE MUST HAVE
JUMPED A CELL OR THAT PARTICLE
//          HAS GONE BEYOND THE COMPUTATIONAL
BOUNDARIES
//
*****************************************************
**********
```

10.3.8 tstep.cpp

```
// ------------------------------------------------------------------------
-
// Calculating the direction vectors

#include "tophead.h"
#include "ext_struct.h"

extern int cparticle;
extern double time_step;
extern int ccell;
extern double x[MAX_N+1];
extern double y[MAX_N+1];

void tstep()
{
double centerx=0.0,centery=0.0;
double distance=0.0, velocity=0.0;
int a=0,b=0,c=0,d=0;
int t=0;

a=cells[ccell].nodes[0];
b=cells[ccell].nodes[1];
c=cells[ccell].nodes[2];
d=cells[ccell].nodes[3];

a=a-1;
b=b-1;
c=c-1;
d=d-1;

// absolute cell center
centerx=0.25*(x[a]+x[b]+x[c]+x[d]);
centery=0.25*(y[a]+y[b]+y[c]+y[d]);

// relative cell center
centerx=centerx-x[a];
centery=centery-y[a];
```

```
t=all_particles[cparticle].last_t;
distance=sqrt(pow(centerx,2)+pow(centery,2));
velocity=sqrt(pow(all_particles[cparticle].pu[t],2)+pow(all_partic
les[cparticle].pv[t],2));
//time_step=(1.0/4.0)*(distance/velocity);
time_step=0.02*(distance/velocity);
//              cout <<"\n----------------------------------------------\n";
}
```

10.3.9 write_dump.cpp

```
// collected information during particle tracks are written to file
#include "tophead.h"
#include "ext_struct.h"

extern int cparticle;

//void write_dump(int ii)
void write_dump(int ii, int count, int status)
{
 const char *file = "output_part.dat";
ofstream par_out(file, ios::out | ios::app);
//ofstream par_out(ios::out | ios::app);
//par_out.open("output_part.txt");
int i=0,t=0;
double temp1=0.0;
int c=0;
par_out << "\n";
//for(i=1;i<=DUMP_T;++i){
for(i=1;i<=count;++i){
//par_out << ii-(DUMP_T-1)+i << "\t";
c=ii-(count-1)+i;
par_out << c << "\t";
par_out << all_particles[cparticle].px[i];
par_out << "\t";
par_out << all_particles[cparticle].py[i];
par_out << "\t";
par_out << all_particles[cparticle].pu[i];
par_out << "\t";
par_out << all_particles[cparticle].pv[i];
par_out << "\t";
par_out << all_particles[cparticle].fu[i];
par_out << "\t";
par_out << all_particles[cparticle].fv[i];
par_out << "\n";
}
par_out.close();
t=all_particles[cparticle].last_t;
//              cout <<"\nLast time=" << t << endl;
temp1=all_particles[cparticle].fu[t];
all_particles[cparticle].fu[0]=temp1;

temp1=all_particles[cparticle].fv[t];
all_particles[cparticle].fv[0]=temp1;

temp1=all_particles[cparticle].pu[t];
all_particles[cparticle].pu[0]=temp1;

temp1=all_particles[cparticle].pv[t];
all_particles[cparticle].pv[0]=temp1;

temp1=all_particles[cparticle].px[t];
all_particles[cparticle].px[0]=temp1;

temp1=all_particles[cparticle].py[t];
all_particles[cparticle].py[0]=temp1;

all_particles[cparticle].last_t=0;
t=all_particles[cparticle].last_t;
}
```

10.3.10 write_dump2.cpp

```
// collected information during particle tracks are written to file
#include "tophead.h"
#include "ext_struct.h"

extern int cparticle;

//void write_dump(int ii)
void write_dump2(int ii)
{
        const char *file1 = "part_info.dat";
        ofstream par_info(file1, ios::out | ios::app);
                par_info << "\n" << cparticle << "\t" << ii;
        par_info.close();
}
```

10.3.11 ext_struct.h

```
// A header file of externs

#include "tophead.h"

struct particle{
        double px[DUMP_T+1];
        double py[DUMP_T+1];
        double pu[DUMP_T+1];
        double pv[DUMP_T+1];
        double fu[DUMP_T+1];
        double fv[DUMP_T+1];
        double time_steps[DUMP_T+1];
        int num_cell;
        int last_particle;
        int last_t;
        int tot_par;
};
extern struct particle all_particles[PAR+1];
struct deriv{
        double dudx[C+1];
        double dudy[C+1];
        double dudz[C+1];
        double dvdx[C+1];
        double dvdy[C+1];
        double dvdz[C+1];
        double dwdx[C+1];
        double dwdy[C+1];
        double dwdz[C+1];
};
extern struct deriv allblock_dev[1+1];
struct ed{
        int vert_num[2];
        int adj;
};
struct ce{
        struct ed edges[EDGE];
        int nodes[4];
};
extern struct ce cells[C+1];
```

10.3.12 structures.h

```
// A header file of structures

#include "tophead.h"

struct particle{
        double px[DUMP_T+1];
        double py[DUMP_T+1];
        double pu[DUMP_T+1];
```

```
        double pv[DUMP_T+1];
        double fu[DUMP_T+1];
        double fv[DUMP_T+1];
        double time_steps[DUMP_T+1];
        int num_cell;
        int last_particle;
        int last_t;
        int tot_par;
};

struct particle all_particles[PAR+1];

struct deriv{
        double dudx[C+1];
        double dudy[C+1];
        double dudz[C+1];
        double dvdx[C+1];
        double dvdy[C+1];
        double dvdz[C+1];
        double dwdx[C+1];
        double dwdy[C+1];
        double dwdz[C+1];
};
struct deriv allblock_dev[1+1];

struct ed{
        int vert_num[2];
        int adj;
};

struct ce{
        struct ed edges[EDGE];
        int nodes[4];
};

struct ce cells[C+1];
```

10.3.13 tophead.h

```
// include all the header files used by all programs


#include <iostream.h>
#include <math.h>
#include <fstream.h>
#include <stdlib.h>

#define C 56930
#define MAX_N 60181
#define EDGE 4
#define MAX_CURV1 5
#define MAX_CURV2 2
#define MN 2            // 3 for 3D, 2 for 2D, size of matrix for
calculation curvilliear
#define SEARCH_LOOP 5
//#define T_STEP 345 // total particle time steps
#define T_STEP 50000 // total particle time steps
#define DUMP_T 100
#define PAR 91          // Total number of particles
```

11 Appendix D

Three dimensional laminar flow fields through a single bifurcation was simulated using commercial software. A grid was generated on the multi-block domain shown in Figure 3-7. Inlet condition was uniform velocity and the outlet conditions were equal relative static pressure. The u-velocity profiles given here are not quantitatively validated, but have qualitatively the same trends as in the experimental profile under equal mass flow rate studies conducted by (Zhao and Lieber 1994). The velocity profiles were evaluated at various cross-sections along the bifurcation at approximate locations as shown in Figure 11-1.

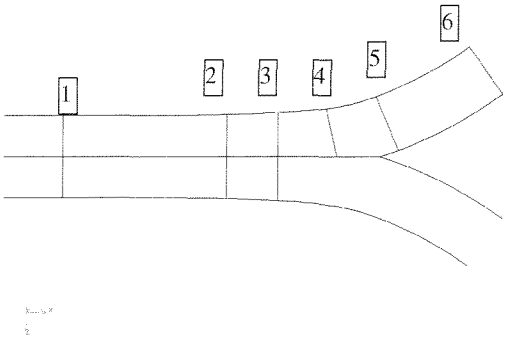
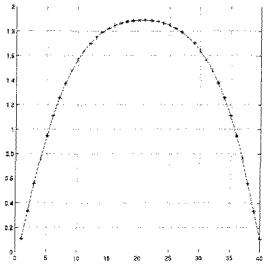
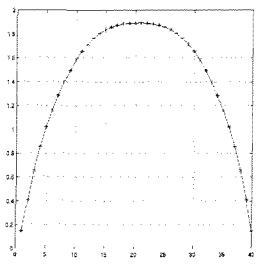


Figure 11-1 Cross-sections from which data are extracted labelled from 1 to 6

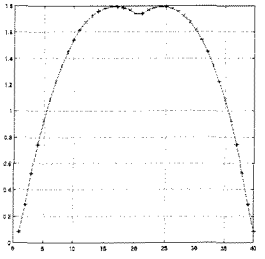
The first six sub-figures in Figure 11-2 show the axial velocity profiles extracted from the six cross-sections shown in Figure 11-1. In Figure 11-3 another set of six sub-figure show the velocity profiles extracted from a plane perpendicular to the bifurcation plane.



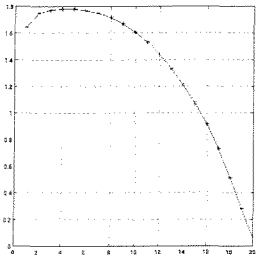
(1)



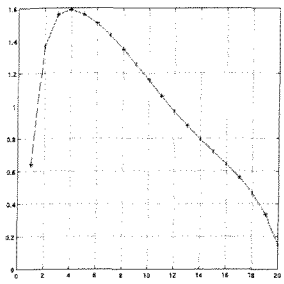
(2)



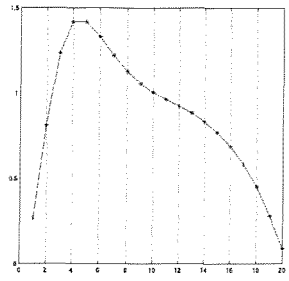
(3)



(4)

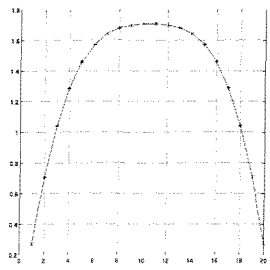


(5)

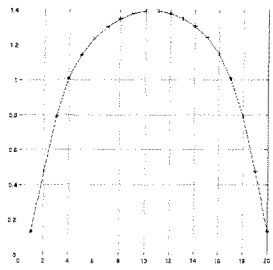


(6)

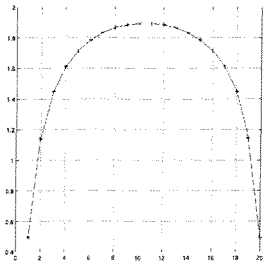
Figure 11-2 Six sub-figures shows six axial velocity profiles at six cross-sections (shown in Figure 11-1) in the three dimensional bifurcation



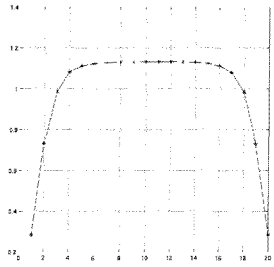
(1)



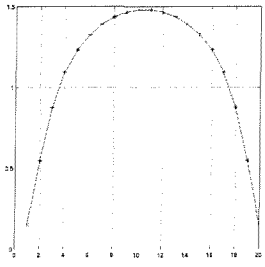
(4)



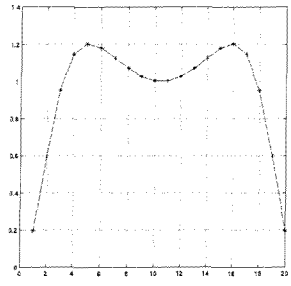
(2)



(5)



(3)



(6)

Figure 11-3 Six sub-figures shows the velocity profiles normal to the horizontal bifurcation plane at six cross-sections (shown in Figure 11-1) in the three dimensional bifurcation

11.1 Reference

Zhao, Y. and B. B. Lieber (1994). "Steady inspiratory flow in a model symmetric bifurcation." Journal of Biomechanical Engineering, (TASME) **116**: 488-496.

12 Appendix E

The equations used by (Zhao Yao 1994a) describing the bifurcation geometry had some anomalies. Some equations were modifications with permission of (Zhao Yao 1994a) after author upon personal communication.

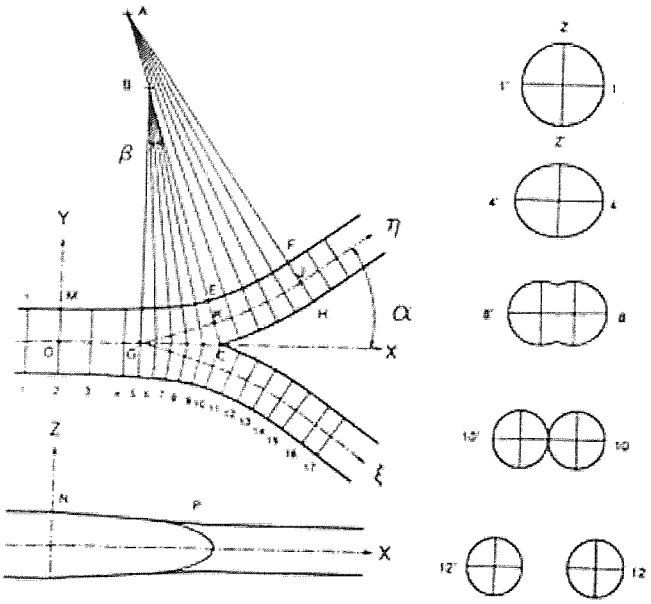


Fig. 1 Test section geometry in the bifurcation plane ($z=0$) and the transverse plane ($y=0$). Typical cross-sections at station 1, 4, 8, 10, and 12 are shown on the right.

Figure 12-1 Geometrical specifications of the three dimensional bifurcation

An airway bifurcation has three sections: parent tube, flow divider and daughter branches. The geometrical parameters that define the bifurcation are as follows.

- Parent tube diameter, D
- Daughter tube diameter, d
- Length of flow divider, L
- Branch angle, 2α
- Curvature of daughter tube centreline, R

Another curvature ratio, β, has also been used to describe the flow divider in order for the flow divider to be independent of the curvature of the daughter tubes.

Line ME:

Equation 12-1

$$y = a_1 x^2 + b_1 x^8 + 0.5D$$

Equation 12-2

$$a_1 = \frac{8d\cos\beta - (L - d\sin\beta)\tan\beta - 4D}{6(L - d\sin\beta)^2}$$

Equation 12-3

$$b_1 = \frac{(L - d\sin\beta)\tan\beta - 2d\cos\beta + D}{6(L - d\sin\beta)^8}$$

Line EF:

Equation 12-4

$$y = (R + 0.5d)\cos\beta - \{(R - 0.5d)^2 - (x - L + (R + 0.5d)\sin\beta)^2\}^{\frac{1}{2}}$$

Line GK:

Equation 12-5

$$y = a(x - 0.5L)^{\lambda}$$

Equation 12-6

$$a = \frac{0.5d\cos\beta}{(0.5L - 0.5d\sin\beta)^{\lambda}}$$

Equation 12-7

$$\lambda = \frac{\tan\beta(L - d\sin\beta)}{d\cos\beta}$$

Line KJ:

Equation 12-8

$$y = (R + 0.5d)\cos\beta - \{R^2 - (x - L + (R + 0.5d)\sin\beta)^2\}^{\frac{1}{2}}$$

Line CH:

Equation 12-9

$$y = (R + 0.5d)\cos\beta - \{(R + 0.5d)^2 - (x - L + (R + 0.5d)\sin\beta)^2\}^{\frac{1}{2}}$$

Line NP:

Equation 12-10

$$z = \frac{0.5(d - D)x}{L - 0.5d\sin\beta} + 0.5D$$

Line QR: (The line that separates the flow divider)

Equation 12-11

$$x = \left(1 + \frac{\tan\theta}{\tan\beta}\right)\frac{L}{2}$$

Equation 12-12

$$z = b_2\left(1 - \frac{\delta^2}{a_2^2}\right)^{\frac{1}{2}}$$

Equation 12-13

$$a_2 = \psi_2 - \psi_1$$

Equation 12-14

$$b_2 = \frac{0.5(d - D)}{L - 0.5d\sin\beta}\left(\psi_2\sin\theta + \frac{L}{2}\right) + 0.5D$$

Equation 12-15

$$\delta = \frac{L}{2\tan\beta\cos\theta} - \psi_2$$

Equation 12-16

$$\frac{L}{2\tan\beta} = \psi_1\cos\theta + a_1\left(\frac{L}{2} + \psi_1\sin\theta\right)^2 + b_1\left(\frac{L}{2} + \psi_1\sin\theta\right)^8 + 0.5D$$

Equation 12-17

$$\frac{L}{2\tan\beta} = \psi_2\cos\theta + a(\psi_2\sin\theta)^{\lambda}$$

Some things to note is that 'Line NP' start from O and end at K, and 'Line QR' start at G and end at C.

The geometrical parameter values are as follows.

- D=3.81

- $2d^2/D^2 = 1$ (Constant cross-sectional area at the flow divider)
- $2\alpha=70°$
- $\beta=18°$
- R=7d

Cartesian coordinates of points A and B are $(L-(R+0.5d)\sin\beta,(R+0.5d)\cos\beta,0)$ and $(L/2,\tan\beta(L/2),0)$ respectively.

Zhao Yao, L. B. B. (1994a). "Steady inspiratory flow in a model symmetric bifurcation." Journal of Biomechanical Engineering, (TASME) **116**: 488-496.