11-1-2013 12:00 AM

Numerical evaluation of aerodynamic roughness of the built environment and complex terrain

Daniel S. Abdi
The University of Western Ontario

Supervisor
Girma T. Bitsuamlak
The University of Western Ontario

Graduate Program in Civil and Environmental Engineering
A thesis submitted in partial fulfillment of the requirements for the degree in Doctor of Philosophy

NUMERICAL EVALUATION OF AERODYNAMIC ROUGHNESS OF THE BUILT ENVIRONMENT AND COMPLEX TERRAIN

(Thesis format: Monograph)

by

Daniel Abdi

Graduate Program in Civil and Environmental Engineering

A thesis submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy

The School of Graduate and Postdoctoral Studies
The University of Western Ontario
London, Ontario, Canada

Abstract

Aerodynamic drag in the atmospheric boundary layer (ABL) is affected by the structure and density of obstacles (surface roughness) and nature of the terrain (topography). In building codes and standards, average roughness is usually determined somewhat subjectively by examination of aerial photographs. For detailed wind mapping, boundary layer wind tunnel (BLWT) testing is usually recommended. This may not be cost effective for many projects, in which case numerical studies become good alternatives. This thesis examines Computational Fluid Dynamics (CFD) for evaluation of aerodynamic roughness of the built environment and complex terrain.

The present study started from development of an in-house CFD software tailored for ABL simulations. A three-dimensional finite-volume code was developed using flexible polyhedral elements as building blocks. The program is parallelized using MPI to run on clusters of processors so that micro-scale simulations can be conducted quickly. The program can also utilize the power of latest technology in high performance computing, namely GPUs. Various turbulence models including mixing-length, RANS, and LES models are implemented, and their suitability for ABL simulations assessed.

Then the effect of surface roughness alone on wind profiles is assessed using CFD. Cases with various levels of complexity are considered including simplified models with roughness blocks of different arrangement, multiple roughness patches, semi-idealized urban model, and real built environment. Comparison with BLWT data for the first three cases showed good agreement thereby justifying explicit three-dimensional numerical approach. Due to lack of validation data, the real built environment case served only to demonstrate use of CFD for such purposes.

Finally, the effect of topographic features on wind profiles was investigated using CFD. This work extends prior work done by the research team on multiple idealized two-dimensional topographic features to more elaborate three-dimensional simulations. It is found that two-dimensional simulations overestimate speed up over crests of hills and also show larger recirculation zones. The current study also emphasized turbulence characterization behind hills. Finally a real complex terrain case of the well-known Askervein hill was simulated and the results validated against published field observations. In general the results obtained from the current simulations compared well with those reported in literature.

Keywords: Computational fluid dynamics, Aerodynamic roughness, Complex terrain, Atmospheric boundary layer, Parallel CFD, Turbulence modeling

Co-Authorship Statement

In Chapter 3, a conference paper titled 'Application of an artificial neural network model for boundary layer wind tunnel profile development' is extracted and published with Simon Levin, who provided the BLWT data for training and validation of the ANN model.

Other papers have been extracted from the cores of Chapters 3,4,5 and submitted for publishing in Journal articles.

Acknowlegements

I would like to thank my major advisor Dr. Girma T. Bitsuamlak for supervising my work here at Western and also at FIU where the research is started. This work would not have been possible with out his active support and daily involvement. I would also like to express my sincere gratitude to Dr. Ashraf El Damatty, Dr Horia M. Hangan, Dr. Fernando Miralles, Dr. Arindam G. Chowdhury and Dr. Ping Zhu for their constructive input in the completion of this research.

I would like to acknowledge the Civil and Environmental Engineering Department at Western and FIU, CEATI International Inc., and the National Science Foundation (NSF) for their financial support of my doctoral research.

I would also like to thank the SHARCNET and MAIDROC high performance computational facility centers that were instrumental for the completion of the study. The help and the moral support I have received from my friends and colleagues is also appreciated.

Finally, I would like to thank my parents and sisters for their unconditional love, encouragement and support to my academic career.

Contents

Abstract ii

Co-Authorship Statement iii

Acknowlegements iv

Nomenclature x

Abbreviations xiii

List of Figures xvi

List of Tables xxii

1 Introduction 1
- 1.1 Overview . . . 1
  - 1.1.1 Methods for investigation of atmospheric flow over topography . . . 1
  - 1.1.2 Effect of roughness on atmospheric boundary layer flow . . . 3
- 1.2 Objectives and scope . . . 4

2 Background 7
- 2.1 Atmospheric boundary layer . . . 7
- 2.2 Modification of ABL by topographic features . . . 8
- 2.3 Modification of ABL by surface roughness . . . 11
- 2.4 ABL stratification and stability . . . 13
- 2.5 Coriolis force . . . 14
- 2.6 Statistics on wind turbulence . . . 15
  - 2.6.1 Spectral content of wind . . . 17
  - 2.6.2 Mean wind speed and turbulence intensity models . . . 19
    - 2.6.2.1 The log law model . . . 19
    - 2.6.2.2 The power law model . . . 20
- 2.7 Surface roughness models . . . 20
  - 2.7.1 Empirical formulas . . . 20
  - 2.7.2 BLWT methodology . . . 25
  - 2.7.3 CFD methodology . . . 27
- 2.8 Computational wind engineering . . . 28
- 2.9 Overview of CFD . . . 29

2.9.1 Governing equations . . . 30
2.9.1.1 Mass conservation law . . . 30
2.9.1.2 Momentum conservation law . . . 30
2.9.2 Turbulence models . . . 32
2.9.2.1 Reynolds Averaged Navier Stokes . . . 32
2.9.2.2 Linear eddy viscosity models . . . 33
2.9.2.3 Non-linear eddy viscosity models . . . 36
2.9.2.4 Reynolds stress models (RSM) . . . 36
2.9.2.5 Modeling flow near wall . . . 37
2.9.2.6 Large eddy simulations . . . 39
2.9.3 Finite volume discretization . . . 39
2.9.3.1 Convection discretization . . . 41
2.9.3.2 Diffusion discretization . . . 43
2.9.3.3 Source term discretization . . . 43
2.9.3.4 Temporal discretization . . . 44
2.9.4 Boundary conditions . . . 45
2.9.5 Calculation of flow field . . . 46

3 Implementation of 3D CFD program 48
3.1 Tensors . . . 48
3.2 Fields . . . 50
3.3 Equation discretization . . . 51
3.4 Overview of components of CFD tool . . . 52
3.4.1 Partial differential equation solvers . . . 52
3.4.1.1 Wall distance solver . . . 52
3.4.1.2 Potential flow solver . . . 53
3.4.1.3 Parabolic diffusion solver . . . 54
3.4.1.4 Transport equation solver . . . 54
3.4.1.5 Navier-Stokes solver . . . 54
3.4.2 Meshing . . . 55
3.4.3 Solution and turbulence modeling . . . 55
3.4.4 Parallelization . . . 56
3.5 Development of high performance CFD code . . . 57
3.5.1 Domain decomposition . . . 57
3.5.2 Platform for high end simulation . . . 57
3.5.3 Parallel computing . . . 59
3.5.3.1 Coarse grained parallelism . . . 59
3.5.3.2 Fine grained parallelism . . . 60
3.5.4 Relaxation algorithms . . . 60
3.5.5 Preconditioning . . . 62
3.5.6 Parallel implementations . . . 64
3.5.7 Asynchronous implementation . . . 67
3.5.8 Scalability study . . . 69
3.5.8.1 Coarse-grained scalability study . . . 69
3.5.8.2 Fine grained scalability study . . . 69

3.5.9 Validation with benchmark problems . . . 71
3.5.9.1 Lid-driven cavity . . . 71
3.5.9.2 Flow around a bluff body . . . 74

4 Numerical evaluation of roughness effects 77
4.1 Complexity 0: Empty domain . . . 78
4.1.1 Computational domain . . . 78
4.1.2 Boundary conditions . . . 79
4.1.3 Simulation for different cases . . . 81
4.2 Complexity 1: Homogeneous roughness evaluation . . . 84
4.2.1 Test setup . . . 84
4.2.2 Analysis . . . 86
4.3 Complexity 2: Inhomogeneous roughness evaluation . . . 88
4.3.1 Homogeneous roughness wind speed models . . . 89
4.3.1.1 Roughness estimation . . . 89
4.3.1.2 Models . . . 90
4.3.2 The ESDU model . . . 91
4.3.2.1 Wind speed model (ESDU 82026) . . . 91
4.3.2.2 Turbulence intensity model (ESDU 84030) . . . 92
4.3.3 The WS model . . . 93
4.3.3.1 Wind speed model . . . 93
4.3.3.2 Turbulence intensity model . . . 94
4.3.4 Comparison of WS and ESDU models . . . 94
4.3.5 Three dimensional CFD simulations . . . 96
4.3.5.1 Simulations on a row of roughness elements . . . 97
4.3.5.2 Simulation of a BLWT with spires and barriers . . . 99
4.3.5.3 Simulation of multiple cases with a virtual Wind tunnel . . . 101
4.3.5.4 Simulation of WS cases using simplified 3D models . . . 105
4.4 Complexity 3: Semi-idealized built environment . . . 109
4.4.1 Computational domain setup and grid generation . . . 109
4.4.2 Boundary conditions . . . 110
4.4.3 Results and discussion . . . 110
4.5 Complexity 4: Built environment . . . 114
4.5.1 Computational domain setup and grid generation . . . 115
4.5.2 Boundary conditions . . . 115
4.5.3 Results and discussion . . . 115
4.6 Prediction with artificial neural networks . . . 118
4.6.1 Data acquisition . . . 118
4.6.2 Artificial neural network model . . . 120
4.6.3 Results and discussion . . . 121
4.6.3.1 Wind profile prediction . . . 121
4.6.3.2 Estimation of tunnel surface roughness and spire dimensions . 121
4.6.4 Conclusions . . . 121

5 Numerical evaluation of orographic effects 124

5.1 Wind speed up over topography . . . 125
5.1.1 Building codes and standards . . . 125
5.1.2 Numerical studies . . . 126
5.1.3 Analytical study of flow over low hills . . . 128
5.1.4 BLWT studies . . . 130
5.1.5 Description of test cases of the current study . . . 130
5.1.6 Ground surface representation and mesh generation . . . 133
5.1.7 Computational domain setup . . . 134
5.1.8 Grid independence study . . . 135
5.1.9 Results and discussion . . . 135
5.1.10 Conclusions . . . 138
5.2 Turbulence structure . . . 148
5.2.1 Background . . . 148
5.2.2 Turbulence models . . . 150
5.2.2.1 Mixing length model . . . 150
5.2.2.2 K-epsilon models . . . 151
5.2.2.3 LES models . . . 152
5.2.3 Wall models . . . 153
5.2.4 Simulation results and discussions . . . 154
5.2.4.1 Effect of turbulence models . . . 154
5.2.4.2 Roughness effects . . . 169
5.2.4.3 Scheme sensitivity . . . 171
5.2.5 Conclusions . . . 172
5.3 Wind flow simulations on real complex terrain . . . 173
5.3.1 Askervein hill case study . . . 174
5.3.1.1 Computational domain setup and grid generation . . . 174
5.3.1.2 Grid independence study . . . 176
5.3.1.3 Different turbulence models . . . 176
5.3.1.4 Comparison with field measurements . . . 178
5.3.2 A second complex hill simulation . . . 179

6 Conclusions and future work **182**
6.1 High performance CFD code . . . 182
6.2 Effect of roughness . . . 183
6.3 Effect of topographic features . . . 184
6.4 Future work . . . 186

Bibliography **196**

A Plots of wind speed model **197**

B Artificial neural network source code **225**

C CFD program **234**
C.1 Brief information on usage . . . 234

C.2 Source code 238

Curriculum Vitae **418**

Nomenclature

α	Power law coefficient
Δ	LES filter width
ΔS	Relative speed up ratio
Δt	Time step
ϵ	Turbulent energy dissipation rate
$\hat{g}$	Peak factor
κ	von Karmann constant
λ_f	Frontal area density ratio
λ_p	Planar area density ratio
μ	Dynamic viscosity
ν	Kinematic viscosity
ν_t	Turbulent viscosity
Ω	Vorticity
$\overline{U}$	Mean wind speed
ϕ	Any flow variable
ρ	Density of air
τ	Surface shear stress
$\tau_{Reynolds}$	Surface Reynolds shear stress

$\tau_{xy}, \tau_{xz}, \tau_{yz}$ Surface shear stresses on different planes

A_d Total area of obstacles

A_f Frontal area of obstacles

A_p Planar area of obstacles

c, d Inverse power law coefficients of ASCE-7

C_D Drag coefficient

C_s Smagorinsky constant

C_{ks} Roughness constant

Co Courant number

d Displacement height

F Body force

f_c Coriolis parameter

G Gradient height

g Gravitational acceleration 9.8 m/s^2

H Height of obstacle such as hill or blocks

I Integral time scale

I_u Longitudinal turbulence intensity

I_v Vertical turbulence intensity

I_w Transverse turbulence intensity

k Turbulent kinetic energy

K_s Sand grain roughness

L_u Longitudinal length scale of turbulence

l_{mix}, l_m Mixing length

M_t	Topographic modification factor
p	Pressure
P_e	Peclet number
Re	Reynolds number
Ro	Rossby number
T	Deviatoric component of stress tensor
U	Instantaneous wind speed
u'	Fluctuating component of wind speed
$U+$	Dimensionless velocity
U_*	Friction velocity
U_p	Horizontal velocity component at the first near wall cell
V	Vertical velocity component
v'	Fluctuating component of vertical wind speed
W	Transverse velocity component
w'	Fluctuating component of transverse wind speed
x	Longitudinal axis
X_f	Fetch length
y	Transverse axis
$y+$	Dimensionless wall coordinate
Y_p	Perpendicular distance to the wall from nearest cell
z	Vertical axis
z_0	Surface roughness length
z_{ref}	Reference height at which wind speed is measured

Abbreviations

ABL Atmospheric Boundary Layer.

ANN Artificial Neural Network.

AS/NZS 1170-2 Australian/New Zealand Standard.

ASCE7 American Society of Civil Engineers - 7.

BLWT Boundary Layer Wind Tunnel.

CCNN Cascade Correlation Neural Network.

CFD Computational Fluid Dynamics.

CPU Central Processing Unit.

CUDA Compute Unified Device Architecture.

CWE Computational wind engineering.

DNS Direct Numerical Simulation.

ESDU Engineering Science Data Unit.

Eurocode I European Standard.

FDM Finite Difference Method.

FEM Finite Element Method.

FSUR Fractional Speed Up Ratio.

FVM Finite Volume Method.

GPGPU General Purpose Graphic Processing Unit.

GPU Graphic Processing Unit.

HPC High Performance Computing.

IBL Internal Boundary Layer.

IHRC International Hurricane Research Center.

LDV Laser Doppler Velocimetry.

LES Large Eddy Simulation.

MPI Message Passing Interface.

MPNN Multilayer Perceptron Neural Network.

NBCC National Building Code of Canada.

PBiCG Preconditioned Bi Conjugate Gradient.

PCG Preconditioned Conjugate Gradient.

PDE Partial Differential Equation.

PISO Pressure Implicit with Splitting Operators.

PIV Particle Image Velocimetry.

RANS Reynolds Averaged Navier-Stokes.

Re Reynolds number.

RMS root mean square.

Ro Rosby number.

RWDI Rowan Williams Davies and Irwin Incorporation.

S-BLWT Symmetric Virtual Boundary Layer Wind Tunnel.

SHARCNET Shared Hierarchial Academic Research Computing Network.

SIMPLE Semi Implicit Method for Pressure Linked Equations.

SOR Successive Over Relaxation.

TVD Total Variation Diminishing.

USGS United States Geological Survey.

V-BLWT Virtual Boundary Layer Wind Tunnel.

WAsP Wind Atlas Analysis Application Program.

WRF Weather Research and Forecasting.

WS Wang and Sthatopoulos Model.

List of Figures

2.1 Structure of Atmospheric Boundary Layer (ABL) (Crasto (2007)) 8
2.2 Speed up on isolated hill (National Building Code of Canada (NBCC) 1995) . . 9
2.3 Hills, escarpments and valleys of different slope 10
2.4 Double hills, funneling between two hills, and speed up inside a valley 11
2.5 Effect of stability on wind flow over hill 14
2.6 Time series of wind turbulence (left) and its spectrum (right) (Stull 1988) . . . 16
2.7 The von Karman-Harris spectra for longitudinal wind speed 18
2.8 Roughness length and displacement height for square and staggered blocks (Peterson 1994) 22
2.9 Comparison of different empirical models for roughness length 23
2.10 Displacement height for different convexity 24
2.11 Roughness length for different convexity 25
2.12 The law of the wall expressed with wall coordinates y^+ and U^+ 37

3.1 Contour map of wall distance from the surface of a 2D hill 53
3.2 MAIDROC tesla cluster at FIU with 2 x 64=128 cores 58
3.3 SHARCNET cluster, a network of high-performance computers 58
3.4 A 5-point stencil with halo layer for exchanging information between processors 65
3.5 Red-black colored graph for parallel Gauss-Siedel 66
3.6 Graphic Processing Unit (GPU) speed up relative to Central Processing Unit (CPU) for fixed number of iterations 70
3.7 Streamlines for different Reynolds numbers showing progressive formation of eddies at the bottom right corner → bottom left corner → top right corner . . . 72
3.8 Streamlines (left) and pressure contours (right) of lid-driven cavity flow at Re=1000 72
3.9 Horizontal(u) and vertical(v) velocity profiles along mid vertical and horizontal sections respectively 73

3.10 Solution of 3D lid-driven cavity problem solved parallely with 16 sub-domains (left), and the resulting 3D iso-surface plot that shows the flow pattern (right) . 73
3.11 Grid for a cube in a boundary layer case of Kose & Dick (2010) 74
3.12 Plots of instantaneous and mean velocity contours showing vortex shading behind the cube 75
3.13 Pressure coefficients along vertical section of cube. Adapted from Bitsuamlak et al. (2010) 75

4.1 Profiles of horizontal velocity, turbulent kinetic energy and dissipation for case-1 83
4.2 Profiles of horizontal velocity, turbulent kinetic energy and dissipation for case-2 83
4.3 Profiles of horizontal velocity, turbulent kinetic energy and dissipation for case-3 83
4.4 Profiles of horizontal velocity, turbulent kinetic energy and dissipation for case-4 83
4.5 Plan of three symmetric configurations: Staggered arrays(left), regular arrays (middle) and 45^0 wind attack on uniform array (right) 85
4.6 Plan of regular array of cubes with height H and spacing 1.5H also showing location of probes 85
4.7 Spatial variation of velocity profiles: longituidinal (left) and transverse(right) . 87
4.8 Sample measured and logarithmic fitted velocity profiles 87
4.9 Comparison of CFD with different roughness models 87
4.10 Effect of staggered placement on z_0 (left) and comparison of CFD and Theurer model for d (right) 87
4.11 Schematics of the growth of internal boundary layer for single roughness change 89
4.12 Schematics of change in velocity profile for three roughness patches.(Wang & Stathopoulos 2007*b*) 93
4.13 Comparison of Wang and Sthatopoulos Model (WS) and Engineering Science Data Unit (ESDU) models on selected cases 96
4.14 A look inside of a 3D symmetrical computational domain for regular array of cubes. The 2D plan of the model is previously explained in Fig.4.5 97
4.15 Velocity contours for different roughness characteristics showing isolated (open-terrain), wake-interference (sub-urban) and skimming flow (urban). 98
4.16 Virtual Boundary Layer Wind Tunnel (V-BLWT) simulation results with surface roughness blocks 100
4.17 Virtual Boundary Layer Wind Tunnel (BLWT) simulation with spires, barrier and roughness blocks 102
4.18 Comparison of U and Iu profiles for different roughness features 103

4.19 Open country(OC), Suburban(S) and Urban(U) roughness representation 103
4.20 Inlet and outlet horizontal velocity profiles for open surface roughness 104
4.21 Perspective view computational domain of a virtual BLWT 105
4.22 Horizontal velocity comparison of CFD with existing models for cases 1-8 . . . 106
4.23 Turbulence intenisy comparison of CFD with existing models for cases 1-8 . . . 107
4.24 Horizontal velocity contour for V-BLWT configuration of cases 1-8 108
4.25 Semi-idealized urban model from CEDVAL database 111
4.26 Plan of the semi-idealized urban model 111
4.27 Inside view of the mesh generated for the semi-idealized urban model 111
4.28 Velocity contours at different elevations 112
4.29 Velocity vectors at the core of the urban canyon 112
4.30 Comparison between Computational Fluid Dynamics (CFD) and BLWT for some of probe locations inside the model 113
4.31 Surface model of a region in downtown Miami 116
4.32 Building edges and corresponding mesh generated by snappyHexMesh 116
4.33 Velocity contours at 5m height for different grid sizes in the vertical direction . 117
4.34 Velocity contours at different heights 117
4.35 Rowan Williams Davies and Irwin Incorporation (RWDI) wind tunnel working Section, spire and roughness blocks 119
4.36 Neural network model with CCNN architecture for roughness estimation: 3 input, 20 hidden and 2 output neurons 120
4.37 Measured versus predicted velocity and turbulence intensity profiles 122

5.1 Transmission line with multiple towers crossing a hill 125
5.2 Speed up factors at x=0(crest), x = L/2 and x = L of a 2D steep hill using various building codes. 127
5.3 Speed up factors at x=0(crest), x = L/2 and x = L of a 2D shallow hill using various building codes. 127
5.4 Flow regimes for flow over a low hill. Adapted from Jackson & Hunt (1975) . . 130
5.5 Wind speed up over a single hill (NBCC) 131
5.6 An escarpment 132
5.7 Double hills 132
5.8 An isolated valley 133
5.9 Computational domain for double 2D hills 134

5.10 Mesh refinement around hills: Background mesh (top-left), box refinement around hill (bottom-left), Planar view of refinement for triple hills (top-right), and close up view of layers towards the ground (bottom-left). 136
5.11 Grid independence study on single 2D hill: wind profiles at crest (left) and close-up view of maximum speed up region(right) 136
5.12 Single shallow hill Fractional Speed Up Ratio (FSUR) color maps and line plots and comparison of 2D and 3D simulation results 139
5.13 Single steep hill FSUR color maps and line plots and comparison of 2D and 3D simulation results 140
5.14 Double shallow hills FSUR color maps and line plots and comparison of 2D and 3D simulation results 141
5.15 Double steep hills FSUR color maps and line plots and comparison of 2D and 3D simulation results 142
5.16 Triple shallow hills FSUR color maps and line plots and comparison of 2D and 3D simulation results 143
5.17 Triple steep hills FSUR color maps and line plots and comparison of 2D and 3D simulation results 144
5.18 Single shallow valley FSUR color maps and line plots and comparison of 2D and 3D simulation results 145
5.19 Single steep valley FSUR color maps and line plots and comparison of 2D and 3D simulation results 146
5.20 Empty domain FSUR color maps and line plots 147
5.21 Escarpement FSUR color maps and line plots 147
5.22 Horizontal velocity fluctuation on upstream(dotted) and crest(solid) of sinusoidal hills (Miller & Davenport 1998) 148
5.23 Horizontal normal stress σ_h/U_∞ profiles(Takeshi et al. 1999) 149
5.24 Mean horizontal velocity for shallow and steep a) isolated hill b) double hills c) triple hills d) isolated valley at 20m height from full scale simulations 156
5.25 Results for single shallow hill: TKE, horizontal velocity U,fluctuations u' and w', and Reynolds stresses 157
5.26 Results for single steep hill: TKE, horizontal velocity U,fluctuations u' and w', and Reynolds stresses 158
5.27 Results for escarpment: TKE, horizontal velocity U,fluctuations u' and w', and Reynolds stresses 159

5.28 Results for double shallow hills: TKE, horizontal velocity U,fluctuations u' and w', and Reynolds stresses . . . 160
5.29 Results for double steep hills: TKE, horizontal velocity U,fluctuations u' and w', and Reynolds stresses . . . 161
5.30 Results for triple shallow hills: TKE, horizontal velocity U,fluctuations u' and w', and Reynolds stresses . . . 162
5.31 Results for triple steep hills: TKE, horizontal velocity U,fluctuations u' and w', and Reynolds stresses . . . 163
5.32 Results for single shallow valley: TKE, horizontal velocity U,fluctuations u' and w', and Reynolds stresses . . . 164
5.33 Results for single steep valley: TKE, horizontal velocity U,fluctuations u' and w', and Reynolds stresses . . . 165
5.34 Horizontal velocity contour plots for model scale simulations (Re=12000) using LES and RANS models . . . 166
5.35 Instantaneous velocity contours of Large Eddy Simulation (LES) simulations for all the 2D hills: single shallow hill, single steep hill, double shallow hill, double steep hill, triple shallow hill, triple steep hill . . . 167
5.36 Mean velocity contours of LES simulations for all the 2D hills: single shallow hill, single steep hill, double shallow hill, double steep hill, triple shallow hill, triple steep hill . . . 168
5.37 Mean velocity contours of Reynolds Averaged Navier-Stokes (RANS) simulations for all the 2D hills: single shallow hill, single steep hill, double shallow hill, double steep hill, triple shallow hill, triple steep hill . . . 169
5.38 Mean horizontal velocity for shallow and steep a) isolated hill b) double hills c) triple hills d) isolated valley at 20m height from model scale simulations . . . 170
5.39 Mean velocity and TKE for different roughness lengths . . . 171
5.40 Mean velocity and TKE comparison for different convection discretization schemes 172
5.41 Contour map of Askervein hill showing Line-A and the hill top (HT) . . . 175
5.42 Elevation map of Askervein hill including surrounding . . . 175
5.43 Coarse and fine meshes of Askervein hill with surrounding hills . . . 177
5.44 Normalized horizontal velocity for different size of grids . . . 177
5.45 Normalized horizontal velocity for different turbulence models . . . 178
5.46 Normalized horizontal velocity comparison with field measurements . . . 179

5.47 A randomly selected complex hill with sharp edges and its mesh (top), elevation contour (bottom) 180
5.48 Contours of horizontal velocity and TKE 181

A.1 Horizontal velocity comparison of CFD with existing models for cases 1-8 . . . 198
A.2 Horizontal velocity comparison of CFD with existing models for cases 9-16 . . 199
A.3 Horizontal velocity comparison of CFD with existing models for cases 17-24 . 200
A.4 Horizontal velocity comparison of CFD with existing models for cases 25-32 . 201
A.5 Horizontal velocity comparison of CFD with existing models for cases 33-40 . 202
A.6 Horizontal velocity comparison of CFD with existing models for cases 41-48 . 203
A.7 Horizontal velocity comparison of CFD with existing models for cases 49-56 . 204
A.8 Horizontal velocity comparison of CFD with existing models for cases 57-64 . 205
A.9 Horizontal velocity comparison of CFD with existing models for cases 65-69 . 206
A.10 Turbulence intenisy comparison of CFD with existing models for cases 1-8 . . . 207
A.11 Turbulence intenisy comparison of CFD with existing models for cases 9-16 . . 208
A.12 Turbulence intenisy comparison of CFD with existing models for cases 17-24 . 209
A.13 Turbulence intenisy comparison of CFD with existing models for cases 25-32 . 210
A.14 Turbulence intenisy comparison of CFD with existing models for cases 33-40 . 211
A.15 Turbulence intenisy comparison of CFD with existing models for cases 41-48 . 212
A.16 Turbulence intenisy comparison of CFD with existing models for cases 49-56 . 213
A.17 Turbulence intenisy comparison of CFD with existing models for cases 57-64 . 214
A.18 Turbulence intenisy comparison of CFD with existing models for cases 65-69 . 215
A.19 Horizontal velocity contour for V-BLWT configuration of cases 1-8 216
A.20 Horizontal velocity contour for V-BLWT configuration of cases 9-16 217
A.21 Horizontal velocity contour for V-BLWT configuration of cases 17-24 218
A.22 Horizontal velocity contour for V-BLWT configuration of cases 25-32 219
A.23 Horizontal velocity contour for V-BLWT configuration of cases 33-40 220
A.24 Horizontal velocity contour for V-BLWT configuration of cases 41-48 221
A.25 Horizontal velocity contour for V-BLWT configuration of cases 49-56 222
A.26 Horizontal velocity contour for V-BLWT configuration of cases 57-64 223
A.27 Horizontal velocity contour for V-BLWT configuration of cases 65-69 224

List of Tables

2.1 Revised Davenport roughness classification (Wieringa 1992) 12
2.2 Summary of empirical formulas for roughness parameters 25

3.1 Speed ups for 256 x 256 case 70
3.2 Speed ups for 1024 x 1024 case 70

4.1 Multiple roughness patch cases considered 95
4.2 Roughness features dimensions 119
4.3 Measured and Artificial Neural Network (ANN) predicted roughness length bottom spire width difference 123

5.1 NBCC parameters for speed up ratio 126
5.2 Wind speed data at 10m height 179

Chapter 1

Introduction

1.1 Overview

Computational wind engineering (CWE) is an inter-disciplinary field that uses Computational Fluid Dynamics (CFD) as the basic tool for studying wind effects on structures and the built environment in general. Some commonly conducted CWE studies include design of buildings for wind loads, assessment of wind hazards due to hurricanes and tornadoes, wind energy production assessment, pollutant dispersion in the built environment etc. Advances in computing technology allow for conducting bigger and more refined simulations with in a short time, thus CFD programs should be written to exploit future computing hardware. New numerical model development and validation with experiment is also an important aspect of CWE. The current work focuses on aspects of CWE regarding atmospheric simulations in the boundary layer and exploiting future computing hardware to accelerate CFD simulations. The particular subject of study is estimation of aerodynamic roughness, which is important in the development of models that represent atmospheric processes ranging from the mircoscale to the mesoscale (Hansen 1993).

1.1.1 Methods for investigation of atmospheric flow over topography

Numerical modeling of the atmosphere has been successfully used for weather prediction (NWP) since it was first introduced by Richardson (1922). One such NWP model is the Weather Research and Forecasting (WRF) model that is used for weather forecasting as well as research in extreme weather conditions such as hurricanes. NWP simulations concern the upper part of the atmosphere and usually have resolutions in the order of 5km. The resulting resolution is usually too coarse to fully understand flow behavior near the ground in a complex

terrain or urban exposures, and as such cannot be directly used for wind engineering purposes that require more detail about wind flow characteristics. Despite these limitations, mesoscale ($\geq 5km$) simulations can be used as boundary conditions for detailed microscale ($\leq 5km$) simulations. Also if the terrain is flat and has no obstacles, extrapolation of results from a height of 4km to the ground may be acceptable (Rasoulli 2010). However that is rarely the case, and such extrapolations on complex terrain are often incorrect. Therefore it is necessary to conduct micro-scale simulations that take in to consideration topographic features and built environment as a whole when detailed wind flow characteristics are required, which is the topic of the current research.

Among experimental methods in wind engineering, Boundary Layer Wind Tunnel (BLWT) testing is industry accepted and most widely used for studying wind flow characteristics in an area of size 5-30 km, mainly because it is relatively cheaper and takes less time compared to field investigations. Numerical simulation is an attractive alternative to BLWT that can further reduce the associated cost and time of investigation. BLWT studies over scaled models of microscale size have been used to obtain detailed wind maps. Many such studies have been carried out at University of Western Ontario BLWTs. A recent study conducted on complex terrain used recent technology for instrumentation, namely Particle Image Velocimetry (PIV), to make simultaneous measurements of the wind field (Rasoulli 2010). Commonly used instrumentation methods, such as hot-wire anemometers and Laser Doppler Velocimetry (LDV), are limited to point measurements unlike the PIV method. The PIV method, despite low sampling rate, has been found to give comparable results with the other instrumentation techniques. The major problems in BLWT testing are model preparation, terrain roughness modeling, and instrumentation.

Field observations of wind characteristics using cup-anemometers placed at specific locations usually take months to complete and are also expensive. Moreover data is gathered only at specific locations in the area, hence the data for rest of the study area have to be extrapolated from those few point measurements at meteorological towers. However field measurements have been successfully used to measure maximum wind speed up factors over hills, escarpments, ridges and other topographic features. The results from these experiments are incorporated in building codes and standards such as American Society of Civil Engineers - 7 (ASCE7) and National Building Code of Canada (NBCC). Extensive field measurements over a real complex terrain is rarely conducted due to associated high cost and time needed for the investigation, but in some cases such data is necessary for benchmarking CFD simulations and BLWT testing. For example, Taylor & Teunisson (1986) conducted extensive field measure-

ments over the Askervein hill which is now commonly used as a benchmark for validating complex terrain CFD codes. Larger scale field studies have been conducted by Grant & Mason (1990), that have studied boundary layer structure over complex terrain by flying balloons at high altitudes and taking simultaneous measurements.

1.1.2 Effect of roughness on atmospheric boundary layer flow

A typical urban surface is extremely complex, with towns and cities consisting of large buildings with various shapes, sizes, and distributions which protrude into the atmosphere and interfere with the atmospheric aerodynamic and radiative heat transfer processes (Arnfield 2003). According to the modeling results of Coceal & Belcher (2005), either a change of building density or a change of building height has a direct impact on the mean flow, with the largest difference occurring near ground level. Buildings exert a resistive drag on the air flow and also complicate interactions through turbulent wakes and mutual sheltering. Therefore the relationship between wind profiles, roughness length and surface morphological characteristics is important. Surface roughness characteristics of an urbanized area can be predicted from wind speed data obtained from meteorological towers (EPA 1987). Wind speed measurements at different locations and height can be collected for long periods of time from which roughness parameters can be calculated.

In current practice, codes and building standards such as ASCE7-05 provide 3-s gust basic design wind speeds for open terrain conditions at 10m elevation, derived largely from meteorological stations at nearby airports. The wind speed at a particular study site will then have to be derived from the basic wind speeds through proper exposure corrections that reflect the roughness of the ground surface. The ground surface roughness lengths are usually estimated visually by examining aerial photographs or satellite images for each wind direction. These visually estimated values will be used in the logarithmic wind velocity profiles at a particular study site. For inhomogeneous upwind terrain condition and dense urban areas this task is even more complicated. Usually simplistic formulas are used to approximate the drag force based on average frontal and planar area of the obstacles (Counihan 1971, Lettau 1969, MacDonald et al. 1998, Theurer 1993). Some wind consulting offices use the Engineering Science Data Unit (ESDU) wind speed model (ESDU-82026 1993). ESDU uses equations fitted to data obtained by the Deaves & Harris (1978) numerical model over changes in surface roughness using a very simplified form of flow equation. An equivalent roughness can be obtained by considering the fetch length and associated roughness length for a particular wind direction. This equivalent roughness is then applied at the floor of a wind tunnel to generate wind test

profiles. In some case the change of roughness can be dramatic. For example, in downtown Miami the characteristic of the wind coming from the ocean will experience a sudden change from open (ocean) to urban (coastal community) and then to suburban within a few miles. These local, small scale roughness changes have significant effect on the velocity as well as turbulence profile (Wang & Stathopoulos 2007*a*).

1.2 Objectives and scope

The major objective of this thesis is to evaluate aerodynamic roughness of the built environment and complex topography by conducting micro-scale numerical simulations. To achieve this goal several specific goals will be pursued.

1. To develop a high performance CFD software tailored for Atmospheric Boundary Layer (ABL) simulations over the built environment and complex topography. The program will be parallelized using Message Passing Interface (MPI) to run on a cluster of processors such as the Shared Hierarchial Academic Research Computing Network (SHARCNET) cluster at Western. Latest technology in High Performance Computing (HPC), namely Graphic Processing Units (GPUs), will be exploited using NVIDIA's Compute Unified Device Architecture (CUDA). Different turbulence models suitable for ABL simulation will be implemented and tested for suitability of complex terrain simulations. The turbulence models to be implemented include linear mixing-length model, many Reynolds Averaged Navier-Stokes (RANS) models including k-epsilon and RNG k-epsilon, and the Smagornisky Large Eddy Simulation (LES) model. The code will be validated against well known benchmark cases including problems specific to wind engineering. The scope of the program does not include mesh generation for complex terrain even though structured mesh generation for simple models is supported. Instead the program imports mesh from advanced meshing software such as 'snappyHexMesh' of OpenFOAM or Gambit meshing software of ANSYS Fluent. The program will use flexible polyhedral meshing format that are robust for CFD simulations and also allow for easy refinements in regions of interest.

2. To analyze the effect of roughness on wind speed and turbulence using CFD simulations. Different levels of complexity of roughness element configuration will be considered. First simplified models with regularly arranged blocks, similar to the case in a BLWT, will be tested for wind coming from different directions. The simulation results will then

be compared against empirical formulas that use average frontal and planar area density ratios. Then the problem of multiple roughness patches on the upstream side of a building will be investigated. It is known that the roughness patches closest to the building have the most effect on wind loading (pressure distribution). In literature this effect has been tested mainly in BLWT but numerical modeling attempts were usually limited to simplified 2D simulations that model the effect of roughness using empirical formulas. This work will investigate 3D explicit modeling of roughness elements. First a Virtual Boundary Layer Wind Tunnel (V-BLWT) will be simulated by replicating all the roughness features such as spires, barrier and roughness blocks to examine the effect of each roughness element. Then spires and barrier will be dropped in the latter simulations, with the blocks remaining as the only roughness feature. Instead a fully developed boundary layer profile is directly applied at the inlet to account for the effect of the removed roughness features. This setup will be used to evaluate the effect of multiple roughness patches on wind profile using many test setups found in literature. Furthermore for roughness blocks that are arranged in a regular manner, the inherent symmetry is exploited to reduce the computational domain to a single row of blocks. Finally the complexity of the test models will be increased further so that they become more and more representative of a real urban environment. A semi-urbanized model from CEDVAL-LES (2011) will be used for validation against BLWT data.

3. To analyze the effect of topographic features on wind flow using CFD simulations. Topographic features are responsible for most of the modification of ABL flow. This fact is recognized in building codes and standards through specification of wind speed up factors based on the slope and height of orography for simple hill, escarpment and valley geometries. Most codes do not have recommendations for multiple topographic features placed one after the other. This work will extend the work done by Bitsuamlak et al. (2004) on multiple topographic features using 2D simulations to a more elaborate 3D CFD simulations and using various turbulence models. Comparisons will be made with results available in literature. Then simulation on a real complex topography for which field measurements are available will be conducted for validation. Parametric studies will be conducted for different resolutions of grid, different turbulence models and dimension of the computational domain.

The thesis is organized as follows. Following the introduction in Chapter 1, brief literature review is carried out on the effect of roughness and topographic features in Chapter 2. The chapter also discusses background on CFD and its applications in wind engineering. Most

relevant literature to the objectives of this thesis are discussed at the beginning of the following sections. Chapter 3 discusses the implementation and validation of a high performance CFD software using C++ and MPI/CUDA for parallelization. Chapter 4 analyzes the effect of roughness on wind speed using different arrangement of roughness elements. After gaining enough experience with simplistic models, a full virtual wind tunnel is simulated using different roughness features through which the effect of multiple roughness patches is assessed. Then the complexity of the model is increased further to an idealized built environment to complete the study on urban flow characterization. Chapter 5 discusses the study on the effect of topographic features on wind flow. Simulations on 2D and 3D isolated and multiple hills are carried out . Speed up factors are calculated along lines at different heights which is a useful information for design of long-span structures such as transmission towers. The turbulence structure behind hills is examined using different turbulence models. Chapter 6 summarizes the findings and conclusions of this research work.

Chapter 2

Background

2.1 Atmospheric boundary layer

The Atmospheric Boundary Layer (ABL) is the lowest portion of the atmosphere (1-2 km) that is under the direct influence of the surface of the earth and responds to surface forcing in an hour or less. It is one order of magnitude smaller than the troposphere ($\sim 10km$), that consists of about 80% of the atmosphere, and two orders of magnitude smaller than the atmosphere ($\sim 100km$). In this region flow quantities display rapid fluctuations, unlike in the free atmosphere that is turbulence free. The terrain shape, roughness, thermal conditions, evaporation are some factors that affect behavior of the ABL. The free atmosphere, shown in Fig. 2.1, is the region above the ABL where the effect of surface friction is negligible and the wind is geostrophic, i.e purely driven by pressure gradient and Coriolis force. The transition zone between the ABL and free atmosphere, from 100m to 1km, is known as the Ekman layer (Outer layer). It is a part of the ABL because surface friction still plays a role, but the effect of Coriolis force can no longer be ignored as in the surface layer. Usually Boundary Layer Wind Tunnel (BLWT) can not accommodate for Coriolis force hence the height up to which ABL can be simulated accurately is limited to 100m. However micro-scale simulations on complex terrain that fall in the Ekman layer and upper portions of the ABL should incorporate Coriolis effect.

A typical characteristic of an ABL flow, compared to uniform flow, is the development of a gradient in the tangential wind speed due to a no-slip condition at the surface. The dragging action of the surface on the wind, also known as aerodynamic drag, takes away momentum from the wind causing a velocity (momentum) deficit near the surface. The effect extends few hundred meters above the surface in which wind speed increases from zero at the ground to a maximum value at the gradient height above which it remains constant with height. The

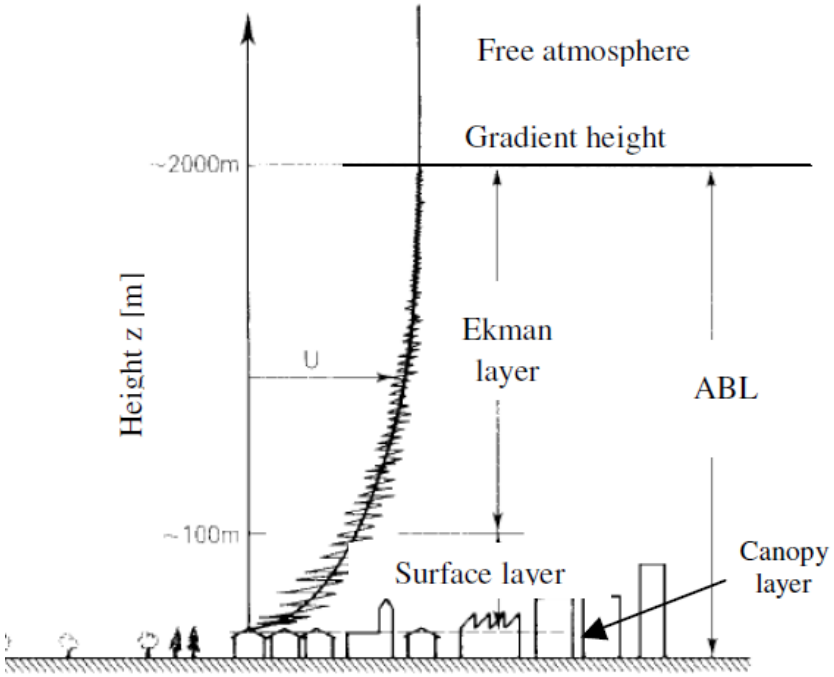


Figure 2.1: Structure of ABL (Crasto (2007))

roughness of the surface determines the gradient height above which the effect of surface drag is negligible. The power-law and log-law are commonly used to approximate mean wind speed profiles with in the surface layer for given surface roughness conditions.

2.2 Modification of ABL by topographic features

Topographical features such as escarpments, embankments, ridges, cliffs and hills can have a more profound effect on the flow in the ABL than any other single factor. It is known that wind speed increases significantly at the top of hills and ridges as shown in Fig.2.2. This phenomenon is exploited in wind energy to place wind turbines at optimal locations for maximum power production. Wind speed increases up the slope and reaches maximum at the crest or slightly upwind of it. Depending on the degree of steepness of the slope, flow separation may occur on the leeward side. It can also occur on the upstream side or any other location where there is a significant change in slope.

Building codes such as American Society of Civil Engineers - 7 (ASCE7) take into consideration the speed up effect over hills by the use of topographic multipliers (M_t in equation 2.1). The value of M_t is calculated from coefficients ($K_1 = f(H), K_2 = f(x), K_3 = f(z)$) that are read

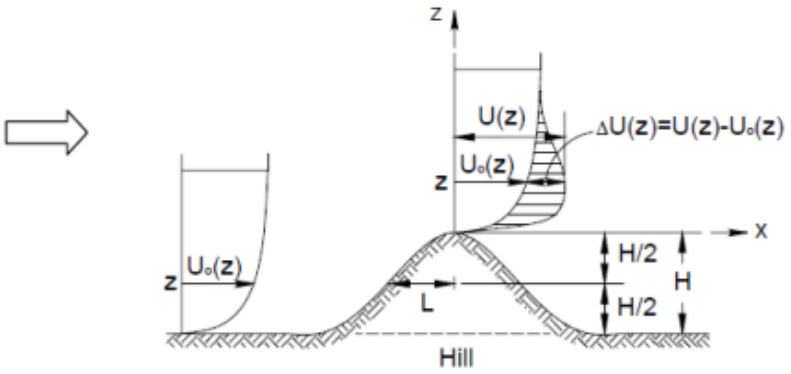


Figure 2.2: Speed up on isolated hill (NBCC 1995)

from a table for given dimensions of a topographic feature.

$$M_t = \frac{U_z(\text{at topographic feature})}{U_z(\text{at flat ground upstream})}$$

$$M_t = 1 + K_1 K_2 K_3 \tag{2.1}$$

ASCE7 states that wind speed-up effects shall be included in the design when all the following conditions are met.

1. The hill, ridge, or escarpment is isolated and unobstructed upwind by other similar topographic features of comparable height for 100 times the height of the topographic feature (100H) or 2 mi, whichever is less. This distance shall be measured horizontally from the point at which the height H of the hill, ridge, or escarpment is determined.

2. The hill, ridge, or escarpment protrudes above the height of upwind terrain features within a 2 mi radius in any quadrant by a factor of two or more.

3. The structure is located in the upper one-half of a hill or ridge or near the crest of an escarpment.

4. H/L ~ 0.2 and H is greater than or equal to 15 ft (4.5 m) for Exposures C and D and 60 ft (18 m) for Exposure B.

It is clear that the code is rather limited and does not extend beyond simple topographical features. Other codes such as NBCC also have similar limitations. These codes usually recommend BLWT experiments for complex terrains that do not meet the criteria.

Wind speed up over topography has both a good and bad side to wind engineers. The increase in wind speed over hills and escarpments causes structural failures if not properly accounted for. The wind load increases in proportion to the square of wind speed ($\sim U^2$), thus a 20% increase in wind speed translates to a 40% increase in wind load. For this reason, buildings and standards such as NBCC and ASCE7 provide guidelines for calculating wind speed up over idealized topography. Some of the orographic features for which codes provide guidelines are shown in Fig. 2.3. Most building codes do not have provisions for multiple

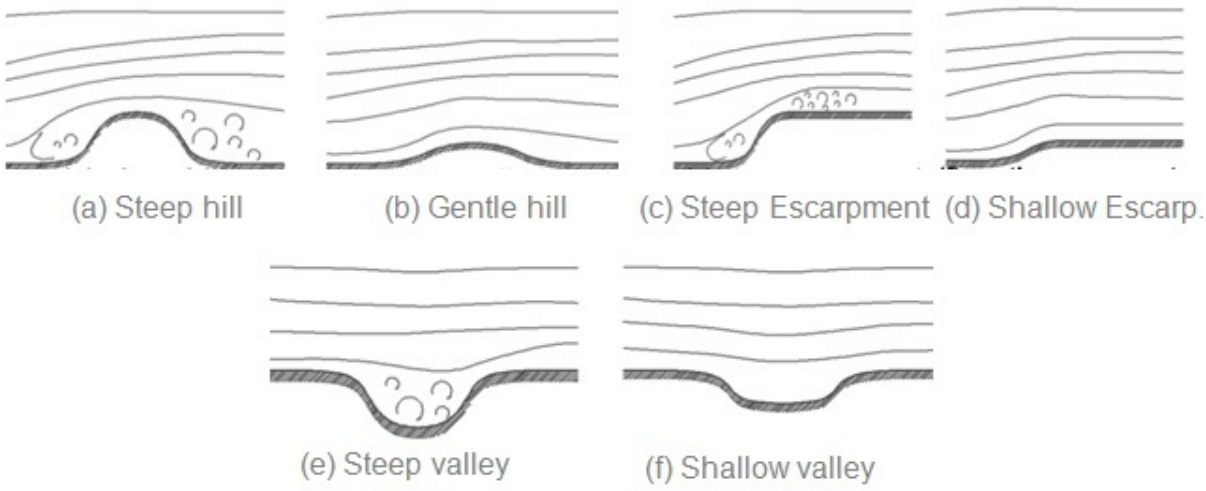


Figure 2.3: Hills, escarpments and valleys of different slope

hills and valleys placed consecutively. The wind speed typically reduces from the second hill towards associated with an increase in turbulence (Miller & Davenport 1998). When hills are placed side by side, funneling effects could significantly increase wind load for a structure placed in between the hills. Similar scenario is also observed inside a valley as shown in Fig. 2.4. These cases are not covered well in building codes and standards. An advantage of wind speed up over topographic features is that wind energy production can be maximized by placing wind turbines at locations where the wind speed is maximum (e.g. crest of hills) and the turbulence intensity is lowest. It is common to conduct Computational Fluid Dynamics (CFD) simulations to get a detail wind map of the area for micro-sitting (e.g.Uchida & Ohya (2008)). For these reasons, wind farm locations are usually located along shorelines, at highest elevations and area at which the surface cover is minimum (e.g. open roughness).

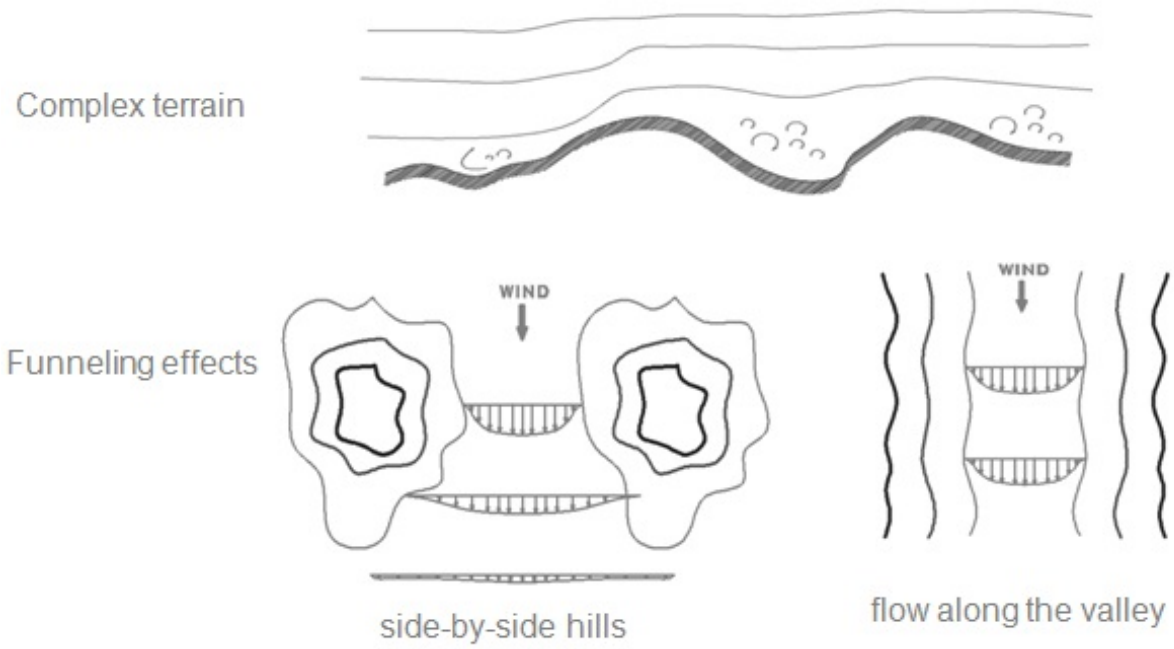


Figure 2.4: Double hills, funneling between two hills, and speed up inside a valley

2.3 Modification of ABL by surface roughness

Aerodynamic roughness comprises of both the effect of the terrain surface and its roughness elements. For a no-slip surface condition, the wind velocity drops from a large value at the free stream to zero at the surface, with in what is known as a boundary layer. The vertical gradient of horizontal velocity is a function of surface roughness. If the surface is smooth, the boundary layer is very thin. However as the surface roughness increases, the surface shear stress and hence velocity deficit in the wind profile also increase. A rough surface imposes larger aerodynamic drag than a smooth surface. For turbulence generated by wind shear, the magnitude of the surface Reynolds stress can be used as a scaling parameter. The shear stress and friction velocity are defined as shown in equations 2.2 and 2.3. The friction velocity U_* is incorporated in the log-law wind speed model as a scaling term.

$$|\tau_{Reynolds}| = [{\tau_{xz}}^2 + {\tau_{yz}}^2]^{1/2} \tag{2.2}$$

$$U_*^2 = |\tau_{Reynolds}|/\rho = [\overline{u'w'}^2 + \overline{v'w'}^2]^{1/2} \tag{2.3}$$

There is a disparity in the definition of roughness parameters z_0 and d. Panofsky & Dutton (1984) believed that the surface roughness length z_0 represents the size of the eddies produced

Table 2.1: Revised Davenport roughness classification (Wieringa 1992)

Class	Z_0(m)	Type	Landscape description
1	0.0002	Sea	open water, tidal flat, snow with fetch above 3 km
2	0.005	Smooth	featureless land, ice
3	0.03	Open	flat terrain with grass or very low vegetation, airport runway
4	0.1	Roughly open	cultivated area, low crops, obstacles of height H separated by at least 20 H
5	0.25	Rough	open landscape, scattered shelter belts, obstacles separated by 15 H or so
6	0.5	Very rough	landscape with bushes, young dense forest etc separated by 10 H or so
7	1.0	Closed	open spaces comparable with H, eg mature forest, low-rise built-up area
8	≥ 2	Chaotic	irregular distribution of large elements, eg city center, large forest with clearings

from the wind moving over a rough surface: the larger the eddies the larger the z_0 and vice versa. It can also be defined as the height above the physical surface at which flow starts to occur (Gardner 2004, Stangroom 2004). At this height z_0, the velocity is theoretically zero even though turbulence maybe present. Roughness length can be as high as 5m in city centers and large forests (Hansen 1993, Wieringa 1992). Building codes and standards such ASCE7-02 provide tables from which values of z_0 and α can be read based on land-use categories. This method can be quite effective in establishing representative roughness lengths (Hansen 1993). Hansen also suggests that this method can be expanded to include terrain features such as hills in a form (pressure) drag contribution. Three roughness categories (namely open, suburban and urban) are commonly considered for the purpose of determining wind loads on structures. Davenport et al. (2000) has proposed fine grained classification of terrain roughness using eight classes. Wieringa (1992) updated the Davenport roughness classification as shown in Table 2.1

There is also some disparity in the definition of the displacement height. This thesis uses the definition that the zero plane displacement height d is the height above the surface at which turbulent exchanges begin to occur, and is comparable to the depth of trapped air (Hansen 1993, Monteith 1965, Stangroom 2004, Thom 1972). In a closed city center, the density of the obstacles may prevent any flow effects from occurring between the buildings, thereby forming a canopy layer where the air is effectively trapped inside. A phenomenon known as 'skimming flow' occurs, where the ground plane is effectively displaced up by an amount d in to the rough-

ness elements. The displacement height does not change the shape of the velocity profile but only displaces it upward by an amount d. The roughness length in this case is measured from the plane of zero-displacement or in other words at $z_0 + d$ above the ground. Both roughness length and displacement height are directly incorporated in the log-law wind speed model. The power law model has a similar parameter (α) to reflect the effect of surface roughness on wind speed.

Raupach et al. (1991) defines rough flow as one whose shear stress is dominated by drag of roughness elements, compared to smooth flow whose shear stress is dominated by viscous drag. A rough ABL flow has the following layers that are depicted in Fig. 2.1.

1. Roughness sublayer in which velocity is influenced by the roughness elements. It is about 2 to 3 times the average height of roughness elements (H). With in this region a canopy layer may be present when the roughness elements are big; forest canopy, urban canopy etc. As discussed in previous sections, the velocity is zero at height of $d + z_0$.

2. Surface layer ($\sim 100m$) in which wind speed varies only with height, i.e. horizontally homogeneous, as governed by the log-law. The shear stress with in this region is constant.

2.4 ABL stratification and stability

Atmospheric stability refers to the resistance of the atmosphere to vertical motion. Temperature usually decreases at a roughly constant rate, commonly known as lapse rate, as one goes away from the surface of earth. This situation leads to an unstable atmospheric condition due to lighter air being at the bottom of cooler air. The lighter air moves upwards and denser air moves downwards due to the action of gravity. A circulation with in the ABL ensues as long as the surface keeps being heated. This is the case in the day time where the sun continuously heats the surface of the earth. During the nighttime,the earth looses its heat through radiation and the air at the bottom also cools down thereby a stable ABL is formed.

Stability plays an important role on wind flow over hills and other topographic features where the air is forced to move upwards. As the wind moves towards the crest on the upwind side, whether the flow remains attached or separated afterwards depends on stability. In a stable condition, the air remains attached to the surface because it has higher density than the surrounding air at the crest. For a neutral condition where the gradient of temperature is zero, the wind moves straight horizontally once it reaches the crest. For an unstable condition, the

wind moves straight upwards until it reaches heights where the surrounding air has the same temperature as the moving air parcels. Flow over a hill for the the three ABL stability cases, namely stable, unstable and neutral, are shown in Fig. 2.5.

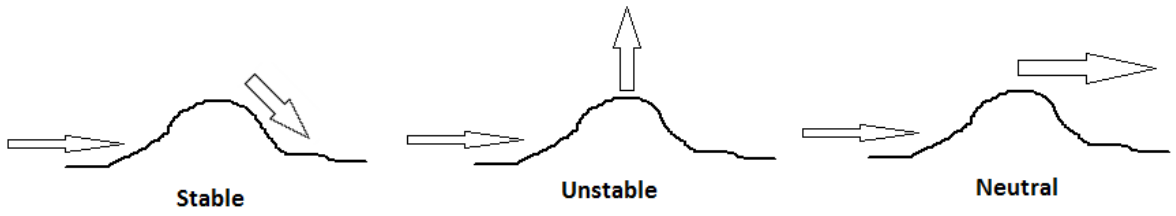


Figure 2.5: Effect of stability on wind flow over hill

A neutral ABL is assumed in this work because vertical velocity is negligible compared to horizontal velocity for moderate to high wind speeds. Therefore the governing wind flow equations used for this work ignore stability effects by making hydrostatic assumption Stangroom (2004), Xabier (2009). For situations where buoyant forces are important, it can be incorporated by introducing density variations due to temperature. An important simplification known as Boussinesq's approximation allows density variations to be ignored in all parts of the governing equations except where they are multiplied with gravity. Despite the serious simplification, the model is found to be very accurate for buoyancy driven flows in the ABL. As a result, buoyancy effects can be incorporated to the governing equations as a body force.

2.5 Coriolis force

Earth's rotation introduces fictitious forces on a moving mass of air that exist only in a rotating frame of reference. These forces are known as Coriolis and Centrifugal forces. The Centrifugal force acts outwards and is usually ignored in ABL simulations. The Coriolis force acts perpendicular to the velocity of wind and deflects it either to the right or left depending on location (latitude (ϕ)). The bulk of moving air is deflected to the right of its direction of motion on the northern hemisphere, to the left of its direction of motion on the southern hemisphere, and remains unaffected on the equator. The wind speed also affects the amount of deflection because the object has to be in motion to experience these fictitious forces. Similar to Reynolds number (Re), a parameter known as the Rosby number (Ro) is defined to quantify the magnitude of Coriolis force relative to inertial forces.

$$Ro = \frac{U}{Lf_c} \tag{2.4}$$

where f_c is the Coriolis parameter $f_c = 2\Omega sin\phi$, Ω is angular velocity of earth's rotation, ϕ latitude, and L is length scale. A small Rossby number of $Ro \sim 1$ indicates dominance of Coriolis force over inertial forces, hence its contribution can not be ignored. Before conducting micro-scale simulations on a hill or any other topographic feature, the Rossby number should be calculated to check whether the effect of Coriolis force can be ignored or not.

2.6 Statistics on wind turbulence

Most engineering flows around bluff bodies, such as buildings and other structures with sharp edges, are turbulent. A laminar flow becomes turbulent above a certain critical Reynolds number. Reynolds number is defined as the ratio of inertial forces to viscous forces.

$$Re = \frac{\rho UL}{\mu} \tag{2.5}$$

Turbulent flows are intrinsically unsteady even with constant imposed boundary conditions. A turbulent flow has the following characteristics:

1. *Irregularity*: Turbulent flow consists of eddies of different size (scale) ranging from the largest eddies whose size is dictated by the geometry (boundary layer thickness), down to the smallest energy dissipating eddies of Kolmogorov scale whose size is a function of viscosity of the fluid.

2. *Diffusivity*: As a flow becomes more turbulent, the boundary layer thickness also increases. Rapid mixing and increased mass, momentum, heat transfer occurs. A flow that is irregular but does not spread (not diffusive) is not turbulent.

3. *High Reynolds number*: Turbulent boundary layer flow occur at high Reynolds number in the order of $10^5 - 10^6$ or higher, where the inertial terms dominate viscous terms. At very low Reynolds numbers, viscous forces dominate inertial forces, and the resulting flow is known as creeping flow (Stokes flow).

4. *Three dimensional and anisotropic*: Turbulent flows consist of rotational vortices. Vortex stretching is at the core of the turbulent energy cascade in which energy is transfered from large to small eddies.

5. *Dissipative*: The smallest scales dissipate kinetic energy into internal energy. Unless the flow is maintained by incoming flow at boundaries turbulence will eventually die out.

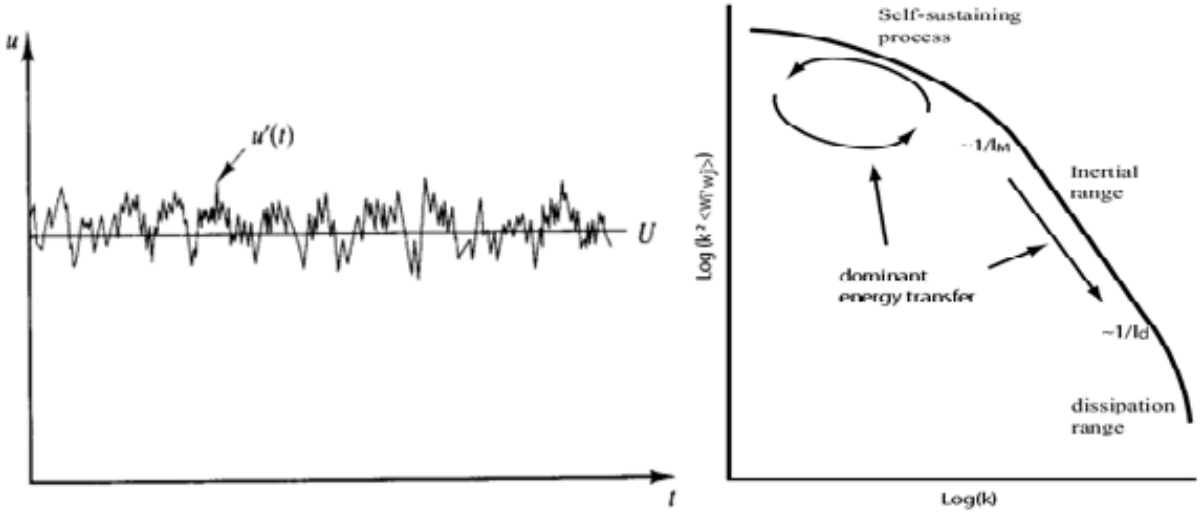


Figure 2.6: Time series of wind turbulence (left) and its spectrum (right) (Stull 1988)

Wind flow is turbulent and hence can only be described using statistical methodology. A time series of wind speed and its frequency content are shown in Fig. 2.6. The mean and root mean square (RMS) fluctuations of wind are usually used to describe wind intensity. Field observations using cup anemometers mounted on meteorological towers can provide point measurements at high frequency, that go a long way to characterize wind flow in an area. The fluctuating component of velocity is usually specified via turbulence intensity, in which the fluctuating component is normalized with the mean velocity.

$$I_u = \frac{\sigma_u}{U} \tag{2.6}$$

$$U' = U + \hat{g}\sigma_u \tag{2.7}$$

In building codes and standards, the peak wind load to be used for design purposes is calculated from a peak factor (g), mean wind speed and RMS fluctuation. Different averaging times can be used to calculate the mean wind speed. Peak 3-sec gust, 5-sec gust, 1min, 10 min, 1 hour averages are commonly used for different purposes in wind engineering. Wind speed averaged over 3 sec can be 1.53 times as large as the hourly average. Mean wind speed in the ABL tend to follow a standard Gaussian distribution while peak values follow extreme value distributions. Thus given records of wind speed from field observations, the probability of occurrence of any wind speed can be determined from the properties of the underlying distributions. Peak factors are used in building codes and standards to calculate peak wind loads.

2.6.1 Spectral content of wind

The largest scales of turbulence extract kinetic energy from the mean flow. Through the cascade process this energy is transferred to progressively smaller scales until it is totally dissipated to internal energy. All scales of turbulence dissipate some amount of energy through friction but it is assumed that about 90% of the energy is dissipated at the smallest Kolmogorov scales. For a given rate of energy dissipation ε per unit mass and kinematic viscosity ν, the velocity , time and length scales of the smallest eddies are as follows

$$U = (\nu\varepsilon)^{1/4},\ L = (\nu^3/\varepsilon)1/4,\ T = (\nu/\varepsilon)^{1/2} \tag{2.8}$$

The energy dissipation ε at the smallest scales can be estimated from that obtained from the largest scales of turbulence. Kolmogorov introduced the idea that the smallest scales of turbulence are the same for every turbulent flow while the largest scales are affected by geometry. In wave number space the energy of eddies from κ to $\kappa + \mathrm{d}\kappa$ can be expressed as $\mathrm{E}(\kappa)\ \mathrm{d}\kappa$

The total kinetic energy contained in all eddies can be found by integrating over the whole wave number space. The wave number of an eddy is proportional to the inverse of its radius ($\kappa \sim 1/r$)

$$K = \int_0^\infty E\,(\kappa)\,d\kappa \tag{2.9}$$

In the intermediate range (inertial range) the energy coming from the largest eddies is in equilibrium with the energy transferred to the smaller eddies. The inertial region exists for all fully turbulent flows (high Re). Kolmogorov, through dimensional analysis, came up with a relation for the energy contained by eddies in the inertial region. The energy in this range exhibits what is known as a '$-5/3$ decay':

$$E(\kappa) = constant * \varepsilon^{2/3} * \kappa^{-5/3} \tag{2.10}$$

The Kolmogorov law is often used in experiment, large eddy simulation and direct numerical simulation, to verify that a flow has become fully turbulent.

To understand the turbulence (fluctuations) in wind better, it is convenient to transform wind speed measurements in time domain to the frequency domain using Fourier decomposition. The resulting spectral density function provides a description of the frequency content of wind speed fluctuations. The most commonly used spectrum for longitudinal velocity component is the von Karman-Harris spectral density shown in Fig.2.7. The spectral density equations for

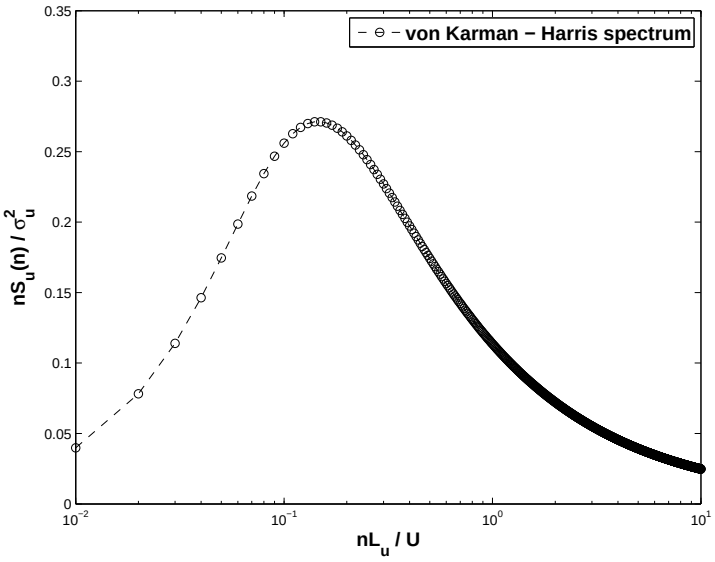


Figure 2.7: The von Karman-Harris spectra for longitudinal wind speed

the three velocity components are

$$\frac{nS_u(n)}{\sigma_u^2} = \frac{4n_u}{(1+70.8n_u^2)^{5/6}} \tag{2.11}$$

$$\frac{nS_v(n)}{\sigma_v^2} = \frac{4n_v(1+755.2n_v^2)}{(1+283.2n_v^2)^{11/6}} \tag{2.12}$$

$$\frac{nS_w(n)}{\sigma_w^2} = \frac{4n_w(1+755.2n_w^2)}{(1+283.2n_w^2)^{11/6}} \tag{2.13}$$

where $n_i = nL_i/U$

The time scale of turbulence is calculated by integrating auto-correlation of fluctuations at a fixed location. The integral time scale indicates the rate at which turbulence decays at a given location.

$$f(\tau) = \frac{\overline{u'(t)u'(t+\tau)}}{\overline{u'^2}} \tag{2.14}$$

$$I = \int_0^\infty f(\tau)d\tau \tag{2.15}$$

Another important property of wind turbulence in relation to wind loading on structures concerns its spatial variation. If the wind speeds at two different heights do not reach peak values simultaneously, it is possible to get a reduction in design wind loads. Cross-correlation of velocity components at different heights reveal approximate size of coherent structures or

eddies. The length scale of turbulence L_u is calculated by first calculating cross-correlation coefficients using equation 2.16 and then calculating the area under the curve by integration.

$$f(\zeta) = \frac{\overline{u'(x)u'(x+\zeta)}}{\overline{u'^2}} \tag{2.16}$$

$$L_u = \int_0^\infty f(\zeta)d\zeta \tag{2.17}$$

2.6.2 Mean wind speed and turbulence intensity models

2.6.2.1 The log law model

Wieringa (1993) proposed a semi-empirical equation for wind speed in neutral conditions and homogeneous roughness, commonly known as the log-law model. The model has some theoretical basis because it can be derived from mixing length theory making reasonable simplifying assumptions. It gives good estimates of wind speed within the inertial sublayer i.e. with in the lowest 100m of ABL. The aerodynamic roughness length (z_0) is used as a correction for the effect of roughness.

$$U(z) = \frac{u_*}{\kappa}[\ln(\frac{z-d}{z_0})] \tag{2.18}$$

The model can be modified to account for atmospheric stability by adding an extra term to it.

$$U(z) = \frac{u_*}{\kappa}[\ln(\frac{z-d}{z_0}) + \psi(z, z_0, L)] \tag{2.19}$$

where L is the Monin-Obukhov stability parameter. Similarly Deaves & Harris (1978) modified the model to account for Coriolis effect and make it applicable in the outer region as well.

$$U(z) = \frac{u_*}{\kappa}[\ln(\frac{z-d}{z_0}) + 5.75(\frac{z}{G}) - 1.88(\frac{z}{G})^2 - 1.33(\frac{z}{G})^3 + 0.25(\frac{z}{G})^4] \tag{2.20}$$

where $G = u_*/6f_c$ is the gradient height. The equation can be simplified to

$$U(z) = \frac{u_*}{\kappa}[\ln(\frac{z-d}{z_0}) + \frac{34.5 f_c z}{u*}] \tag{2.21}$$

The corresponding equation for turbulence intensity is

$$I_u(z) = \frac{k}{\ln(\frac{z-d}{z_0})} \tag{2.22}$$

where k is a coefficient dependent on roughness. It takes values of 1, 0.92, 0.88 for smooth, open and closed roughness respectively.

2.6.2.2 The power law model

An alternative wind speed model for the upper portion of the surface layer is the power law model. This simpler wind speed formula is commonly used in wind power calculations in which wind turbines reach heights of $\geq 50m$. Given wind speed measurements at a reference height (usually 10m) , the power law can be used to calculate wind speed at any other height using the following equation. Because the power law index α is usually chosen to fit the upper portion of the surface layer better, its prediction in the lowest portion could be relatively poor compared to that of the log-law. The power law index α is a function of roughness of terrain and turbulence. This method does not have a theoretical background unlike the log-law.

$$U(z) = U_{ref}(\frac{z-d}{z_{ref}})^{\alpha} \tag{2.23}$$

Similarly an inverse power law equation is used to estimate turbulence intensity profile in ASCE 7. The coefficients c and d are dependent on the roughness of the terrain and can be read from a table.

$$I_u(z) = c(\frac{z}{z_{ref}})^{-d} \tag{2.24}$$

2.7 Surface roughness models

Aerodynamic roughness can be estimated analytically, experimentally (full scale and wind tunnel) and numerically. A brief review of literature of roughness estimation is given in the following sections.

2.7.1 Empirical formulas

For well defined obstacle shapes, surface roughness parameters can be determined from density of obstacles, frontal (wall) area and/or planar (floor) area densities. Grimmond & Oke (1999) review various empirical models to determine aerodynamic characteristics of a site through analysis of its surface form (morphometry). Different roughness models have been proposed in literature to determine roughness parameters z_0 and d based on average area density of obstacles. This section discuses some of well known empirical formulas for estimating z_0.

A simple approximation for z_0 can be obtained from average height of obstacles: buildings, bridges, crops, forests etc. A value of c = 0.1 have been found to give good results in many situations however it is established that z_0 is generally not constant.

$$\frac{z_0}{H} = c \tag{2.25}$$

Theurer (1993) noted that z_0 and d are related to two secondary parameters of the obstacles.

$$\lambda_f = \frac{A_f}{A_d} \tag{2.26}$$

$$\lambda_p = \frac{A_p}{A_d} \tag{2.27}$$

Lettau (1969) provided an empirical formula to determine z_0 from frontal area density ratio of obstacles

$$\frac{z_0}{H} = 0.5\lambda_f \tag{2.28}$$

Peterson (1994) tested this model in wind tunnel and found good agreement when roughness elements do not interfere strongly with each other. For λ_f or λ_p greater than 20 to 30 % , the model fails to give good predictions due to interference between obstacles and development of displacement height d that is not accounted for in Lettau's expression. For example when the surface is completely covered with obstacles ($\lambda_p = 1$), Lettau's expression predicts a maximum roughness length. But in reality a new smooth surface displaced with a height of $d = H$ is formed and $z_0 \to 0$. Raupach (1992) have shown that peak value of roughness length occur in the range of $0.2 < \lambda_f < 0.3$, with $z_0 \to 0$ as $\lambda_f \to 1$.

Counihan (1971) measured z_0 in wind tunnel from velocity profiles over regular arrays of cubic elements and arrived at an expression that includes the effect of limited fetch length.

$$\frac{z_0}{H} = 8.2\frac{H}{X_f} + 1.08\lambda_p - 0.08 \tag{2.29}$$

X_f is the fetch length. In Counihan's experiment $A_f = 0.6\ A_p$ from whichλ_f can be obtained with which it is better correlated than it is with λ_p. For large fetch lengths, his equation reduces to

$$\frac{z_0}{H} = 1.08\lambda_p - 0.08 = 1.08\lambda_f - 0.08 \tag{2.30}$$

He claimed this expression is valid $0.06 < \lambda_f < 0.15$. Similar results as that of Lettau's are obtained for $\lambda_f < 0.06$. Both the above models fail to capture the non-linear reduction of $\frac{z_0}{H}$

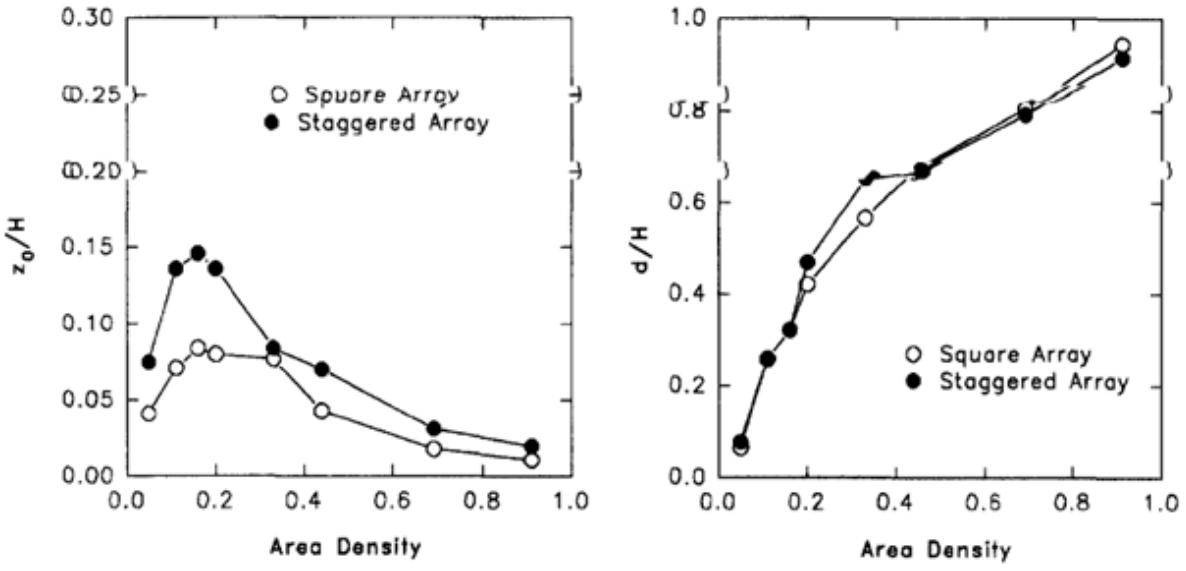


Figure 2.8: Roughness length and displacement height for square and staggered blocks (Peterson 1994)

as λ_f goes beyond 0.3. Hall et al. (1996) conducted wind-tunnel experiments over arrays of 0.1m cubes placed in a regular and staggered manner. They measured mean velocity profiles for $X_f \sim 22H$ and varying λ_p, and calculated z_0 and d using similar methods as that used by Peterson (1994). The variation of these parameters with λ_p is shown in Fig. 2.8. Theurer (1993) found expressions for z_0 and d from full scale measurements in cities and wind tunnel experiments.

$$\frac{d}{H} = 1.67\lambda_p \tag{2.31}$$

$$\frac{z_0}{H} = 1.6\lambda_f(1 - 1.67\lambda_p) \tag{2.32}$$

These equations are valid for up to $\lambda_p < 0.6$. Theurer limited λ_f to 0.25 to avoid skimming flow effects. For cubical obstacles Theurer's expression for z_0 becomes quadratic with a peak value of 0.24 as show in Fig. 2.9. All of the above methods fail to perform adequately in urban areas where λ_f exceeds 20%.

MacDonald et al. (1998) proposed an improved model which tackles the following limitations of the Lettau's model: low roughness element densities, lack of non-linear decrease of z_0 at high area density, drag differences caused by different obstacle shapes or layouts. The model is derived from Lettau's expression which they proved can be derived from fundamental principles assuming negligible wake interference between surface obstacles. Then the mean velocity approaching each obstacle can be obtained using log-law, which is the main reason

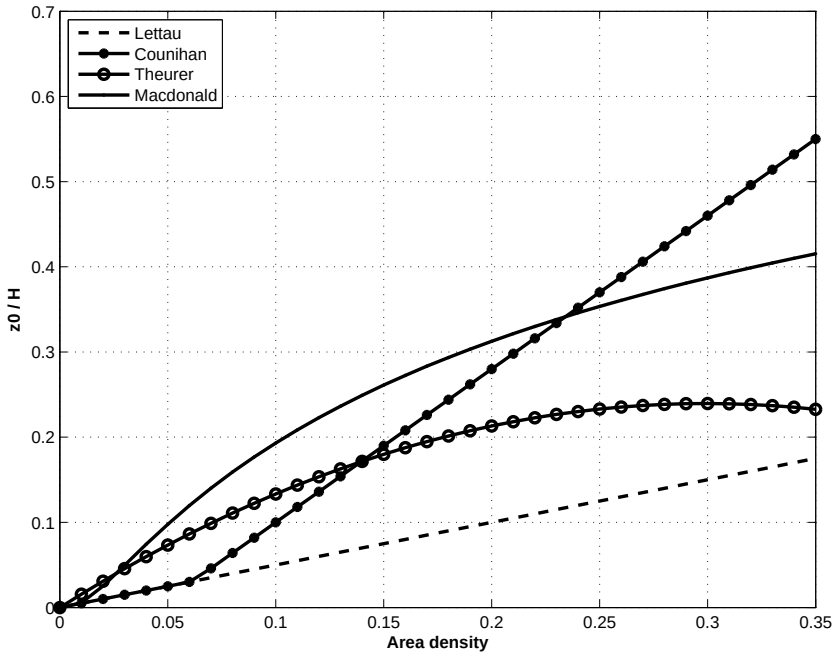


Figure 2.9: Comparison of different empirical models for roughness length

why the Lettau's expression fails to give good results for high area density ratios, in which wake interference effect is significant. The final expression after including the drag effect by different obstacle shapes is given below

$$\frac{z_0}{H} = \exp(-(0.5\frac{c_d}{k^2}\lambda_f)^{-0.5}) \tag{2.33}$$

For $C_D = 1.2$ (Engineering Science Data Unit (ESDU) recommendation for cube) and $\kappa = 0.4$, the above expression is simplified

$$\frac{z_0}{H} = \exp(-(0.52\lambda_f)^{-0.5}) \tag{2.34}$$

This expression shows better agreement with the Counihan's relation as shown in Fig. 2.9. However it still predicts monotonic increase of z_0 with λ_f. Hall et al. (1996) have shown that z_0 reaches a peak around $\lambda_f = 20\%$ from wind tunnel experiments. The derivation is redone with the logarithmic law which considers the effect of displacement height d.

$$\frac{z_0}{H} = (1 - \frac{d}{H})\exp(-(0.5\frac{c_d}{k^2}((1 - \frac{d}{H})\lambda_f)^{-0.5}) \tag{2.35}$$

Jackson (1981) has shown that the minimum displacement height is the height of an equivalent surface obtained by flattening out the obstacles in to a smooth one with a uniform cross-

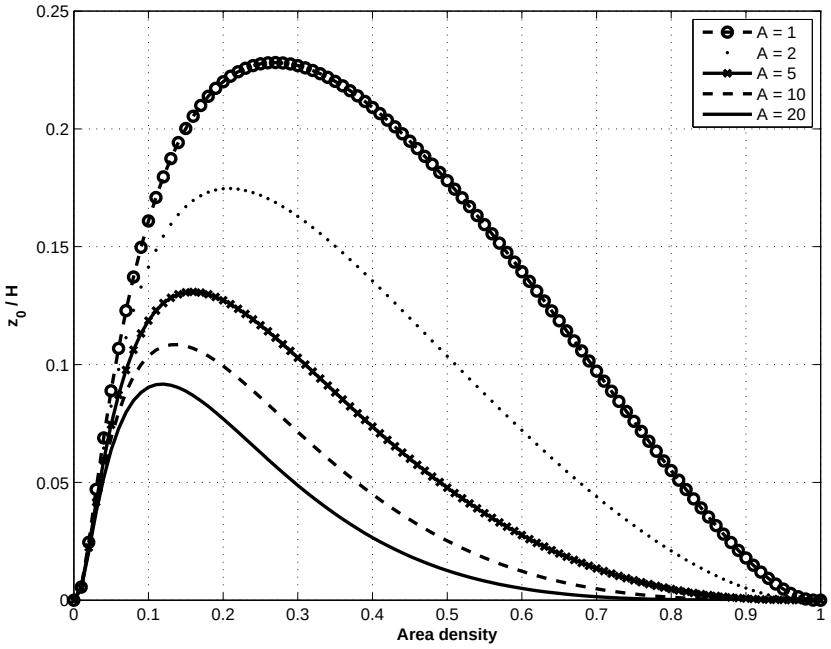


Figure 2.10: Displacement height for different convexity

sectional area, $\frac{d}{H} >= \lambda_p$. Approximate values for d can be obtained by the following equations that satisfy the requirement that $d = 0$ at $\lambda = 0$, and $d = H$ at λ=1.The parameter A controls the convexity of the curve as shown in Fig. 2.10.

$$\frac{d}{H} = 1 + A^{-\lambda}(\lambda - 1) \tag{2.36}$$

Using this model, MacDonald et al. (1998) found excellent agreement with Hall's wind tunnel data collected for staggered array obstacles. However the square array data is over-predicted due to enhanced sheltering effect. Various correction factors on the drag coefficient can be applied to account for different obstacle shapes and flow conditions: factor for velocity profile shape (k_s), incident turbulent intensity (k_i), turbulent length scales (k_l), incident wind angle (k_θ), and round corners (k_r).

$$C'_d = C_d\beta = C_d k_s k_i k_l k_\theta k_r \tag{2.37}$$

This modified drag coefficient can be used to determine z_0. Using $\beta = 0.55$, they were able to get good fit to the wind tunnel data for the staggered array.

The empirical roughness models we have discussed so far are summarized in Table 2.2. For a terrain with high density of obstacles of uniform height, 'skimming flow' occurs in which the roughness length continuously decreases to zero while the displacement height increases. The Lettau (1969) and Counihan (1971) models disregard this effect, hence their use is limited to

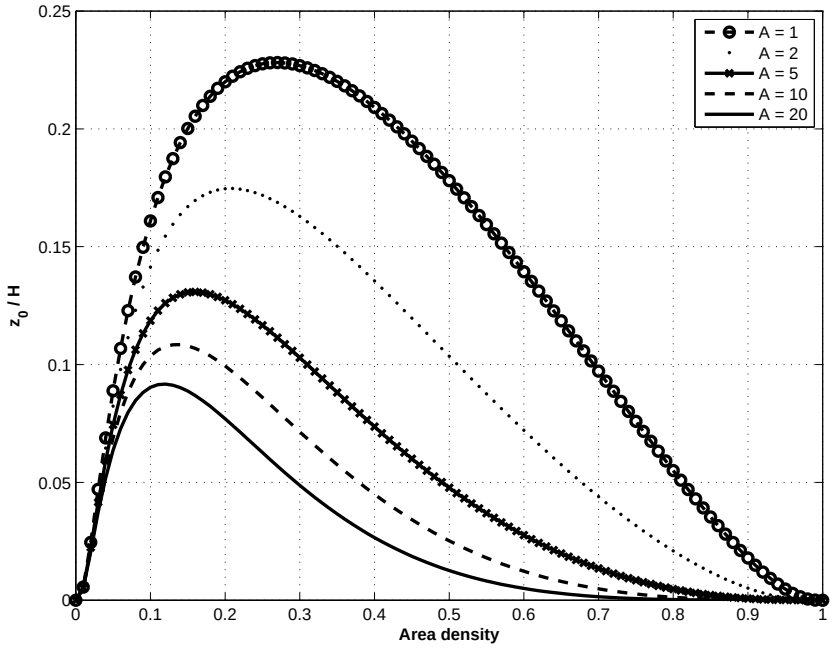


Figure 2.11: Roughness length for different convexity

Table 2.2: Summary of empirical formulas for roughness parameters

Model	$\frac{z_0}{H}$	$\frac{d}{H}$
Lettau (1969)	$0.5\lambda_f$	None
Counihan (1971)	$1.08\lambda_f - 0.08$	None
Theurer (1993)	$1.6\lambda_f(1 - 1.67\lambda_p)$	$1.6\lambda_p$
MacDonald et al. (1998)	$(1 - \frac{d}{H})\exp(-(0.5\frac{c_d}{k^2}((1 - \frac{d}{H})\lambda_f)^{-0.5})$	$1 + A^{-\lambda}(\lambda - 1)$

low density of obstacles not more than 30%. Peak values of z_0 occur approximately at area density ratio of 20%.

2.7.2 BLWT methodology

Peterson (1994) conducted wind tunnel tests on different models to evaluate surface roughness parameters from observed velocity profiles. Using the database of wind tunnel tests on three refinery models and two uniform roughness modes, the Lettau, Counihan and simplified Counihan models are evaluated. Among the seven different methods tested to determine roughness length z_0 from velocity profiles, only two were deemed adequate. Using statistical analysis to evaluate predicted roughness by the above mentioned methods, he found out that the Lettau model provides good estimates within a factor of 0.5-1.5 and 95% confidence.

First profiles of mean wind speed and turbulence intensity are obtained at several locations on the center line, left and right of center line. Then three different methods are used to

determine roughness length from velocity profiles.

1. The first method uses best fit to the logarithmic profile without displacement height.

$$ln(z) = \frac{k}{u*}u(z) + ln(z_0) \tag{2.38}$$

   a linear fit can be done to obtain z0 and u* simultaneously.

2. EPA (1987) on-site meteorological program provides the following relation for determination roughness length from turbulence intensity measurements.

$$z_0 = \frac{z}{exp(\frac{u}{u'})} \tag{2.39}$$

   for $20z_0 < z < 100z_0$.

3. Lo (1990)'s method: Technically this is the best method to estimate the actual roughness length. However it has deficiencies in that the estimation is based on measurements at only two heights, which can introduce large errors from small statistical errors. By applying the logarithmic profile with displacement height at two points Lo arrived at the following equations using normalized variables with respect to U_{n+1} and Z_{n+1}.

$$z_0 = \frac{(z_n - d)^\alpha}{(1-d)^\beta} \tag{2.40}$$

$$\begin{aligned}&(1 - A - d)ln(1 - d) - (1 - d)+\\ &(A - 1 + d)[\alpha ln(z_n - d) - \beta ln(1 - d) + \frac{(z_n - d)^\alpha}{(1-d)^\beta}] = 0\end{aligned} \tag{2.41}$$

$$\alpha = \frac{1}{1-u_n}, \beta = \frac{u_n}{1-u_n}, \text{ and } A = \int_0^1 U(z)dz \tag{2.42}$$

The solution proceeds by first solving iteratively for the displacement height from the second equation, and then the roughness length is determined from the first equation. Peterson also discussed other methods which can be used for roughness evaluation, among which two are found to be adequate. One of them is Lo's method with some modifications to avoid the problem of statistical errors.

Zaki et al. (2010) conducted wind tunnel measurements of roughness parameters of building arrays with random geometries. The randomness is featured in the form of vertical ran-

domness of height of blocks, and horizontal randomness of the rotation angle of each block. The study has found that the 'skimming flow' effect observed at high area density ration with uniform height elements is absent when the height of roughness elements show variations. This effect is attributed to the fact that flows around the taller blocks do not interfere with each other due to large separation on average, and also because randomly rotated blocks are less streamed than a regular arrays.

2.7.3 CFD methodology

Many researchers have used CFD to study aerodynamic drag using different arrangement of obstacles: shapes, size and layouts. The height and arrangement of roughness elements in wind tunnels is fixed in accordance with the required roughness that produces a desired velocity and turbulence intensity profiles at the turntable. When the terrain has multiple roughness patches or obstacle shapes are not clearly defined, analytical methods are difficult to use and in general do not give good results. In that case CFD can be used to conduct simulations, from the results of which can be estimated roughness parameters for different angle of wind attack. Explicit roughness modeling, as opposed to implicit modeling via wall functions, is used throughout this work (Chapter 4) to investigate the effect of homogeneous and inhomogeneous roughness on wind profiles.

Idealized models can be used to replace a complex built environments with simplified models that have equivalent aerodynamic roughness. Usually cubes with a regular or staggered arrangement and certain packing density are used in areas where resolving detailed flow characteristics is not required. This homogeneous model is exploited in CFD and Wind tunnel models where the area with in a certain radius of the study object is modeled as perfectly as possible, while the rest of the area is replaced with blocks that have similar aerodynamic properties. The next higher level modeling adds desired features of typical urban environment which is heterogeneous and morphologically consistent with the actual environment. This kind of models have been used in urban pollution and pedestrian comfort studies using CFD (e.g. (CEDVAL-LES 2011)).

Rasheed (2010) conducted CFD simulations to compare complex urban environment with a simplified model consisting of regular array of cubes. This is similar to the case in BLWT where the less important buildings away from the test building are modeled with regular array of roughness blocks. This transformation is necessary to take advantage of existing urban parameterization models that are developed for regular array of obstacles. The simplification also helps in increasing quality of meshes because the usually problematic tetrahedral meshes can

be avoided. He found that the stream wise velocity components for the two models show good agreement but turbulent kinetic energy profiles show significant differences.

To summarize the above reviews,

1. Most empirical models fail to correctly predict roughness parameters for high area density ratios. The Macdonald model seems to give the best results in this regard and can be modified to account for other factors through the inclusion of drag coefficient.

2. Wind tunnel testing and full scale testing have been used to study and validate roughness models. In most BLWT testing of a built environment, detailed wind flow characteristics are sought inside the built environment instead of just average roughness parameters.

3. Simplification of model by transformation to an equivalent regular array of blocks can be helpful if mean quantities are of the most interest.

2.8 Computational wind engineering

CFD can be used to analyze problems involving complex flow by solving governing partial differential equations of fluid flow. In wind engineering, experimental methods such as full scale and boundary wind tunnel tests have been the most successful ones so far. In other related fields such as aeronautical and mechanical engineering, CFD has been highly successful.

Computational wind engineering (CWE) is relatively young compared to other fields where CFD has made considerable progress. One cannot ignore the ever increasing computational power of computers which has motivated use of complex mathematical models that allow accurate flow predictions. For instance, Large Eddy Simulation (LES) has been mostly unused for high-Reynolds number flows due to its high computational demand. Nowadays it is becoming more and more common due to availability of high end computers and improvements in CFD modeling techniques. The commonly used Reynolds Averaged Navier-Stokes (RANS) models are known to give poor results in adverse pressure gradient flows, but LES gives very good results for separated flows that dominate bluff body aerodynamics.

CWE have also proven to be a reliable supplement to experimental methods. For instance it has been successfully used in the design and daily tests of the Wall of Wind (WoW) facility at Florida International University (Bitsuamlak 2006). Other applications where it has proven successful include: prediction of pedestrian level wind flows (Blocken & Carmeliet 2004), estimation of wind load on main wind force resisting system (Wright & Easom 2003), and

simulation of flows over complex topography for micro-sitting (e.g. using Wind Atlas Analysis Application Program (WAsP)). A review of the current state of art in CFD for wind engineering applications is summarized in (Bitsuamlak 2006, Dagnew & Bitsuamlak 2013).

2.9 Overview of CFD

The governing equations of fluid flow are the Navier-Stokes equations for momentum conservation and the continuity equation for mass conservation. These equations are coupled and non-linear that makes obtaining analytical solution very difficult, if not impossible, for most practical engineering problems. Also most flows of wind engineering interest are turbulent in nature, which is inherently chaotic, hence seeking for closed form solutions for these problems is rather meaningless. Statistically averaged solution of turbulence can be obtained by the use of so called 'turbulence models'. RANS have proven to be quite successful in this aspect for many engineering problems. An engineer is usually satisfied with the averaged property of the flow in many cases. For those particular cases where accurate results are required, better turbulence models can be used to capture instantaneous properties of turbulence at least for the largest eddies. A method exists that is able to capture all eddying motions down to the smallest scales (Kolmogorov scales), but it has a huge computational demand for many high-Re flows that are common in wind engineering. This aspect of compromise between accuracy and computational requirements manifests itself in other stages of CFD as well.

The mathematical model consists of governing partial differential equations of conservation laws and specified boundary conditions. Usually the equations are solved numerically by first dividing the whole domain into smaller regions and then forming linear equations that relate the quantities in each cell. There are mesh free methods that can solve the equations directly on the specified geometry without the need for discretization but these methods are still in development stage and not commonly used. The discretization method gives a set of algebraic equations at a number of discrete points in space and time. The Finite Volume Method (FVM), Finite Element Method (FEM) and the Finite Difference Method (FDM) are a few of these numerical methods. The most commonly used discretization method for fluid flows is FVM. Its popularity comes from the fact that conservation of mass and momentum is achieved in each cell during all stages of solving. The conservative nature of FVM is more appealing to engineers and gives a certain level of confidence on the results obtained from simulation carried out on coarse grids. Another advantage of the FVM over FDM is the relative ease in which it can be used on unstructured grids. Ideally structured meshes are preferable for fluid flow

simulations, but that is rarely the case in many practical problems.

2.9.1 Governing equations

The governing equations relevant to wind flow over complex terrain and built environment are described in this section. Many conservation laws applicable to other fields of engineering are excluded from the discussion. Thus an incompressible and dry atmosphere that is neutrally stratified is assumed.

2.9.1.1 Mass conservation law

The principle of mass conservation states that the fluid going out of a closed system is equal to the fluid getting into the system. This principle is expressed by the continuity equation shown below

$$\frac{\partial U}{\partial x} + \frac{\partial V}{\partial y} + \frac{\partial W}{\partial z} = 0 \tag{2.43}$$

The above expression assumes that the density of the fluid is constant. This assumption is acceptable for wind engineering applications. Boundary layer wind flow is usually incompressible even in the case of many extreme cases like hurricanes and cyclones. In aerospace applications, where the fluid is compressible, the form of the continuity equation which conserves ρU is used instead.

2.9.1.2 Momentum conservation law

This law states that if a closed system is not affected by external forces, its total momentum can not change. This law is basically an expression of Newton's second law to fluid motion. When Newton's law is combined with the assumption that the fluid stress is the sum of diffusive viscous stress which is proportional to velocity gradient, and a pressure term, it gives rise to a set of equations known as the Navier-Stokes equations. These equations are used to describe the physics of fluid flow such as weather, ocean currents, pipe flow, and air flow in and around a moving or stationary obstacle.

$$\rho \frac{DV}{Dt} = -\nabla p + \nabla . T + \vec{F} \tag{2.44}$$

$$\rho (\frac{\partial V}{\partial t} + V . \nabla V) = -\nabla p + \nabla . T + \vec{F} \tag{2.45}$$

The right hand side represents the forces applied on a closed domain of fluid. Surface forces applied due to the normal pressure gradient and viscous shear stresses are explicitly represented. The additional term on the right hand side $\vec{F}$ represents body forces per unit volume. For instance, it can be used to represent the effect of gravity and Coriolis force as shown in Eq.(2.46)-(2.48). For large scale movement of air in the atmosphere and oceanic movements, Coriolis force is an important contributor and is included in weather prediction systems.

$$\frac{dU}{dt} = \frac{-1}{\rho}\frac{\partial p}{\partial x} + fV + \tau_{vx} \tag{2.46}$$

$$\frac{dV}{dt} = \frac{-1}{\rho}\frac{\partial p}{\partial y} + fU + \tau_{vy} \tag{2.47}$$

$$\frac{dW}{dt} = \frac{-1}{\rho}\frac{\partial p}{\partial z} - g + \tau_{vw} \tag{2.48}$$

The hydrostatic assumption, where the atmosphere is assumed to be free from vertical acceleration, is commonly used in wind engineering. As discussed in section 2.4, temperature variations affect density of air and thus non-hydrostatic model of the atmosphere is appropriate for meso-scale or global simulations in meteorological applications.

$$0 = \frac{-1}{\rho}\frac{\partial p}{\partial z} - g \tag{2.49}$$

The left hand-side of Eq.(2.44) is momentum in material derivative form. This is an important concept in fluid dynamics where the Eulerian frame of reference is commonly used. Changes in fluid properties at a fixed location are observed instead of following trajectory of particles as in Lagrangian frame of reference. However once the solution is obtained through Eulerian frame of reference, trajectories (streamlines) can be computed for visualization purposes.

At very low Reynolds numbers, viscous forces dominate inertial forces, and the resulting flow is known as creeping flow (Stokes flow). The limiting case of Navier-Stokes equations where the inertial terms are dropped, for flow approaching $Re \to 0$, form the Stokes equations. The other limiting case where the viscous terms are dropped instead, for flow approaching $Re \to \infty$, result in the Euler equations. The flow in the upper portions of the atmosphere can be modeled using Euler equations, but inside the ABL viscous effects cannot be ignored, hence full Navier-Stokes equations should be solved there.

2.9.2 Turbulence models

A major problem with the otherwise very important set of equations is difficulty of getting a closed form solution except for very simple cases. This is due to the convective acceleration term $V.\nabla V$ in Eqs.(2.45) that introduces non-linearity and also couples all components of velocity. As a result the vast majority of flows can be studied only numerically after discretization of the domain and governing equations. Steady state solutions are usually sufficient for most wind engineering applications. However in cases where peak values of pressure and velocity are required, unsteady solutions can be carried out using unsteady RANS (uRANS) or LES turbulence models. The numerical solution of turbulent flows is extremely difficult. The most straight forward solution of turbulent flows involves using a very fine mesh and a laminar flow solver (i.e. no turbulence model) to resolve all flow scales, also known as Direct Numerical Simulation (DNS). The smallest mesh size required to resolve all flow scales is proportional to Kolmogorov length scale η.

$$\eta = \frac{(\nu^3)^{1/4}}{\epsilon} \tag{2.50}$$

The number of floating point operations required for a DNS solution is proportional to cube of Reynolds number (Re^3) , resulting in very high computational cost. The most powerful computers cannot solve practical flows with large Reynolds number which in the case of wind engineering is in the order of $Re \sim 10^6$. Hence this method is rarely used in practice and its use is limited to fundamental research in developing and verifying new turbulence models. RANS equations are used in practice where knowledge of average quantities is sufficient. LES is a computationally expensive alternative that is starting to gain ground for studying bluff body aerodynamics. DNS, which do not use turbulence models, have only been carried out for very simple cases. Moreover the engineer is mostly interested in the mean quantities of flow parameters and sometimes in peak values, and not in instantaneous values. Hence it is convenient to break down the turbulent quantities into mean and fluctuating components using Reynolds decomposition. Different turbulence models are discussed in the following sections.

2.9.2.1 Reynolds Averaged Navier Stokes

For a steady flow the mean can be calculated over an infinite Δt or a value large enough that exceeds the time scales of the largest eddies. For unsteady flow, the average of instantaneous values of the flow quantity over a large number of repeated identical experiments, so called ensemble average, is used.

$$U = \overline{U} + u \tag{2.51}$$

$$p = \overline{p} + p \tag{2.52}$$

Using the Reynolds decomposition for all variables and substituting into the instantaneous Navier Stokes equations, modified equations for the mean values can be obtained which has an additional term that accounts for the effect of turbulent fluctuations on the mean flow. This new equations are the RANS equations.

$$\nabla.\overline{U} = 0 \tag{2.53}$$

$$\frac{\partial \overline{U}}{\partial t} + \nabla.(\overline{UU}) = g - \frac{1}{\rho}\nabla\overline{p} + \nabla.\nu\nabla\overline{U} + \overline{U'U'} \tag{2.54}$$

The new term that appears on the right hand side is the Reynolds stress tensor R = $\overline{U'U'}$ which depends on the velocity fluctuations induced by turbulence. The Reynolds stress tensor is a symmetric tensor with six components which are unknown. But we have only four equations (three momentum equations for each direction and continuity equation). This problem is known as the turbulence closure problem. To close the system of equations turbulence models are used to model the Reynolds stress tensor in terms of known quantities. RANS turbulence models can be categorized as:

- Linear eddy viscosity models
- Non-linear eddy viscosity models
- Reynolds stress models

2.9.2.2 Linear eddy viscosity models

The Reynolds stress tensor R is computed using the Boussinesq assumption which prescribes linear relation between R and viscous stresses.

$$-\rho\overline{U'U'} = 2\mu_t S - \frac{2}{3}\rho k\mathbf{I} \tag{2.55}$$

where S is the mean strain rate and k is the mean turbulent kinetic energy.

$$\mathbf{k} = \frac{\overline{U'.U'}}{2} \tag{2.56}$$

$$\mathbf{S}=\frac{\left(\nabla U + (\nabla U)^T\right)}{2} - \frac{\nabla.U}{3} \tag{2.57}$$

The new viscosity term ν_t is called turbulence viscosity (eddy viscosity). It can be solved in many ways by solving additional transport equations. Earlier models approximated turbulence viscosity directly from the flow variables without solving additional equations.

1. *Zero equation models:*

The eddy viscosity is computed using an algebraic equation to close the system of equations. No additional transport equations are solved hence the name zero-equation. One such model for boundary layer type flows is the mixing length model developed by Prandtl. Using dimensional analysis

$$\nu_t(m^2/s) \sim U(m/s) * l(m) \tag{2.58}$$

where U and l are characteristics of the largest turbulence scales. In the mixing length model velocity gradient is used as the velocity scale.

$$\nu_t = {l_{mix}}^2 |\frac{\partial U}{\partial y}| \tag{2.59}$$

A problem with this model is that the mixing length is unknown; hence the model is hardly used in practice nowadays. There are other algebraic models commonly used in aerospace engineering to get quick results when robustness in design iterations is more important than capturing all details of turbulence. The Baldwin-Lomax and Cebeci-Smith are such models which prescribe the eddy viscosity in terms of local boundary layer velocity profile.

2. *One equation models:*

All zero equation models can not properly account for history effects of turbulence due to convection and diffusion of turbulent kinetic energy. The one equation models solve a transport equation, which is usually turbulent kinetic energy k. Prandtl's one equation model is shown below.

$$\mu_t = \rho k^{1/2} l \tag{2.60}$$

$$\frac{\partial k}{\partial t} + \nabla.(\rho \mathbf{V} k) = \nabla.\left(\left[\mu_{lam} + \frac{\mu_t}{\sigma_k}\right]\nabla k\right) + \mu_t G - \rho C_D \frac{k^{3/2}}{l} \tag{2.61}$$

where G is the turbulence generation rate G = 2**S.S** and $C_D = 0.08$, $\sigma_k = 1$.

The last two terms on the right hand side account for production and destruction of turbulent kinetic energy respectively. The length scale is, for example, taken to be proportional to the

boundary layer thickness. However, the main disadvantage of one equation models is that the length scale is not universal for all type of flows.

3. *Two equation models:*
Two equation models are most commonly used RANS models in industry. Usually one of the equations solved is turbulent kinetic energy k as was the case for one equation models. The other equation is solved to determine the length scale of turbulence, which was a major problem of the one equation models. The turbulent dissipation ε or specific dissipation ω are common choices for the second transport equation. The standard k-epsilon model equation for high Reynolds number flow are shown below

$$\mu_t = \rho C_\mu \frac{k^2}{\epsilon} \tag{2.62}$$

$$\frac{\partial k}{\partial t} + \nabla.(\rho \mathbf{V} k) = \nabla.\left(\left[\mu_{lam} + \frac{\mu_t}{\sigma_k}\right]\nabla k\right) + \mu_t G - \rho\epsilon \tag{2.63}$$

$$\frac{\partial \epsilon}{\partial t} + \nabla.(\rho \mathbf{V} \epsilon) = \nabla.\left(\left[\mu_{lam} + \frac{\mu_t}{\sigma_\epsilon}\right]\nabla \epsilon\right) + C_{1\epsilon}\mu_t G\frac{\epsilon}{k} - C_{2\epsilon}\rho\frac{\epsilon^2}{k} \tag{2.64}$$

$$C_{1\epsilon} = 1.44, \quad C_{2\epsilon} = 1.92, \quad C_\mu = 0.09, \quad \sigma_k = 1.0, \quad \sigma_\epsilon = 1.0 \tag{2.65}$$

Wilcox proposed a series of k-omega models that solve specific dissipation (omega) equation to determine length scales. Omega is proportional to the ratio of ε and k i.e. $\omega\ \alpha\ \varepsilon$ / k. This helps in regions of low turbulence where both ε and k goes to zero. In the standard k-epsilon equation the non-linear term (ε^2/k) causes stability problems as k approaches zero.

$$\mu_t = \frac{k}{\omega} \tag{2.66}$$

$$\frac{\partial k}{\partial t} + \nabla.(\rho \mathbf{V} k) = \nabla.\left(\left[\mu_{lam} + \sigma^* \mu_t\right]\nabla k\right) + \mu_t G - \rho\beta^*\omega k \tag{2.67}$$

$$\frac{\partial \omega}{\partial t} + \nabla.(\rho \mathbf{V} \omega) = \nabla.\left(\left[\mu_{lam} + \sigma \mu_t\right]\partial \omega\right) + \alpha\mu_t G\frac{\omega}{k} - \rho\beta\omega^2 \tag{2.68}$$

$$\beta = \frac{3}{40}, \quad \beta^* = \frac{9}{100}, \quad \alpha = \frac{5}{9}, \quad \sigma = \frac{1}{2}, \quad \sigma^* = \frac{1}{2}, \quad \varepsilon = \beta^*\omega k \tag{2.69}$$

The Boussinesq approximation has an inherent weakness manifested in commonly used two

equation models. In strongly accelerated / decelerated flows and flows with strong curvature the assumption is not valid. Hence the models are incapable of correctly predicting strongly rotating flows. Usually an overproduction of turbulent kinetic energy is observed in those regions. To solve this problem, different modifications to two equation models have been proposed. Normally turbulent kinetic production P in the k equation is specified as

$$P = 2\mu_t(S.S) \tag{2.70}$$

Kato & Launder (1993) proposed an ad-hoc modification to this term by introducing vorticity Ω into the equation. The modified P equation is

$$P = 2\mu_t\left(\sqrt{(S.S) * (\Omega.\Omega)}\right) \tag{2.71}$$

This modification can be applied to all two equation models to alleviate the problem of over production of turbulent kinetic energy in strongly rotating zones. Other modifications on the standard k-epsilon equation to include swirling component of the flow resulted in two modified turbulence models Realizable k-epsilon and RNG (Renormalization Group) k-epsilon. The effect of spin on turbulence, which was missing from the standard k-epsilon model, is incorporated in the equations.

2.9.2.3 Non-linear eddy viscosity models

The linear Boussinesq approximation is dropped in favor of a non-linear relation that includes vorticity to improve the poor performance observed in the two equation model near flow stagnation zones. Hence the Reynolds stress is rewritten as

$$-\rho\overline{U'U'} = 2\mu_t\, f(S, \Omega, \dots) \tag{2.72}$$

2.9.2.4 Reynolds stress models (RSM)

This is the most elaborate RANS turbulence model which directly calculates Reynolds stresses without the need of modeling. Transport equation for the Reynolds stress (six) is solved together with an equation for dissipation rate (one). The seven additional transport equations make the method very expensive compared to the two equation models, however the benefit obtained from correct solution in rotating flows may balance the cost in some cases. A detailed discussion of this method can be found in Launder et al. (1975).

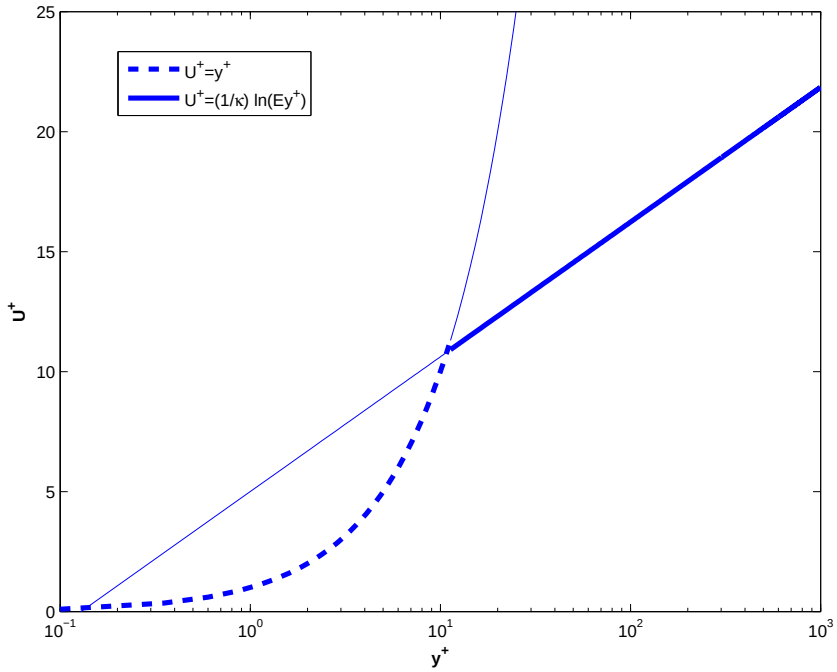


Figure 2.12: The law of the wall expressed with wall coordinates y^+ and U^+

2.9.2.5 Modeling flow near wall

Close to walls where turbulence is generated, a fine mesh should be used to resolve the details of turbulent motion due the prevailing sharp gradients. This imposes heavy computational requirements even when using two equation RANS models. The behavior of fully developed turbulent boundary layer flow near wall regions is well established from experiments. Hence, a significant saving on computation can be obtained by developing wall models (wall functions) to predict the near wall behavior for high Reynolds number flows. The law of the wall was first published by Theodore von Karman. The flow adjacent to the wall is dominated by viscous stresses (linear sub-layer) while the one on the top is dominated by turbulent Reynolds stresses (log law layer). In the middle is a buffer layer where both stresses are equally important. For a smooth no-slip wall the logarithmic law of the wall is shown in Fig.2.12. The equations describing the flow near wall are prescribed using dimensionless quantities $u+$ and $y+$. The viscous and log-law layer are separated at y^+ value of about 11.

$$y^+ = \frac{u_* y}{\nu} \tag{2.73}$$

$$u^+ = \frac{U}{u_*} \tag{2.74}$$

$$u^+ = \begin{cases} y^+, & \text{viscous layer} \\ \frac{1}{\kappa}\ln(Ey^+)\,, & \text{log-law layer} \end{cases} \tag{2.75}$$

where y the perpendicular distance between the nearest wall and center of nearest cell. For smooth walls values of E = 9.8 and $\kappa = 0.41$ are commonly used. In high Reynolds number RANS models, the production and rate of dissipation of kinetic energy are calculated and fixed at the nearest cell to the wall. During the solution phase, the turbulent kinetic energy k from the previous iteration is used to calculate a wall function friction velocity $u*$ as follows. This is a different friction velocity than atmospheric boundary layer friction velocity used in the log-law.

$$u* = C_\mu^{1/4} k^{1/2} \tag{2.76}$$

Then the turbulent dissipation (ε) or specific dissipation (ω) rates can be calculated as follows to be used for k-epsilon and k-omega models respectively.

$$\varepsilon = \frac{u^{*3}}{\kappa y} \tag{2.77}$$

$$\omega = \frac{u^*}{\kappa y \sqrt{C_\mu}} \tag{2.78}$$

Then a Dirichlet boundary condition is applied at the nearest cell to the wall with the above values. The turbulent energy production at the walls is calculated using an eddy viscosity coefficient

$$\mu_t = u^* \frac{y^+}{u^+} \tag{2.79}$$

The log-law can be modified for rough wall surfaces by adding an extra term on the right hand side ΔB which is a function of sand grain roughness K_s. Nikurdase (1933) conducted extensive experiments on rough wall surfaces and found out that the log-law has still the same slope when plotted on semi-log scale i.e $1/\kappa$. The plot is just shifted by an amount ΔB which is 0 for smooth walls.

$$u^+ = \frac{1}{\kappa}\ln\left(Ey^+\right) \; - \; \Delta B \tag{2.80}$$

For fully rough flow with ($K_s^+ \geq 90$), the following approximation for ΔB is suggested by Cebeci and Bradshaw. K_s^+ is dimensionless sand grain roughness ($K_s u* /\nu$).

$$\Delta B = \frac{1}{\kappa}\ln(1 + C_{ks}\, K_s^+) \tag{2.81}$$

2.9.2.6 Large eddy simulations

In LES, the small universal eddies are filtered out and modeled using sub-grid scale models (SGS models). This filtering process can be thought of as separating the velocity field into a resolved and sub-grid component. The filtering operation is convolution of velocity with a filtering kernel G.

$$\overline{u_i}(\vec{x}) = \int G(\vec{x} - \vec{\epsilon}) u(\vec{\epsilon}) d\vec{\epsilon} \tag{2.82}$$

$$u_i = \overline{u}_i + u_i' \tag{2.83}$$

The simplest kernel is a box filter which results in the grid itself acting as a spatial filter. That means the values of velocity on the grid are the filtered values. This implicit filtering is easy to program and is commonly used. For dynamic SGS models explicit filtering with a different filtering kernel such as Gaussian filter is used. Applying the filtering on the Navier-Stokes equation results in filtered equations with an addition stress term (SGS stress).

$$\frac{\partial \overline{u}_i}{\partial t} + \overline{u}_j \frac{\partial \overline{u}_i}{\partial x_j} = -\frac{1}{\rho}\frac{\partial \overline{p}}{\partial x_i} + \frac{\partial}{\partial x_j}([\nu + \nu_t]\frac{\partial \overline{u}_i}{\partial x_j}) \tag{2.84}$$

The sub-grid scale turbulence models usually employ the Boussinesq hypothesis to calculate the deviatoric part of the SGS stress.

$$\tau_{ij} - \frac{1}{3}\tau_{kk}\delta_{ij} = -2\mu_t \overline{S}_{ij} \tag{2.85}$$

The sub-grid scale turbulent viscosity for the Smagorinsky - Lilly model

$$\mu_{sgs} = \rho(C_s \triangle)^2 \left|\overline{S}\right| \tag{2.86}$$

where the filter width $\Delta = (Volume)^{1/3}$ and the constant Cs = 0.1 – 0.2

2.9.3 Finite volume discretization

The three most common ways of discretizing the governing equations are the Finite Difference Method, Finite Volume Method and Finite Element Method. In FDM the partial derivatives are replaced with terms usually taken from truncated Taylor series. Its disadvantage is difficulty of applying the method to an irregular grid. The FVM and FEM work with integral forms of the governing equations, and thus can be easily extended to support irregular grids. Moreover the

FVM balances fluxes across faces of cells and hence governing conservation equations (mass, momentum and energy) are always satisfied locally and globally at any stage of the solution. This property makes FVM preferable for engineers which are used to conservation laws. In FEM Galerkin's method of weighted residuals, where the weights have the same form as the shape function, is used. FEM is more mathematically involved than FVM and also the terms of the algebraic equations does not have any physical significance unlike the FVM approach. All approaches can be viewed as variations of method of weighted residuals: FDM as collocation method using Dirac-Delta weights where $w_i = 1$ at nodes and zero everywhere else, FVM as a subdomain method where $w = 1$ in a subdomain and the integral of the weighted residual is forced to zero for each subdomain, and FEM as the Galerkin variation of weighted residuals. From here only the FVM approach is discussed.

In FVM, the differential form of a general transport equation is converted to integral form by integrating over a closed control volume

$$\frac{\partial \rho\phi}{\partial t} + \nabla.(\rho\phi\mathbf{u}) = \nabla.(\Gamma\nabla\phi) + S_\phi \tag{2.87}$$

$$\oint \frac{\partial \rho\phi}{\partial t} + \oint \nabla.(\rho\phi\mathbf{u}) = \oint \nabla.(\Gamma\nabla\phi) + \oint S_\phi \tag{2.88}$$

The volume integrals for the convection and diffusion terms can be re-written into a surface integral form by using Gauss's theorem

$$\oint \nabla.\mathbf{a}dV = \oint \mathbf{n}.\mathbf{a}d\mathbf{A} \tag{2.89}$$

$$\frac{\partial}{\partial t}\left(\oint \rho\phi dV\right) + \int_A \mathrm{n}.(\rho\phi\mathbf{u})\,\mathrm{dA} = \int_A \mathbf{n}.(\Gamma\nabla\phi)\,\mathrm{d}\mathbf{A} + \oint S_\phi dV \tag{2.90}$$

where $\mathbf{n}$ the surface normal vector. For transient simulation, integration in time is applied on top. Because time is a one way coordinate, solution marches forward from initial prescribed conditions at t_0. Temporal discretization is division of the total time in to small time steps of Δt.

The accuracy of the discretization method depends on the assumed variation of ϕ in space and time around the center of the control volume. If a stepwise profile, where a constant value of ϕ is assumed throughout the control volume, a first order accurate discretization scheme is arrived. To get second and higher order accuracy, a profile assumption , such as piece wise linear, that couples the values of ϕ in adjacent control volumes is required. Not all terms of

the equation above have to be discretized the same way. For instance, the convection term is usually discretized with first order accurate stepwise profile, while the diffusion term uses second order accurate piece wise linear profile. Which ever method is used, the model should posses certain properties to be useful.

1. Consistency: Truncation error should vanish as $\Delta t \to 0$ and $\Delta x \to 0$
2. Stability: Errors in the course of the simulation should not magnify, and cause divergence.
3. Conservation: Conservation of physical quantities on both local and global scales.
4. Boundedness: Solutions must lie within proper bounds dictated by the boundary values.
5. Realizability: The solutions obtained should be realistic

2.9.3.1 Convection discretization

As mentioned before, Gauss's theorem can be used to convert the volume integral into a surface integral with interpolated quantities at the surface (Jasak (1996))

$$\begin{aligned} \oint \nabla.(\rho U\phi)\, dV &= \sum S.(\rho U\phi)_f \\ &= \sum S.(\rho U)_f \phi_f \\ &= \sum F_f \phi_f \end{aligned} \tag{2.91}$$

where F represent the mass flux though the face, and S the surface normal vector with magnitude equal to the area of the face.

$$F = S.(\rho U)_f \tag{2.92}$$

Three basic discretization schemes are considered below. ϕ_f represents the value at the shared face between two control volumes P and N.

1. Central difference scheme (CDS) : Second order accurate but unbounded

$$\phi_f = f_x \phi_P + (1 - f_x)\phi_N \tag{2.93}$$

2. Upwind scheme (UDS): Bounded but first order accurate

$$\phi_f = \begin{cases} \phi_P, & F \geq 0 \\ \phi_N, & F < 0 \end{cases} \tag{2.94}$$

3. Blended scheme (BS): A compromise between accuracy and boundedness by blending the above two methods

$$\phi_f = \gamma\phi_P + (1-\gamma)\phi_N \tag{2.95}$$

4. Hybrid scheme: Picks either of CDS or UDS depending on how strong convection is compared to diffusion. The Peclet number is defined as $P_e = F/D$ where D is diffusion conductance $D = \gamma/\delta x$. The method takes advantage of the good properties of CDS and UDS; it is bounded and has better accuracy than UDS, however its accuracy is still first order.

$$\phi_f = \begin{cases} CDS, & |P_e| \leq 2 \\ UDS, & |P_e| > 2 \end{cases} \tag{2.96}$$

5. Total Variation Diminishing (TVD) schemes: The schemes discussed so far are either unbounded (e.g. CDS) or of first order accuracy (e.g. Hybrid). There is a need for high resolution(HR) schemes that are bounded and non-oscillatory. These schemes are especially important for shock predictions, in which all the previous schemes can give false predictions when used on coarse grids. A class of HR schemes known as TVD schemes start from implicit UDS scheme for the sake of boundedness, and then add explicit source term (difference of higher order scheme (HOS) and UDS scheme) to improve accuracy in each iteration via what is known as a 'deferred correction' approach. In this sense, all the previous convection discretization schemes are implicit because the final values are obtained after exactly one iteration without correction. Deferred correction can also be used in other situations where it is difficult or impossible to handle terms implicitly. For example non-orthogonality of mesh can be handled by starting from the assumption of orthogonality and then adding explicit corrective terms in every iteration. Before the time step is updated to the next one, the deferred corrections should be iterated until convergence (or acceptable level of accuracy) with the only changes coming from deferred corrections.

$$\phi_f = UDS \tag{2.97}$$

$$S_u = \sum [(HOS - UDS) * F] \tag{2.98}$$

If the HOS is CDS then the TVD scheme becomes a bounded central difference scheme. Other TVD schemes are Linear upwind scheme (LUD), Monotone Upstream-centered

Schemes for Conservation Laws (MUSCL), VanLeer etc.

2.9.3.2 Diffusion discretization

The diffusion term is discretized similarly with the central difference scheme.

$$\begin{aligned} \oint \nabla.\left(\rho\Gamma_\phi \nabla\phi\right) dV &= \sum S.(\rho\Gamma_\phi \nabla\phi)_f \\ &= \sum (\rho\Gamma_\phi)_f \, S.(\rho\Gamma_\phi)_f \end{aligned} \tag{2.99}$$

where the surface gradient is approximated by

$$S.(\nabla\phi)_{\mathbf{f}} = |\mathbf{S}| \frac{\phi_N - \phi_P}{|\mathbf{d}|} \tag{2.100}$$

For non-orthogonal meshes such as tetrahedrons and pyramids, a deferred correction approach is used as discussed in the previous section. S is split into two components one always parallel to the line connecting the centroids (Δ) and another chosen in different ways ($\mathbf{k}$).

$$\mathbf{S}.(\nabla\phi)_{\mathbf{f}} = \mathbf{\Delta}.(\nabla\phi)_{\mathbf{f}} + \mathbf{k}.(\nabla\phi)_{\mathbf{f}} \tag{2.101}$$

$$\mathbf{S} = \mathbf{\Delta} + \mathbf{k} \tag{2.102}$$

This splitting can be done in three different ways

1. Minimum correction: makes the correction as small as possible by making $\mathbf{\Delta}$ and $\mathbf{k}$ orthogonal to each other, i.e $\mathbf{\Delta}.\mathbf{k} = 0$.

2. Orthogonal correction: This approach keeps the contribution of ϕ_P and ϕ_N same as that on an orthogonal mesh despite the amount of non-orthogonality, i.e. $|\mathbf{\Delta}| = |\mathbf{S}|$.

3. Over-relaxed: This approach increases the contribution of ϕ_P and ϕ_N with non-orthogonality in such a way that $\mathbf{S}.\mathbf{\Delta} = 0$.

The non-orthogonal correction may not preserve boundedness of a scheme, hence the correction should be limited or completely abandoned if preservation of boundedness is more important (Jasak 1996).

2.9.3.3 Source term discretization

Source term may have non-linear terms which need transformation to linear forms (linearization). For example, the $k - \epsilon$ turbulence mode has a highly non-linear dissipation term in the

transport equation for dissipation ϵ. The linearization can be carried out in many ways that give different values of S_u and S_p. Convergence rate and stability of solution depends on the selected linearization scheme.

$$S_\phi(\phi) = S_u + S_p\phi \tag{2.103}$$

$$\oint S_\phi(\phi)\ dV = S_u V_P + S_p V_P \phi_P \tag{2.104}$$

2.9.3.4 Temporal discretization

Temporal discretization of spatial derivatives can be carried out in three ways.

1. Implicit: Current values of ϕ are assumed to persist through out the time step. The advantage of this method is that it is bounded and unconditionally stable, however since the unknown current ϕ values are coupled with each other, a set of simultaneous equations need to be solved at each time step.

2. Explicit: Old values of ϕ are assumed to persist through the time step. The new ϕ values depend only on the old values, hence explicit updates can be carried out. However it comes with a price of using small time step Δt to insure stability of solution. The Courant number $Co = U.\Delta t/\Delta x$ should be less than one for stability.

3. Crank Nicholson: A linear variation of ϕ between old and new values is assumed with in the time step to get a second order accurate scheme. It is unconditionally stable but unbounded. Patankar (1980) notes that unconditionally stability refers to the fact that the oscillations will eventually die out, not to the absence of them

The above discretization concern the time at which the values of spatial derivatives are to be evaluated. The first time derivative itself use implicit first order Euler scheme, that becomes second-order accurate if the Crank Nicholson scheme is used for the spatial derivatives.

$$\frac{\partial}{\partial t}\oint \rho\phi dV = \frac{(\rho\phi V)_i - (\rho\phi V)_{i-1}}{\Delta t} \tag{2.105}$$

Higher order Runge-Kutta or other implicit/explicit time integration schemes can be used for more accuracy. First and second time derivatives can also be discretized with backward differencing scheme that require values of ϕ in the previous two time steps. This results in second order scheme for the first derivative and only first order for the second derivative.

$$\frac{\partial}{\partial t}\oint \rho\phi dV = \frac{3(\rho\phi V)_i - 4(\rho\phi V)_{i-1} + (\rho\phi V)_{i-2}}{2\Delta t} \tag{2.106}$$

$$\frac{\partial}{\partial t}\oint \rho \frac{\partial \phi}{\partial t} dV = \frac{(\rho\phi V)_i - 2(\rho\phi V)_{i-1} + (\rho\phi V)_{i-2}}{\Delta t^2} \tag{2.107}$$

2.9.4 Boundary conditions

Boundary conditions are required to obtain a well-posed problem and complete the solution. The basic boundary conditions are the Dirichlet type where the value of ϕ is prescribed and the Neumann type where the value of the gradient $\nabla\phi$ is prescribed. Other boundary conditions can be derived from these basic boundary conditions. Some of those implemented in the software are discussed below

1. Dirichlet: Fixed values applied on faces of a boundary. This include different profiles such as uniform, power-law, log-law, parabolic etc. In addition, these boundary conditions have turbulence intensity parameters to change their values with time. The fluctuations can take on a prescribed turbulence intensity profile or simple random fluctuations about a mean value of ϕ. Correlated fluctuations such as one that satisfies the von Karman spectrum can also be imposed but are not implemented in current software.

2. Neumann: The value of ϕ at the boundary face is calculated from its value at the adjacent cell center and the specified gradient.

$$\phi_b = \phi_P + \mathbf{d}_n.(\nabla\phi)_b \tag{2.108}$$

   The value at the boundary ϕ_b can be eliminated from the set of equations by substituting the above equation for a given value of gradient.

3. Symmetry: When the domain is symmetrical in geometry and boundary conditions, the flow is also symmetrical with no flux through the symmetry plane. Hence this boundary condition sets the normal component $\nabla\phi_n$ to 0. In this case ϕ_b can not be eliminated from the set of equation, hence a deferred correction approach is used instead such that the boundary condition is obeyed gradually with iterations.

4. Cyclic: This boundary condition is applied when the domain wraps around and the value at one end is the same as the value at the other. This is implemented the same as the Neumann methods above where half of the cells in the boundary are used as inputs to the other half for all flow quantities.

5. Ghost: When a decomposed domain is solved in parallel, the values at the boundary are exchanged through ghost cells. This is similar to cyclic boundary condition but is a two way update.

6. Derived: Other derived boundary conditions such as Robin, turbulence generating wall boundary conditions etc.

2.9.5 Calculation of flow field

Once the set of algebraic equations relating values of ϕ at control volume centers are obtained, we set out to solve the equations but there are some more difficulties to overcome. First the convective acceleration term in the momentum equations, i.e. 'velocity being transported by itself', results in non-linear terms with squared velocities. However this non-linearity is not a problem for iterative solvers that work on linearized equations formed from guessed values U. In other words one of the U's is treated explicitly and is no more different than other coefficients. What is problematic is the apparent absence of an equation for obtaining pressure, that appears only in the momentum equation. If the pressure was specified directly, the momentum equation could be solved with no difficulty. Instead the pressure field is indirectly specified through the continuity equation, which couples pressure and velocity.

For efficiency reasons segregated solvers are commonly used. Each component of velocity and pressure (via continuity equation) are solved separately. This decoupling of pressure and velocity can sometimes result in few problem that are discussed later. To ensure convergence towards a solution, a strategy to iteratively link the equations is required. If direct methods are used, all the components can be solved simultaneously. However the cost of direct solution is significantly larger than that of iterative solvers both in terms of memory and floating point operations.

A related problem is the representation of the pressure gradient term. If both u and p are stored at the centers of same control volumes, part of the pressure gradient term will cancel out. This results in a situation where pressure is being effectively solved at twice a coarser grid than velocity. This is a partial decoupling of pressure and velocity which can result in unrealistic oscillating solutions between alternating grids (checkerboard pattern). This problem can be solved by storing pressure and velocity at adjacent grid points (staggered grid) to prevent the decoupling. This has been the preferred method for decades but it is problematic for implementation in non-orthogonal and unstructured grids. Also two grids have to be maintained for solution of velocity and pressure.

In collocated grid arrangement, all the variables are stored at the same location and is very convenient for programmers. Collocated grids have become popular since the discovery of Rhie and Chow interpolation for pressure velocity coupling. The velocity at the faces is interpolated in such a way that pressure and velocity remain coupled. This method is usually seen as a correctional approach between the pressure gradient at the face and the interpolated pressure gradient (Ferziger & Peric (2001))

$$u_j = u_j - \triangle\overline{\left(\frac{1}{A_p^{u_j}}\right)}(\frac{\partial p}{\partial x_j} - \overline{(\frac{\partial p}{\partial x_j})}) \tag{2.109}$$

First the momentum equations are solved with old pressure values. This stage is the momentum prediction step which is necessary when there are other scalar transport equations to be solved. To solve for pressure using continuity equation the pressure contribution to momentum is separated from the rest. Following Ferzigers notations

$$[U] = \frac{H}{A} - \frac{1}{A}\nabla[p] \tag{2.110}$$

$$U* = \frac{H}{A} \tag{2.111}$$

H represents contributions from neighboring cells, previous time step, other sources but the pressure gradient term. Applying the divergence operator to the above equation, and setting the left hand side to 0 due to continuity, we arrive at pressure Poisson equation

$$\nabla.([U*]) = \nabla.(\frac{1}{A}\nabla[p]) \tag{2.112}$$

Once the pressure equation is solved, velocity that satisfy continuity can be obtained by adding back the pressure contribution. This is the explicit velocity correction step.

The two commonly used segregated solvers are Semi Implicit Method for Pressure Linked Equations (SIMPLE) for steady state simulations and Pressure Implicit with Splitting Operators (PISO) for transient simulations. The major difference between these two methods is that PISO solves the pressure equation more than once, while SIMPLE relies on a severe under-relaxation of pressure equation for convergence. For steady state simulations, non-linearity of the system becomes more important than pressure-velocity coupling since changes of ϕ between successive iteration is large anyway.

Chapter 3

Implementation of 3D CFD program

The basics of the Computational Fluid Dynamics (CFD) program developed in this work is briefly described in the following sections. The program solves continuum mechanics problems using the Finite Volume Method (FVM). An Object Oriented Programming approach (OOP) using C++ is used which is inspired by the design of OpenFOAM (Jasak et al. 2007, OpenFOAM 2013, Weller et al. 1998). This helps to significantly reduce the time required to write new solvers for any Partial Differential Equation (PDE). The code developed in this work is about 7300 lines and is available in Appendix C.2. The development started from the lowest units of tensor and field manipulations. The reason for this choice is to gain expertise in developing CFD software and also have the utmost freedom in implementing performance enhancements using latest technology such as Graphic Processing Units (GPUs). The code has been parallelized to run on a homogeneous cluster of Central Processing Units (CPUs) and also on a single GPU. Validation is an important step in development of any CFD program. Therefore for every new feature added to the program, such as turbulence models or new discretization methods, validation has been carried out with well known benchmark cases. Some of the test cases are described at the end of the chapter.

3.1 Tensors

Problems in continuum mechanics can be concisely expressed using tensors and associated linear field operations. For example the second order stress and strain tensors can be represented by a 3x3 array. The program represents tensors using template classes with parameters describing the rank of a tensor and a parameter specifying storage for each element of the tensor. Storage can be either single precision or double precision. Common operations for all rank

tensors such as addition, subtraction, dot product etc are optimized by unrolling loops to allow parallel operations on each element of the tensor. Use of templates for representing tensors allows production of optimized code for each instance of the template with as little effort as possible. In older CFD codes using FORTRAN, separate code had to be written even for the case of increasing the precision of solution.

An example illustrating this advantage is shown below for calculating the dot product of any size tensor.

```
template <int N>
struct Unroll {
    static FORCEINLINE Scalar dot(const Scalar* p,const Scalar* q) {
  return (*p) * (*q) + Unroll<N - 1>::dot(p + 1,q + 1);
    }
}
```

Operations specific to a given rank tensor such as finding the transpose, symmetric, skew tensors of a second rank tensor are implemented taking into consideration the known size and nature of the tensor. For example, most tensors encountered in fluid mechanics are symmetric which reduces the number of elements that need to be stored from nine to six. Such optimization are taken advantage of whenever possible.

Some of the implemented tensor operations are:

1. Inner product: The dot product on two vectors, double inner product on two second rank tensors and triple inner product on third rank tensors all give a scalar. This is concisely implemented as shown in the above code snippet. Other inner products yield vectors and tensors that requires special handling. For example, inner product of a vector and second rank tensor gives a vector, and that of two second rank tensors yield a second rank tensor.

2. Outer product: The outer product of two vectors yield a second rank tensor while that of a vector and second rank tensor give a third rank tensor.

3. Exclusive operations to a tensor of given rank :

   - First rank tensors: Cross product
   - Second rank tensors: Transpose, symmetric and skew components, hydrostatic and deviatoric components etc.

$$\begin{aligned} A &= 1/2 * (A + A^T) + 1/2 * (A - A^T) &&= symm(A) + skew(A) \\ A &= A - 1/3(trace(A)) + 1/3(trace(A)) &&= dev(A) + hyd(A) \end{aligned} \tag{3.1}$$

3.2 Fields

A tensor represent values of physical quantities or their derivatives at a point in space and time. Solution of a PDE such as the Navier-Stokes equations involves discretization of the domain into what is commonly known as a grid or mesh. The set of tensors of each point in the domain forms a tensor field of the physical quantity over the mesh that varies both in space and time. Thus a tensor field is implemented as an array (vector) of tensors of size equal to the number of nodes or faces of the mesh. The values may be stored at cell centers, vertices or face centers. All tensor operations for single grid points are extended for the tensor fields as well. Important differential operations on tensor fields that are required to formulate any PDE are described in the following paragraphs.

1. Gradient: The derivative (gradient) of a continuously differentiable scalar field gives a second rank tensor field (vector field) .

$$grad(s) = \nabla s = (\frac{\partial s}{\partial x}, \frac{\partial s}{\partial y}, \frac{\partial s}{\partial z}) \tag{3.2}$$

   Similarly the gradient of a second or higher rank tensor field can be derived by taking the gradient of each scalar component to get a tensor field one rank higher.

2. Divergence: Divergence operation on a vector field gives a scalar field that represents the net outward flow at each grid point.

$$div(v) = \nabla.V = \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z} \tag{3.3}$$

   Similarly the divergence of a second rank or more tensor field yields a tensor field one rank lower.

3. Curl: The curl of a vector field represent the rotation (vorticity) of the flow field.

$$curl(v) = \nabla xV = (\frac{\partial w}{\partial y} - \frac{\partial v}{\partial z}, \frac{\partial u}{\partial z} - \frac{\partial w}{\partial x}, \frac{\partial v}{\partial x} - \frac{\partial u}{\partial y}) \tag{3.4}$$

4. Laplacian: Laplacian is the divergence of gradient of a tensor field.

$$lap(S) = \nabla.\nabla S = \nabla^2 = \frac{\partial^2 S}{\partial x^2} + \frac{\partial^2 S}{\partial y^2} + \frac{\partial^2 S}{\partial z^2} \tag{3.5}$$

5. Temporal derivative: The total derivative measures the rate of change of a quantity ϕ as an infinitesimally small volume of material (particle) moves .

$$\frac{D\phi}{Dt} = \lim_{\triangle t \to \infty} \frac{\triangle \phi}{\triangle t} \tag{3.6}$$

In fluid mechanics the rate of change observed at a fixed point in space, i.e spatial time derivative $\partial\phi/\partial t$, is preferred

$$\frac{D\phi}{Dt} = \frac{\partial \phi}{\partial t} + U.\nabla\phi \tag{3.7}$$

3.3 Equation discretization

The partial differential equations for fluid flow can be compactly expressed by field operators discussed in the previous section. Equation discretization involves conversion of components of the PDE (first time derivative, convection, diffusion and source term) into linear algebraic equations. Also non-linear source terms have to be converted in to an equivalent linear form. The final set of equations can then be represented in matrix form as

$$[A]\{\phi\} = \{B\} \tag{3.8}$$

where [A] is a matrix of coefficients , ϕ is a vector of the unknown quantity at cell centers and b is a vector of source terms.

A field operation can be explicit in which case a tensor field is transformed into another without contributing to the coefficient matrix. The gradient (∇), divergence ($\nabla.$) and curl (∇x) operation are examples of such explicit operations. On the other hand, the operation can be implicit in which case values of ϕ at neighboring cells are coupled through the coefficient matrix A. The coefficient matrix is extremely sparse with zeros filling up most of the matrix. This is because when two cells do not share the same face the corresponding coefficients are both set to zero . Various specialized methods for efficient storage and solution of sparse matrices are available. The storage method used in this work is suitable for polyhedral meshes in which a control volume can contain any number of faces. After finite volume discretization, coefficients are obtained for each cell : a_p for the parent cell and a_n for each of the neighboring cells sharing a face with the parent.

$$a_p\phi_{\mathrm{p}} = \sum \mathrm{a_n}\phi_{\mathrm{n}} + \mathrm{S} \tag{3.9}$$

The sparse matrix format used in this study stores a_p and a_n separately in different scalar fields. Hence the non-zero elements are not stored resulting in tremendous saving of memory, and also solution with fixed point iterative methods becomes straight forward. This form of matrix representation is not suitable for computations on the GPU, hence another type of representation known as Compressed Sparse Row (CSR) is used instead. CSR is a popular general purpose format that stores non-zero values and corresponding column indices.

Finite volume discretization integrates each term of the PDE in a control volume. The volume integration is converted into a surface integral (summation) over the faces of the polyhedral mesh.

$$\oint_V \nabla * \phi \, dV = \oint_S \mathrm{dS} * \phi = \sum S * \phi \tag{3.10}$$

where S is the surface area vector and (*) represents any tensor operation.

3.4 Overview of components of CFD tool

A short summary of the components of the developed CFD program is given in the following sections. The software is developed in C++ using an object oriented programming approach (OOP). Classes are provided for different field calculus such as divergence, laplacian, temporal derivative etc. This makes the software suitable for solving PDE other than Navier-Stokes equations. Templates are extensively used to avoid duplication of code. The design of the program keeps the implementation of the physics (Navier stokes equations, turbulence model etc) isolated from other parts of the program, so that different mathematical models can be tried conveniently.

3.4.1 Partial differential equation solvers

Writing solvers for many PDEs becomes easy once the basic field and tensor operations are programed. These include divergence, laplacian, temporal derivative, gradient among others. Some of the PDE solvers implemented that were necessary for this work are briefly described as follows

3.4.1.1 Wall distance solver

It is necessary to calculate the distance of a grid cell to the nearest wall for some turbulence models and other applications. This can be obtained by solving a differential equation first proposed by Spalding (1994). The following equations are solved with boundary conditions

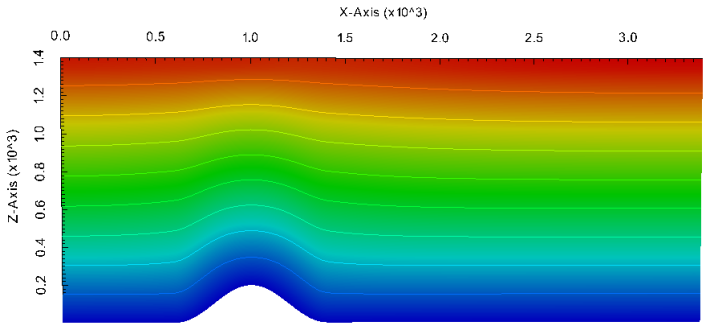


Figure 3.1: Contour map of wall distance from the surface of a 2D hill

for ϕ set as Dirichlet at ground surface and Neumann elsewhere.

$$\nabla.\nabla\phi = -V \tag{3.11}$$

Then the distance to nearest wall is calculated as

$$y = \sqrt{\nabla\phi.\nabla\phi + 2\phi} - |\nabla\phi| \tag{3.12}$$

These two equations are implemented as follows. This is the simplest solver implemented but other complex solvers do not pose more difficulty.

```
void Mesh::calc_walldist(Int step,Int n_ORTHO) {
 ScalarCellField& phi = yWall;
 /*poisson equation*/
 ScalarFacetField one = Scalar(1);
 for(Int k = 0;k <= n_ORTHO;k++)
  Solve(lap(phi,one) == -cV);
 /*wall distance*/
 VectorCellField g = grad(phi);
 yWall = sqrt((g & g) + 2 * phi) - mag(g);
 /*write it*/
 yWall.write(step);
}
```

An example simulation result using this solver for a 2D hill is shown in Fig. 3.1.

3.4.1.2 Potential flow solver

In potential flow theory, the velocity field is assumed to be gradient of velocity potential $V = \nabla\phi$ and also that the fluid is inviscid (no viscosity $\nu = 0$) and irrotational (no vorticity $\nabla xV = 0$).

For an incompressible flow where $\nabla.V = 0$, the previous two equations reduce to a single Laplace equation from which all flow parameters can be determined

$$\nabla.\nabla\phi = 0 \tag{3.13}$$

The specified initial flow field will inevitably not satisfy continuity due to imposed boundary conditions ($\nabla.V \neq 0$), hence a pressure Poisson equation is solved and the velocity is corrected with the gradient of p, which is the velocity potential ϕ. At the end of solution, continuity equation will be satisfied so that $\nabla.V = 0$. This solver can be used for initializing flow field as exemplified by its use in OpenFOAM (2013).

$$\begin{aligned} \nabla.\nabla p &= \nabla.U \\ U &-= \nabla p \end{aligned} \tag{3.14}$$

3.4.1.3 Parabolic diffusion solver

The parabolic heat equation is solved using implicit or explicit temporal discretization schemes. The steady state version drops the temporal derivative to becomes Laplace's equation for temperature, in which case under-relaxation is necessary to avoid divergence of solution.

$$\frac{dT}{dt} = -\alpha * \nabla.\nabla T \tag{3.15}$$

3.4.1.4 Transport equation solver

Once the flow field (velocity) field is established, transport of pollutants, dies and even turbulent flow quantities themselves (k and $epsilon$) can be obtained by solving a 'transport' equation for any tensor.

$$\frac{\partial \epsilon}{\partial t} + \nabla.(\rho V \epsilon) = \nabla.\mu\nabla\epsilon \tag{3.16}$$

3.4.1.5 Navier-Stokes solver

The details of this solver will be explained in the following sections but it basically solves transport equation for momentum and the continuity equation. Source terms in the form of surface and body forces such as pressure gradient, Coriolis force and others are added to the momentum transport equations. Different turbulence models are used for closure.

$$\frac{\partial V}{\partial t} + \nabla.(VV) = -\nabla p + \nabla.\nu\nabla V + F \tag{3.17}$$

3.4.2 Meshing

The issue of mesh generation is a vast topic beyond the scope of this study however we give a glimpse of what is required. It is known that the quality of mesh plays a major role in the quality of simulation results. For finite volume discretization, Hexahedral elements are known to give much better results compared to tetrahedrals. Hexahedral grid leads to faster solutions and requires lower cell count than tetrahedral grid, while keeping the same quality of results. Unfortunately most of the existing grid generation software is adapted to finite element codes in which tetrahedral elements are popular. Tetrahedral grid can be generated for a complex terrain using algorithms such as Delaunay triangulation. This has proven to be successful in the finite element field, but not so much in the finite volume field mainly due to the problems mentioned above. It is also very difficult , if not impossible, to generate Hexahedral meshes for an irregular geometry. Tetrahedral meshing of an irregular geometry is relatively easy and many free software are available for that purpose. Automatic hexahedral mesh generation suitable for finite volume solutions is still an active research area. This study does no attempt to produce a grid generator for complex surfaces but simply adds support to import grid from other grid generating software. For 2D grids, body fitted grid methodology using transfinite interpolation is used which has proved rather useful in some of the 2D hill simulations. The same method is used for simple 3D mesh generation when the surface is not too complex. This tool was enough for most of the study cases considered in this research such as simple rectangular buildings, staggered/regular array of cubes etc. To improve quality of grid with elongated and skewed cells, special care is taken during the discretization steps to account for mesh non-orthogonality and skewness as suggested in Jasak (1996).

3.4.3 Solution and turbulence modeling

Boundary layer wind flow is incompressible even in the case of many extreme cases such as hurricanes and cyclones. Like many other engineering flows, it is also of a high Reynolds type flow. Hence the program is geared towards solving an incompressible Navier Stokes equation at high Reynolds number. An incompressible pressure based solver is used as opposed to a general type compressible density based solver. The Navier Stokes equations are non-linear and coupled at the same time. Both problems can be tackled using segregated iterative solvers, in which partial solution of velocity and pressures are sought one after the other. For this study, the Semi Implicit Method for Pressure Linked Equations (SIMPLE) and Pressure Implicit with Splitting Operators (PISO) algorithms are implemented.

The linear solvers that are implemented include Successive Over Relaxation (SOR), Preconditioned Conjugate Gradient (PCG) and Preconditioned Bi Conjugate Gradient (PBiCG) methods. An algebraic multi grid solver (AMG) is planned for the future. The turbulence model implemented include many high-Re versions of Reynolds Averaged Navier-Stokes (RANS) models and the Smagorinsky Large Eddy Simulation (LES) model. The standard k-epsilon model has proven to be a cost effective solution to many wind engineering flows. However it fails to give accurate results in regions of flow separation such as at corners of buildings. Many large scale experiments have been conducted at the Wall of Wind (WoW) facility at FIU which consistently demonstrated this deficiency of RANS models. LES models, including the simplest one implemented in this study (Smagorinsky model), have shown good agreements with experimental data. The use of sub-grid scale (SGS) two equation models without the use of wall functions gives the best results, but the temporal and spatial resolution requirements for high-Re flows limits its applicability.

3.4.4 Parallelization

Even for simulation on a simple cubical building, the computational demand may be very high depending on the accuracy required (Kose & Dick 2010). Flow around bluff bodies is extremely unsteady and turbulent and require fine resolution in both time and space. For instance, Lim et al. (2009) used about 10 million cells to model a flow around a single building for a flow with a Reynolds number of 20,000. This Reynolds number (Re) is in fact too low compared to typical values of a few millions in wind engineering. Hence simulation of realistic situation would require much more grid cells. The demand for simulations on complex terrain is even more severe. Therefore parallel computing can be helpful in a wide spectrum of flow problems.

The CFD software developed is parallelized using the Message Passing Interface (MPI) communication protocol to exchange information between different sub-domains. The communication is kept as low as possible to account for the relatively slow Ethernet network connections that are common in commodity clusters. The software is also parallelized to run on the state-of-the-art High Performance Computing (HPC) technology using General Purpose Graphic Processing Units (GPGPUs). A speed up of up to 100 times as fast as a serial version has been reported in literature (Julien & Senocak 2009). However this was for simple benchmark problems that are not representative of simulations carried out in practical wind engineering. Nowadays, regular desktop computer with GPUs of up to thousands of cores can be bought for couple of hundred dollars and give cluster-level performance. The top-most supercomputers in the world use both GPUs and CPUs to reach peak performance in the order of

peta FLOPS (Meuer 2013).

3.5 Development of high performance CFD code

Large scale simulation of wind flow over complex topography requires tremendous amount of computational resources: CPU hours and memory. Also the use of mesh refinement close to walls or use of more complex turbulence models, for example LES instead of K-epsilon model, will add to the computational demand. As mentioned before, even simulations around a single building may require tens of millions of grid cells to fully resolve the flow. Hence it is usually necessary to take a cut in accuracy of flow simulations close to walls by assuming the law of the wall to hold there. Parallel computation on cluster of machines can help to get quick results without degrading quality of results.

3.5.1 Domain decomposition

Complex terrain simulations produce mega bytes of data at each time step of the simulation, making it impossible to simulate the whole domain all in one computer. The high performance CFD software uses domain decomposition methods in which each processor takes care of part of the terrain, while exchanging information during the solution stage. Domain decomposition is a 'divide and conquer' strategy that is commonly used when either the problem is too big to fit in memory or the sub-domains are easily solved than the original. The method is extensively used in aerospace engineering to conduct finite element and finite volume CFD simulations on parts of an air-plane. The program uses a non-overlapping domain decomposition methods to parallelize the solution of the Navier Stokes equations. The details of the parallelization are given in the following sections.

3.5.2 Platform for high end simulation

From 2009-2012, the Tesla-128 cluster at Florida international university is used for development and validation of the code. The cluster is composed of 64 nodes as shown in Fig. 3.3, each with a fast Ethernet interface and a gigabit Ethernet interface. All 64 nodes are connected by a 48-way fast Ethernet switch. From May 2012 onwards, the multi-institutional Shared Hierarchial Academic Research Computing Network (SHARCNET) is used. It is much more powerful than the Tesla cluster and also have GPU clusters on which the code is tested.

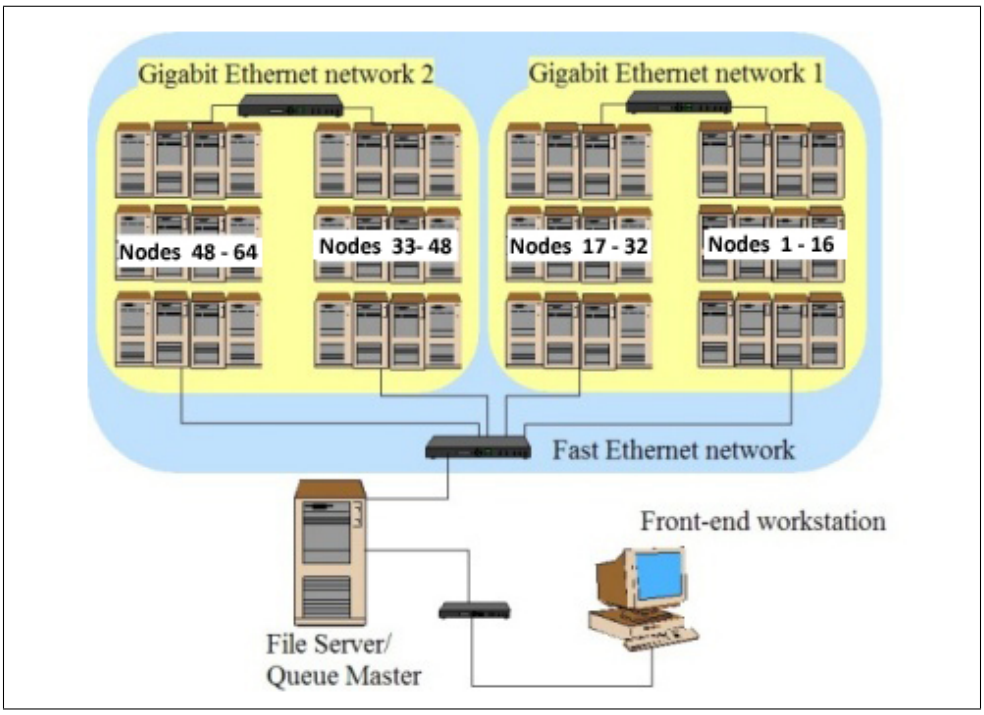


Figure 3.2: MAIDROC tesla cluster at FIU with 2 x 64=128 cores

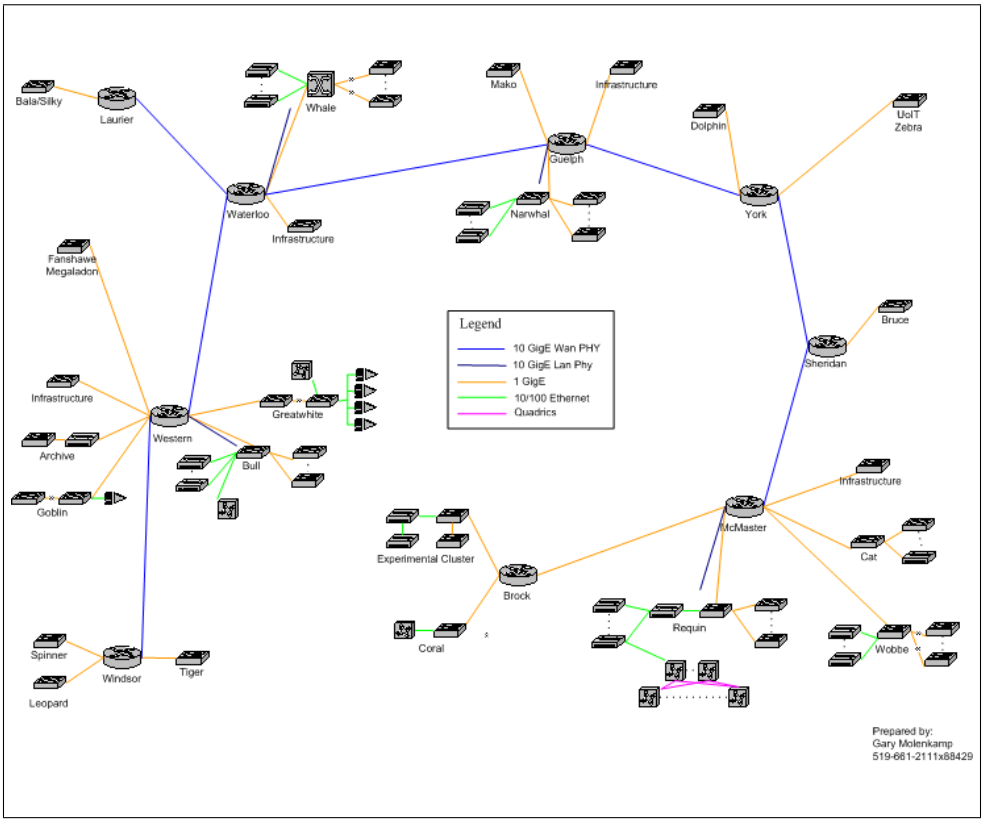


Figure 3.3: SHARCNET cluster, a network of high-performance computers

3.5.3 Parallel computing

Coarse grained parallelism in a distributed memory cluster is traditionally achieved by domain partitioning strategies. The whole domain is partitioned into smaller sub-domains which are assigned to one processor in a cluster. In this study a non-overlapping type of domain decomposition method is implemented where information such as pressure and velocity is exchanged at the boundary through ghost cells during the calculation phase. The MPI is used to exchange information between sub-domains.

GPGPU are overtaking CPUs in the HPC market. They are especially suitable for solving linear system of equations such as those obtained from fluid flow problems. Julien & Senocak (2009) reported speed ups of up to 100 times compared to a CPU implementation. The program is parallelized using NVIDIAs Compute Unified Device Architecture (CUDA) programming toolkit to harness the fine grained parallelism offered by GPUs. Both methods of parallelization are combined at the solver level so that a mixed CPU-GPU computation is possible. Finally the speed up numbers obtained for different size problems are compared.

3.5.3.1 Coarse grained parallelism

Parallel computing using domain decomposition (DD) methods have been used extensively in finite element methods used in aerospace engineering. Even when the computational resources were very limited, the decomposed sub-domains are solved one by one on a regular desktop computer by imposing special boundary conditions suitable for the kind of problem being solved. Some of the non-overlapping DD methods are the Dirichlet - Neuman, Neumann-Neumann, and other adaptive variations of these methods suitable for hyperbolic convection problems. While the motivation for these methods was to solve large size problems which do not fit in the memory space of a desktop computer, our motivation in this study is to exploit concurrency using a cluster capable of holding the whole computational domain. Thus synchronization between the sub-domains is done while all sub-domains are being solved simultaneously. The domain partitioning strategy adopted in this study is done in two ways. First synchronizing all the working processors at each and every iteration of the solver using barriers $MPI\_Barrier()$. Gropp et al. (1999) describes a way of parallelizing Poisson equation using this method. An asynchronous communication method is also implemented and tested in this study. The details of this unique implementation is given later in this chapter.

3.5.3.2 Fine grained parallelism

GPUs are the latest technology in HPC that broadens the scope of graphic co-processors to number crunching besides rendering graphics. GPUs with hundreds of processors are very cheap to set up compared to cluster of CPUs. GPGPU computing is at its infancy compared to distributed computing using MPI. However, excellent acceleration of the fluid simulations on GPUs have been reported in many fields including wind engineering (Corrigan et al. 2009, Julien & Senocak 2009, Selvama & Landrus 2010).

The first generation of GPGPUs were difficult to program because one has to use graphics rendering operation to do number crunching as well. This changed with the introduction of NVIDIAs CUDA programming language and OpenCL which are extensions to traditional programming languages such as C. NVIDIAs CUDA programming language is used to parallelize three solvers SOR, PCG and PBiCG. These three solvers are used to solve incompressible Navier-Stokes equations, Poisson pressure equation and transport equations used in turbulence modeling. Among the above equations, the solution of the elliptic Poisson-pressure equation is the most time consuming which makes it a good candidate for computation on the GPU. All of the solvers mentioned can be implemented on the GPU with relative ease, but in some cases sacrifices are made for ease of implementation and better parallelization. Algorithms that are hard to parallelize on the GPU include pre-conditioners of the Incomplete Cholesky type. Selvama & Landrus (2010) reported speed up of up to 24x using a simple Jacobi preconditioner thus that is also used in our program. Corrigan et al. (2009) reported a speed up of upto 33x times over the equivalent serial code on an unstructured grid. Use of shared memory and coalesced memory access are reported to accelerate GPU solver significantly, but no attempt is made in this study to optimize implementations to the fullest. In general structured grid solvers have a regular memory access pattern that can be exploited during optimization, but unstructured grid requires re-numbering to ensure two neighboring cells remain close in memory. Most of the comparisons in literature on CPU vs GPU computations are done on structured grid that heavily benefit from the above memory optimization techniques, hence those reported numbers may not be representative of expected performance on practical problems that use unstructured grids.

3.5.4 Relaxation algorithms

Relaxation methods are iterative methods suitable for solving sparse linear systems of equations. Although they are hardly used for solving system of equations all by themselves, they

can be good preconditioners for other methods that have fast convergence properties. All relaxation algorithms can be formulated as updates of a solution vector starting from initial guess x_0 as follows

$$x^{(k+1)} = Tx^{(k)} + c \tag{3.18}$$

Given a decomposition of matrix A = L + D + U, the most common relaxation methods namely Jacobi, Gauss-Seidel and Successive over relaxation (SOR) are formulated as follows.
Jacobi:

$$x^{(k+1)} = D^{-1}(b - (U + L)x^{(k)}) \tag{3.19}$$

Gauss-Seidel:

$$x^{(k+1)} = (L + D)^{-1}(b - Ux^{(k)}) \tag{3.20}$$

SOR:

$$x^{(k+1)} = (1 - \omega)x^{(k)} + (\omega){x^{(k)}}_{GS} \tag{3.21}$$

Jacobi is inherently parallel because the new values are computed solely from old values. The stencil used to compute new value of the jth component ${x^k}_j$ depends on the type of differential equation being solved, nonetheless all the values are taken from the old iteration. This method is easily parallelizable with the only challenge coming from stencils which have points lying in a different processor. The Gauss-Seidel method uses values from the current iteration as they become available, adding to the challenge of parallelization. Depending on the order of computation, different results can be obtained leading to different Gauss-Seidel methods. This is problematic for validation of parallelly computed results against serially computed results. However Gauss-Seidel method has superior convergence properties than Jacobi,and is proven to converge twice as fast asymptotically. The SOR method is an extension of Gauss-Seidel method that tries to further accelerate convergence by over-relaxation. It combines newly computed values and old ones with a factor $\omega > 1$. The method is equivalent to the basic Gauss-Seidel for $\omega = 1$.

The sequential nature of Gauss-Seidel can be broken by selecting specific order of computation that allows for parallel computation. One such method is the wavefront ordering where all points on the same diagonal are calculated in parallel. The downside of this method is that it is difficult to load balance because of unequal length of the diagonals. The degree of parallelism increases from the shortest diagonals at corners to the longest diagonal in the middle. Another alternative is to use graph coloring algorithms to form computation stencils of nodes containing of only one color. For example in simple two dimensional grid for solving Poisson equation, red-black coloring of adjacent nodes give 5-point stencils of same color neighbors.

First updates for stencils with red points at the center are done parallely and then the same can be done for the black nodes. A third alternative is to not care about order of updates at all. This method is sometimes known as chaotic relaxation Chazan & Miranker (1969). The stochastic behavior limits analysis of convergence properties. The method may also diverge solely due to the way updates are done, even though the convergence conditions of the Gauss Seidel method are met.

SOR is convergent for $0 < \omega < 2$ for symmetric positive definite (SPD) matrices. A value of $\omega = 1.7$ gives good acceleration for many problems, while maintaining convergence properties. However we are mostly interested in faster convergence rather than just convergence,thus higher values may be used. A symmetric version of the method does a forward SOR sweep followed by another sweep in reversed order. This usually converges slower than standard SOR with optimal ω value. The motivation for this method is the symmetry of the iteration matrix which allows it to be used as a pre-conditioner for SPD matrices. The convergence rate of fast solvers such conjugate gradient method (CG) and generalized minimal residual (GMRES) is highly dependent on the condition number of the matrix. Infact all the above relaxation methods are too slow for practical calculations so they are mostly used as preconditioners or as smoothers to remove low frequency errors.

3.5.5 Preconditioning

Matrix preconditioning is a procedure to reduce the condition number of the matrix so that it becomes more suitable to numerical algorithms. The preconditioner M is usually a partial inverse of the matrix itself that can be calculated fast enough. The range of possible preconditioners is from the identity matrix I to the actual matrix inverse itself A^{-1}. The former is a no preconditioning case, while the later is an extreme case where the solution can be found in one iteration. There are a bunch of preconditioners in between with different cost-to-benefit ratio and suitability for parallelization. The procedure of preconditioning is outlined in algorithm 1 with one of fastest solvers for sparse linear systems: the preconditioned conjugate gradient method. The preconditioning is applied on the residual by multiplication with the preconditioner M^{-1}. In practice this procedure is done in such way that neither the matrix M nor its inverse need to be stored, because it will be dense even for sparse matrix A. Also the matrix is not inverted , rather forward and backward substitutions are used to solve triangular system $Mz_{k+1} = r_{k+1}$ where M is usually some incomplete LU-factorization of A. For diagonally dominant matrices,the Jacobi preconditioner $M = D$ is effective. The preconditioner scales rows of the matrix such that elements on the diagonal are one. This method is good for parallel precon-

Algorithm 1 Preconditioned conjugate gradient

procedure PCG(A)
  $r \leftarrow b - Ax_0$
  $z_0 \leftarrow M^{-1}r_0$
  $p_0 \leftarrow z_0$
  while $r \neq$ *small* **do**
    $\alpha_k \leftarrow \frac{r_k^T z_k}{p_k^T A p_k}$
    $x_{k+1} \leftarrow x_k + \alpha_k p_k$
    $r_{k+1} \leftarrow r_k - \alpha_k A p_k$
    $z_{k+1} \leftarrow M^{-1} r_{k+1}$ ▷ Preconditioning: solve $Mz_{k+1} = r_{k+1}$
    $\beta_k \leftarrow \frac{z_{k+1}^T r_{k+1}}{z_k^T r_k}$
    $p_{k+1} \leftarrow z_{k+1} + \beta_k p_k$
  end while
  return x
end procedure

ditioning because a processor can compute relevant slice of the preconditioner by itself. Also sequentially computed result will be exactly the same as its parallelly computed counter part which may be an important advantage during solver development stage. The symmetric gauss siedel preconditioner is given in equation 3.22. Here L and U are not exact LU decompositions but the upper and lower triangular parts of A = L + D + U. Thus usually only the inverse of the diagonal is stored, and that is usually done for efficiency reasons i.e. to avoid division in the inner loops.

$$M = (D + L)D^{-1}(D + U) \tag{3.22}$$

Similarly the SSOR preconditioner can be formulated by introducing ω. Optimal value of ω will lead to lower number of iterations for solution.

$$M = \frac{1}{2 - \omega}(\frac{D}{\omega} + L)D^{-1}(\frac{D}{\omega} + U) \tag{3.23}$$

The SSOR preconditioner is among the best general preconditioners for sparse matrices, and usually gives much better results than the Jacobi preconditioner. However it is very difficult to parallelize due to the sequential nature of gauss-siedel sweeps.So far the preconditioners discussed formulate M from components of A itself: D , L, U. Better preconditioners can be obtained by conducting an incomplete factorization of A.

$$M = L_* D_* U_* \tag{3.24}$$

The standard LU or Cholesky factorization is followed with dropping of elements that do not have a corresponding entry in A. If all such elements are dropped i.e. no fill-ins allowed, the preconditioner so obtained is ILU-0. For SPD matrices the cholesky decomposition is applied in a similar manner. Incomplete factorization methods require separate storage of the preconditioner matrix, which in the case of 0 fill ins is same size as the matrix A itself, but are among the best general preconditioners for sparse matrices. Better preconditioners that do not preserve the same sparsity as matrix A can be obtained, but the cost-benefit ratio should be examined because the factorization stage consumes significant amount of time. To avoid computation and storage of off-diagonal elements, one could opt for finding incomplete factorization of only the diagonal elements. This method , also known as the D-ILU, assumes the off diagonal components are same as the original matrix A and the preconditioner becomes

$$M = (D_* + L)D_*^{-1}(D_* + U) \tag{3.25}$$

3.5.6 Parallel implementations

The suitability of relaxation algorithms and preconditioners for parallel implementation has been discussed in the previous sections. The Jacobi sweeps are the simplest to parallelize but even those are not embarrassingly parallel due to the need for values of neighboring points which could be in a separate processor. A 5-point stencil with off processor neighbors is shown in Fig. 3.4. If the value of the neighbor on core 2 is fetched every time it is needed, the parallel performance will degrade due to frequent small chunk exchanges. This problem can be solved by exchanging values for a layer of cells around the boundary,*halo* layer, at once.With this change, the calculation for a stencil at the border become exactly same us those in internal cells. After each Jacobi sweep, each processor updates values at the halo layer from the neighboring processor. Wide halo layers (two or more halos) may be used when the stencil encompasses neighbors two or more steps away from the central cell. Thicker layers also help to reduce the communication overhead since the inner halo layer's values can be locally updated without exchanging data in every iteration (Fredrick & Marc 2010). With this approach and an n-halo layer,communication need to be done only once every n-th iteration. Information exchange can be done using MPI_send call with a corresponding MPI_recieve in each processor ordered in such a way that when one processor sends halo layer values, the other should await for the message with a corresponding receive call. Besides the complication associated with order of messages, this blocked communication method adds additional synchronization points that are avoidable. Minimizing synchronization points is

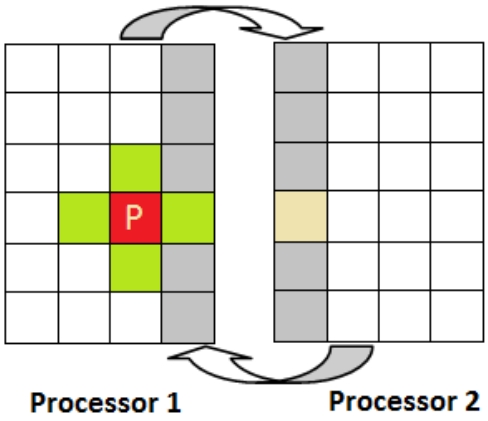


Figure 3.4: A 5-point stencil with halo layer for exchanging information between processors

crucial for good performance in massively parallel systems. Both problems can be solved using asynchronous communication via MPI_isend and MPI_irecieve calls as outlined below. The synchronization is done once with an MPI_Waitall call at the end instead of being at every send and receive call as in the synchronous case.

Algorithm 2 Asynchronous communication

```
procedure EXCHANGE
    for all to ← neighbors do
        MPI_isend(to)
        MPI_irecieve(to)
    end for
    MPI_Waitall()
end procedure
```

The difficulties associated with parallelizing Gauss-Siedel and SOR algorithms have been discussed in previous sections. Using graph coloring algorithms, one sweep of SOR can be broken down to two or more equivalent sweeps that can be applied in parallel. The wavefront method exploits parallelizability on the diagonals as shown in Fig. 3.5. A source of concern with these methods is load balancing of work between different processors. The coloring algorithm should ensure approximately equal amount of nodes is assigned to each color on all processors, otherwise the time spent waiting for other processors to finish their share of work becomes a bottleneck. The wavefront method is predisposed to have unequal work at different diagonals thus it inherently suffers from this problem. An advantage of wavefront method over graph coloring is that it preserves the original order, and thus have the same convergence rate as its sequential counterpart. It is known that re-ordered gauss siedel converges slower than the

sequential counterpart that has a natural ordering. The first sweep in a red-black Gauss-Seidel is basically a Jacobi iteration since no values from the current iteration are used. Thus the overall red-black algorithm will have convergence rate equivalent to a Jacobi-GS sweeps. The wavefront method uses values from the current iteration, but it offers significantly less parallelization than graph coloring algorithms do. Asynchronous Gauss Siedel (chaotic) relaxation may be easier alternative that can avoid the above complications if convergence can be ensured somehow.Besides ease of implementation, asynchronous method do not need to exchange halo layer values at every iteration thereby completely avoiding the associated latency. Synchronization is avoided at all stages of solution, however the method may take larger number of iterations to converge, or sometimes not converge at all. Halo layers are updated randomly, i.e. as the neighbor processor sends them,therefore it is difficult to analyze convergence property of chaotic relaxation methods. Parallelization of PCG solver involve different stages with varying

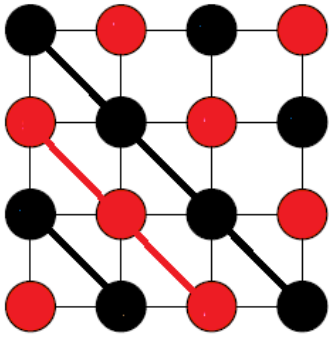


Figure 3.5: Red-black colored graph for parallel Gauss-Siedel

degree of difficulty.These stages are outlined in the pseudo code below in algorithm 3. The scalar operation SAXPY ($y \leftarrow \alpha * x + y$) is embarrassingly parallel with no communication required whatsoever.However matrix-vector product and preconditioning stage are very difficult to parallelize and are usually bottlenecks of performance. The EXCHANGE operation at the beginning makes sure that halo layers have the latest values before local matrix-vector multiplications are done. The operation has an implicit barrier at the end that further adds to synchronization overheads. The local DOT products can be done in parallel however the ensuing summation of local products i.e. REDUCE operation introduces many synchronization points. This operation is commonly done through smart algorithms that are able to do the calculation in $O(log_2(N))$ time. The matrix preconditioning stage is difficult to parallelize except for the simplest case where Jacobi preconditioner is used. Due to complexity of implementing a parallel preconditioning algorithm that gives same result as its sequential counterpart, domain decomposition method with local matrix preconditioning are commonly used. Similar to the

case with asynchronous gauss siedel method, this may result in more number of iterations for convergence.

Algorithm 3 Parallel PCG

$EXCHANGE(p)$
$z \leftarrow M * p$
$oo_r \leftarrow DOT(p, z)$
$oo_r \leftarrow REDUCE(oo_r)$
$\alpha \leftarrow \frac{o_r}{oo_r}$
$x \leftarrow SAXPY(x, p, alpha)$
$r \leftarrow SAXPY(r, z, -alpha)$
$z \leftarrow M^{-1} * r$
$oo_r \leftarrow o_r$
$o_r \leftarrow DOT(r, z)$
$o_r \leftarrow REDUCE(type, o_r)$
$\beta \leftarrow \frac{o_r}{oo_r}$
$p \leftarrow SAXPY(p, z, beta)$

3.5.7 Asynchronous implementation

In the previous section different methods of implementing a parallel algorithm that strictly follow the same computational path as the sequential counterparts have been discussed. The work associated for strict implementation of this requirement can sometimes be overwhelming. At times a significant reduction in complexity can be achieved by relaxing this requirement. For example opting for asynchronous gauss siedel avoids the need for complex algorithms such as graph coloring and wavefront method. Local preconditioning through domain decomposition avoids the need for a parallel ILU preconditioner with graph coloring or wavefront method. The number of synchronization points introduced for parallelizing PCG solver also suggests scalability issues on massively parallel systems. Given all the above problems, it is worthwhile to investigate asynchronous algorithms. In these methods, each processor does its own calculations with no synchronization whatsoever. As long as halo layers are updated regularly, one processor could be solving fluid equations while the other solves solid equations, one processor could be using PCG and the other SOR etc.. This complete freedom comes at the price of increased number of iterations or even divergence of solution, non-reproducibility in the sense that sequential computation follows different path than its parallel counterpart. However its advantage regarding scalability can be a deciding factor with the ever increasing computational power with thousands of processors,and load balancing problems due to non-uniformity of clusters.An asynchronous implementation of solvers is outlined in algorithm 4. Processors do

not exchange information at designated synchronization points, unlike the case of synchronous computation where information is exchanged at the end of each iteration and other places. Each processor continually probes for messages from its neighbors by MPI_iprobe. When a processor receives a halo layer data from neighboring processor, it sends back data of its own halo layer at the shared boundary or an END message to indicate convergence on its local problem. Each processor also keeps count of how many of its neighbors reached convergence and then stops calculations when all of them and itself reach convergence.

Algorithm 4 Asynchronous solution

```
Initialize halo layer exchange
nConverged = 0
while converge is not reached do
    Do one sweep of solver asynchronously: Jacobi,SOR, PCG etc.
    while nConverged ≠ nNeighbors do
        MPI_iprobe(message)
        if message is NULL then
            Do nothing
        else if message is HALO then
            MPI_recieve(HALO)
            Update halo layer
            Calculate residual
            if Converged then
                if END not sent before then
                    MPI_send(END)
                    Mark we have sent END message
                end if
            else
                MPI_send(HALO)
            end if
        else if message is END then
            MPI_recieve(END)
            nConverged = nConverged + 1
            if END not sent before then
                MPI_send(END)
                Mark we have sent END message
            end if
        end if
    end while
end while
```

3.5.8 Scalability study

Scalability study with number of computing units is a necessary step to evaluate the efficiency of parallelization. A badly parallelized program can give very poor performance due to poor algorithm, nature of the problem, communication latency etc Numerical calculations in CFD usually give good parallel speed ups due to the relative ease CFD can be parallelized. An embarrassingly parallel problem does not incur any performance loss due to communication alone. However, CFD computations do require communication of pressure, velocity and other quantities at the boundaries of the domain and quite frequently too. For an iterative solver, , communication at each step of the iteration is usually necessary.

3.5.8.1 Coarse-grained scalability study

The speed up of the coarse grained parallelism using message passing was tested on a cluster of the following specification; 2 cores per node, AMD 1.6GHz 2GB RAM, fast Ethernet connection. The lid-driven problem is run with a grid 256 x 256 decomposed into sub-domains. Run time is measured from the start of loading the cases to end of iterations. The loading time is decreased from the total run time which otherwise would have biased the result. For example, the one node test took too long to load the case compared to that of sixteen node case as shown in the table 3.1. The speed up to 16 processors is very good but it starts to flatten out onwards as evidenced by the 36 processors case. The total number of cells is 64k which is relatively small, hence better speed up numbers are expected with cases of bigger size. The cavity problem is run again with a grid of 1024 x 1024 resulting in a total of 1 million cells. As expected, much better scaling numbers are obtained for larger number of processors due to a larger computation to communication ratio. The 36 processors case showed an improvement of 50%, and the 25 processor case a 16% increase. The single processor case could not solve this bigger problem due to memory constraints; hence the percentages are calculated relative to 16 processors case. This test demonstrates an advantage of domain decomposition to solve large problems which are impossible to do on one processor. And also the point where the scaling shows diminishing returns differs based on the problem size.

3.5.8.2 Fine grained scalability study

Speedup test for the fine grained parallelism is conducted on an Intel quad core with one Nvidia Quadro FX 3700m workstation GPU and Intel Core 2 quad 3.0 GHZ cpu. The GPU has 128 processors and 1 GB memory. The test is done separately for the SOR and conjugate gradient

Table 3.1: Speed ups for 256 x 256 case

Processors	Time(ms)	Speed-up
1	1427135	1.00
2	772862	1.85
4	427012	3.34
9	198425	7.19
16	124985	11.42
25	92477	15.43
36	84422	16.90

Table 3.2: Speed ups for 1024 x 1024 case

Processors	Time(ms)	Speed-up	Improvement
16	4100112	1	
25	2819571	1.45	15.61%
36	2037217	2.01	51.98%

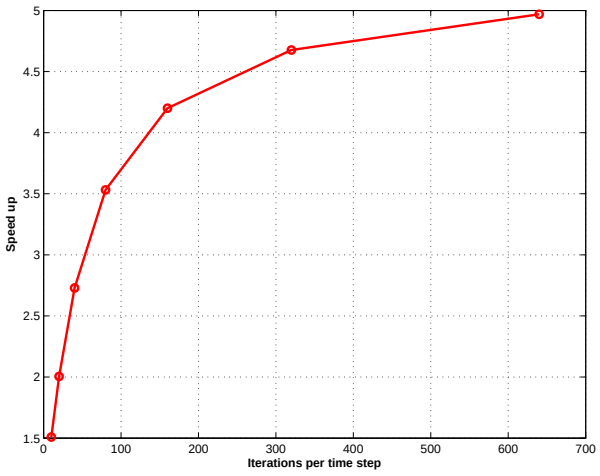


Figure 3.6: GPU speed up relative to CPU for fixed number of iterations

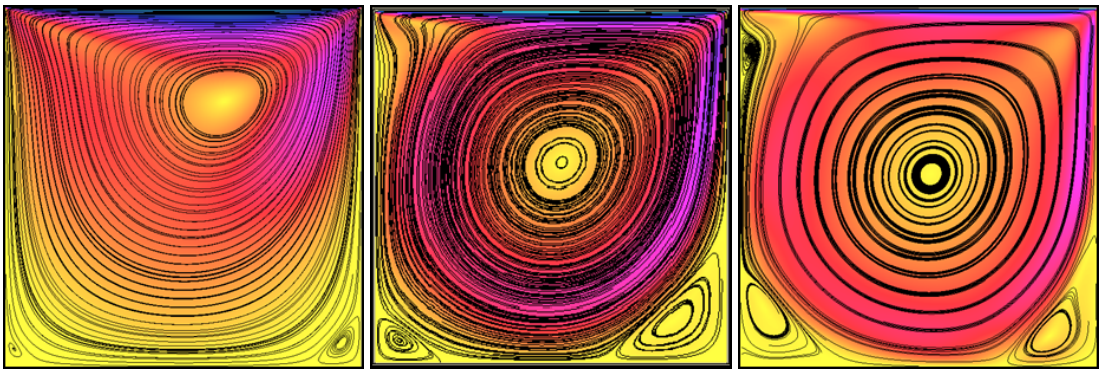
solvers. In general, the SOR solver shows a much better scaling than the PCG solver because it does more iteration per time step. In each time step, matrices are copied from the CPU to the GPU which severely degrades performance. To compensate for this latency the solver should be doing many iterations in each time step. Otherwise most of the number crunching will be done by the CPU and performance may not improve at all or even degrade in some cases. A steady-state problem getting close to overall convergence or a transient problem with very small time step (e.g.LES simulations) are some examples where GPU may not scale well. Pre-allocation of workspace once on the device (for all the matrices and vectors that will be used inside the iterations) and updating directly on the device has been used by Julien & Senocak (2009) to avoid this latency.

For a 3D lid driven cavity test with a grid of 128 x 128 x 32, a speedup in the range of 1.3 - 5 relative to the single CPU is obtained on the machine specified above. This is rather disappointing compared to what is reported in literature, but it should be noted that the number of iterations done per time step was relatively small in both cases especially towards the end. To illustrate this point, the above problem is solved with fixed number of iterations per time as shown in Fig. 3.6. Latency between host and device memory and between shared and global memory in the device are bottlenecks for GPU solvers. Both optimizations were not done for our implementation.

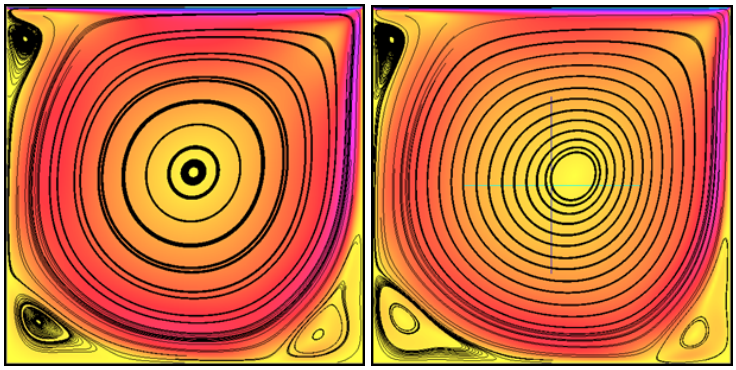
3.5.9 Validation with benchmark problems

3.5.9.1 Lid-driven cavity

The well known lid-driven cavity test is used to validate the implementation of both fine grained and coarse grained parallelism. The streamline plots for this two dimensional flow at different Reynolds number and a grid of 128 x 128 are shown in Figs. 3.7-3.8. Botella & Peyret (1998) conducted spectral analysis of the lid-driven cavity flow for CFD benchmarking purpose. Plots of u on vertical section and v on horizontal section show excellent agreement as shown in Fig. 3.9. Streamlines and pressure contours for higher Reynolds number are also compared with the plots found in Goyon (1996). The flow structure for the primary and secondary vortices shows very good similarity. Validation of the parallel code for both CPU and GPU implementations is done indirectly by comparing the result of decomposed cases against the serial code's result, which is already validated. Different problems with different number of sub-domains have been tested and the results are in good agreement which proved that both parallel implementations are correct. For illustration, a 2D and 3D lid-driven cavity problems



(a) Re=100 (b) Re=1000 (c) Re=3200



(d) Re=5000 (e) Re=7500

Figure 3.7: Streamlines for different Reynolds numbers showing progressive formation of eddies at the bottom right corner → bottom left corner → top right corner

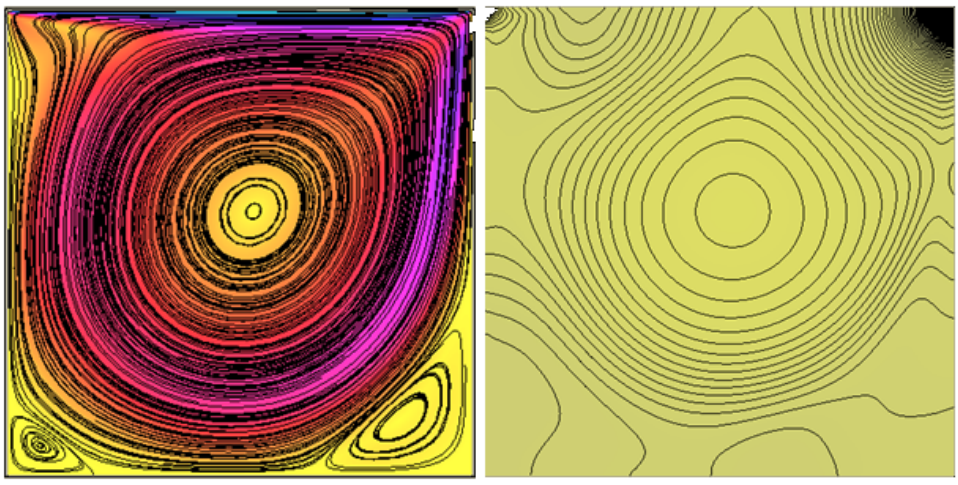


Figure 3.8: Streamlines (left) and pressure contours (right) of lid-driven cavity flow at Re=1000

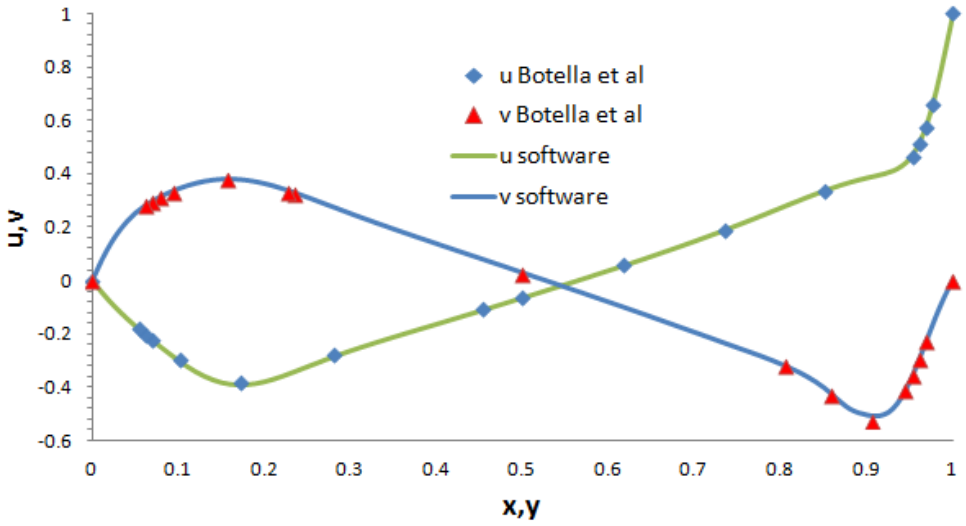
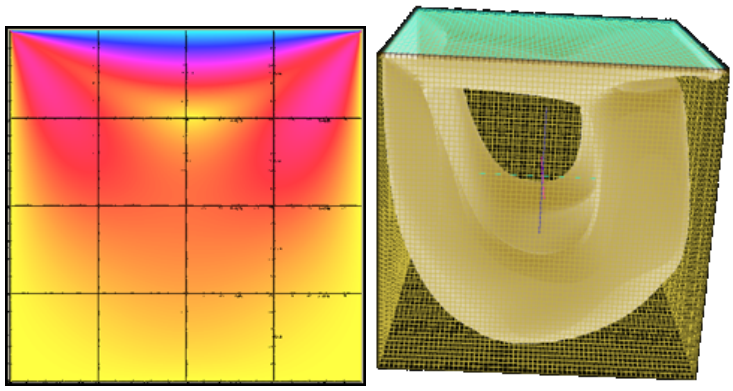


Figure 3.9: Horizontal(u) and vertical(v) velocity profiles along mid vertical and horizontal sections respectively



(a) Decomposed domain (b) Iso-surface

Figure 3.10: Solution of 3D lid-driven cavity problem solved parallely with 16 sub-domains (left), and the resulting 3D iso-surface plot that shows the flow pattern (right)

are solved decomposed into 16 sub-domains as shown in Fig. 3.10. In all cases, no mismatches are observed at interfaces, that indicate iterations in each sub-domain have been done until full convergence is reached. The implementation has also been tested on more complex problems with unstructured mesh. The result found from the asynchronous implementation are in agreement with that of synchronous implementation in all cases. As discussed in previous sections, asynchronous algorithm may sometimes diverge where a synchronous algorithm would not, and this has been observed in some of the other tests

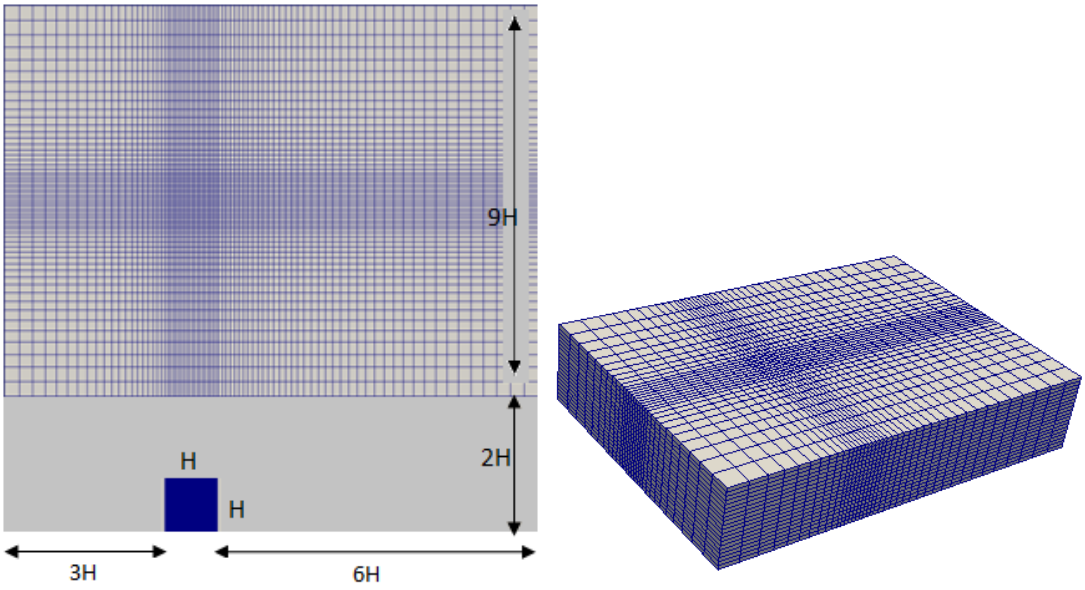
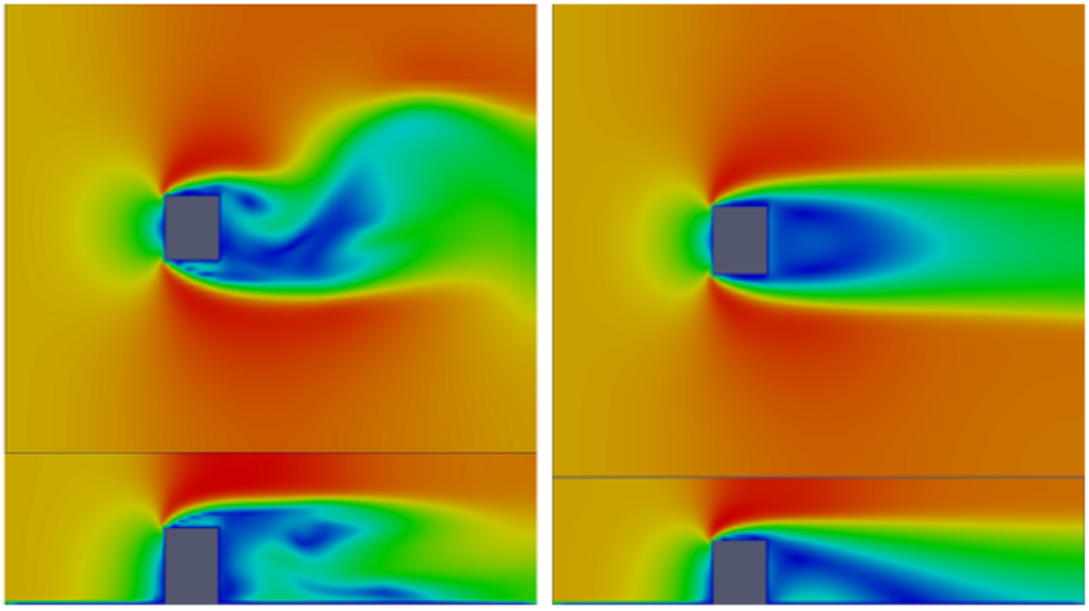


Figure 3.11: Grid for a cube in a boundary layer case of Kose & Dick (2010)

3.5.9.2 Flow around a bluff body

The RANS and LES turbulence models are validated with a practical wind engineering application of flow around a bluff body, namely a cube immersed in a boundary layer. The external pressure distribution around the cube is sought using RANS and LES turbulence models. The setup used for this test is similar to the one used by Kose & Dick (2010); cube height H = 4 cm, bulk velocity 10 m/s and molecular viscosity at 10^{-5} kg/ms, and Re = 40000. The mesh consists of about 200000 cells. The cells are expanded away from the cube as shown in Fig. 3.11. Appropriate boundary conditions are applied as specified in Kose & Dick (2010); Richard and hoxey inlet profiles for $k-\epsilon$ model and a turbulent inlet with random fluctuations for the LES model, symmetry boundry conditon on the left,right and top walls, a pressure-outlet conditon, and simulations are carried out using standard k-epsilon and Smagornisky LES turbulence models. A Smagorinsky LES model with a time step of $2x10^{-4}$ sec is used for the simulation.



(a) LES (b) RANS

Figure 3.12: Plots of instantaneous and mean velocity contours showing vortex shading behind the cube

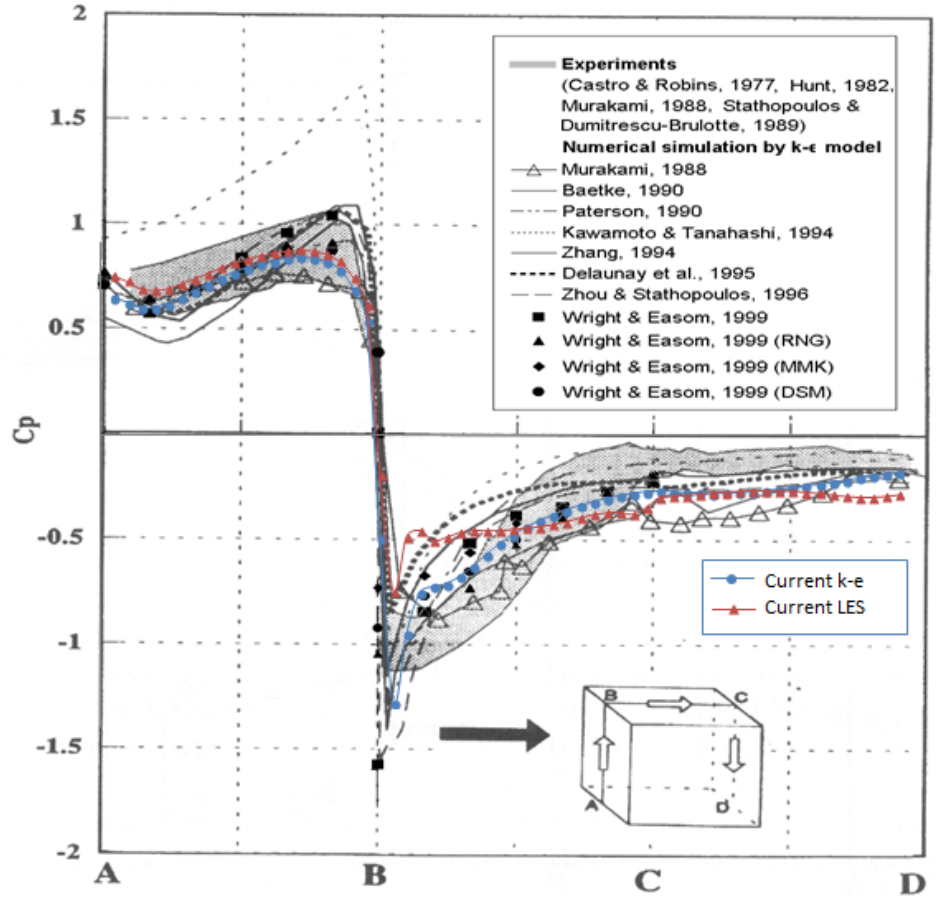


Figure 3.13: Pressure coefficients along vertical section of cube. Adapted from Bitsuamlak et al. (2010)

Formation of Karman vortex street behind the cube is captured by the simulation as shown in the instantaneous profile of Fig. 3.12. Pressure coefficients are calculated for a vertical section passing through center lines of the front, top and back faces of the cube. From LES simulations pressure values can be obtained at any instant of time, while RANS gives only time averaged (mean) pressure values. The pressure values are normalized by the dynamic head according to Eq. 3.26. The reference pressure P_0 is usually taken as atmospheric pressure.

$$C_p = \frac{P - P_0}{\rho U^2} \tag{3.26}$$

The pressure coefficient (Cpe) distributions from the current study are shown in Fig. 3.13 along with other experimental and CFD investigations by many researchers including Bitsuamlak et al. (2010). The results from the current CFD study on the upstream side of the onset of flow separation lie with in the shaded area that signifies limits of acceptable range. The current standard k-epsilon model over shoots at the leading edge, where flow separation occurs, similar to the results of Wright & Easom (2003) who used the same turbulence model. This confirms the suspicion that RANS models can indeed have problems at flow separation zones. On the other hand, the current LES model gives reasonable values even at the leading edge. On the top wall, the current LES model seem to underestimate the suction pressure compared to $k - \epsilon$. This could be due to the use of a simple LES model with a constant C_s that needs to be adjusted based on the flow behaviour.

Chapter 4

Numerical evaluation of roughness effects

Atmospheric Boundary Layer (ABL) flow is affected by aerodynamic roughness that consists of the effect of surface cover (roughness) as well as the shape of the terrain (topography). This chapter examines the effect of roughness alone by conducting Computational Fluid Dynamics (CFD) simulations over various roughness setups. Given velocity and turbulence intensity measurements at a certain location, it is possible determine roughness parameters z_0 and d by fitting suitable profiles of either the log-law or power-law type. Some methods of fitting, with different degree of accuracy, have been discussed in section 2.7.2. Therefore the task of determining roughness parameters can be considered to be equivalent to determining velocity and turbulence intensity profiles either from field observations or numerical simulations, which is the case in the current work. The investigation of roughness effects is conducted beginning from the lowest level of complexity, namely a flat terrain, and progresses to the case of a real built environment. Each model will be validated against existing literature and wind tunnel tests when available.

1. Complexity 0: Preliminary investigations on an empty domain

2. Complexity 1: Regularly arranged array of blocks similar to that used in wind tunnels. Empirical formulas for estimating roughness parameters based on density of obstacles are compared with current CFD results.

3. Complexity 2: The effect of inhomogeneous roughness, i.e. multiple roughness patches upstream of a site, is evaluated using three dimensional CFD simulations and results are compared against existing wind speed models. The simulations are carried out in a Virtual Boundary Layer Wind Tunnel (V-BLWT) by duplicating all roughness features used namely spires, blocks and barrier. Sixty nine cases tested by Wang & Stathopoulos

(2007*a*) in wind tunnel are simulated and results are compared with wind speed models.

4. Complexity 3: The flow characteristics in a semi-idealized urban environment is studied by conducting model scale simulations and results are compared with existing Boundary Layer Wind Tunnel (BLWT) test data

5. Complexity 4: Simulations over a real urban environment are conducted in an area in Downtown Miami. There is usually a lack of validation data for such kind of simulations thus the only qualitative discussion of results is made.

Finally Artificial Neural Network (ANN) are considered as an alternative to setup roughness features in an actual BLWT for a required wind profiles at the turn table. A neural network is trained with half of the dataset obtained from Rowan Williams Davies and Irwin Incorporation (RWDI), and then the model is tested for prediction ability on the rest of the dataset.

4.1 Complexity 0: Empty domain

CFD enjoys a wide spread use in the wind engineering community however many parameters that influence the simulation results are not well understood (Franke & Hirsch 2004). A rather trivial case that is commonly used to demonstrate disparity between simulation results of different CFD software is the case of an empty domain. Since there are no obstacles, the characteristics of the wind should be maintained along the whole length of the domain. It may seem at first that simulation on an empty terrain is trivial but is quite challenging. The problem stems from difficulty of achieving horizontally homogeneous flow unless proper boundary conditions are used (Blocken et al. 2007, Hargreeves & Wright 2007, Richards & Hoxey 1993). This investigation also helps to outline the steps involved in a typical Computational wind engineering (CWE) simulation.

4.1.1 Computational domain

The computational domain used for this experiment is the same as the one used by Hargreeves & Wright (2007). It has dimensions of 5000m X 100m X 500m. The domain is meshed with 500x50x5 cells and the mesh is expanded in the the vertical direction in such a way that the the size of nearest cell to the ground is 1m. This satisfied the $Y_p > K_s$ criterion for roughness conditions of $z_0 = 0.01m$. A reference wind speed of 10m/s at a height of 6m is used. Different boundary conditions at the ground surface, inlet and top of the domain are tested until

a horizontally homogeneous flow is obtained using k-epsilon turbulence model. Hargreeves & Wright used commercial CFD software CFX and Fluent to demonstrate the problem of ABL simulations on an empty fetch. Boundary conditions are modified progressively through user defined functions (UDF) until a horizontally homogeneous flow is obtained for all flow quantities (U,k and ϵ). Here similar procedure is followed to check if the software developed in this work can overcome the problem.

4.1.2 Boundary conditions

Boundary conditions are very important for any CFD simulation because they are cutoff planes that divide the area we are interested in simulating from that we do not want to include in the simulation. In other words they are used to incorporate the influence of the surrounding to our model. The type of boundary condition also affects the placement of the cutoff planes relative to the central region where obstacles are placed. For example, it is well known that use of symmetry boundary condition at the top and sides of the domain introduces artificial accelerations unless blockage ratio is kept to a minimum. The computational domain is usually divided into three regions (Blocken et al. 2007), namely, the central region where the obstacle is modeled as best as possible, and the upstream and downstream regions where the effect of obstacles is modeled by regular roughness elements. The other issue concerns consistency of boundary conditions with the wind profiles specified at the inlet and the turbulence model (O'Sullivan et al. 2011, Richards & Hoxey 1993).

At the inlet of the computational domain fully developed equilibrium velocity and turbulence intensity profiles are applied. The inlet profiles should be consistent with the upstream surface roughness characteristics (Miller & Davenport 1998, Wieringa 1993), and they should be maintained within the computational domain until the flow reaches the face of the test building. This is very important for determination of wind load on buildings, that will be significantly different if, for instance, a uniform velocity profile is used instead of logarithmic profile. A peculiar problem in ABL simulations is that maintaining horizontal homogeneity is very difficult to achieve with current breed of CFD software. Richards & Hoxey (1993) have investigated this problem thoroughly and suggested boundary conditions (Eqs. 4.1-4.3) to be specified at the inlet that will ensure horizontal homogeneity for the standard k-epsilon turbulence model. Their formulas have been used by the wind engineering community for many years. However, it is not enough to specify just inlet conditions to get a stream-wise homogeneous flow. The wall functions used at the surface should be compatible with the roughness of the upstream fetch outside the domain. Otherwise an internal boundary layer will develop

starting from the inlet at which the roughness change occurs.

$$u = \frac{u*}{\kappa} \ln \frac{z + z_0}{z_0} \tag{4.1}$$

$$k = \frac{u*^2}{\sqrt{C_\mu}} \tag{4.2}$$

$$\epsilon = \frac{u*^3}{\kappa(z + z_0)} \tag{4.3}$$

Richards & Hoxey found that the transport equations for the standard k-epsilon model can be satisfied with above relations only when a different σ_ϵ is used than the standard value of 1.3. The formula for calculating σ_ϵ given vonKarman constant is

$$\sigma_\epsilon = \frac{\kappa^2}{(C_{\epsilon 2} - C_{\epsilon 1})\sqrt{C_\mu}} \tag{4.4}$$

Nikurdase's modified log-law equations 4.5-4.6 are used as rough wall functions in many CFD code. As described in Blocken et al. (2007), the first cell's center should be placed higher than the equivalent sand grain roughness height i.e. $Y_p > K_s$. This constraint is in conflict with using a fine mesh close to walls where high velocity gradients are present.

$$u^+ = \frac{1}{\kappa} \ln(Ey^+) - \Delta B \tag{4.5}$$

$$\Delta B = \frac{1}{\kappa} \ln(1 + C_{ks} K_s^+) \tag{4.6}$$

For a horizontally homogeneous flow, i.e. one in which same velocity profile is maintained, the wall function should approximately yield the same profile as the inlet profile as specified by Richards and Hoxey.

$$\begin{aligned} u^+ &= \tfrac{1}{\kappa} \ln \tfrac{z+z_0}{z_0} && \text{, Inlet} \\ u^+ &= \tfrac{1}{\kappa} \ln(\tfrac{Ey^+}{1+C_{ks}K_s^+}) && \text{, Wall} \end{aligned} \tag{4.7}$$

Equating the above two equations we get relations between K_s and z_0

$$\frac{z + z_0}{z_0} = \frac{Ey^+}{1 + C_{ks} K_s^+}$$

$$\frac{z}{z_0} = \frac{Ey}{C_{ks}K_s}$$

$$K_s = \frac{Ez_0}{C_{ks}}$$

$$K_s \sim 20z_0 \tag{4.8}$$

At the sides and top of the domain, a symmetry boundary condition that prevents inflow or outflow is usually applied. This boundary conditions results in a parallel flow at the boundary which could sometimes lead to artificial acceleration if enough space is not provided between the obstacles and the boundary plane. To solve this problem the domain is sized in such a way that blockage ratio is set at a certain limit below which the effect is minimal. Another solution is to replace the boundary condition with one that allows flow outwards through the boundary (Franke & Hirsch 2004).

The common use of symmetry boundary condition at the top of the boundary is rather unfortunate since it ignores the contribution of geo-strophic wind in driving the ABL flow. Many researchers have noted that use of symmetry boundary condition results in stream-wise gradients of velocity profile. However there are many reasons why symmetry is assumed in many wind engineering problems. The major physical reason is that log layer in the ABL extends only up to a certain depth above which the gradient of velocity becomes zero. Also it is not known a priori what the values would be set at the top if symmetry boundary condition is not used. A shear stress boundary condition ($\tau = \rho u^2$) should be applied at the top to get a homogeneous (non-decaying) profile (Hargreeves & Wright 2007, Richards & Hoxey 1993). Another approach used by Blocken et al. (2007) is to apply Dirichlet boundary condition for velocity and turbulence quantities at the top.

4.1.3 Simulation for different cases

Simulations are conducted by varying the boundary conditions at the ground, inlet and top of the computational domain, and the results are examined with regard to maintaining a horizontally homogeneous flow. The four different test cases considered are briefly described in the following sections.

Case 1 - Incompatible wall roughness

The first case applies the Richard and Hoxey boundary conditions at the inlet but assumes a smooth ground surface thereby creating a situation where the surface roughness exhibits a sud-

den change at the inlet. Due to this incompatibility, stream wise gradients are observed in the profiles of U, k and epsilon as shown in Fig.4.1. Close to the ground, both the velocity profile and turbulence dissipation show large changes as one goes downstream; while the profiles towards the top remain somewhat constant. On the other hand, the turbulent kinetic energy shows variations throughout. The difficult of maintaining the turbulent kinetic energy along the fetch has been noted by Richards & Hoxey especially on the first cell close to the ground where many CFD software show peak values.

Case 2 - Compatible wall roughness

When surface roughness conditions compatible with the inlet profiles are applied, both velocity and turbulence intensity profiles are maintained throughout the domain as shown in Fig.4.2. The sand grain roughness used for the simulation is determined according to the relation $K_s = 20z_0 = 0.2$ and $C_{ks} = 0.5$. However the calculated turbulent kinetic energy profile still shows variations from the expected constant vertical profile.

Case 3 - Fixed U,k and ε at the top

From the previous simulations, we observe that the flow quantities at the top show some variations due to the imposed symmetry boundary condition. Blocken et al. has suggested using Dirichlet boundary condition to make sure that the flow quantities remain the same at least at the top of the boundary. The result for this case is shown in Fig. 4.3. While velocity and turbulence dissipation show an almost perfect fit from start to finish of the fetch, the turbulent kinetic energy profile show a rather distorted profile compared to the previous cases. Other simulations have been carried out which confirm the same observation.

Case 4 - Uniform k and epsilon at the inlet

It is customary to specify constant values of k and epsilon at the inlet for convenience. The assumption is correct for k but not for epsilon. The simulation result for this case shows a developing epsilon profile along the fetch, before reaching more or less the same values at the outlet, as shown in Fig.4.4.

So far we have managed to get homogeneous velocity and turbulent dissipation profiles. To get a homogeneous turbulent kinetic energy profile, further modifications to wall functions are necessary. Most commercial CFD code do not usually offer wall functions that can maintain k profile this way, however we note that it is possible to implement modifications to wall functions to achieve horizontal homogeneity for k as described in Hargreeves & Wright (2007).

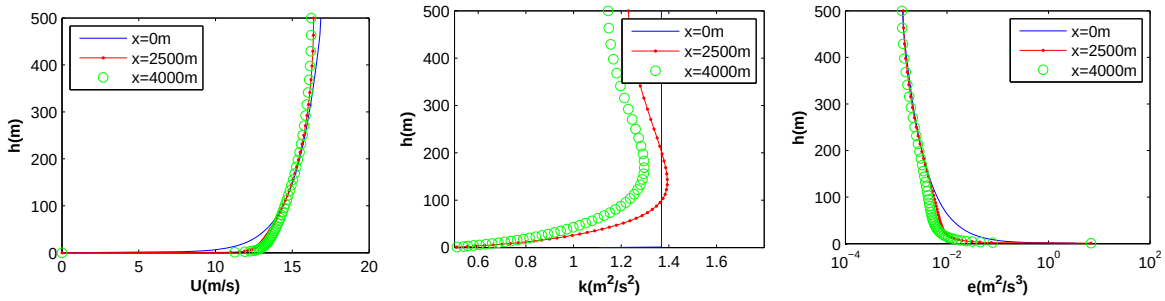


Figure 4.1: Profiles of horizontal velocity, turbulent kinetic energy and dissipation for case-1

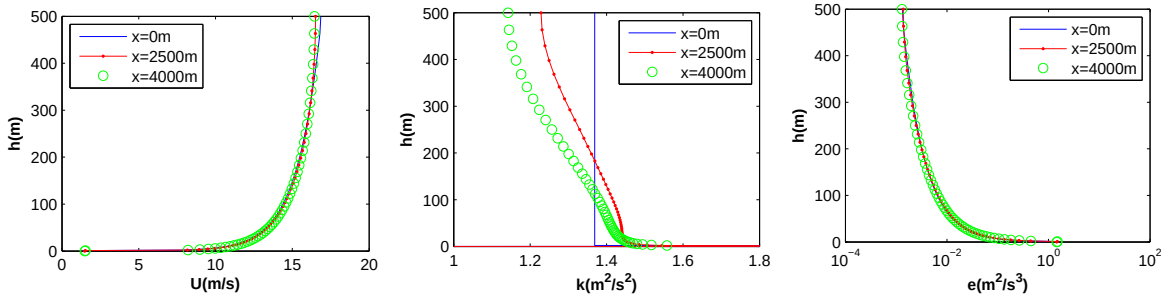


Figure 4.2: Profiles of horizontal velocity, turbulent kinetic energy and dissipation for case-2

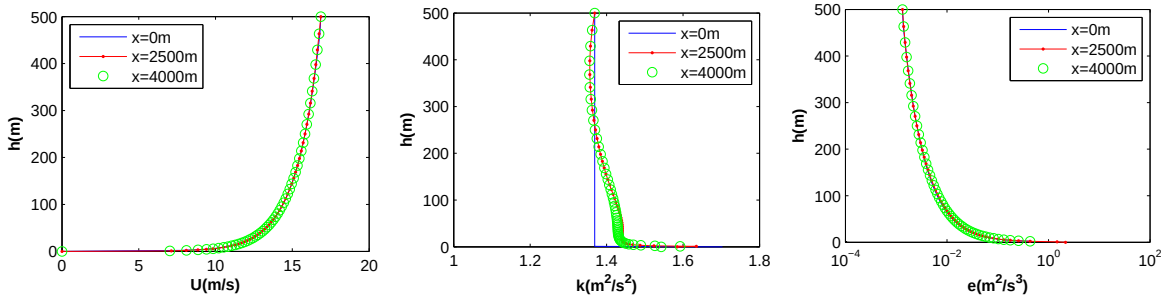


Figure 4.3: Profiles of horizontal velocity, turbulent kinetic energy and dissipation for case-3

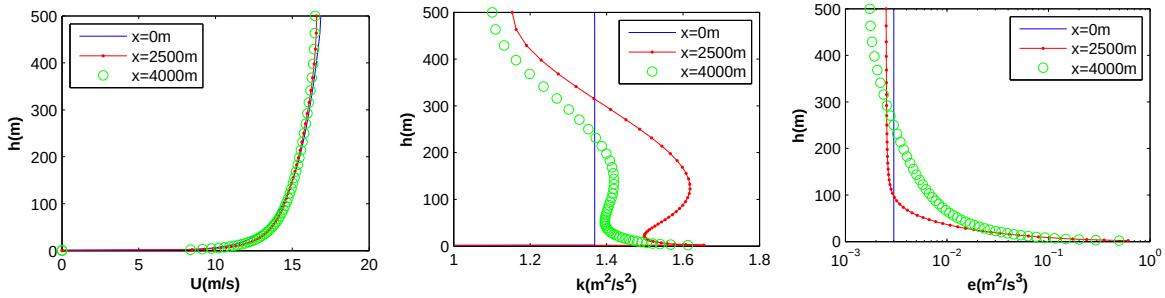


Figure 4.4: Profiles of horizontal velocity, turbulent kinetic energy and dissipation for case-4

4.2 Complexity 1: Homogeneous roughness evaluation

The next level of complexity concerns uniform (homogeneous) roughness due to array of obstacles. Simulations are carried out on regular and staggered array of blocks for wind coming from different angles. The arrangements of the roughness blocks considered are all symmetric, which allows for a reduction of computational domain to a much smaller section of one or two rows as shown in Fig.4.5. The test setups and results obtained are described in the following sections.

4.2.1 Test setup

The test setup used in this study is similar to that used by MacDonald et al. (1998). Regular or staggered arrays of cubes are exposed to wind coming from different directions, and velocity profiles are recorded at different sections behind the obstacles. Wind speed profiles show variations in the transverse direction because some of the locations are sheltered by the blocks while others lie in the gap between the blocks. Therefore multiple measurement points are considered in the transverse direction, and results are averaged to get a representative velocity profile for that section. This approximation is acceptable for regular arrays of cubes but it may be inaccurate for irregular array of obstacles. Close to the ground and right behind an obstacle, negative velocity profiles can develop due to re-circulation, while at locations close to center line of gap the velocity is positive. The averaging operation removes these variations and positive velocity values are observed also at heights where recirculation happens.

The area density ratio for the configurations considered can be approximated by the following formula.

$$\lambda = \frac{1}{(1 + \frac{S}{H})^2} \tag{4.9}$$

For example, a spacing $S = 1.5H$ between blocks gives $\lambda = 0.16$. The Lettau (1969) model predicts a roughness length $z_0 = 0.5\lambda H = 0.08H$ The test is conducted for various configurations of obstacles with different spacing, regular and staggered arrangement, rectangular obstacle shapes, and different wind angle of attacks as shown in Figs.4.5-4.6. Symmetry of arrangement of obstacles is exploited to reduce computational domain. Then models are prepared for six area density ratios (0.05, 0.11, 0.16, 0.2, 0.33, and 0.5) for every configuration of obstacles considered. A series of 32 blocks of height 20 m are arranged in different ways, and a steady state solution of the flow problem is sought.

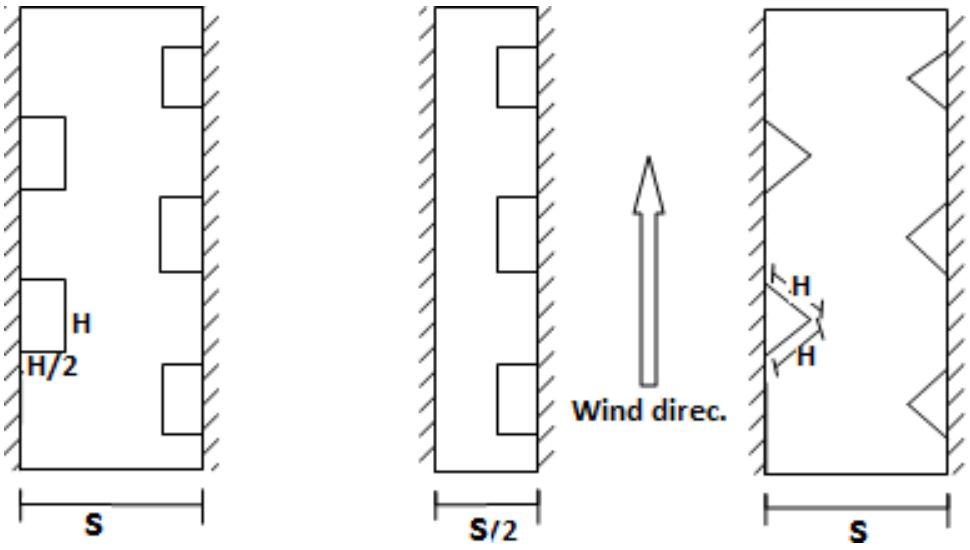


Figure 4.5: Plan of three symmetric configurations: Staggered arrays(left), regular arrays (middle) and 45^0 wind attack on uniform array (right)

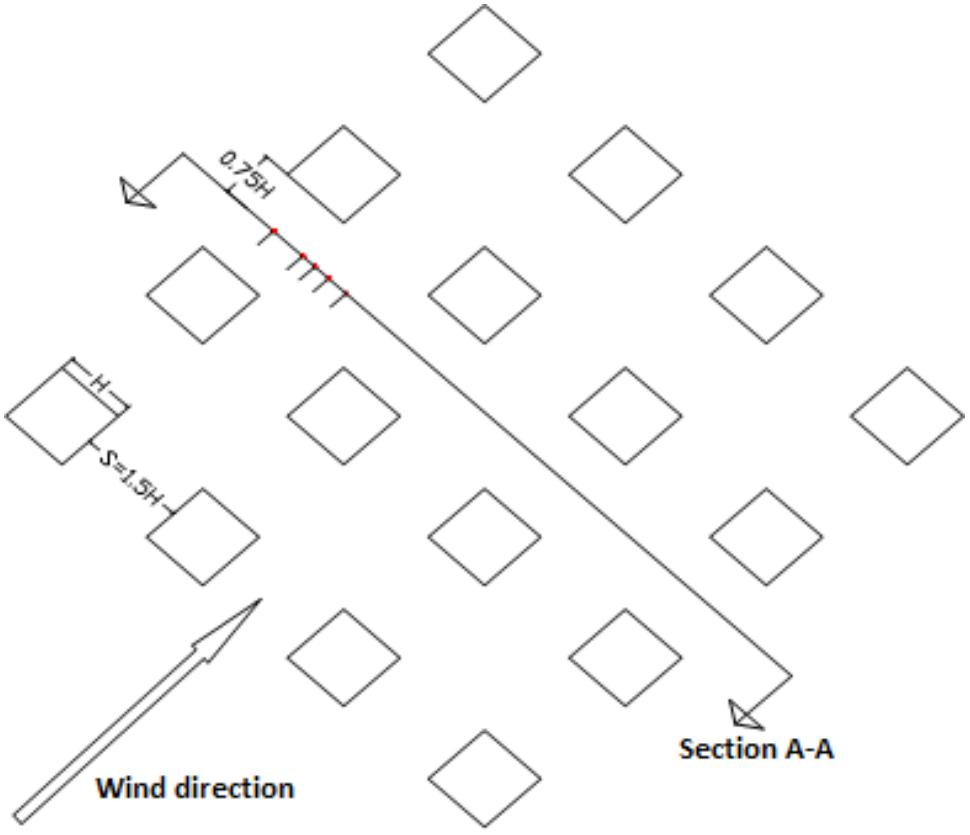


Figure 4.6: Plan of regular array of cubes with height H and spacing 1.5H also showing location of probes

4.2.2 Analysis

The objective is to calculate roughness length and displacement height from velocity and turbulence intensity profiles obtained from CFD simulations. For this purpose average of five velocity profiles measurements at locations shown in Fig.4.6 is considered instead of a single profile. First the displacement height is determined iteratively using Eq.(2.41) from Lo (1990). The value of d obtained using this method is usually satisfactory, however value of z_0 is very sensitive to the selected reference heights because it is based on measurements at two heights in the inner layer.

As the fetch length becomes larger, the internal boundary layer grows until it becomes equal or greater than the height of the computational domain. This stabilization of flow is usually achieved earlier than the last row of blocks. The average velocity within the viscous layer increases with fetch length, while the velocity in the inner and outer layers decrease. The roughness length and displacement height obtained from averaged velocity profile measured at the last row of a series of blocks is shown in Fig.4.8, along with predictions from different roughness models. The McDonald roughness model is tested in two ways in which the displacement height is calculated differently. The first method determines roughness length from displacement height calculated using Lo (1990)'s equation ($McDonald1$). The second method uses d calculated from Theurer (1993)'s equation ($McDonald2$). The results from the analysis are briefly summarized as follows. The $McDonald1$ method gives the best fit to the CFD calculated result as shown in Fig. 4.9. The Theurer model also shows good fit up to area density ratio of 20%. Lettau's and Counihan's models hugely underestimate the roughness for area density below 20% and overestimate it for area density larger than 20%. The staggered obstacle arrays and regular arrays with 45 degree wind angle of attack resulted in higher roughness compared to the simple case of regular arrays as shown in Fig. 4.10. The staggered placement of obstacles increase roughness due to relatively larger exposure of faces of the cubes to on coming wind. A regular array of cubic obstacles exposed to a 45 degree on coming wind is equivalent to a staggered array of triangular obstacles as shown in Fig.4.5. We can also observe that the deviation of Lettau's and Counihan's models from CFD model is less pronounced on staggered array of blocks compared to the regular arrangement. This is a reasonable observation because of much less wake interference in staggered arrangment that has large spacing (small λ), in which case the flow becomes effectively isolated for each block. However Lettau's and Counihan's model still show significant deviations from the CFD model, which is partly explained by the larger drag imposed by the cubic obstacles (C_D = 1.2).

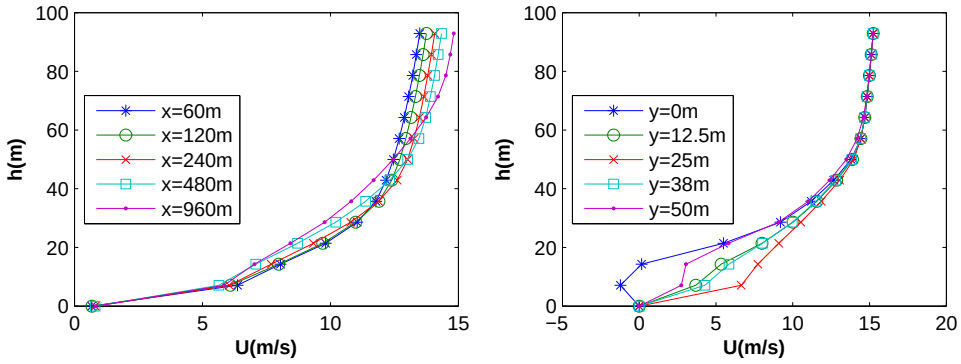


Figure 4.7: Spatial variation of velocity profiles: longituidinal (left) and transverse(right)

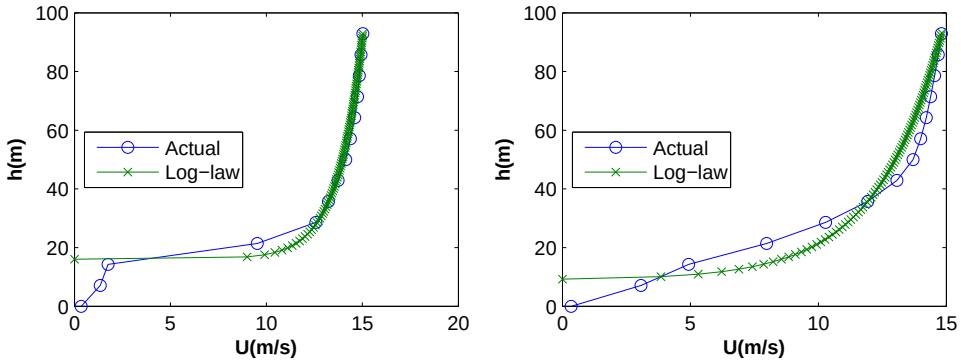


Figure 4.8: Sample measured and logarithmic fitted velocity profiles

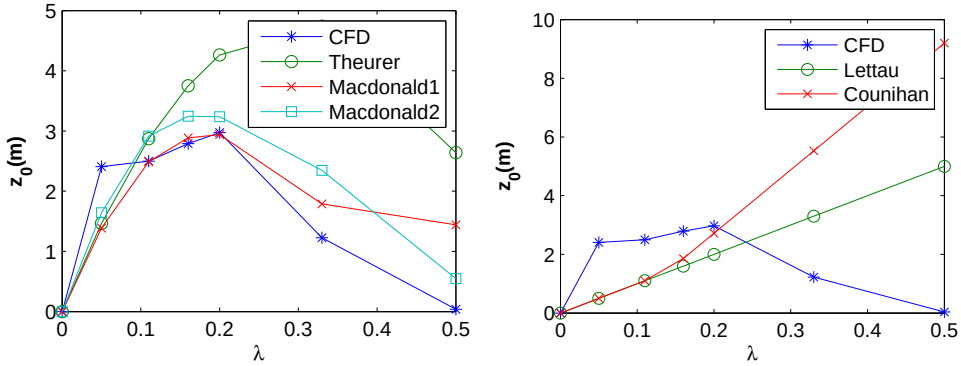


Figure 4.9: Comparison of CFD with different roughness models

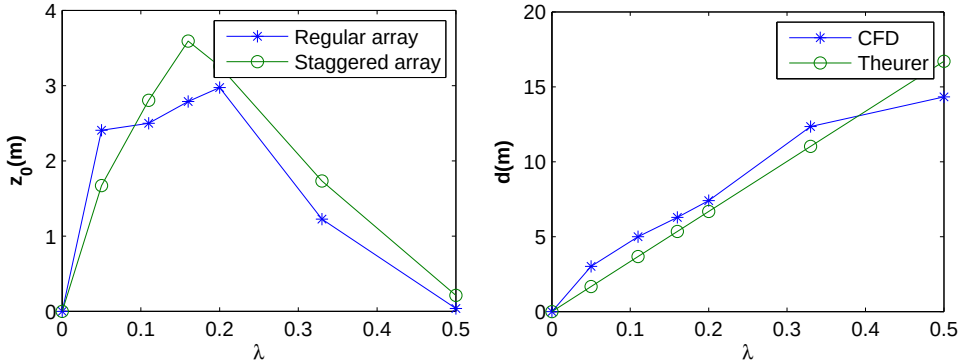


Figure 4.10: Effect of staggered placement on z_0 (left) and comparison of CFD and Theurer model for d (right)

4.3 Complexity 2: Inhomogeneous roughness evaluation

Proper evaluation of wind speed and turbulence intensity profiles is important for correct determination of wind loads on buildings. Both profiles are sensitive to upwind roughness changes especially close to the building. In design of structures usually a level terrain is first assumed, and then departures from this caused by topographic changes and surface roughness inhomogeneities are assessed. The significance of the effect of inhomogeneous roughness within the pertinent fetch was overlooked in building codes and standards before the Engineering Science Data Unit (ESDU) model is introduced.The earliest investigation of this effect was carried out by Deaves (1981), Deaves & Harris (1978) using CFD simulations over single changes of roughness. The result of this work is incorporated in the ESDU model which is recommended methodology in many building codes and standards for the case of multiple roughness changes close to building.

Recently Wang & Stathopoulos (2007*a*) put forward wind speed and turbulence intensity models that improved upon the ESDU model. Their model, henceforth called Wang and Sthatopoulos Model (WS), is validated with wind tunnel experiments and simplified 2D CFD simulations over multiple roughness changes. The motivation for this work is that the ESDU model can sometimes overestimate wind speed by as much as 20% , which means a 40% increase in wind load. In this work the performance of three dimensional CFD simulations for predicting wind speed and turbulence intensity profile will be compared with the above mentioned models.

Inhomogeneous roughness within the pertinent fetch length of the building site affects both wind speed and turbulence intensity profiles. The boundary layer for multiple roughness changes is stratified with an upper boundary layer up to the gradient height G and as many inner boundary layers as there are patches, with a possible transitional layer in between. The case of a single roughness change with a transition layer is shown in Fig. 4.11. Three distinct regions can be seen namely the outer layer, the transition layer and Internal Boundary Layer (IBL).
Deaves & Harris divide the flow horizontally in to three regions.

- $x < 0$**:** The upstream region where flow is characterized solely by roughness conditions there.

- $0 < x < F$**:** The region of influence of the roughness change where IBL is still growing on the new roughness z_0. The friction velocity $U_*(x)$ is a function of distance from transition point.

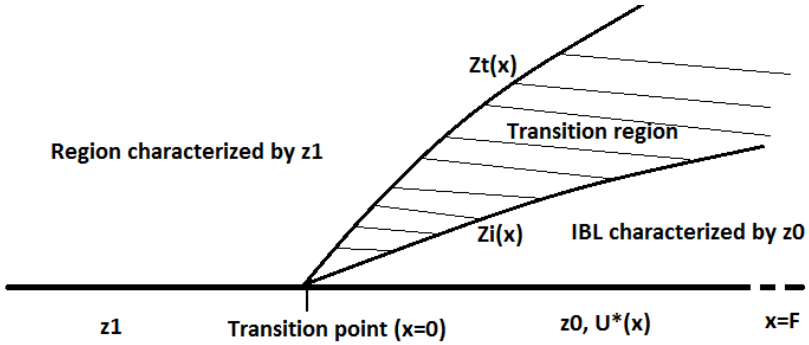


Figure 4.11: Schematics of the growth of internal boundary layer for single roughness change

- $x > F$**:** IBL has fully developed and is in equilibrium with new parameters U_* and z_0.

Similarly at any position in $0 < x < F$, the flow can be divided vertically in to three regions.

- $0 < z < z_i(x)$**:** The flow is in equilibrium with the new surface so any of the homogeneous wind speed models can be used to determine using the new surface roughness parameters.

- $z_i(x) < z < z_t(x)$**:** The flow here is neither in equilibrium with the new roughness nor does it retain upstream character. Velocity profiles should be smoothly interpolated between $z < z_i$ and $z > z_t$.

- $z > z_t(x)$**:** The upstream flow is unmodified as the disturbance has not reached there yet.

In the following sections, a review of different models for both homogeneous and inhomogeneous terrain is given.

4.3.1 Homogeneous roughness wind speed models

4.3.1.1 Roughness estimation

An estimate for roughness length in a homogeneous terrain can be obtained from Davenport roughness classifications. If the obstacles are big with measurable dimensions e.g.buildings, a better estimate can be found using empirical formulas as discussed in 2.7.1. A brief overview of simple formulas to approximate roughness parameters follows. Given mean height of obstacles: buildings,bridges, crops,forests etc., roughness length is estimated as

$$\frac{z_0}{H} = c_1 \tag{4.10}$$

where $c_1 = 0.1$ gives good results in many situations, however it is established that z_0 is not constant. Similarly an approximation for the zero-plane displacement height is

$$\frac{d}{H} = c_2 \tag{4.11}$$

where c_2=0.75. Lettau (1969) provided an empirical formula to determine z_0 from frontal area density ratio of obstacles

$$\frac{z_0}{H} = 0.5\lambda_f \tag{4.12}$$

This simple approximation fails to give good results for moderately dense regions. MacDonald et al. (1998) suggested a model that tackles limitations of Lettaus and other similar empirical models. The MacDonald model improvements include a non-linear decrease of z_0 at high area density ratio , and different obstacle shapes and layouts.

4.3.1.2 Models

The log-law and power law wind speed models have been discussed in section 2.6.2.1 and 2.6.2.2, but we repeat the relevant equations here for convenience.

$$\frac{U(z)}{u*} = \frac{1}{\kappa}\ln(\frac{z-d}{z_0})$$

$$U = U_{ref}(\frac{z-d}{z_{ref}})^{\alpha} \tag{4.13}$$

$$I_u(z) = c(\frac{z}{z_{ref}})^{-d}$$

Homogeneous roughness wind speed models from ESDU 82026 ,that are based on the work of Deaves & Harris (1978), incorporate the effect of Coriolis force. The simplified homogeneous model (for $z \leq 300m$) has an additional term over the log-law model that relates with the gradient height G.

$$\frac{U(z)}{u_*} = \frac{1}{\kappa}(\ln(\frac{z}{z_0}) + \frac{34.5 f_c z}{u_*}) \tag{4.14}$$

And the corresponding model for turbulence intensity is given as follows

$$I_u(z) = \frac{u(z)}{U(z)} = \frac{u(z)}{u_*}\frac{u_*}{U(z)} \tag{4.15}$$

$$\frac{u(z)}{u_*} = \frac{7.5\eta[0.538 + 0.09\ln(\frac{z}{z_n})]\eta^1 6}{1 + 0.156\ln(\frac{u_*}{f_c z_0})} \tag{4.16}$$

$$\eta = 1 - \frac{6 f_c z_u}{u_*} \tag{4.17}$$

4.3.2 The ESDU model

The set of equations provided in ESDU-82026, ESDU-84030, for determining wind speed and turbulence intensity respectively for multiple roughness changes, are based on numerical work of Deaves (1981). A comparison of the Deaves model with the log-law and power-law for heterogeneous terrain can be found in Nicholas (1997). The ESDU model is now adopted in several building codes and standards such as American Society of Civil Engineers - 7 (ASCE7) and National Building Code of Canada (NBCC). Deaves conducted CFD simulations using simple eddy-viscosity (mixing length) models for turbulence closure. Contemporary CFD studies dropped the second horizontal derivatives rendering the Navier-stokes equations parabolic and solutions were carried out by 'marching'. Deaves solved the full elliptic set of equations in which Coriolis force is also included using an approximation that allows the equations to remain two dimensional.

4.3.2.1 Wind speed model (ESDU 82026)

A set of equations for U and I_u are proposed for both homogeneous and inhomogeneous terrain. For inhomogeneous terrain with n roughness patches the following set of equations are provided, however ESDU recommends the use of the equations for up to a maximum of three patches. This is partially due to lack of sufficient experimental validation for four or more patches.

The velocity profile within each IBL , $g_n <= z <= g_{n-1}$, can be calculated using the following equation

$$U(z) = K_{x2} K_{x3} K_{x4} \cdots K_{xn} U_n(z) \tag{4.18}$$

The coefficient K is a terrain dependent coefficient calculated differently for smooth to rough (S-R) and rough to smooth transitions (R-S) as follows

$$K_{xi} = \begin{cases} 1 + 0.67 R_i^{0.85} f_{S-R} \\ 1 - 0.41 R_i f_{R-S} \end{cases} \tag{4.19}$$

$$R_i = \frac{[\ln(\frac{z_{0,i-1}}{z_{0,i}})]}{(\frac{u_*}{f u_*})_i^{\beta}} \tag{4.20}$$

$$\beta = \begin{cases} 0.23, & \text{for S-R} \\ 0.14, & \text{for R-S} \end{cases} \tag{4.21}$$

$$f_{S-R} = \begin{cases} 0.1143E^2 - 1.372E + 4.087 & \text{if } E \leq 5.5 \\ 0 & \text{if } E > 5.5 \end{cases} \tag{4.22}$$

$$f_{R-S} = \begin{cases} 0.0192E^2 - 0.550E + 2.477 & \text{if } E \leq 5.6 \\ 0 & \text{if } E > 5.6 \end{cases} \tag{4.23}$$

$$E = \log_{10} X, \text{ where } X = X_2 + X_3 + \cdots + X_i \tag{4.24}$$

Then the IBL depths $g_i(x)$ can be determined by continuity requirement at each transition. Two profiles can be combined into one continuous profile using the following equation

$$g_i(x) = \exp\left(\frac{K_{xi}\left(\frac{u_{*,i}}{u_{*,i-1}}\right)\ln(z_{0,i}) - \ln(z_{0,i-1})}{K_{xi}\left(\frac{u_{*,i}}{u_{*,i-1}}\right) - 1}\right) \tag{4.25}$$

4.3.2.2 Turbulence intensity model (ESDU 84030)

Here equations are provided for determining turbulence intensity profile for inhomogeneous roughness.

$$I_u(x) = \frac{u(x)}{u}\frac{u}{u_*}\frac{u_*}{U(z)}\frac{U(z)}{U(z,x)} \tag{4.26}$$

$$\frac{u - u(x)}{u - u'} = \begin{cases} \cos^2\left[\frac{\pi^2}{4}\left(\frac{\zeta - 0.25}{0.8}\right)\right], & \text{for } 0.25 \leq \zeta \leq 1.85, S - R \\ \cos^2\left[\frac{\pi^2}{4}\left(\frac{\zeta - 0.1}{0.8}\right)\right], & \text{for } 0.10 \leq \zeta \leq 1.70, R - S \end{cases} \tag{4.27}$$

where u', u and $u(x)$ are the upwind, far-downwind and local values of fluctuating velocities respectively.

$$\zeta = \frac{\ln(x) - \ln(g')}{\ln(g) - \ln(g')} \tag{4.28}$$

$$\frac{g'}{z_{0,(n,n-1)}} = \left(\frac{z}{10 z_{0,(n,n-1)}}\right)^{5/3} \tag{4.29}$$

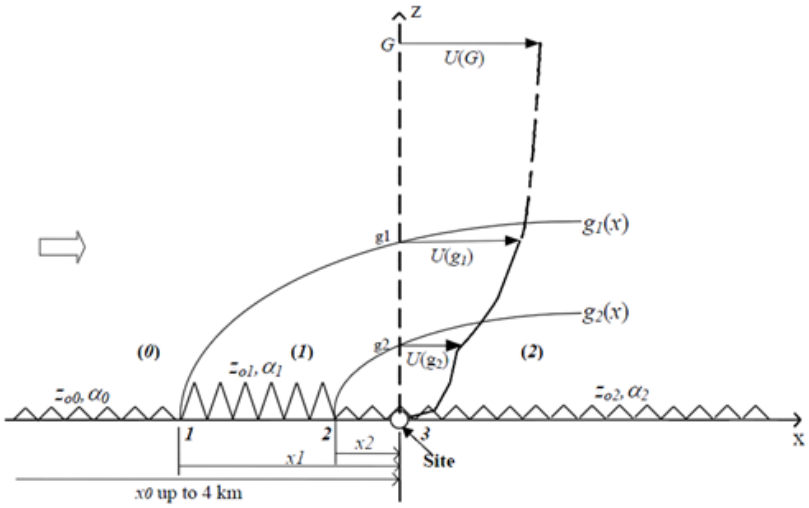


Figure 4.12: Schematics of change in velocity profile for three roughness patches.(Wang & Stathopoulos 2007*b*)

$$\frac{g}{z_{0,n}} = \begin{cases} (\frac{z}{0.36 z_{0,n}})^{4/3}, & \text{for } S - R \\ (\frac{z_{0,n-1}}{z_{0,n}})(\frac{z}{0.07 z_{0,n}}), & \text{for } R - S \end{cases} \tag{4.30}$$

4.3.3 The WS model

4.3.3.1 Wind speed model

The WS model assumes the stratification of the IBL for each patch follows the power law model. Unlike the ESDU model , each segment of the wind profile has a wind speed curve dictated by the power law index of the corresponding patch as shown in Fig. 4.12. The IBL growth is assumed to follow a power law with coefficient 0.8.

$$\begin{aligned} g_0(x) &= G \\ g_n(x) &= 0.5 z_{0,(n,n-1)}^{0.2} x_n^{0.8} \text{ , where } z_{0,(n,n-1)} = \max(z_{0,n}, z_{0,n-1}) \end{aligned} \tag{4.31}$$

The wind speed model for each segment of the profile is

$$U(z) = U(g_n(x))(\frac{z}{z_{g_n}})^{\alpha_n}, \text{ where } g_{n+1} \le z \le g_n \tag{4.32}$$

4.3.3.2 Turbulence intensity model

The corresponding model for turbulence intensity requires that the IBL be subdivided into a transitional and equilibrium sublayer. The total IBL depth g(x), including the transitional sublayer, is still to follow a 0.8 power law but the equilibrium sub-layer depth is assumed to follow a 0.72 and 0.4 power law for smooth-rough and rough-smooth transitions respectively.

$$g'_n(x) = \begin{cases} 0.5 z_{0,(n,n-1)}^{0.2} x_n^{0.72}, & \text{for } S - R \\ 0.5 z_{0,(n,n-1)}^{0.2} x_n^{0.40}, & \text{for } R - S \end{cases} \tag{4.33}$$

The turbulence intensity profiles are then obtained using the following equations based on inverse power law.

$$I_u(z) = \begin{cases} I_{un}(10)(\frac{z}{10})^{-0.4}, & \text{for } g_{n+1} \leq Z \leq g_n \\ I_u(g_n)(\frac{z-g_n}{g'_n-g_n})(I_u(g'_n) - I_u(g_n)), & \text{for } g_n \leq Z \leq g'_n \end{cases} \tag{4.34}$$

Letchford et al. (2001) noted that in general turbulence intensity requires shorter fetch length to forget the upwind patch influence than wind speed.

4.3.4 Comparison of WS and ESDU models

The WS model discussed in the previous section was verified using boundary layer wind tunnel tests on sixty-nine cases of multiple roughness patch detailed in Table 4.1. A roughness patch is characterized by three parameters namely length, distance to building site, and roughness length or (l, x, z_0). The letters c, s and u represent open, sub-urban and urban roughness patches respectively. The number following the letters represent the length of the patch in meters. For the patch upstream of all other patches, a relatively long fetch length of 2km is assumed. The basic single patch roughness cases are case-1 for open terrain, case-8 for sub-urban and case-55 for urban. First a simple program is written to compare the performance of WS and ESDU models using the formulas discussed in the previous sections. Sample results for some of the cases is given in Fig. 4.13. For the open terrain patch there is a good agreement between the two models, but a significant difference is observed for the sub-urban and urban patches. The ESDU model gives conservative results which can overestimate the velocity by as much as 20% as is confirmed by Wang & Stathopoulos, which was one of the motivations for the development of their model.

Table 4.1: Multiple roughness patch cases considered

1	c 2000	2	c 2000 u 125 s 250 u 125
3	c 2000 s 1000	4	s 2000 c 250 s 125 c 125
5	s 2000 c 125 s 125	6	s 2000 u 500
7	c 2000 u 125 s 125 u 125 s 125	8	s 2000
9	s 2000 c 125	10	s 2000 c 125 s 250 c 125
11	s 2000 c 250 s 250	12	s 2000 c 125 s 125 c 125 s 525
13	c 2000 s 125 u 125 s 125 u 125	14	c 2000 s 125
15	s 2000 c 250	16	s 2000 c 375 s 125
17	s 2000 c 125 s 250	18	s 2000 u 500 s 500
19	c 2000 s 125 c 125 s 125	20	c 2000 s 250
21	s 2000 c 375	22	s 2000 c 250 s 125
23	s 2000 u 125 s 125 u 125 s 125	24	s 2000 c 500 s 500
25	c 2000 s 125 u 250 s 125	26	c 2000 s 375
27	s 2000 c 500	28	s 2000 u 250 s 125
29	s 2000 u 125 s 125 u 125	30	c 2000 u 125
31	c 2000 u 125 c 125 u 125	32	c 2000 s 500
33	s 2000 c 750	34	s 2000 c 125 s 125 c 125 s 125
35	s 2000 u 250	36	c 2000 u 250
37	c 2000 u 125 c 250 u 125	38	c 2000 s 750
39	s 2000 c 125 s 125 c 250	40	c 2000 u 375
41	u 2000 c 375	42	u 2000 c 250 u 125
43	c 2000 s 1500	44	s 2000 c 500 s 2750
45	c 2000 u 500	46	u 2000 c 500
47	u 2000 c 125 u 125 c 125 u 125	48	c 2000 s 2250
49	s 2000 c 125 s 125 c 125 s 125 c 125 s 125 c 125 s 125	50	c 2000 u 1000
51	u 2000 c 1000	52	u 2000 c 250 u 250
53	s 2000 c 1500	54	s 2000 c 500 s 1000
55	u 2000	56	u 2000 c 125 s 125 c 125 s 125
57	u 2000 c 500 u 500	58	s 2000 c 2250
59	s 2000 c 125 s 125 c 125 s 125 c 125 s 125 c 125 s 1125	60	u 2000 c 125
61	u 2000 s 500	62	c 2000 u 375 c 250
63	s 2000 c 250 s 250 c 250 s 250	64	u 2000 c 1500
65	u 2000 c 250	66	u 2000 c 375 u 125
67	c 2000 u 1500	68	u 2000 c 2000
69	c 2000 u 2000		

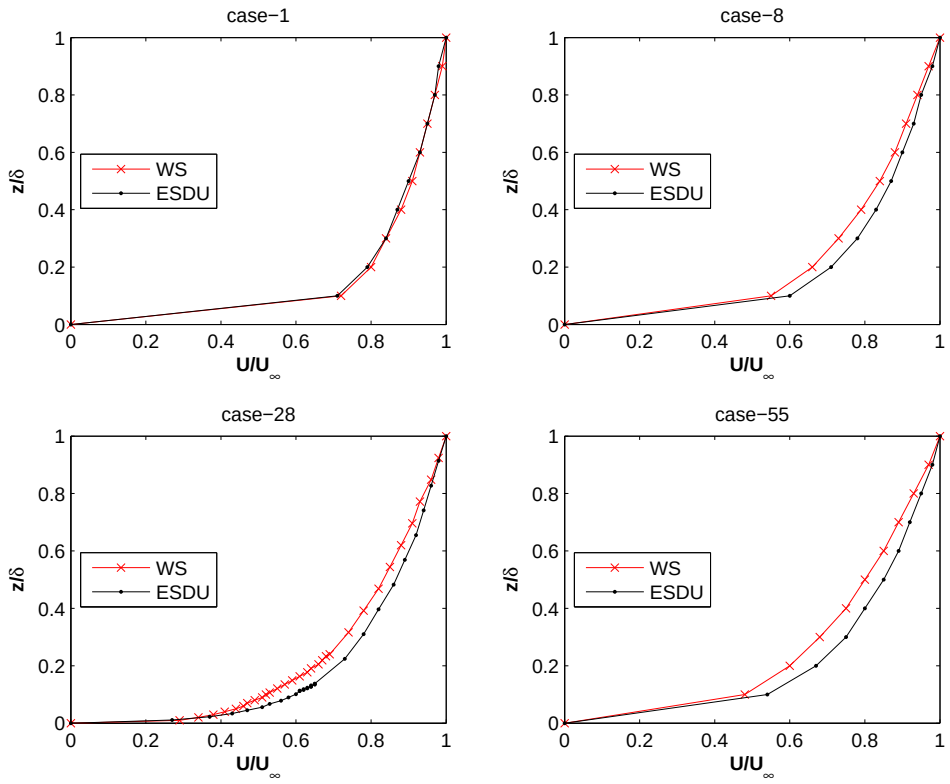


Figure 4.13: Comparison of WS and ESDU models on selected cases

4.3.5 Three dimensional CFD simulations

The conventional method of using wall functions for roughness model has problems when the surface is very rough. Blocken et al. (2007) discusses the problems and gives recommendations for very rough surfaces in which the requirement that the first cell's $Y_p > K_s$ can not be satisfied. There is a conflict with the requirement that a fine mesh need be used close to the wall to resolve the high gradients. In this case, Blocken et al. suggests explicit modeling of roughness elements. This has been done by Miles & Westbury (2003) and leads to a significant improvement of the computed results compared to the results obtained with an approach flow over a smooth flat wall. The roughness blocks used in CFD simulations correspond to those used in wind tunnel study, using only smooth wall boundary conditions. The disadvantage of this methodology is that computational resources are wasted on less important part of the computational domain rather than improving the model of the primary object of study.

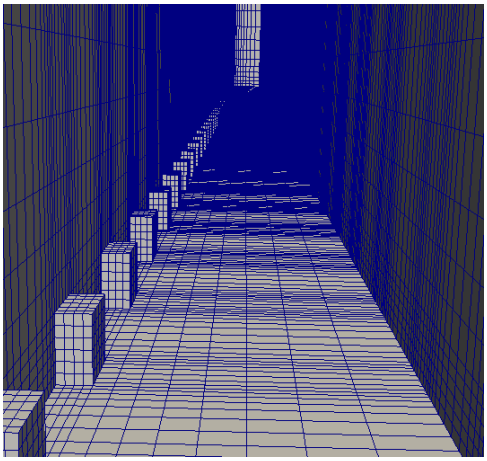


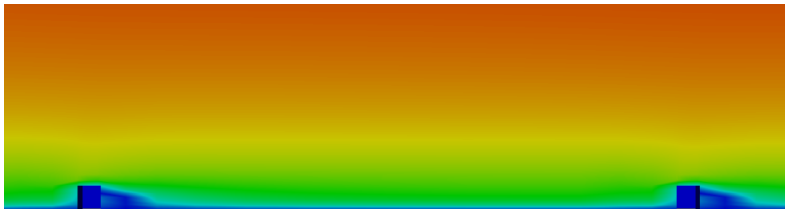
Figure 4.14: A look inside of a 3D symmetrical computational domain for regular array of cubes. The 2D plan of the model is previously explained in Fig.4.5

4.3.5.1 Simulations on a row of roughness elements

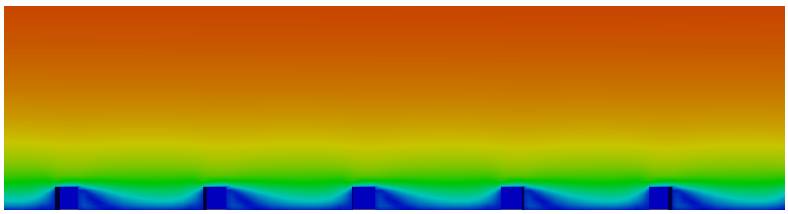
The first step is to determine configurations of roughness elements that yield a desired profile at a target downstream location. This iterative process could sometimes be time consuming if data is not available from previous wind tunnel tests. If a regular array of blocks arranged in simple manner (aligned or staggered) is used to represent roughness explicilty, the inherent symmetry can be exploited to reduce the computational domain. A typical symmetrical computational domain is shown in Fig. 4.14. It is clear that simulating one raw of obstacle arrays is sufficient if a time averaged turbulence model, such as the k-epsilon model, is used. A section passing through the center of the cubes and another one passing through the center of the open space between two raws gives same result as the one shown in the Fig. 4.15. A two dimensional simulation can be used, but it results in larger spacing of roughness blocks because the blocks are assumed to be continuous in the transverse direction.

An approximate formula to relate roughness length with average frontal and planar area of obstacles can be found in MacDonald et al. (1998). The spacing and height of blocks using the formula are usually good estimates for starting the iterative process. The first simulations conducted are for homogeneous roughness patches of open-country, sub-urban and urban roughness characteristics. Steady state simulations with k-epsilon turbulence model are conducted on a 2km long domain for each of the roughness patches. The result of this preliminary analysis are shown in Fig. 4.15.

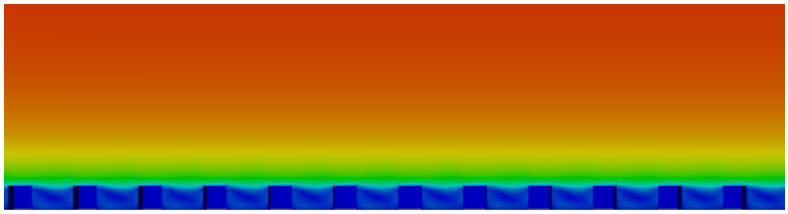
We can observe the three possible flow regimes first predicted by Oke (1998). The first roughness configuration is representative of open terrain ($B/H \sim 24$) is in an isolated flow regime. The wake and the separation bubble behind each obstacle is fully developed with re-



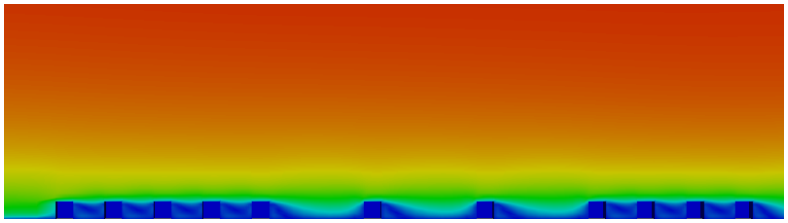
(a) open-no interference



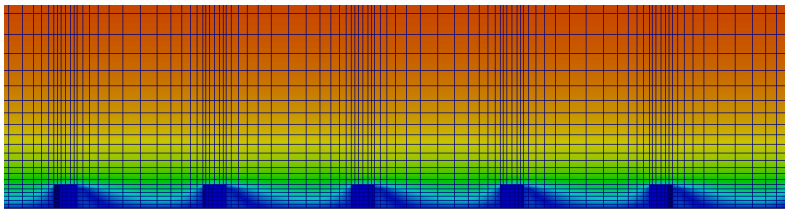
(b) suburban - wake interference



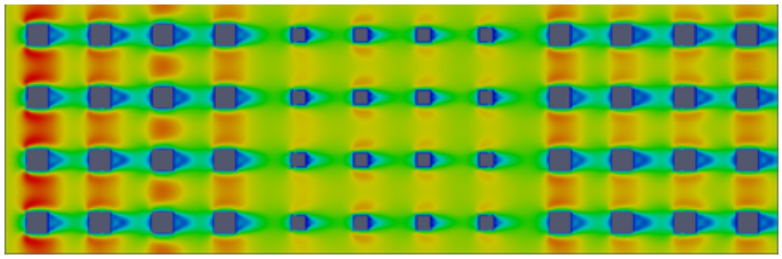
(c) urban-skimming



(d) multiple-patches



(e) Mesh



(f) Full domain simulation

Figure 4.15: Velocity contours for different roughness characteristics showing isolated (open-terrain), wake-interference (sub-urban) and skimming flow (urban).

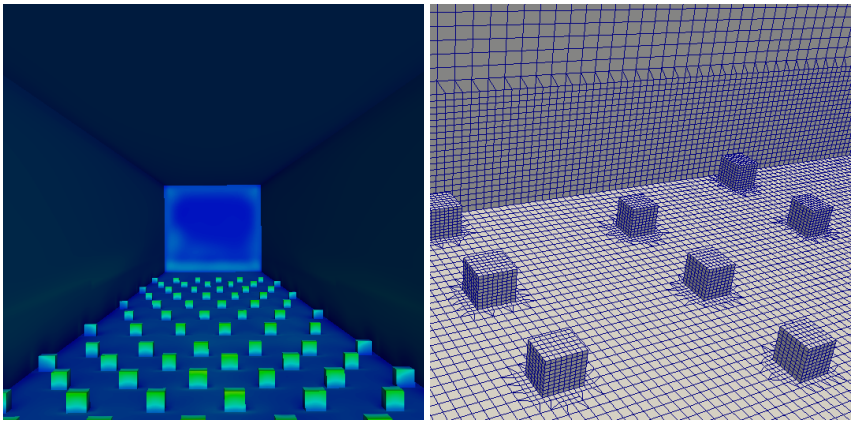
attachment occurring before the next element. With increasing density, the roughness elements become close enough so that the wake behind an obstacle starts to interfere with that of the downstream obstacle. The suburban roughness configuration ($B/H \sim 5$) seems to be in this wake interference regime. If the density increases further to a very rough urban setting ($B/H \sim 1.7$), the flow begins to skim over the elements. Our simulations used a constant height of $H = 10m$ for the blocks, varying only the spacing B for different roughness. As a result the skimming flow effect is more pronounced than it would have been if variable height blocks were used.

In all cases the bulk of the flow is forcibly displaced up and over the obstacle, which causes acceleration or a jet, but once over it is able to expand again and decelerates accordingly. This flow region, disturbed because of the presence of the obstacle, is called the displacement zone. Jimenez (2004) emphasizes the importance of the blockage ratio σ/h to the development of a logarithmic profile. The ratio measures the direct effect of the roughness on the logarithmic layer. For our simulations the boundary layer height $\delta = 500m$ and height of blocks $H = 10m$, hence $\delta/h = 50$. Jimenez notes that the ratio should be larger than 40 before similarity laws can be expected, and experimental results suggest that it should be greater than 80. Flows with higher blockage fractions retain few of the mechanisms of normal wall turbulence, and can better be described as flow over obstacles. Hence it is important to make sure the blockage ratio is with in the acceptable range.

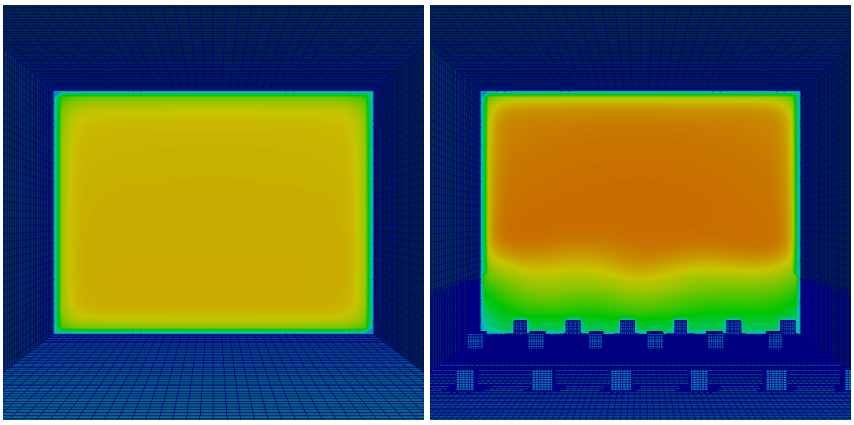
4.3.5.2 Simulation of a BLWT with spires and barriers

Before we conduct a case by case study of multiple roughness patches, we simulate a virtual wind tunnel with and without raised roughness blocks. The flow in a wind tunnel is bounded by walls all around, unlike the case of ABL flow where symmetry boundary is usually assumed at the sides and top of the domain. It is appropriate to use a no-slip boundary condition on all walls since boundary layers develop on all four sides. The wind tunnel in University of Western Ontario has a length of 26m, a width of 2.4m and a variable height from the inlet (1.55m) to the exit (2.15m). The roughness features are spires , barrier and roughness blocks. We first simulate the case where none of these roughness features are used, and evaluate the change in velocity profile due to the expansion of the tunnel alone. This simulation is similar to that of a smooth pipe flow. A grid with about 2.6 million cells (480x40x40) is used which is found to be enough for a grid independent result.

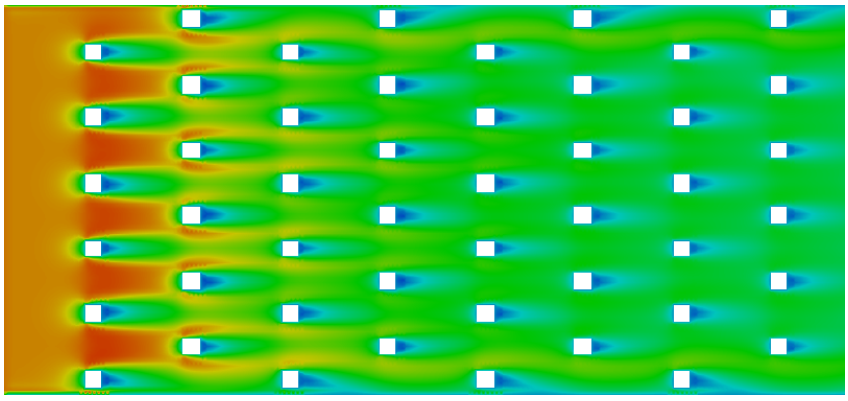
Next we simulate with roughness blocks of 0.1m high placed in a staggered manner. As we can see from Fig. 4.16 and 4.18, the boundary layer thickness on the bottom surface increases



(a) Inside look of surface(left) and close-up of surface mesh(right)



(b) Cross-section of velocity contours: empty domain(left) and with roughness blocks (right)



(c) Planar section of velocity contours

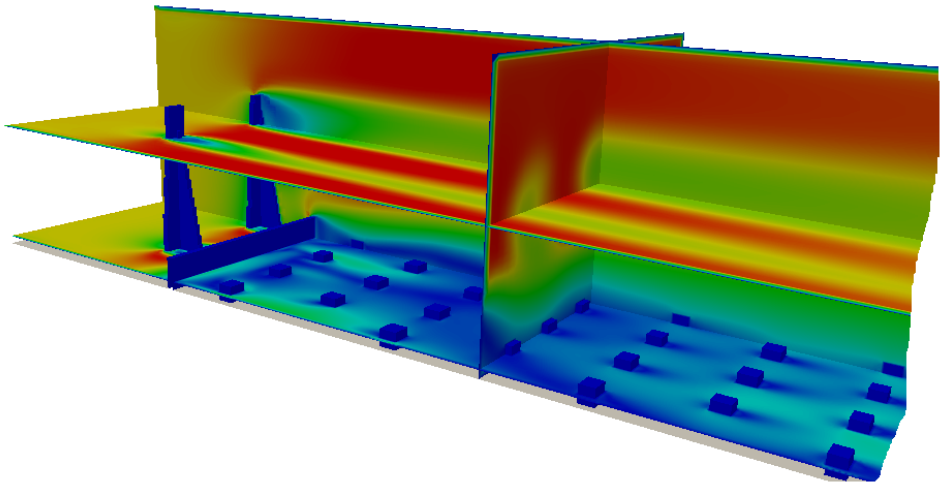
Figure 4.16: V-BLWT simulation results with surface roughness blocks

due to addition of the blocks. This is associated with an increase in turbulent kinetic energy. A planar view at half of blocks height shows that each block develops a wake. The interference effect in staggered arrangement is not as pronounced as that of a regular arrangement where the sheltering effect is maximum. We can observe that the first couple of rows have the longest wakes where the wind adjusts to the new roughness conditions.

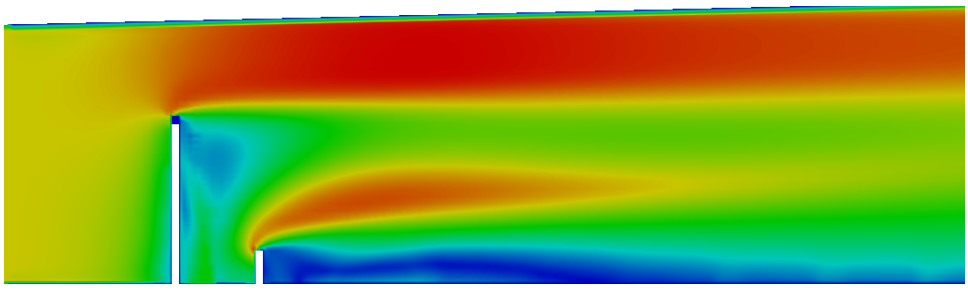
Next three spires and a barrier are added to help in development of boundary layer as soon as possible. If a uniform flow enters the tunnel, it is expected that a boundary layer will develop $6H$ downstream of the spires. Figure 4.17 shows the mesh and result of the analysis after the addition of these new roughness features. It can be observed that the boundary layer depth and turbulent kinetic energy has significantly increased compared to using roughness blocks alone. The dimension of spires and height of barrier have a significant effect over the profile at the turntable. The wake from spires is very elongated as shown in Fig. 4.17.

4.3.5.3 Simulation of multiple cases with a virtual Wind tunnel

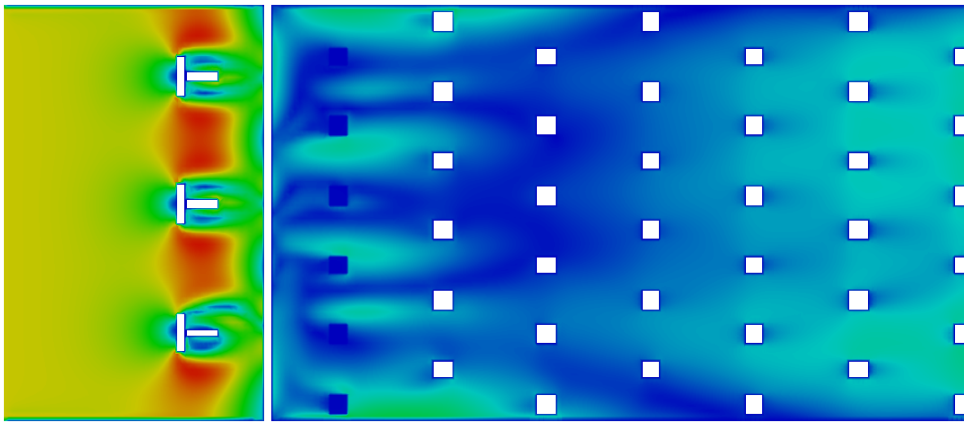
First we consider an approach of simulating a whole boundary layer wind tunnel similar to that done in section 4.3.5.2 but without using spires and barrier as shown in Fig. 4.21. Also the V-BLWT used for this case is from Wang & Stathopoulos's study, Concordia University BLWT. Roughness blocks are used to model suburban and urban roughness, while carpet is used for open country roughness. Some of the V-BLWT setups for multiple roughness patches are shown in Appendix A. We do not incorporate spires,barriers or grids to the models to save on simulation time. Fully developed boundary layer velocity and turbulence intensity profiles are applied at the inlet of the V-BLWT instead. The wind tunnel has a length of 12.2m, width of 1.8m and height of 1.8m. Open country roughness is is directly incorporated by the use of wall functions. This method has a limitation in that the nearest cell to the wall should be big enough, but since $z_0 = 0.024$ is small the requirement is satisfied. For the suburban and urban roughness blocks are used as shown in Fig. 4.19. The blocks used in Wang & Stathopoulos's study were 1in cubes for suburban (S), and 1.5in cubes for urban (U). This results in too many roughness elements for the simulation, so it is decided to double the size of the cubes to 2 in and 3 in respectively. The number of roughness blocks is as a result reduced by four times. This is in accordance with formulas that use area density ratios to determine average roughness characteristics. The modified block sizes result in the same planar and frontal area density ratios as the original, hence they are equivalent. For this simulations we consider blocks to be the only roughness features, and no spires, grids or barriers are used. Instead of a uniform wind profile as used at wind tunnel inlet, an ABL boundary layer profile is applied. The inlet velocity



(a) Velocity contour details close to spires, barrier and roughness blocks



(b) Contour of U at mid vertical section



(c) Contour of U at height of blocks

Figure 4.17: Virtual BLWT simulation with spires, barrier and roughness blocks

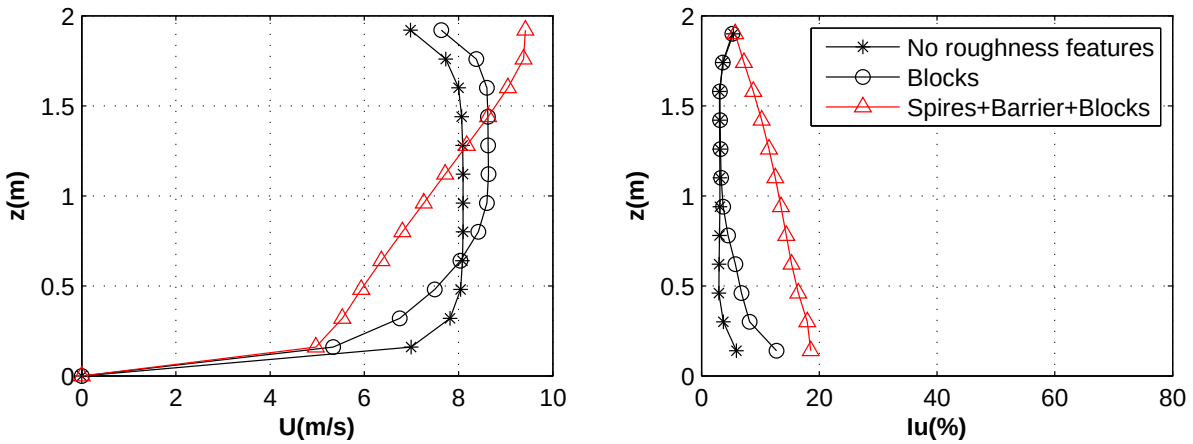


Figure 4.18: Comparison of U and Iu profiles for different roughness features

profile is logarithmic with the gradient height fixed at 600mm and $U_g = 12.5m/s$. The length scale of the BLWT simulations is 1:400 and time scale is 3:400. A preliminary simulation is

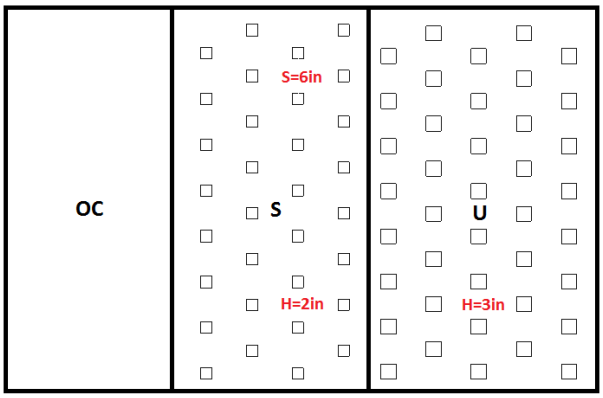


Figure 4.19: Open country(OC), Suburban(S) and Urban(U) roughness representation

carried out on an open country roughness. A sand grain roughness of $K_s = 20z_0 = 0.48$ is used for the wall function.The result is shown in Fig. 4.20. There are two problems with this simulation. First the first cell height $Y_p = 0.48$ is too high compared to the boundary layer thickness $\delta = 0.6m$. The problem of matching roughness in wind tunnel problems is a well known problem. For the simulation of the 69 cases of Wang & Stathopoulos a much lower roughness is assumed for the carpet so that the $Y_p > K_s$ condition is met. The first simulations we carried out with $K_s = 0.48$ for open carpet turned out to be bad where a bulge in the velocity profile is observed close to the ground. Using a lower roughness for OC corrected this problem much better fit are obtained except for the cases where open country roughness dominates the other patches. Second one should not expect horizontal homogeneity as the case of an empty domain because of the no-slip boundary conditions used at the top and side walls.

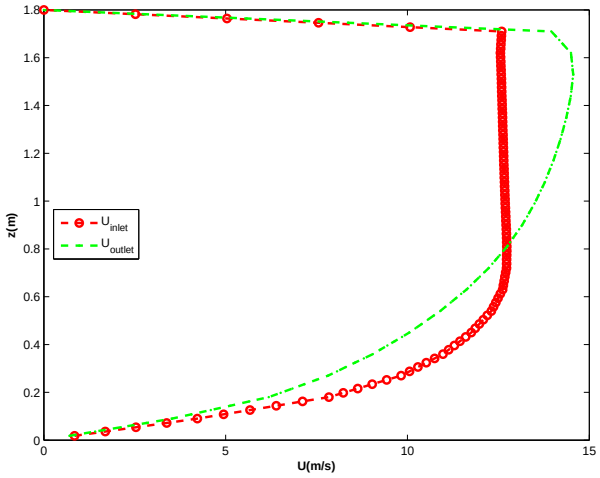


Figure 4.20: Inlet and outlet horizontal velocity profiles for open surface roughness

The results for the 69 cases are given in the following pages and Appendix A. We can observe that in most of the cases the V-BLWT fits the data much better than ESDU model. This is in contrast to the result found by Wang & Stathopoulos using numerical model with 2D simulations, which gave closer result to the ESDU model. The reason for this difference is not the simplified 2D model rather the difference in the shear stress modeling at the wall. Wang & Stathopoulos assumed a model of shear stress variation with fetch suggested by Bradley (1968).

$$u_*(x) \sim x^{-0.1} \quad \text{(for S-R)} \tag{4.35}$$

Garrat (1989) found that the shear stress initially increase to about twice its equilibrium value for S-R change, and decreases to about half its final value for R-S change.

$$\begin{aligned} u_*(x) &\sim 2x^{-0.1}u_* \quad \text{(for S-R)} \\ u_*(x) &\sim 0.4x^{-0.1}u_* \quad \text{(for R-S)} \end{aligned} \tag{4.36}$$

These equations were directly incorporated in Wang & Stathopoulos's numerical model. Our approach does not model shear stress but let it develop from the simulation. Also we should note that it is difficult to incorporate Wang's numerical approach into an existing CFD software due to the shear stress model. The other difference concerns turbulence models. Wang's numerical model uses linear eddy viscosity (mixing length) model for turbulence closure, while the current approach uses two equation Reynolds Averaged Navier-Stokes (RANS) model, namely standard $k-\epsilon$ model. We believe that these two differences , primarly the shear stress model, are the reasons for better result found from virtual wind tunnel simulations.

4.3.5.4 Simulation of WS cases using simplified 3D models

The results of the V-BLWT simulations suggest that computational effort can be reduced by taking advantage of symmetry of arrangement of the roughness elements. This is especially true for the rows in the middle that are farthest from the side walls. If the wind tunnel was infinitely wide, i.e. in the transverse direction, full symmetry can be achieved at all rows. Hence we can exploit the symmetry by considering only two rows with the sides of the domain cutting through the centerline of the rows. If the arrangement was a regular, one row of blocks would have sufficed as outlined in the preliminary investigations and shown in Fig. 4.5. The Symmetric Virtual Boundary Layer Wind Tunnel (S-BLWT) represents an infinitely wide BLWT where as the V-BLWT represents an actual BLWT with limited width in which the side walls retard the flow for a no-slip boundary condition. If the side walls of V-BLWT are also slip walls (symmetry), then the result of V-BLWT and S-BLWT should be exactly the same.

All the 69 cases of Wang are simulated again with this new setup , i.e. S-BLWT. The simulation time decreases tremendously since the width of the tunnel is decreased by almost 35 times. The results are shown along with the V-BLWT simulation results. We can observe that both wind speed and turbulence intensity results for the S-BLWT and V-BLWT are very close to one another. In some cases the V-BLWT wind speed result matches Wang's wind tunnel results better than the S-BLWT, hence V-BLWT is the better model for reproducing actual wind tunnel results. However the S-BLWT may actually be better in the grand scheme of things, because wind speed models over multiple roughness patches assume infinitely wide patches. Both Wang and ESDU model only take into consideration the length of patch (l) and not width of it(b).

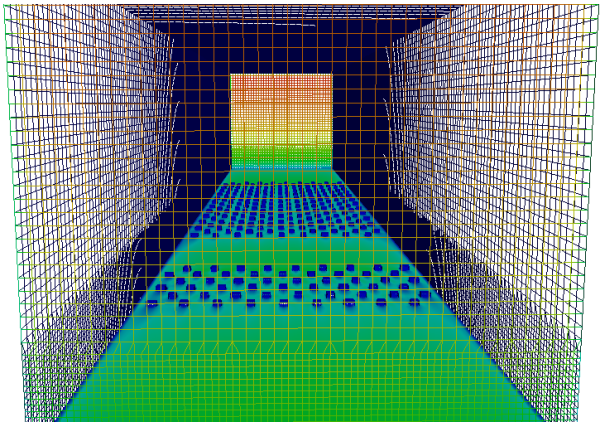


Figure 4.21: Perspective view computational domain of a virtual BLWT

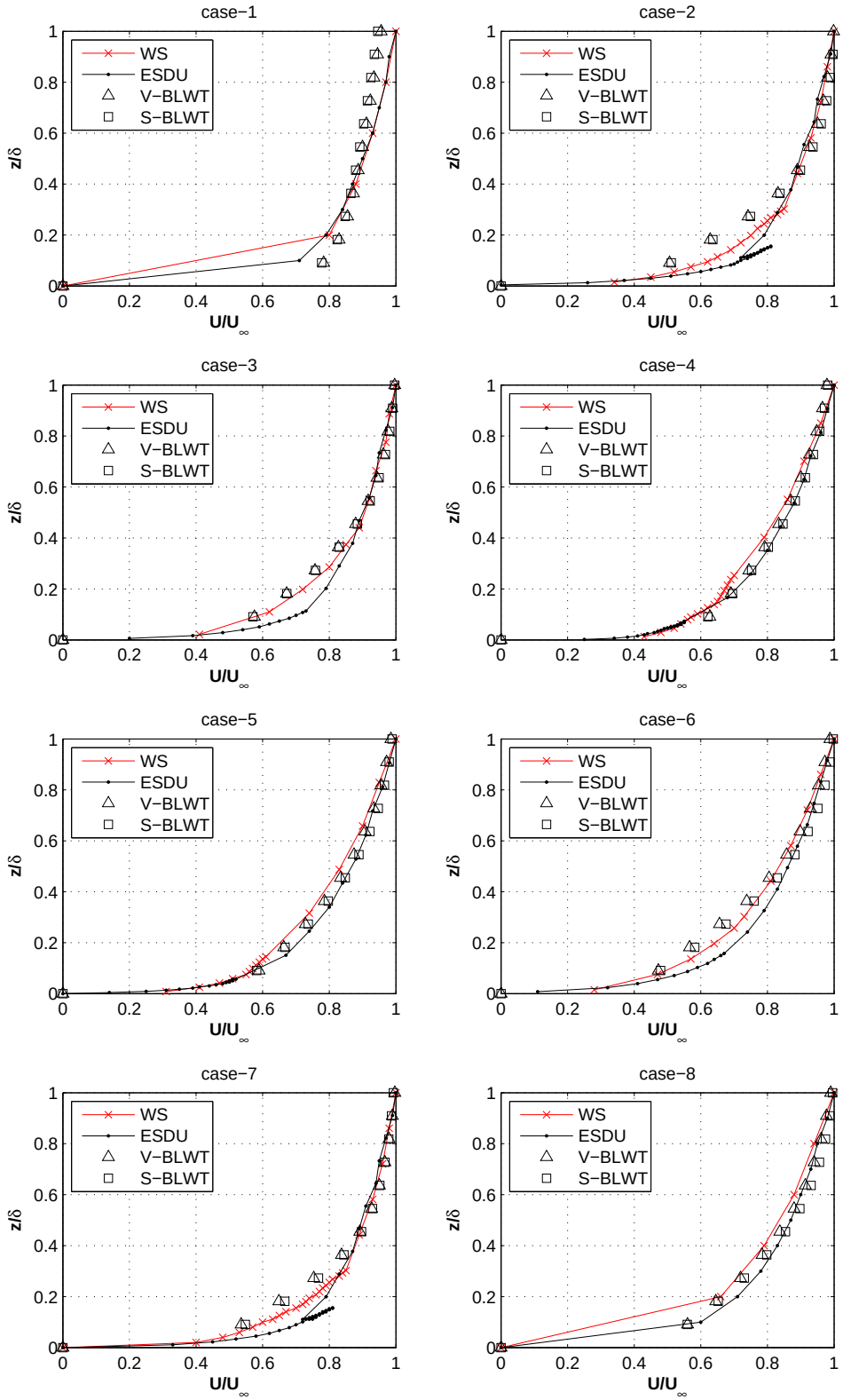


Figure 4.22: Horizontal velocity comparison of CFD with existing models for cases 1-8

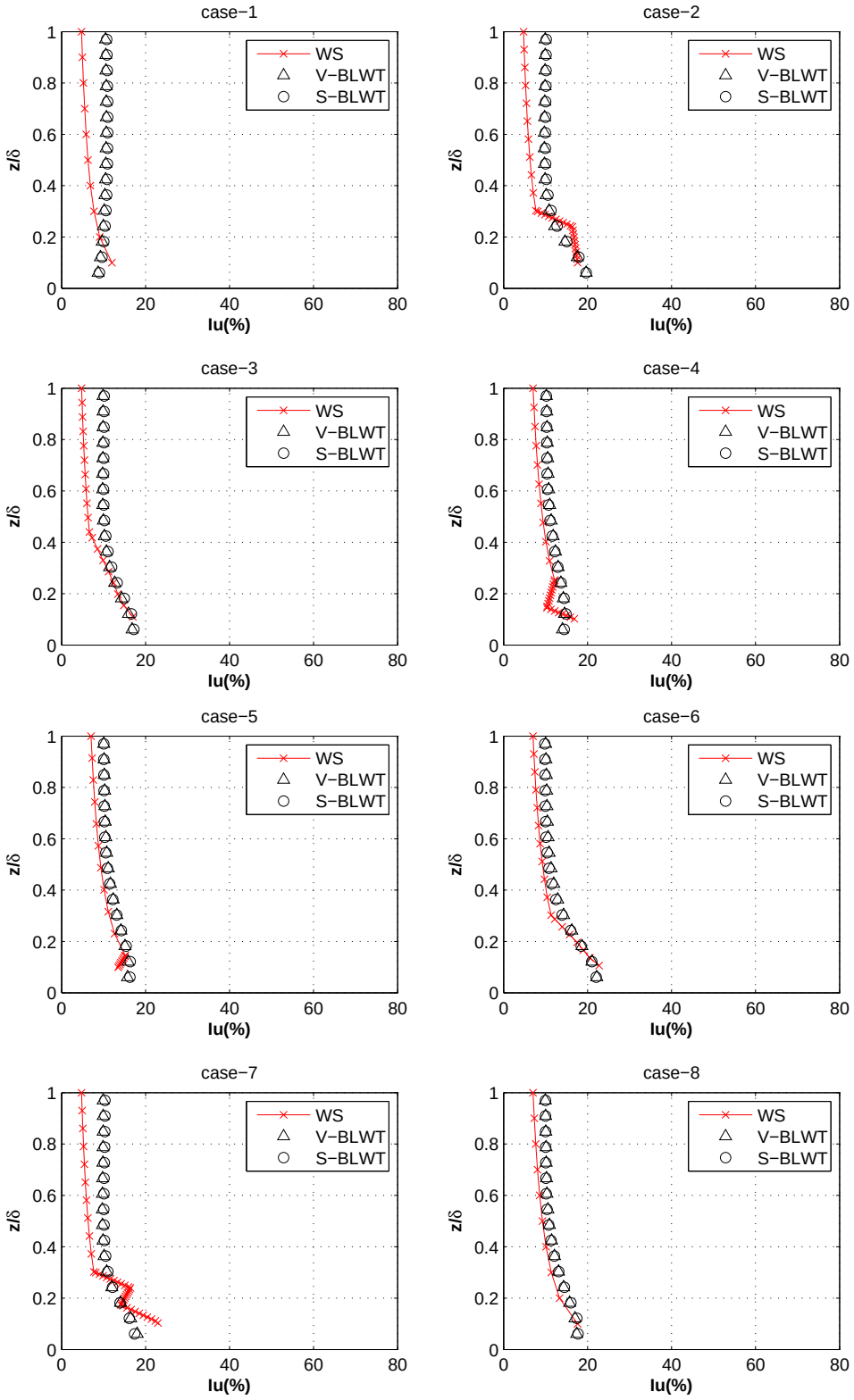
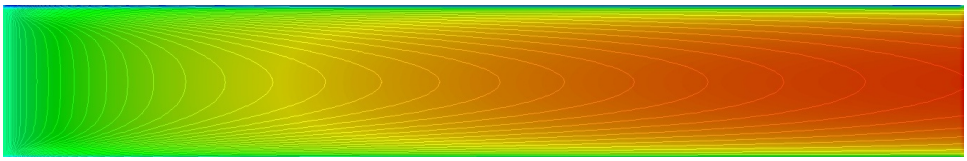
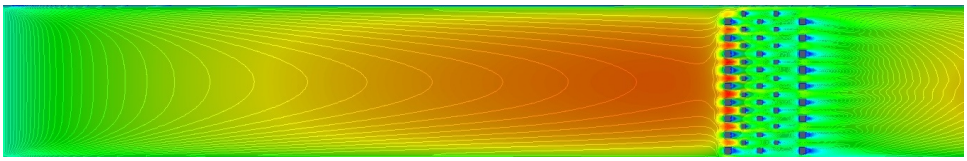


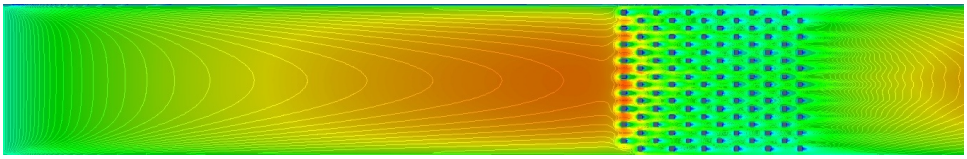
Figure 4.23: Turbulence intenisy comparison of CFD with existing models for cases 1-8



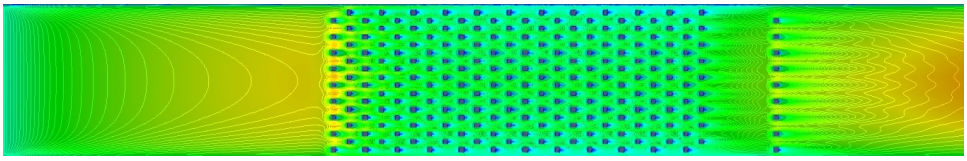
(a) BLWT case-1



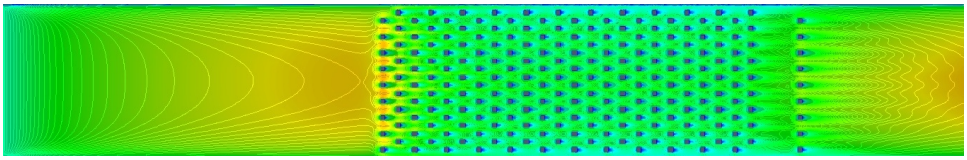
(b) BLWT case-2



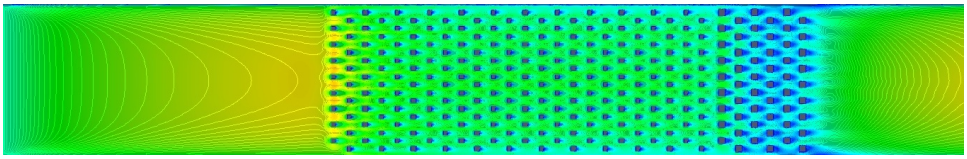
(c) BLWT case-3



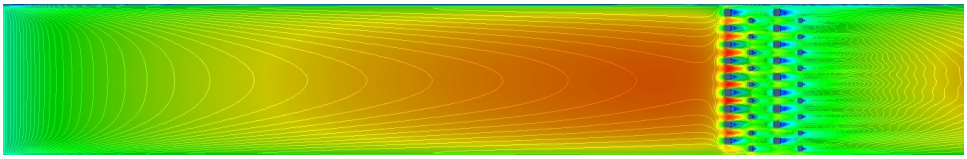
(d) BLWT case-4



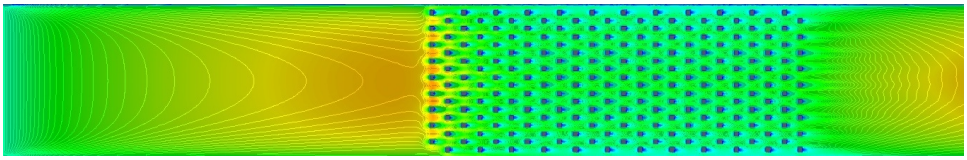
(e) BLWT case-5



(f) BLWT case-6



(g) BLWT case-7



(h) BLWT case-8

Figure 4.24: Horizontal velocity contour for V-BLWT configuration of cases 1-8

4.4 Complexity 3: Semi-idealized built environment

So far the cases considered focused on determination of average roughness characteristics of highly idealized built environment models. This is acceptable in cases where detailed wind flow characteristics inside the built environment are not of high importance. For example, in BLWT testing, the building of interest and its surrounding with in a short radius are modeled as best as possible, whilst the rest of the model is replaced with regular array of blocks that have similar roughness characteristics as the original model. The next higher level of complexity concerns flow in a semi-idealized urban canopy model. Wind tunnel test results are available, for the purpose of validation, from CEDVAL-LES (2011) for the urban model to be considered here. CEDVAL-LES is a compilation of wind-tunnel datasets intended to be used for validation of Large Eddy Simulation (LES)-based numerical flow and dispersion models. The database consists of both time series and and time-averaged statistics against which LES and RANS models can be validated. This study uses RANS turbulence models thus only the time-averaged statistics is used.

The semi-idealized urban model is shown in Fig. 4.25. Hertwig et al. (2012) mentions that the model is so chosen to be heterogeneous and morphologically consistent with a typical central European city characteristics. It has sharp building corners, open courtyards, plazas and complex intersections etc. The on-line database has two cases, one where all roofs are flat and the other where some of the buildings have slanted roofs. The flat roofs case is chosen for this study.

4.4.1 Computational domain setup and grid generation

The computational domain is setup similar to Hertwig et al. (2012), who conducted numerical simulations using various CFD software and compared the results with the CEDVAL-LES database. The model tested in the boundary layer wind tunnel has a scale of 1:225, with the full scale size representing an area of about 1320m X 820m X 24m. The size of the computational domain is 1672m X 1140m X 144m. First a background mesh of 191 X 118 X 41 is applied which is then transfered to snappyHexMesh for molding the urban model from the STL file of the building surfaces. All the three stages of snappyHexMesh are used but there were still some visible problems at the edges of inclined walls as shown in Figs. 4.26-4.27. The total number of cells generated by snappyHexMesh is about 4 million.

4.4.2 Boundary conditions

At the inlet a logarithmic profile with $U_{ref} = 6.537m/s$ at a height of $H_{ref} = 144m$ is applied. A homogeneous roughness of $z_0 = 0.06m$ is used for the ground, and hence the friction velocity is $U_* = 0.346m/s$. At the sides of the computational domain a symmetry boundary condition is used, and at the top the values of U, k and ϵ are fixed to the same value used for the inlet at the same height: $U = 6.537m/s, k = 1.057m^2/s^2, \epsilon = 0.0049m^2/s^3$. The profiles of k and ϵ are determined according to Richards & Hoxey (1993) formulas. At the outlet an outflow boundary condition is used.

4.4.3 Results and discussion

Plots of velocity contours at different heights with in the urban canopy of height 24m is shown in Fig. 4.28. We can observe that wind flow inside the built environment is complex due to the sharp corners, open yards, intersections and other features. The wind speed decreases and flow becomes more chaotic close to the ground thus grid refinement in the lower portions helps to capture the complex flow behavour better. As mentioned before, the purpose of this simulation is to assess performance of CFD for prediction of detailed wind flow characteristics inside a built environment. For this reason, mean wind speed profiles at many locations inside the core are compared with measurements in wind tunnel of the same model.

The wind field is sampled at 40 locations distributed uniformly across the area in an 5 rows X 8 columns. Densely spaced measurements are also available at the core of the model to characterize street canyon flow, but this work compared only normalized vertical velocity profiles at the 40 locations. A comparison between wind tunnel and CFD results are shown in Fig. 4.30. We can observe that there is in general a good agreement between the current CFD results using RANS model and the BLWT measurement. At some of the probe locations, some deviations are observed especially close to the ground where surface roughness effects have pronounced effect. Also use of additional layers of grid that are aligned with the surface can improve the accuracy of results, but as is the case in many CFD simulations there is a trade-off between accuracy and simulation time. The good agreement obtained here also serves as a verification of the current CFD code's RANS model, k-epsilon in this particular case, for built environment studies. The LES model can be verified using this model in the future using the instantaneous measurements in the database for which it is primarily intended for.

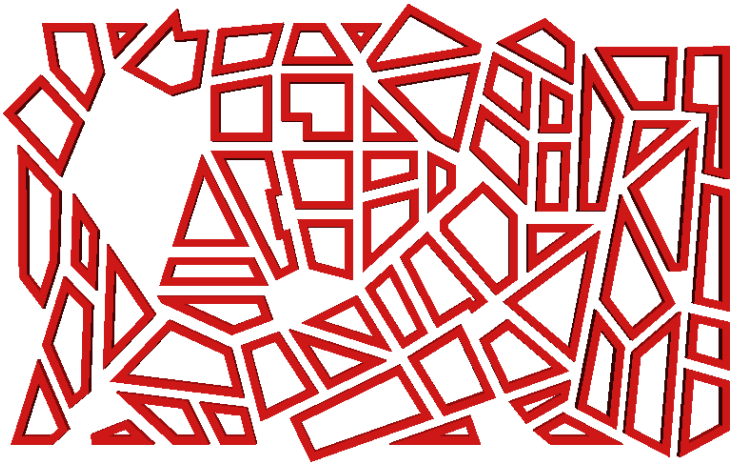


Figure 4.25: Semi-idealized urban model from CEDVAL database

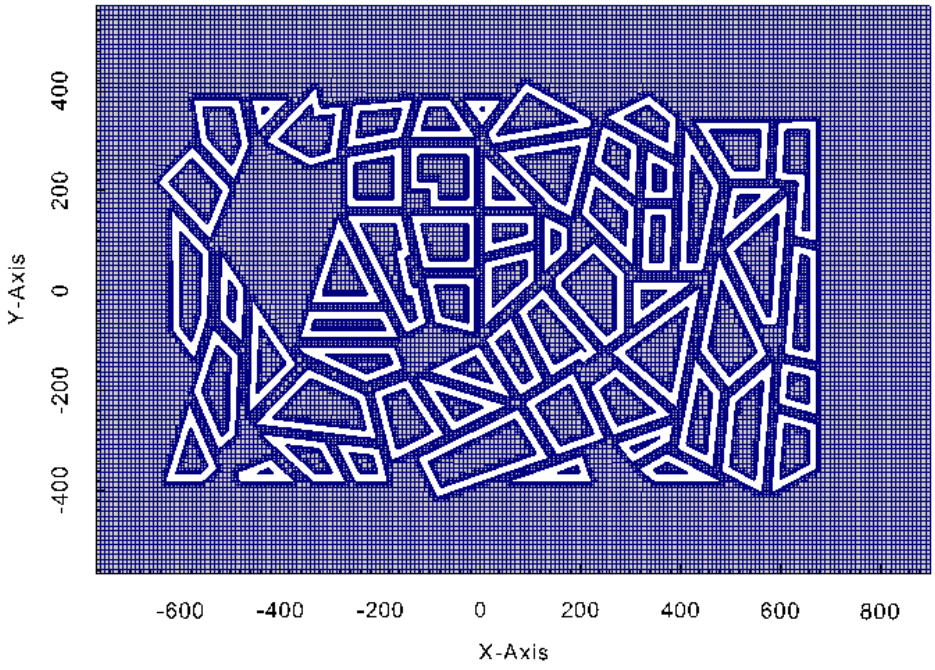


Figure 4.26: Plan of the semi-idealized urban model

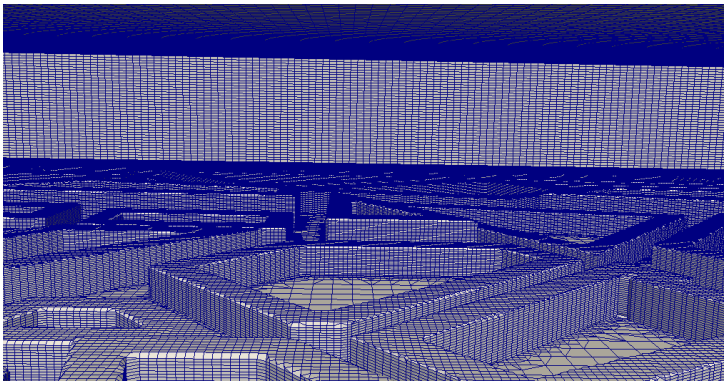
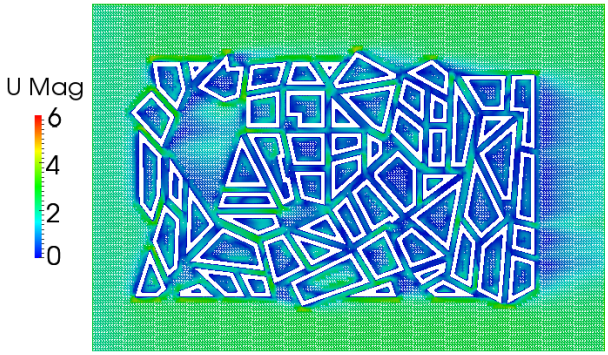
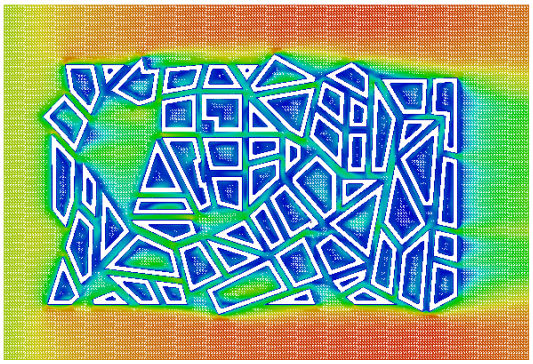


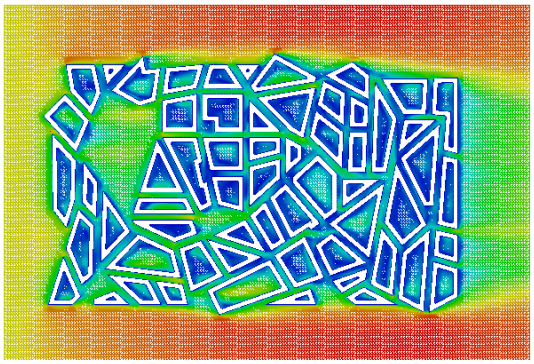
Figure 4.27: Inside view of the mesh generated for the semi-idealized urban model



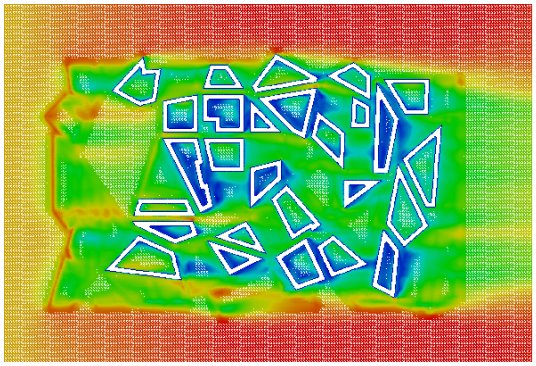
(a) At 2m



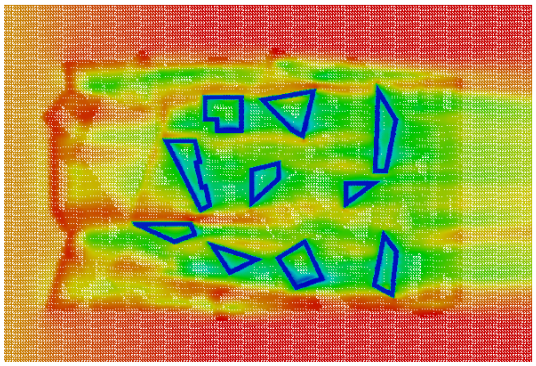
(b) At 9m



(c) At 12m



(d) At 18m



(e) At 24m

Figure 4.28: Velocity contours at different elevations

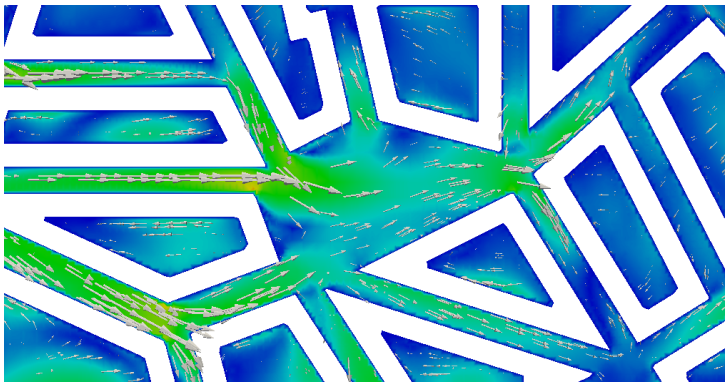


Figure 4.29: Velocity vectors at the core of the urban canyon

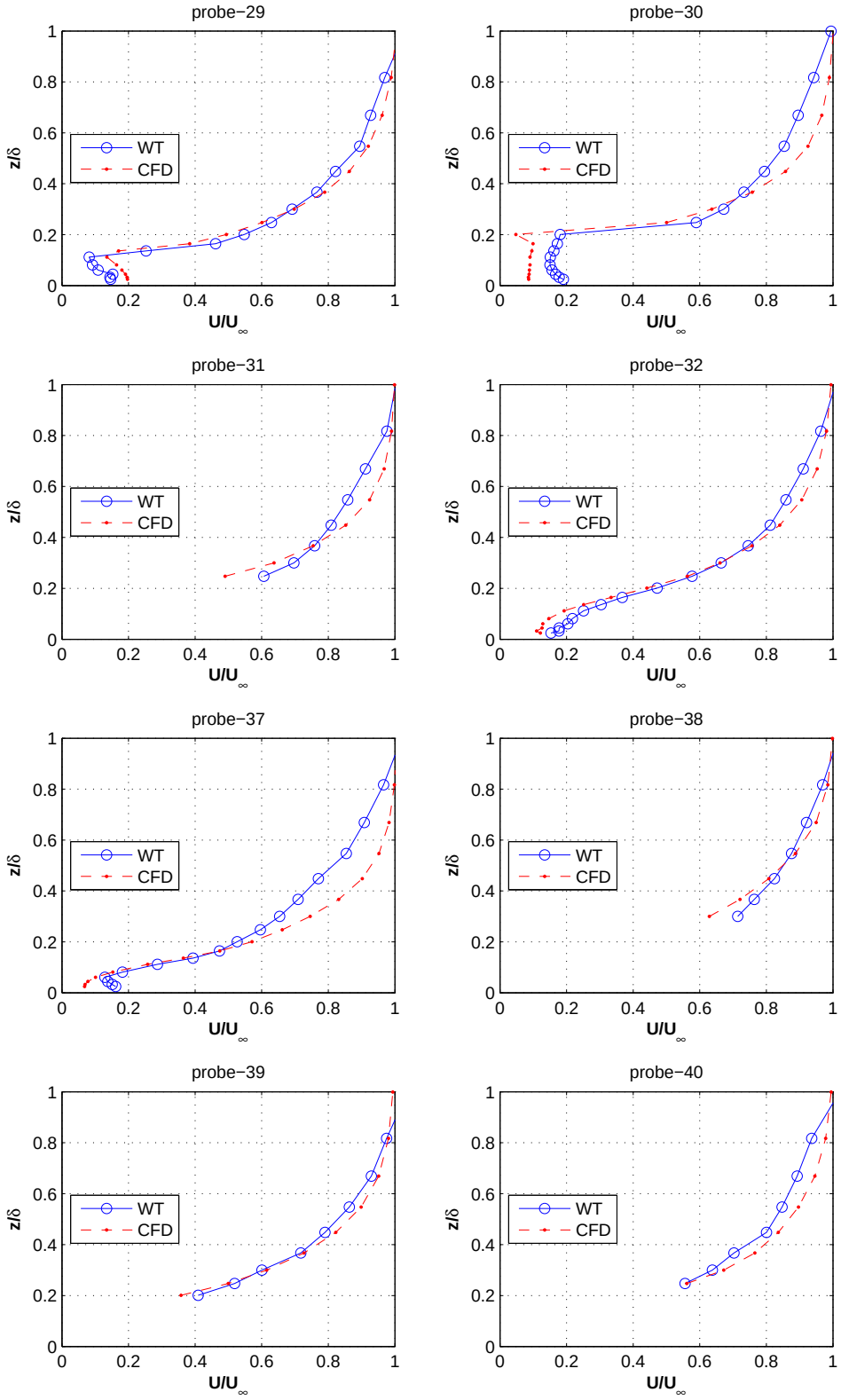


Figure 4.30: Comparison between CFD and BLWT for some of probe locations inside the model

4.5 Complexity 4: Built environment

CFD simulations in urban environment can be grouped in to two categories (Blocken & Carmeliet 2004) : (a) fundamental studies on simple and generic building configurations (b) applied studies on complex case studies. Fundamental studies on isolated cases help to understand flow behavior in and around a building, and also to validate CFD codes against wind tunnel or field measurements. Applied studies on specific urban setting have been conducted by many researchers despite the lack of extensive validation. Blocken et al. (2009) have reviewed the status of CFD in building performance studies of outdoor environment. The four main applications areas are summarized as follows.

- Pedestrian level wind comfort is an important consideration for high rise buildings. Usually wind tunnel studies are done to take point measurements of wind velocity at pedestrian height 1.8m. Area measurement techniques such as sand erosion can be used to spot the problematic points where hot wires are to be placed. CFD can be used to avoid at least this preliminary stage of the investigation. A case study of CFD simulation for pedestrian comfort in a University campus is described in Blocken et al. (2011).

- Air pollutant dispersion around buildings have been carried out using CFD on microscale level of about 5km horizontal length. Due to complexity of the phenomenon, studies are usually focused on two simplified models : urban street canyon and isolated building. Although studies have been carried out on complex urban environments, there is a lack of extensive validation which is usually the case for complex models.

- Wind driven rain (WDR) studies on buildings also benefit from CFD simulations albeit not as much as wind comfort studies do. The physical modeling of WDR requires expensive CFD techniques such as Lagrangian particle tracking of raindrops and LES turbulence model for accurate simulation, however RANS turbulence model are commonly used in practice.

- Convective heat and mass transfer studies on buildings using CFD requires accurate modeling of the boundary layer. Using wall functions as is used for other CFD simulations can overestimate heat transfer coefficients significantly. High Reynolds number simulation without wall function require very fine grids. Therefore the simulations are usually limited to simple cubic models.

The previous section focused on validation of CFD on semi-idealized built environment. The next higher level of complexity is a real built environment that is classified as Complexity

5 by CEDVAL-LES dataset. For this study, validation data is not available hence its purpose is for demonstration of the procedures to be followed. The built environment is an area in downtown Miami which has some high rise buildings. The computational domain has dimensions 4.3km X 4.3km X 2km. Micro-scale simulations of this magnitude need to consider the effect of Coriolis force since some buildings penetrate well into the Ekman layer. However this is ignored for the current simulations. The boundary layer height is chosen to be 2km to reduce blockage effect due to high rise buildings with heights $\geq 200m$ as shown in Fig. 4.34.

4.5.1 Computational domain setup and grid generation

First a background mesh of 120 X 120 X 60 cells is generated, which is then transferred to snappyHexMesh for refinement close to the ground. The final grid consisted of 2.3 million cells. The meshing process involves three stages : clipping, snapping to surface, and layer additions for better boundary layer simulations. The layer addition was problematic for this particular case, producing cells of high skewness and similar low quality cells, so the result after the snapping stage is retained. Snapshot of the background STL surface edges of building and the corresponding mesh is shown in 4.32.

4.5.2 Boundary conditions

At the inlet a logarithmic velocity profile for a rough surface condition is assumed mainly because no field observation data is available and the upstream terrain resembles a mildly rough environment from visual inspection. If field observation data was available for the inlet profile, the correct procedure is to make logarithmic fitting of inlet velocity profile followed by modification of the k-epsilon model constants (Blocken et al. 2011, Martinez 2011). Symmetry boundary conditions are assumed for the sides and top of the computational domain, and a no-slip ground surface with roughness of $z_0 = 0.1$ is assumed.

4.5.3 Results and discussion

Velocity contour plots at pedestrian level and higher are shown in figures 4.34. The simulations are re-run with different grid sizes to check grid independence of results. The number of cells in the vertical direction is changed to 30 and 90 cells for a total of 1.2 million cells and 3.2 million cells respectively. The smaller case result shows some qualitative differences with the current case of 2.3 million cells obtained using 60 cells in the vertical direction, however the larger case did not show much difference with the current result. Therefore it can be said

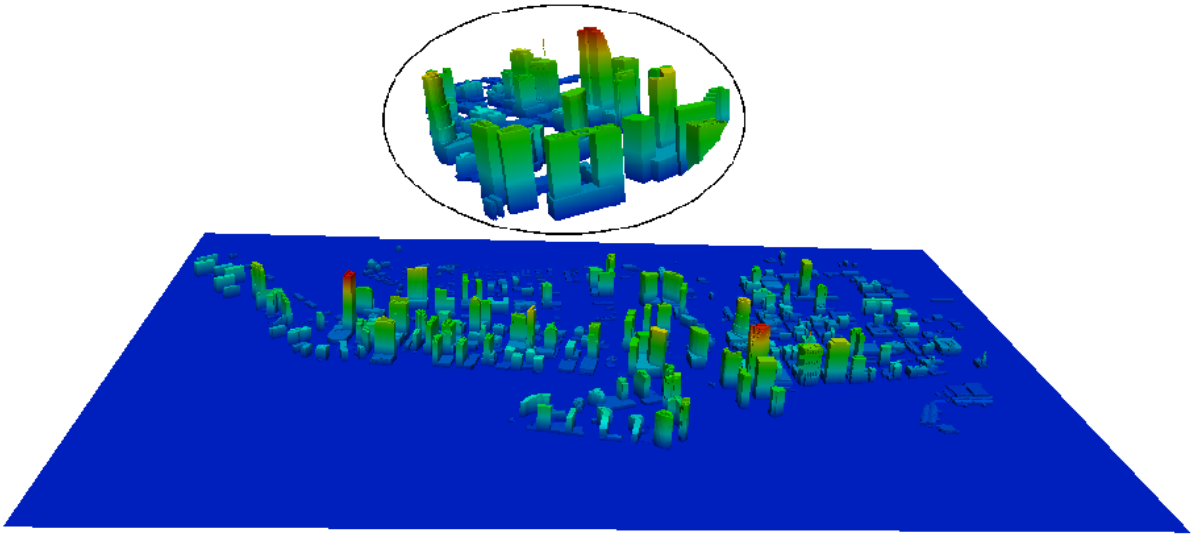


Figure 4.31: Surface model of a region in downtown Miami

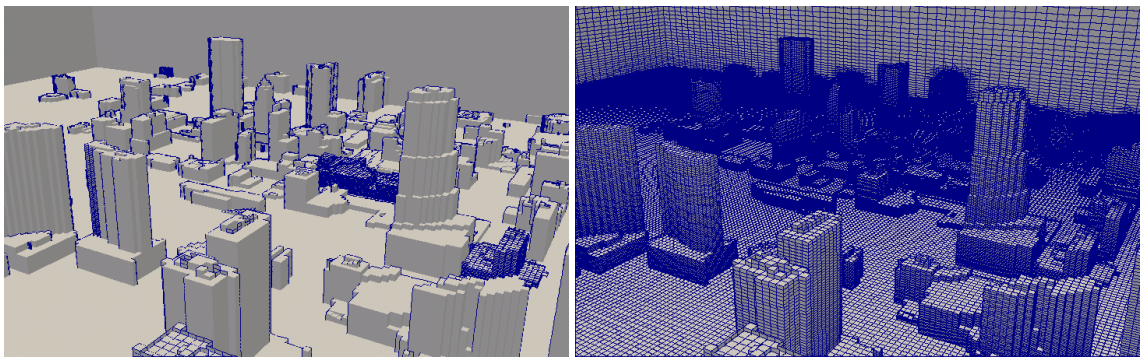
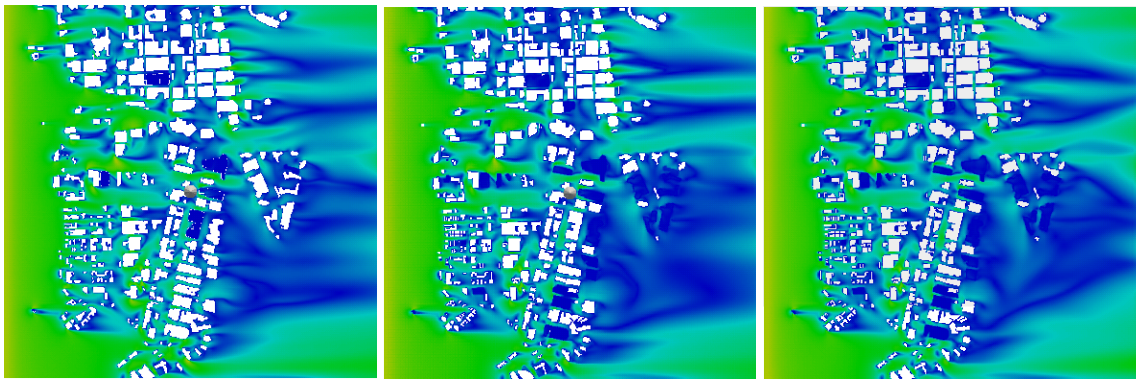


Figure 4.32: Building edges and corresponding mesh generated by snappyHexMesh

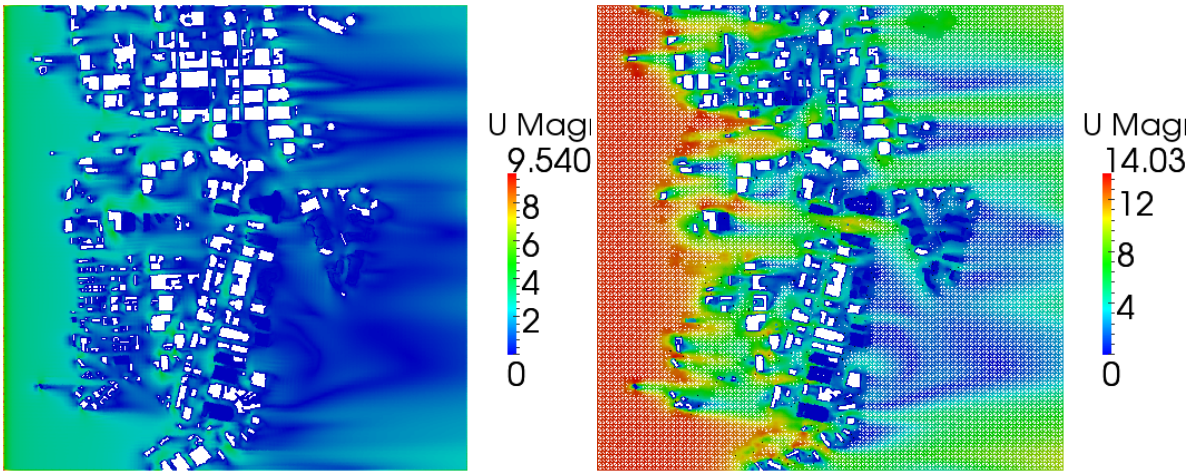
that grid independence has been reached with the current grid size of 60 cells in the vertical direction. The fact that a detailed information can be retrieved from CFD analysis, compared to just point probes in wind tunnel or full scale investigations, is what makes them attractive for many engineering applications. However the lack of validation data, as is the case in this simulation as well, and expertise in CFD modeling leave something to be desired.

The velocity contour at different elevations show that wind speed increases with height. Also only few of the buildings reach the height of 200m, hence the planar density area ratio of obstacles, a parameter that affects roughness, also decreases with height. Micro-scale simulations of built environment of this size, that do not cover the whole area, pose a problem with regard to the boundary conditions. Here we can see that at the sides of the domain there are many buildings thus assuming symmetry boundary condition is not appropriate.

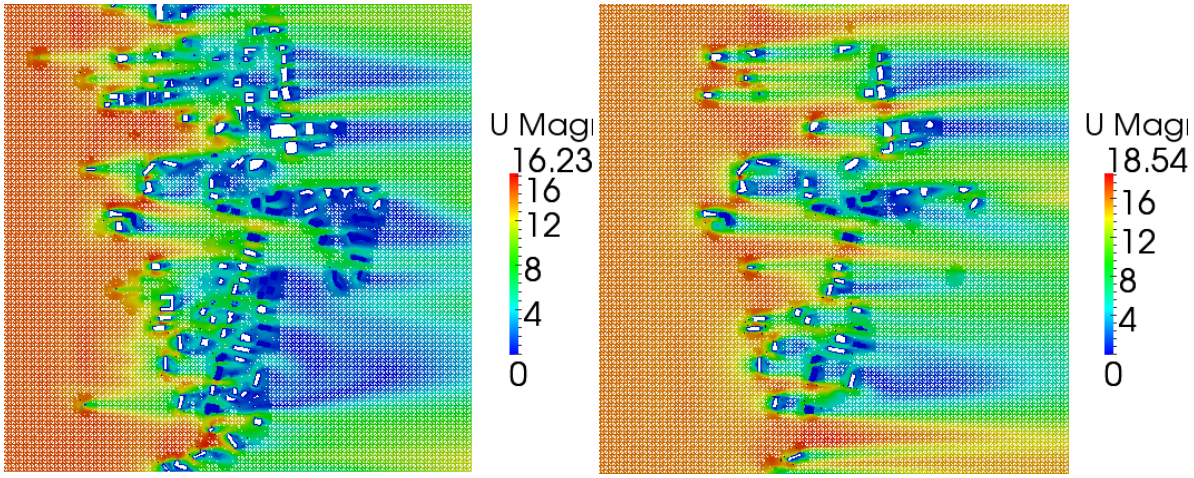


(a) 30 cells in z-direction (b) 60 cells in z-direction (c) 90 cells in z-direction

Figure 4.33: Velocity contours at 5m height for different grid sizes in the vertical direction



(a) At 20m (b) At 50m



(c) At 100m (d) At 200m

Figure 4.34: Velocity contours at different heights

4.6 Prediction with artificial neural networks

It is clear the V-BLWT methodology presented in this research is costly in terms of computational resources, though less expensive than conducting actual BLWT experiments. An approach followed by Bitsuamlak et al. (2004) to reduce the number of simulations is use of neural networks. From a database of simulation results, a reduced model can be built using artificial neural network that can capture non-linear relationships among different parameters. For a quick estimation, the reduced model can be used to get expected outcomes of simulation. The current work did not produce enough V-BLWT data to be used for this purpose, hence a similar study using actual BLWT data is carried out to investigate the relation between roughness elements and wind profiles using a reduced model. From the developed model, roughness element configuration can be predicted from the observed wind velocity and turbulence intensity measurements and vice versa. The details of the procedures folllowed are briefly described in the following sections and in more detail in Abdi et al. (2009).

4.6.1 Data acquisition

The neural network model is first trained with velocity and turbulence intensity measurements, and then the resulting model is used for prediction of configuration of roughness elements. Wind profile data was collected in a recently commissioned BLWT at RWDI USA LLC in Miramar, Florida. The unique characteristic of BLWTs is an extended working section downwind of the contraction over which an appropriate wind profile is developed. This particular wind tunnel is a closed-circuit tunnel with a 40 ft long and 8 ft wide working section upwind of the wind tunnel model, which is mounted on a turntable at the end of the working section. The ceiling height varies from 6 ft to 7ft above the turntable. This wind tunnel employs the spire-roughness technique to develop the wind profile, as described by Irwin (1979).

Figure 4.35 shows the working section of the BLWT. Three trapezoidal spires extending from the wind tunnel floor to ceiling are situated at the entrance to the working section. The floor is covered with triangular roughness elements in 40 staggered rows 1 ft apart. Spires of various dimensions can be interchanged manually as necessary, while the roughness elements are raised lowered by means of mechanical actuators controlled from the wind tunnel control room in order to save testing time. Massing models of the test building, present and future surrounding buildings are mounted on the turntable at the end of the working section, which can rotate 360 degrees to simulate wind from any direction. In the use of the spire-roughness technique for boundary layer wind flow simulation, the fundamental question to be answered

Figure 4.35: RWDI wind tunnel working Section, spire and roughness blocks

is the following: 'What size, shape, location and number of spires, and what floor roughness height is needed to recreate a particular target atmospheric boundary layer wind profile in the wind tunnel?' While there are a multitude of combinations of spire sizes, shapes, locations and floor roughness heights, the problem was reduced to a manageable size through previous experience. Three trapezoidal spires spaced on the centerline and 18in from the tunnel wall, and uniform floor roughness were kept constant. Thus, the remaining design variables were the top and bottom spire widths, and the uniform floor roughness height. These design variables are summarized in Table 4.2, along with the variable ranges that were used. It was desired to collect data for various combinations of these variables in order to train and test the artificial neural network model. Pressure data were collected with a 'pitot rake' positioned at the centerline of

Table 4.2: Roughness features dimensions

Variables	Range
Spire top width	5in-8in
Spire bottom width	10in-19.5in
Block height	0in-3in in increment of 0.5in

the working section, at the upwind edge of the turntable. The rake consisted of 53 pitot tubes. The pitot tubes were spaced at 0.5in intervals up to 5in above the tunnel floor, at 1in intervals up to 30in, and at 2in intervals from 30in to 66in above the tunnel floor. At a typical model scale of 1:400, the uppermost measurement location equates to a full-scale height of 2200 ft. The pressure data were sampled at 512 Hz for 36 seconds. From these time series of pressure, longitudinal velocities and longitudinal turbulence intensities were determined. The velocity ratio was defined as the ratio of the mean velocity at a particular pitot to the mean velocity of the pitot at a reference height of 60in above the tunnel floor. The turbulence intensity was defined

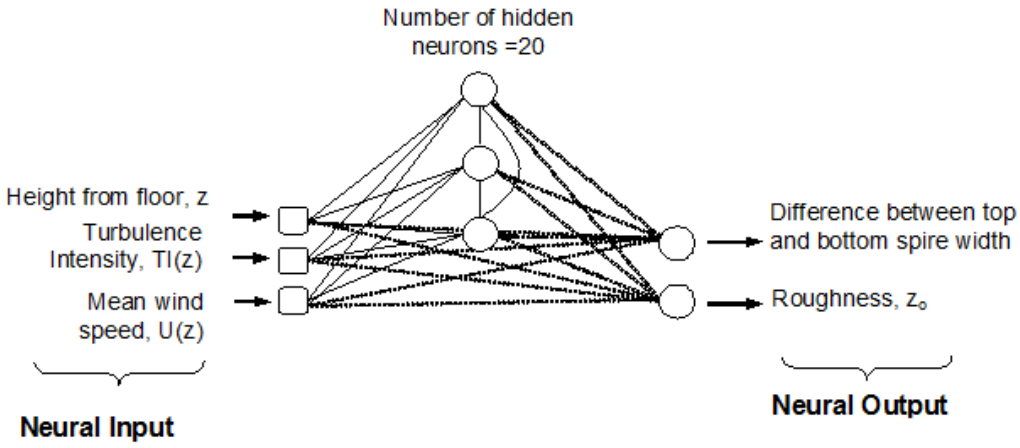


Figure 4.36: Neural network model with CCNN architecture for roughness estimation: 3 input, 20 hidden and 2 output neurons

as the ratio of the r.m.s to the mean velocity at a particular pitot. Thus, for each combination of design variable values, profiles of velocity ratio and turbulence intensity from 1in to 66in above the wind tunnel floor were determined.

4.6.2 Artificial neural network model

The most practical design considerations to build and train a neural network include the selection of an appropriate internal error criterion, efficiency of learning algorithm as well as choice of network topology and optimum stopping criterion for maximum performance. In the present work the neural network tool for prediction of wind profiles or estimation of roughness height and spire dimensions required to generate a specific target profiles is developed based on the cascade correlation algorithm using object-oriented methodology following the methodology described in Bitsuamlak et al. (2006). The architecture of a Cascade Correlation Neural Network (CCNN) is shown in Fig. 4.36. In this algorithm new hidden neurons are installed one at time during run-time as required from a pool of candidate hidden neurons, which are initialized to different weights and trained separately in the background. Note that the candidate neurons are not connected to the rest of CCNN during training. Thus, for each new hidden neuron, the present algorithm tries to maximize the magnitude of the correlation between the new neurons output and the residual error signal of the CCNN. Installation of new hidden neurons is automatically stopped when the network meets the error criteria or exceeds the maximum number of hidden neurons set by the user. For validation and comparison purposes a second flavor of Neural networks is also tested. The Multilayer Perceptron Neural Network (MPNN) uses a

supervised learning technique called back propagation. The major difference with the CCNN method is that the CCNN method works by installing new neurons while MPNN continually adjusts weights of neural network until a desired level of accuracy is reached. The C++ code for the MPNN method is given in Appendix B.

4.6.3 Results and discussion

4.6.3.1 Wind profile prediction

The neural network is trained with the database and then used to predict mean longitudinal velocity and turbulence intensity profiles from four input parameters, namely, height above which velocity measurements are taken, roughness length, top and bottom spire widths. Samples are taken randomly from the available data to train the ANN and then predictions are made on the remaining data. Some of the inputs are normalized with respect to the maximum values for better efficiency. Comparison of the predicted velocity profile and turbulence intensity with observed values showed a very good match, as is shown in the figures 4.37.

4.6.3.2 Estimation of tunnel surface roughness and spire dimensions

The inverse problem of determining roughness length and width of spire is done in the same way as the forward problem but by switching the inputs and outputs. Thus for the inverse ANN modeling the following three inputs are used: Target mean longitudinal velocity profile, target turbulence intensity, and height above which velocity measurements are taken. The outputs include the roughness length (of the wind tunnel floor), and the ratio of width of spire at height z divided by the bottom spire width. The inverse modeling is noticed to require more iteration to converge to the solution for a given tolerance (mean square error). For one test setup, the spire widths and roughness length are kept the same while measurements of velocity are conducted at different height. Hence, it is expected that the inverse ANN model to predict a single value of roughness length and Top and Bottom width of a Spire. Table 4.3 shows the comparison of the measured and ANN predicted values. These values can be used as starting values for further wind tunnel verification thus reducing cycle in the trial and error process.

4.6.4 Conclusions

Artificial neural networks are used to predict wind velocity and turbulence intensity profiles in a wind tunnel for a given floor roughness and spire dimensions with the objective of assisting

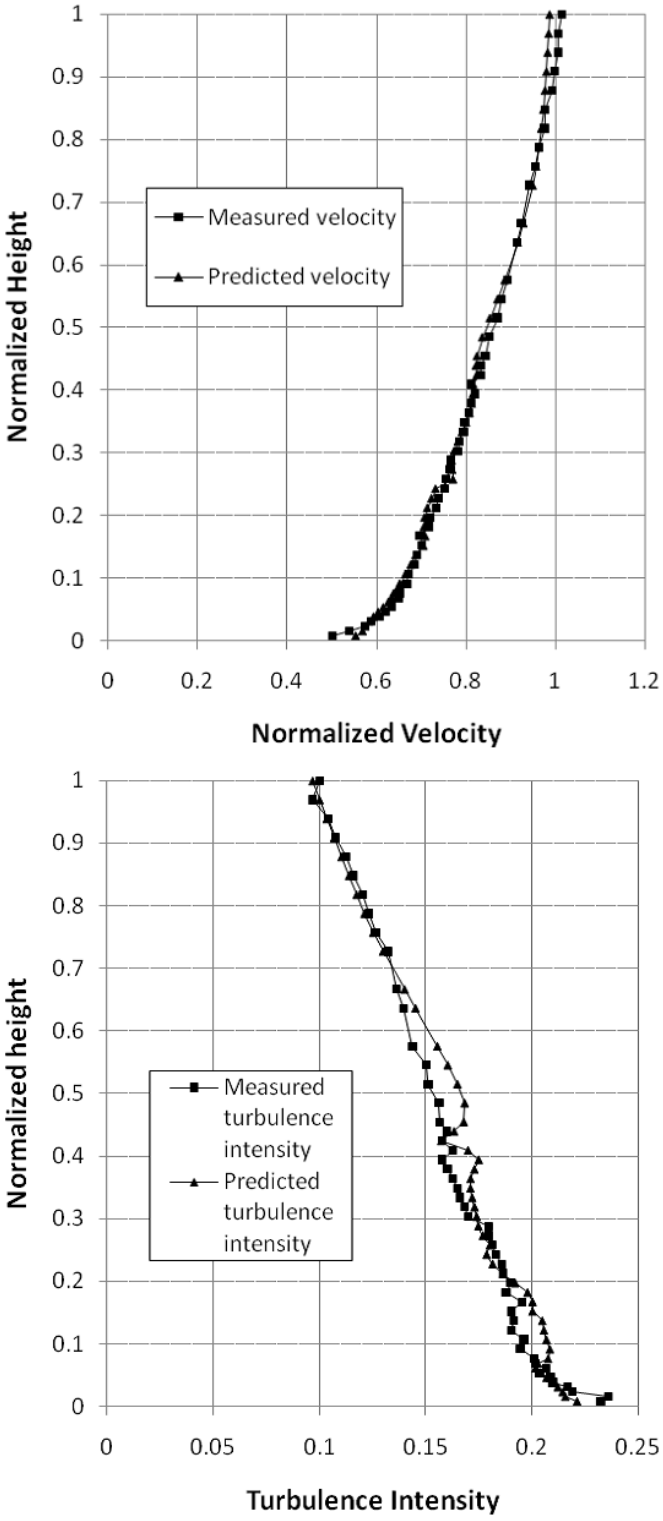


Figure 4.37: Measured versus predicted velocity and turbulence intensity profiles

Table 4.3: Measured and ANN predicted roughness length bottom spire width difference

Test set 1	Actual value	Predicted value
Spire width difference	12in	10.2in
Floor roughness	2in	2.2in
Test set 2		
Spire width difference	5in	6.2in
Floor roughness	1in	1.1in

the flow management process. The neural network model is trained with part of the wind tunnel data collected for various roughness length and spire dimensions. The results predicted by the neural network model have shown excellent agreement with the observed data for both mean longitudinal velocity and turbulence intensity profiles considered in this study. The inverse problem of determining roughness length and spire dimensions has also shown good agreement despite the relatively difficult nature of the problem due to discrete-valued parameters. In future other family error optimization techniques appropriate step functions can be used to improve learning efficiency and performance the inverse ANN models for discrete outputs. The CCNN network is found to be more efficient than MPNN because relatively fewer number of iterations are required for a given tolerance level.

Chapter 5

Numerical evaluation of orographic effects

This chapter focus on evaluating the effect of topographic features such as hills, valleys and escarpments on wind speed and turbulence using Computational Fluid Dynamics (CFD). Wind loading standards provide guidelines to determine wind speed up over hills as a function of the hill slope. The provision is usually for an isolated and symmetrical hill that is a highly idealized scenario (Miller & Davenport 1998). Real topography contains three dimensional topographic features and thus not symmetrical, and also are surrounded by other topographic features and thus not isolated. Design made on complex terrain without considering these deficiencies may be overly conservative in some cases and unsafe in other cases.

First we consider wind speed alone and calculate speed up ratios over many topographic features . We start from simulation on a flat terrain similar to what is done in the previous chapter, and then progressively add topographical features in both 2D and 3D domain. The effect of orography on wind speed is compared by calculating fractional speed up ratios. Multiple topographic features placed one after the other are also investigated to gain insight on sheltering effects. For all the 2D test cases considered, corresponding 3D simulations are carried out using axi-symmetric version of the 2D topographic features and results are compared against each other.

The second part of this chapter discusses turbulence structure over topographic features. Different turbulence models such as mixing-length model, two-equation Reynolds Averaged Navier-Stokes (RANS), and Large Eddy Simulation (LES) models are compared with one another with regard to their ability to predict recirculation zones. Also qualitative comparisons are made with results available in literature. The effect of roughness on wind speed ups and root mean square (RMS) fluctuations is assessed using equivalent sand grain roughness approach. In general roughness impacts RMS fluctuation estimations more than it does wind

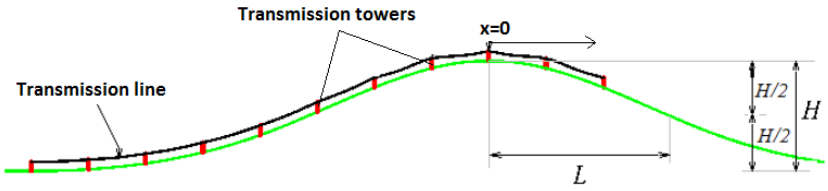


Figure 5.1: Transmission line with multiple towers crossing a hill

speed. Therefore careful consideration of all simulation parameters is mandatory for characterization of the turbulence structure behind topographic features.

5.1 Wind speed up over topography

5.1.1 Building codes and standards

Several building codes and standards incorporate the effect of topography on wind speed using simplified models of isolated two dimensional hills, escarpments and valleys. Design of structures for wind loads requires accurate estimation of wind speed and turbulence intensity at different heights of the site. For infrastructures that span a large length, such as transmission lines, wind speed information is required at many locations. The structure crosses different speed-up regions as shown in Fig.5.1. If the site consists of outstanding orography such as hills and escarpments, the fractional speed up ratio can be high depending on the slope of the orography. Even on hills with gentle slope the speed up can be large enough to cause structural damages if not properly accounted for. Thus many national codes such as National Building Code of Canada (NBCC), American Society of Civil Engineers - 7 (ASCE7), Australian/New Zealand Standard (AS/NZS 1170-2), and European Standard (Eurocode I), provide general guidelines to estimate topography multiplication factors for wind speed over hills and escarpments. As discussed in section 2.2, these codes give recommendations only for simple topographic features. Experimental methods is recommended for a complex terrain that is not covered well in building codes. Methods that can be used to estimate wind speed up factors include : field measurements, boundary-layer wind tunnel testing, analytical methods and numerical methods. This work focuses on a numerical CFD approach to asses the effect of orography features on wind speed.

The Fractional Speed Up Ratio (FSUR), Eq.(5.1), quantifies the effect of orography on the horizontal component of velocity at a given height relative to its value on a flat terrain at the

same height. If there are no topographic features, FSUR should be 1 at every location. At the top and upstream side of hills and ridges, FSUR> 1 indicating a speed up, while FSUR< 1 on the leeward side where back flow occurs. Maximum values of FSUR are reached at crest of hills or a little upstream of it.

$$FSUR = \frac{U(z)}{U_o(z)} \tag{5.1}$$

NBCC defines a relative speed up ratio (ΔS = 1−FSUR) as follows

$$\Delta S = \Delta S_{max}(1 - \frac{|x|}{\kappa_1 L})e^{-\beta z/L} \tag{5.2}$$

where the values of the parameters are taken from the Table 5.1. To show application example

Table 5.1: NBCC parameters for speed up ratio

			κ_1	
Hill shape	ΔS_{max}	β	$x < 0$	$x > 0$
2D ridges(or valleys with $H < 0$)	2.2H/L	3	1.5	1.5
2D escarpments	1.3H/L	2.5	1.5	4
3D axi-symmetrical hills	1.6H/L	4	1.5	1.5

of the above mentioned national codes, wind speed-up ratio ΔS over isolated hills of dimensions L=800, H=200 (steep hill) and L=1600, H=200 (shallow hill) under open country (C) exposure are considered. The formulas provided in the national codes are complicated, thus a program is written to plot speed up factors at different locations with in the lowest 200m of the Atmospheric Boundary Layer (ABL) as shown in Figs. 5.2 - 5.3. We can immediately observe that NBCC and ASCE7 codes give FSUR estimates that are higher than that of AS/NZS 1170-2 and Eurocode I. This is most likely due to the underlying approaches used to generate the codal provisions. Some may have used Boundary Layer Wind Tunnel (BLWT) based methods while others use analytical/numerical approaches.

5.1.2 Numerical studies

A number of numerical studies over complex terrain have been conducted since Jackson & Hunt (1975) first analyzed flow over isolated hills of low slope using linearized forms of fluid flow equations by analytical means. Their approach is still in use for large scale wind mapping where a quick estimation is required for micro-siting or other purposes. One such program developed at Risϕ-DTU is the Wind Atlas Analysis Application Program (WAsP), that includes complex terrain flow model with roughness change, and a separate wake model. On complex

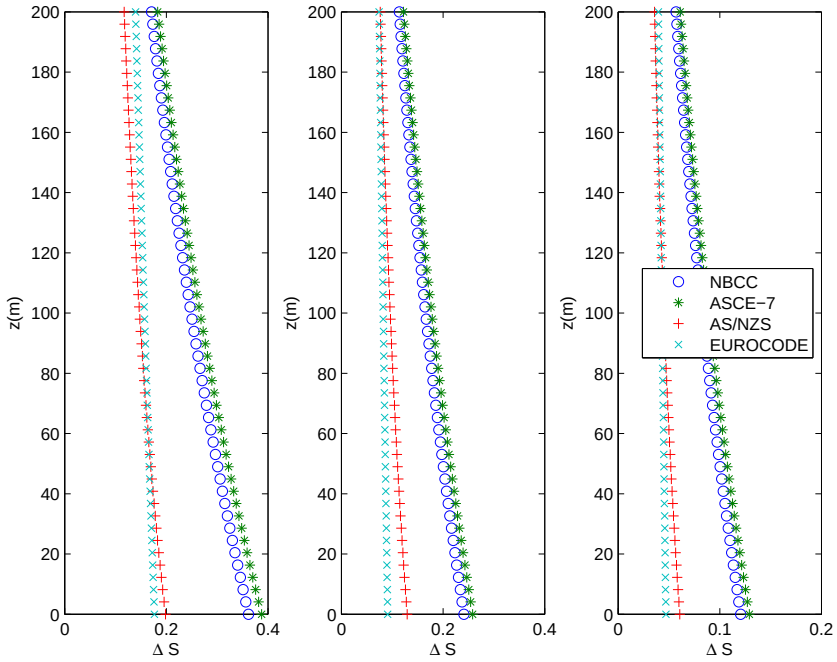


Figure 5.2: Speed up factors at x=0(crest), x = L/2 and x = L of a 2D steep hill using various building codes.

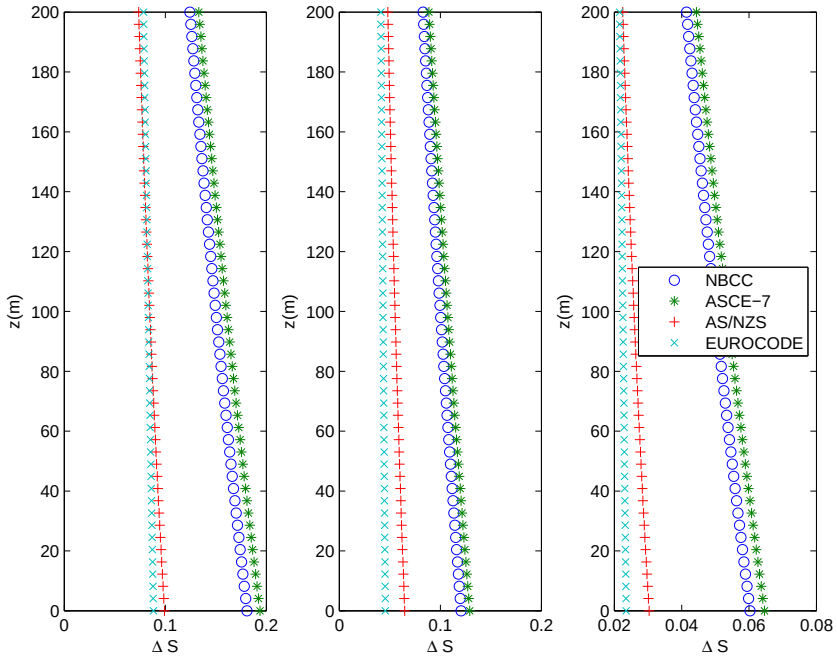


Figure 5.3: Speed up factors at x=0(crest), x = L/2 and x = L of a 2D shallow hill using various building codes.

terrain with high hills and mountains, linear models fail to predict flow separation behind obstacles. Therefore such programs should not be used without modification when flow separation is expected. Castro et al. (2003), Maurizi et al. (1998) conclude that even non-linear steady state numerical models (CFD) have problems in recirculation regions, because orography can induce unsteadiness. The early studies using linear models (Deaves & Harris 1978, Jackson & Hunt 1975, Miller & Davenport 1998) are motivated by limitations in computational resources, but the problem is still present in case of micro-scale wind simulations conducted for micro-siting of turbines. This work investigates the simplest turbulence model, Prandtl mixing length model, besides more complex RANS and LES models. Complex turbulence models such as LES should be used when accurate information is required about turbulent structure and its effects. Some examples of use of LES in literature: pollutant dispersion studies (Lee et al. 2002), complex terrain studies (Dupont et al. 2008, Feng & Fernando 2011, Iizuka & Kondo 2006, Tamura et al. 2007, Tsang et al. 2009), wind loading (Dagnew & Bitsuamlak 2013). There are many studies carried out using RANS turbulence modes: isolated hills (Chung & Bienkiewicz 2004, Takeshi & Hibi 2002), multiple hills in succession(Bitsuamlak et al. 2004, Carpenter & Locke 1999, Lee et al. 2002), real complex topography such as Askervein hill (Rasoulli & Hangan 2013, Stangroom 2004).

5.1.3 Analytical study of flow over low hills

Guidelines for estimation of wind speed up over crest of 2D hills in neutrally stratified flows started with the seminal work of Jackson & Hunt (1975). They derived formulas for estimating fractional speed up ratio (ΔS) for a low hill of arbitrary shape defined by $z = hf(x/L)$ where h and L are the characteristic height and length of the hill as shown in Fig.5.4. L is defined as the upstream length where the height of the hill is half the maximum. A theory is developed for the boundary layer flow over such a hill with surface roughness of z_0 subjected to the following conditions

$$\frac{L}{z_0} \rightarrow \infty$$

$$\frac{h}{L} = \frac{1}{8}\left(\frac{z_0}{L}\right)^{0.1}$$

$$\frac{\delta}{L} \gg \frac{2\kappa^2}{\ln(\frac{\delta}{z_0})}$$

where δ is the boundary layer height. The incident profile is defined with logarithmic law in the boundary layer

$$U_0(z) = (u_*/\kappa)\ln(z/z_0)$$

and a constant value outside the boundary layer

$$U_0(z) = (u_*/\kappa)\ln(\delta/z_0)$$

Also the boundary layer region is divided in to two regions, namely inner and outer region, in which the velocities are calculated differently via 'perturbation' approach.

1. In the inner region of height l, the velocity above the surface of the hill henceforth termed as displacement $\Delta z = z - hf(x/L)$ is calculated from the upstream velocity at same displacement above level ground, and a perturbation of $\Delta\hat{u}$.

$$u = u_0(\Delta z) + \Delta\hat{u}$$

2. In the outer region, the velocity is assumed to be a perturbation of the incident velocity at the same height z, not displacement height Δz.

$$u = u_0(z) + \Delta u$$

   In this region the flow is assumed to be essentially inviscid and thus governed by potential flow theory.

The vertical velocity v can be determined from continuity relations. The boundary conditions in the boundary layer are such that as $z \to \infty$ both Δu and v go to zero, and at $z = l$ the velocities match with that of the inner layer. Then the two dimensional navier stokes equation is linearized by omitting the non-linear terms, followed by substitution of appropriate order scales for the inner and outer region to arrive at the velocity perturbations from which FSUR is determined. Detail mathematical analysis of the solution can be found in the paper. This seminal work has been extended to 3D hills by Mason & Sykes (1979) and used in wind mapping software such as WAsP. The major disadvantage of this method is that it can not be used for steep hill where non-linear models are more appropriate to capture recirculation behind hills. But the fact that solutions can be obtained very quickly makes them still attractive at least for preliminary investigations of micro-siting or similar purposes. Improvements to the model can be obtained by using a better turbulence model than the mixing-length model used by Jackson

& Hunt, avoiding linear approximations of equations and solving the equations numerically instead etc. The current work investigates non-linear models (CFD) for the calculation of speed up ratios and turbulence intensity using different turbulence models.

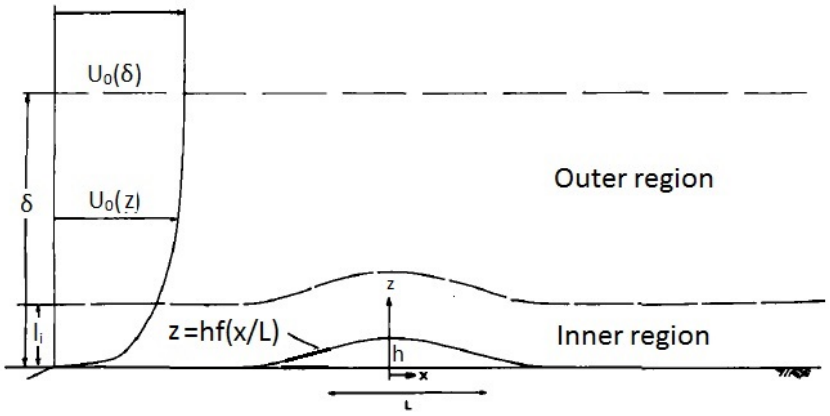


Figure 5.4: Flow regimes for flow over a low hill. Adapted from Jackson & Hunt (1975)

5.1.4 BLWT studies

Miller & Davenport (1998) provided guidelines for the wind speed-up evaluation over complex two dimensional surfaces based on a wind tunnel study. Ishihara et al. (1999) presented the results of measurements of wind speed over a circular hill with a maximum slope of about 62.5%. Cao & Tamura (2007) studied the roughness blocks effect on the atmospheric boundary layer flow over a two dimensional low hill with and without a sudden roughness change. The effects of the roughness blocks were clarified by comparing the flow characteristics over hill models, with emphasis on wind speed-up and turbulence structure. Adding or removing roughness blocks on the hill surface or inflow area changes the velocity deficit and creates a completely different turbulence structure in the wake. Lubitz & White (2007) presented a wind tunnel and field investigation of the effect of local wind direction on speed-up over hills. Other wind tunnel investigations include: Arya et al. (1987), Ayotte & Hughes (2004), Carpenter & Locke (1999), Castro et al. (2003), Ferreira et al. (1995), Finnigan et al. (1990), Gong & Ibbetson (1989), Snyder & Britter (1987).

5.1.5 Description of test cases of the current study

The first case considered is that of flat terrain with no topographic features. While this sounds rather pointless, Richards & Hoxey (1993) and others have demonstrated the difficulty of sim-

ulating wind flow over a featureless terrain. If the inlet wind speed and turbulence intensity profiles are incompatible with wall functions used at the ground, horizontal gradients may develop. Therefore this benchmark case should result in an FSUR of one through out the domain when proper boundary conditions are applied. Any other value indicate artificial speed ups due to inconsistent boundary conditions.

The second set of cases considered are single hills of two dimensions classified as shallow and steep from here on. Both hills have the same height but the shallow hill has twice the length. This geometric configuration has been used by Bitsuamlak et al. (2004), Carpenter & Locke (1999). The expected speed up over a 2D hill is depicted in Figure 5.5. The simulations are carried out at full scale dimensions where the height of the hlls $H = 200$ giving rise to a high Reynolds number (Re_h). This is not a problem when RANS models are used for the simulation, but for LES simulations either the dimension of the hills or the viscosity has to be reduced by the order of 1:1000 to make simulations feasible on current desktop computers.

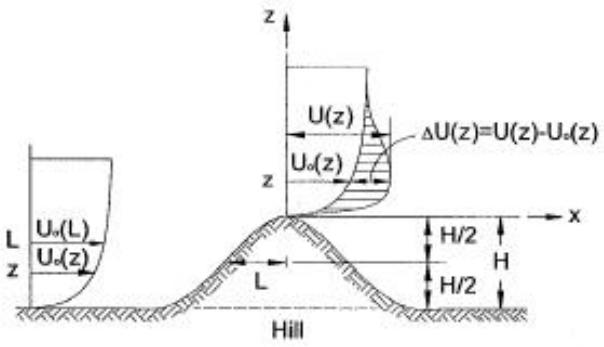


Figure 5.5: Wind speed up over a single hill (NBCC)

Many equations are available to define shapes of hills that may have significantly different effects on the FSUR obtained (Bitsuamlak et al. 2004). This study uses cosine hills with curves defined below. Both hills have the same height(H), and the half length (L) of the hills are $L = 4H$ and $L = 2H$ for the shallow and steep hills respectively. The maximum slope angles are 38^0 and 21^0 for the steep and shallow hill respectively, hence according to Finnigan (1988) flow separation is expected for both cases because the maximum slopes are above critical angle of $\theta_{cr} = 16^0$.

$$z = H(\cos(\frac{\pi x}{L}))^2 \tag{5.3}$$

Some national codes also provide recommendations for 3D hills, hence we consider axisymmetric version of most topographic features considered in this study. It is expected that the wind speed over the hill will decrease on the 3D hill because of more freedom in the span

wise direction, while in the 2D hill case all the fluid has to go over the top of the hill. Depending on the actual shape of the orography, a 2D or 3D model may be appropriate. However the simplicity and conservative FSUR estimate of 2D hills usually makes them preferable in practice.

The third case is that of an escarpment with a constant slope as shown in Figure 5.6. A three dimensional version of this case, a frustum, may be possible but only the 2D case is considered in the present study. The slope of the escarpment is chosen to be 1:2, which is the same setup used by Glanville & Kwok (1997).

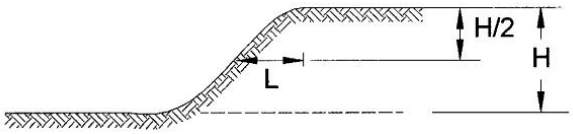


Figure 5.6: An escarpment

The fourth set of cases considered are double and triple hills. Multiple topographic features are not covered in national codes but it is implied that the wind speed up factors applicable for the first hill are to be applied for the following hills as well. The flow characteristics for the second and third hills are fundamentally different from that of an isolated hill, because of flow separation on the upstream hills. As already discussed before, there is a reduction in wind speed associated with more turbulence on the leeward side of the upstream hills. FSUR for the second hill are typically reduced by 20-30%. National codes such as NBCC use an overly conservative approach that may have severe economical consequences in the design of structures for wind loads (Horsfield et al. 2002).

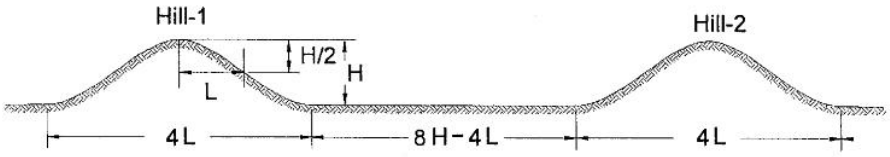


Figure 5.7: Double hills

The last set of cases concerns wind flow over valleys where a slow down is expected unlike the hill cases. A recirculation zone forms inside the valley, therefore wind speed reduces associated with an increase in turbulence similar to what happens on the leeward side of a hill.

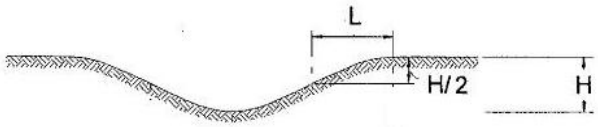


Figure 5.8: An isolated valley

5.1.6 Ground surface representation and mesh generation

The first step in ABL simulation is to prepare a model of the actual terrain as best as possible. While most of the cases considered here are simplistic, the procedure that is followed is applicable to complex geometry cases. It is assumed that geometric information is available in (x,y,z) point-cloud format, from which surface of the terrain can be produced by triangulation methods. The data maybe collected by field surveying and depending on the resolution accurate reproduction of the orography features maybe possible. For large areas that span kilometers ,high resolution LiDAR (light detection and ranging) data can be used if available.

Once the model of the surface is generated usually in STL (Stereo Lithography) format, the computational domain can be meshed with emphasis (refinement) on orographic features. There are many tools available to generate tetrahedral meshes that are suitable for finite element methods, but such non-orthogonal meshes are not suitable for finite volume CFD calculations. Hexahedral (or polyhedral) elements are preferred whenever possible. In most cases purely Hexahedral mesh for arbitrary surface is not possible. Therefore tetrahedral elements with a layer of elements parallel to the surface close to the wall , and Hexahedrons away from the surface are used. The meshing component of OpenFOAM cfd software known as snappyHexMesh is used to mesh an arbitrary terrain.This tool saves the user time by avoiding a lot of manual work, and it is worth explaining the process of meshing using snappyHexMesh. There are four stages in the meshing process. First a background mesh is generated as shown in Figure 5.10, with refinement regions around the hills. Then cells outside of the computational domain are removed. After this stage, the surface boundary is roughly established but it is not smooth enough. Thus a third stage of snapping to the surface is applied by moving vertices. Some of the cells near the surface may be of deformed shapes (tetrahedrals etc). It is crucial to have a body-fitted gridded closer to the wall for convergence and better accuracy, thus a final stage of adding layers of cells parallel to the surface is done.

While it is very difficult to generate a good mesh for an arbitrary 3D terrain, let alone one that satisfies orthogonality requirement, it is possible to produce high quality mesh for the 2D cases as described in Bitsuamlak et al. (2004). The method of meshing used in the present study generates non-orthogonal but body fitted grid. A correction for non-orthogonality is added as

a source term using an approach known as deferred correction for non-orthogonality (Jasak 1996). In both the 2D and 3D grids the grid is stretched in the vertical direction so that the first cell height is roughly equal to $2K_s$.

5.1.7 Computational domain setup

The computational domain is setup following recommendations for the use of CFD in wind engineering (Franke & Hirsch 2004). The domain may be broken down into three regions: upstream, central and downstream regions. The length of the upstream region is fixed at 5H from the center of the first orography. It is recommended to use short distances for the upstream region to avoid horizontal gradients that may develop with inconsistent wall and inlet boundary conditions. In the upstream and downstream regions no obstacles are placed and effect of roughness is taken care of through wall function modifications. The mesh in the central region may be refined to capture the change in wind flow characteristics that occur there. The outlet of the domain is placed far away at 12H from the last orography so that zero gradient boundary condition can be assumed for all flow quantities: $\partial\phi/\partial x = 0$. The distance between hills is fixed at 8H-4L which is zero for the shallow hill case and 4L for the steep hill case. This separation is selected to compare with results available in literature. The sides and top of the computational domain are placed 6H from the center of the hill to reduce blockage effect. The

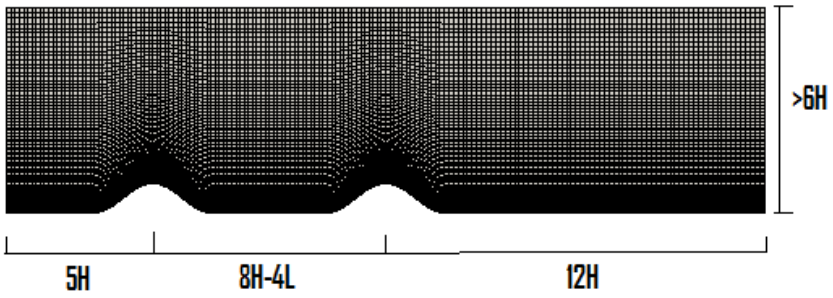


Figure 5.9: Computational domain for double 2D hills

boundary conditions to be applied are as follows. At the inlet the Richard and Hoxey equations for an open terrain roughness of $z_0 = 0.024$ are used. At the outlet zero gradient is assumed for all flow quantities. At the top of the boundary a Dirichlet boundary condition is assumed in which the horizontal component of velocity is fixed to gradient velocity U_g. If the top of the domain is not placed at sufficient distance from the hill top, symmetry boundary condition should be used. The fact that the gradient velocity is known and that a driving shear stress is required to avoid horizontal gradients makes Dirichlet boundary condition appropriate. At the

sides a symmetry boundary conditions is still used.

5.1.8 Grid independence study

Grid independence study for a single 2D hill case is carried out as a benchmark for selection of grid sizes for the other cases. The number of cells in the vertical and horizontal directions are changed to estimate their effect on speedup at the hill top. The speed up values obtained for the different cases are more or less the same. In all the cases stretching in the vertical direction is done in such a a way that the first cell size is the same for all cases. The result for the coarsest mesh is not far away from the result for fine mesh confirming grid independence. A zoomed in plot of velocity at the top of the hill is shown in Figure 5.11 that indicates coarser meshes tend to slightly underestimate speed up. Using extra cells in the vertical direction gives better results than using them to resolve along wind flow. This is due to the fact that resolving near wall flow with high velocity gradient in the vertical direction requires finer grids. For example a coarser 117 X 72 grid shows better performance than a finer 294 X 36 grid because the latter though finer, applied fineness in the wrong direction. However, it is not possible to refine indefinitely in the vertical direction because of limits imposed by the wall function treatment, compatibility of wall roughness with inlet profiles, aspect ratio of cells close to wall etc. In general using more number of cells improves solution,but computation time becomes a constraint. It is possible to solve the 2D cases considered in this work with the finest grid considered, however simulations over 3D topography and/or complex turbulence models such as LES will require major grid optimization to get results within a reasonable time frame.

5.1.9 Results and discussion

First the benchmark case of an empty fetch of 17H X 7H = 3400m X1400m is analyzed with inlet boundary conditions as specified in 4.1-4.3. The velocity profile is more or less sustained throughout the domain as shown in Figure 5.20. The FSUR is 1 in most of the domain except towards the ground where it is difficult to sustain the inlet profile. The inlet velocity profile used for this case follows the log-law equation through out the height of the domain which is rather unrealistic for a height of 1400m. The only reason for this choice is to be consistent with Richards & Hoxey (1993) rough wall functions that have a log-law format. For the other cases to be considered, a gradient height of 270m above the ground is assumed above which the velocity is assumed constant. A disadvantage of the later is that the discontinuity in the velocity gradient at 270m is felt downstream, as will be clear in forthcoming plots of velocity

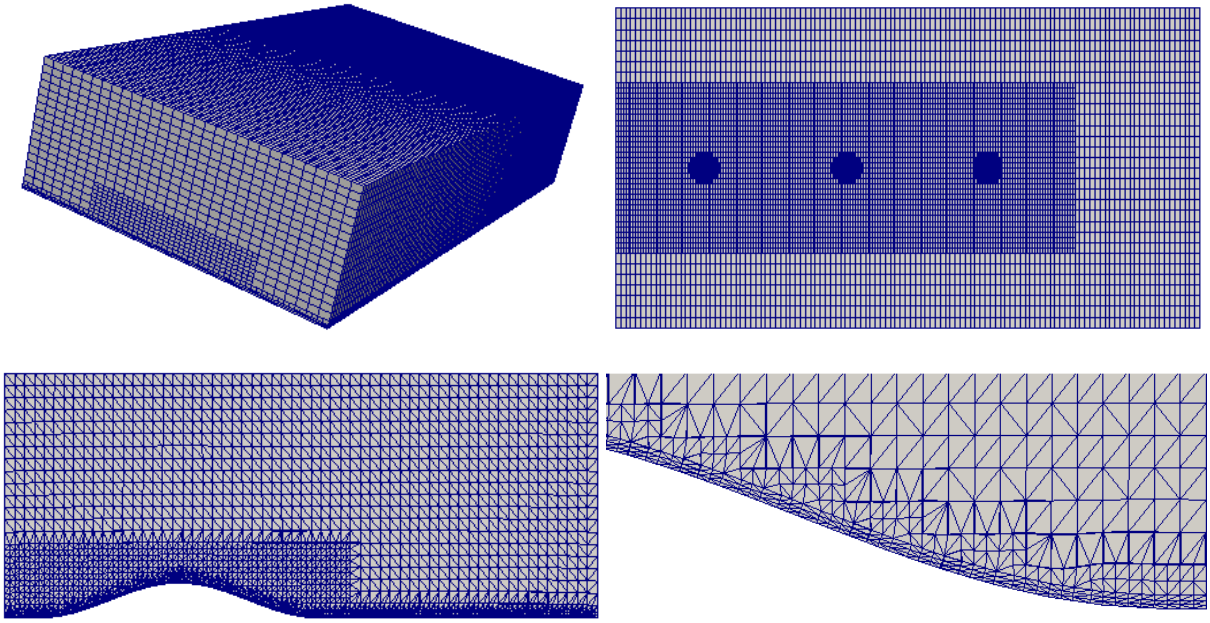


Figure 5.10: Mesh refinement around hills: Background mesh (top-left), box refinement around hill (bottom-left), Planar view of refinement for triple hills (top-right), and close up view of layers towards the ground (bottom-left).

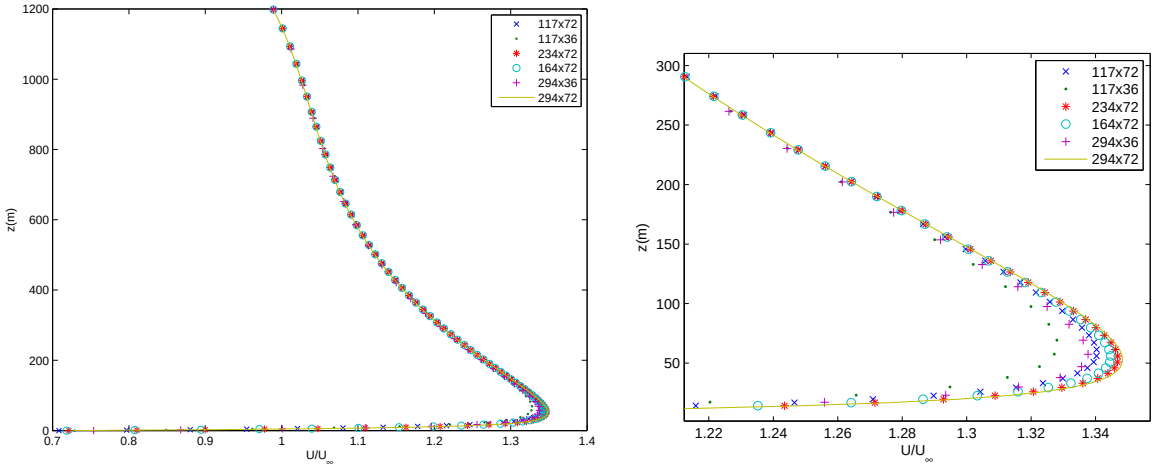


Figure 5.11: Grid independence study on single 2D hill: wind profiles at crest (left) and close-up view of maximum speed up region(right)

profiles.

The second 2D orography considered is an escarpment with a slope of 1:2. Flow separation and recirculation are observed at the top and foot of the escarpment as shown in Figure 5.21. Speed up factors of 2 and 1.8 are observed at 10m and 30m above the crest of the escarpment. Larger values of FSUR may be found at lower depths but those are not to be trusted. In general the values obtained in the present study match with published literature such as CFD solutions of Carpenter & Locke (1999), and analytical solutions of Weng et al. (2000).

From here on, the cases are analyzed using both 2D and 3D orography model. As discussed previously, it is expected that the FSUR values for the 3D cases will be lower than the 2D hills and this is exactly what is obtained for all cases. Plots of FSUR along the center line of the hills at 30m from the ground are shown in Figures 5.12-5.19. A difference of about 20% is observed at the extreme points of hills and valleys where the difference is the largest.

Results for isolated shallow and deep hills are given in Figures 5.12-5.13. A gradual increase of wind speed up the hill followed by a slow down and recirculation on the leeward side are observed. The color plots for the 3D case also show that FSUR reaches peak values on the sides of the hill as well. Th recirculation zone behind the steep hill is larger than behind the shallow hills. Also the 3D cases simulations show a much smaller recirculation zone than their 2D counterparts. After a drop due to recirculation, the FSUR gradually increases to 1 on the downstream. At the outlet ,which is 12H away from the lee of the hill, FSUR reaches values of greater than 0.9 for all the cases. The peak FSUR at 10m above the crest are 1.6 and 1.8 for the shallow and steep 2D hills respectively. The 3D cases show lower values ,by about 15%, of FSUR at the top of the hill as expected.

The next set of simulation results is that of double hills Figures 5.14-5.15. The purpose of this simulation is to determine by how much the FSUR drops from the first hill to the second. For the 2D simulations a drop of about 20% is observed, slightly larger for the steep hill case. This is in accordance with the result of Bitsuamlak (2004). However the 3D simulations do not show that big of a drop which can be explained by reduced sheltering in the lateral direction. For the shallow hill cases where the second hill starts off where the first one stopped, there is a continuity in the FSUR from the first to the second hill. But in between the steep hills there is a long recirculation zone where the FSUR remains roughly constant.

The case of triple hills is investigated further to see if there is further drop in FSUR. The results ,Figures 5.16-5.17,show that there is not a significant drop in FSUR from the second to the third hill. This is again in accordance with results from literature. Therefore this shows that the approach taken by NBCC to design structures on sheltered hills for the same wind load as

the those on the first hill is an overly conservative approach.

The last set of cases analyzed are single valleys with big recirculation zones. Analysis of separated flow requires a better turbulence model as is done in Bitsuamlak (2004) but the standard $k - \epsilon$ model is used for the present study. The wind speed first decreases reaching negative values in the case of steep valley, and then the wind is sped up the second slope reaching or exceeding FSUR = 1 at the ridge. The 3D valleys show less recirculation similar to the case with hills.

5.1.10 Conclusions

A numerical procedure for computing speed up factors for different orography have been described, starting from meshing a complex terrain to post-processing the results. The results obtained using CFD procedure are generally in agreement with those found in literature. Three dimensional orography show reduced FSUR and also reduced recirculation in the lee compared to their 2D counterparts. Steeper slope leads to higher speed up factors over the crest and larger re-circulation zones. The reduction in wind speed up due to sheltering effects from hills in succession has also been investigated. While there is a significant drop in FSUR from the first to the second hill, not much drop is observed from the third hill onwards.

The result obtained from 3D simulation is significantly different from that obtained from 2D simulations to justify the associated cost of simulation. In general the speed up on 3D topographic features are found to be less than those obtained from corresponding 2D simulation. This is attributed to the fact that the flow has more freedom in the lateral direction in a 3D simulation.

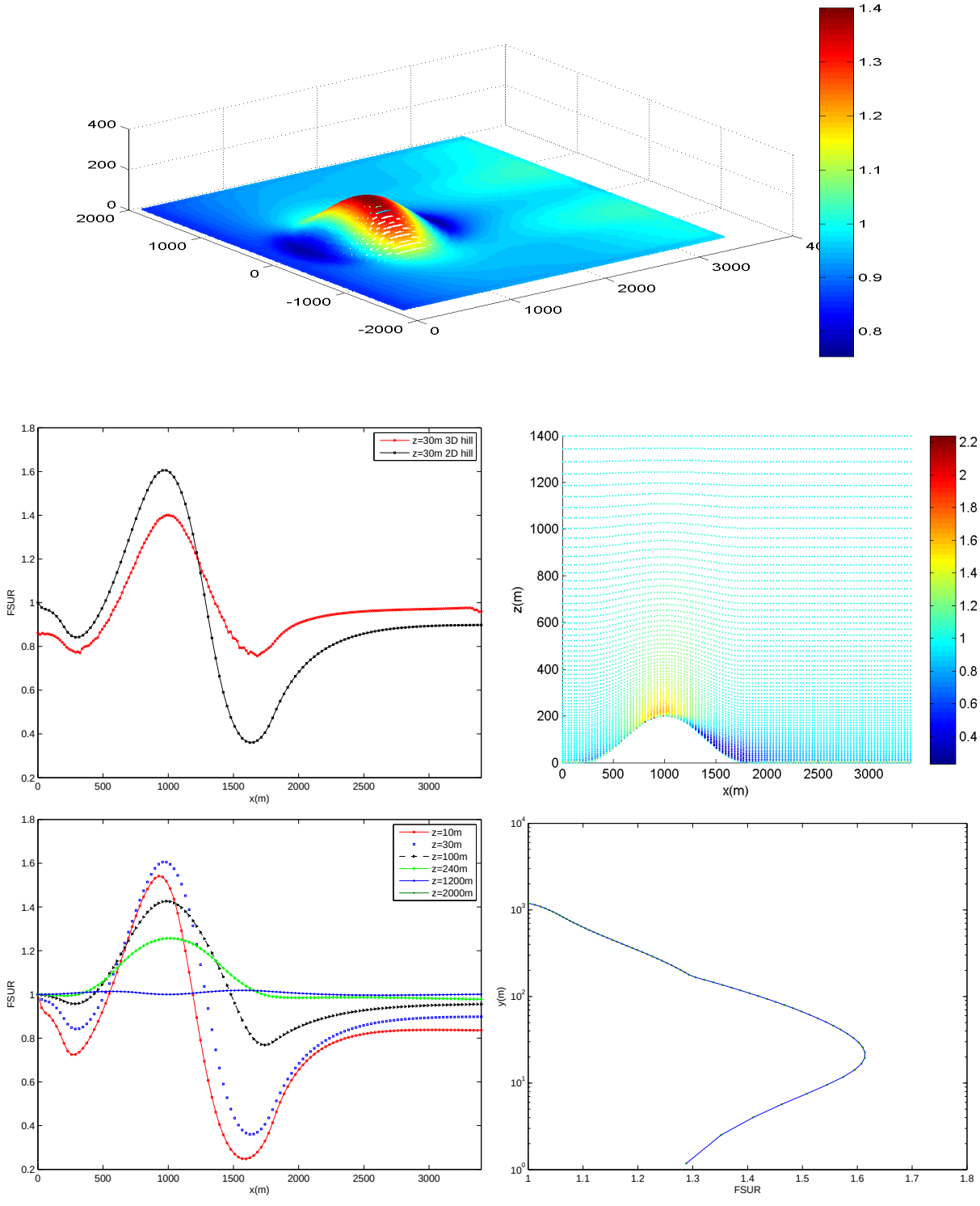


Figure 5.12: Single shallow hill FSUR color maps and line plots and comparison of 2D and 3D simulation results

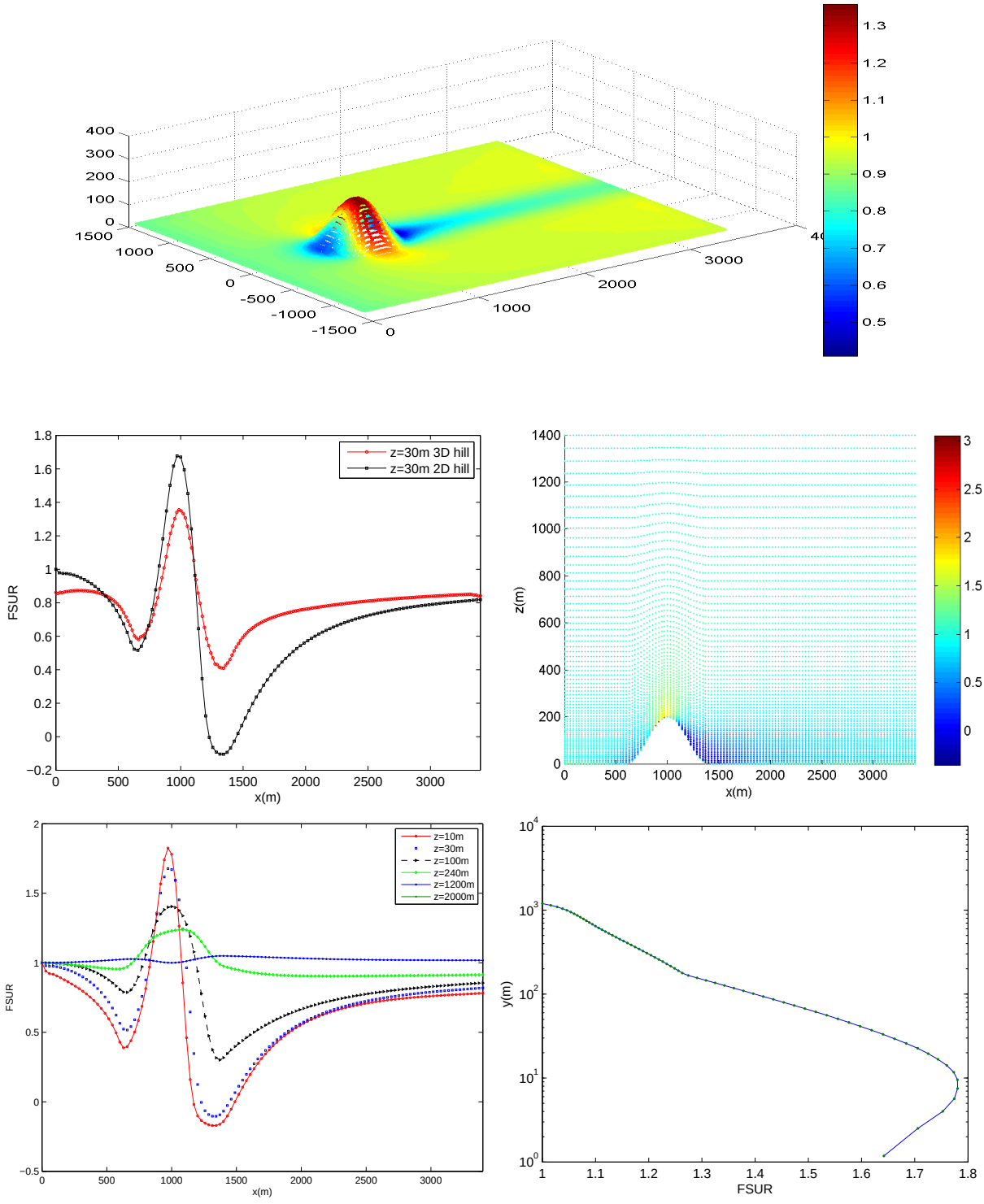


Figure 5.13: Single steep hill FSUR color maps and line plots and comparison of 2D and 3D simulation results

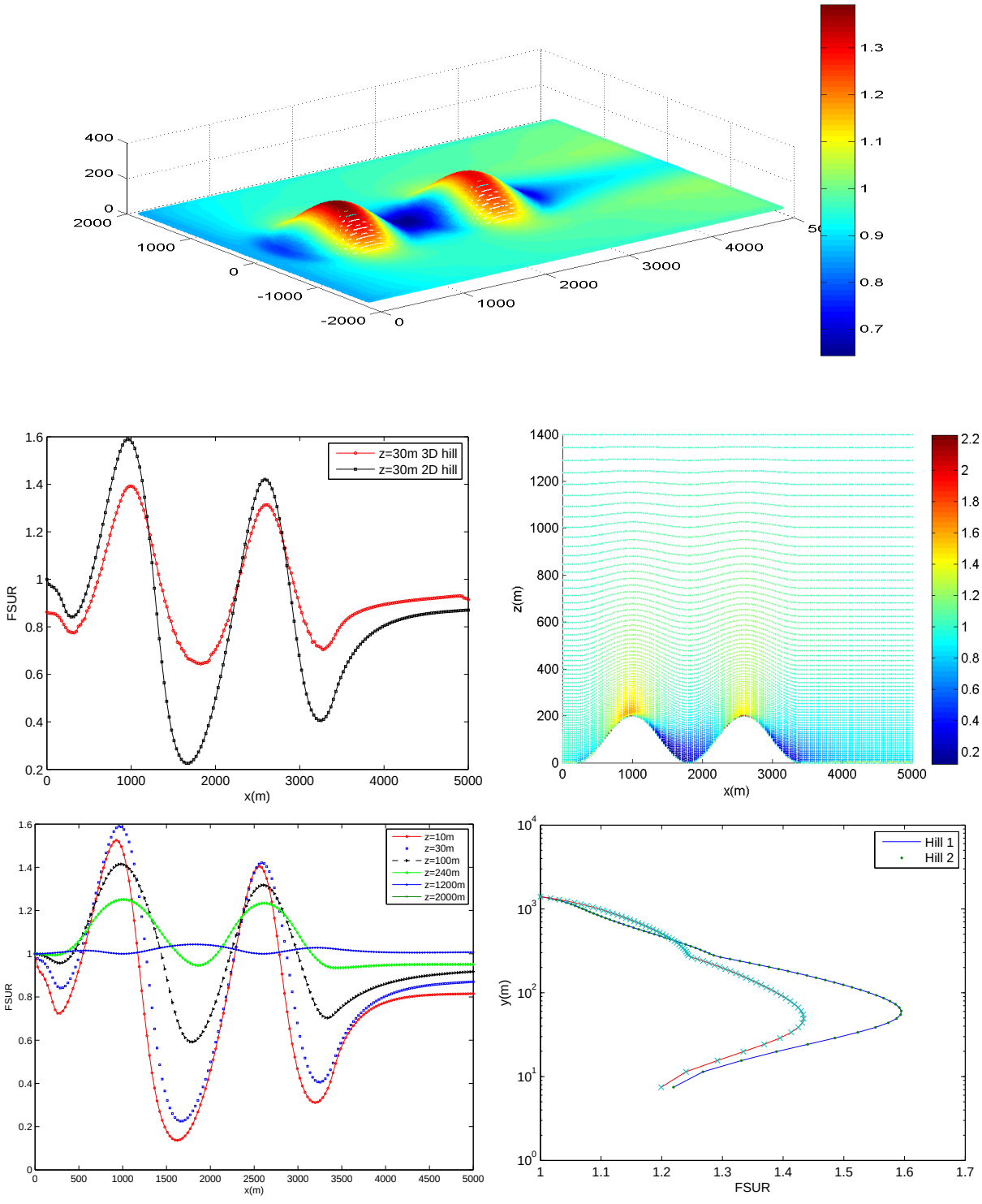


Figure 5.14: Double shallow hills FSUR color maps and line plots and comparison of 2D and 3D simulation results

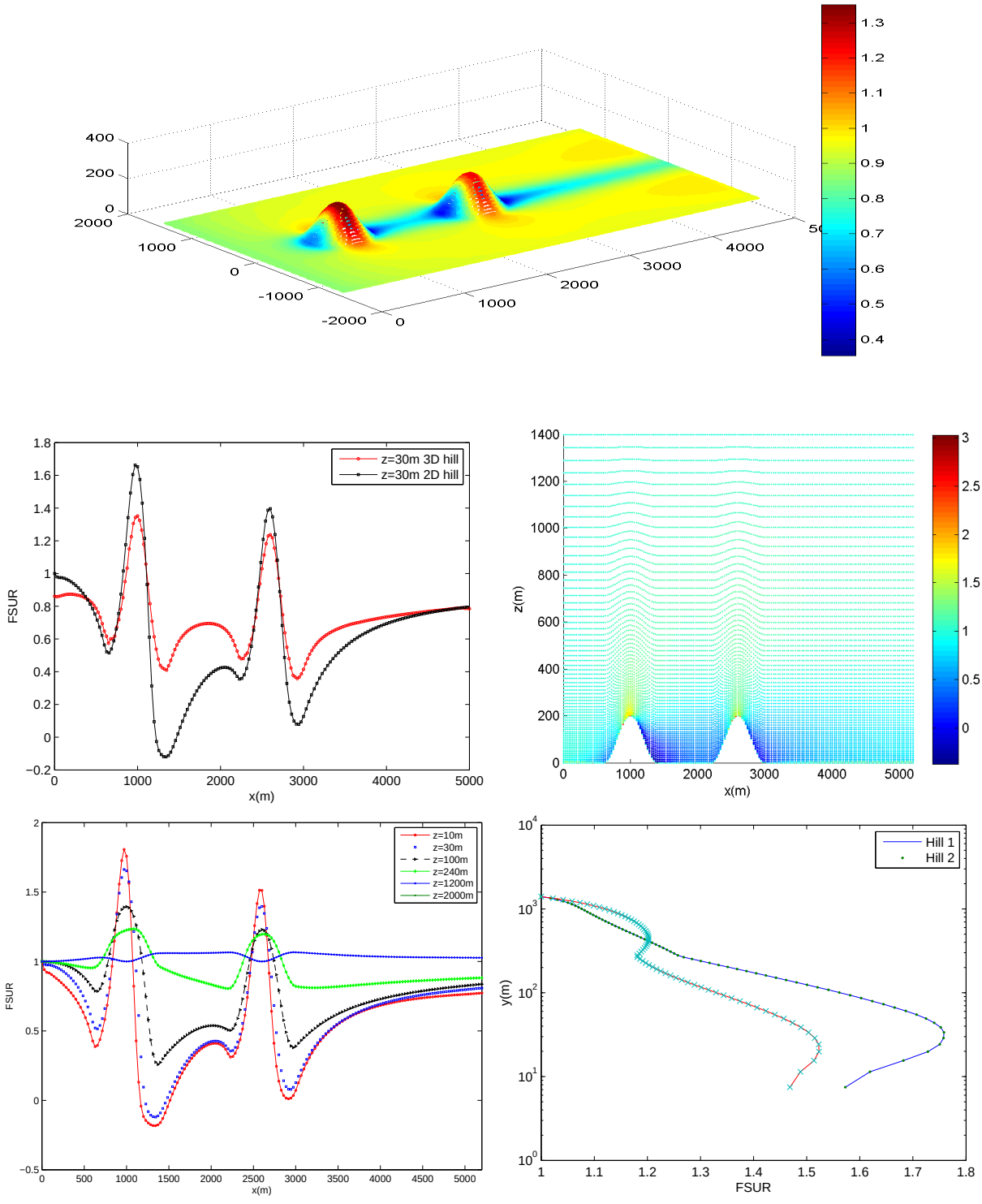


Figure 5.15: Double steep hills FSUR color maps and line plots and comparison of 2D and 3D simulation results

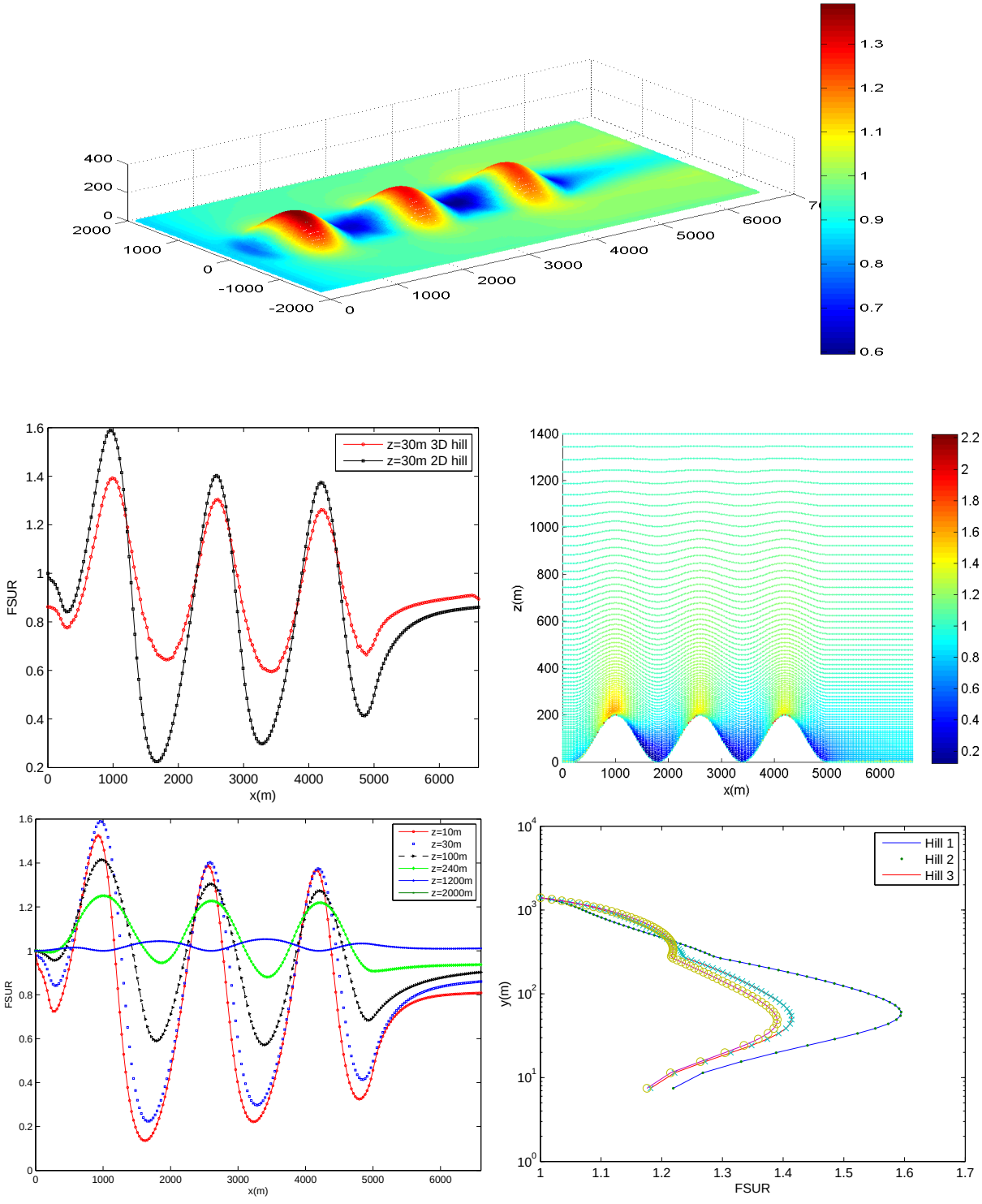


Figure 5.16: Triple shallow hills FSUR color maps and line plots and comparison of 2D and 3D simulation results

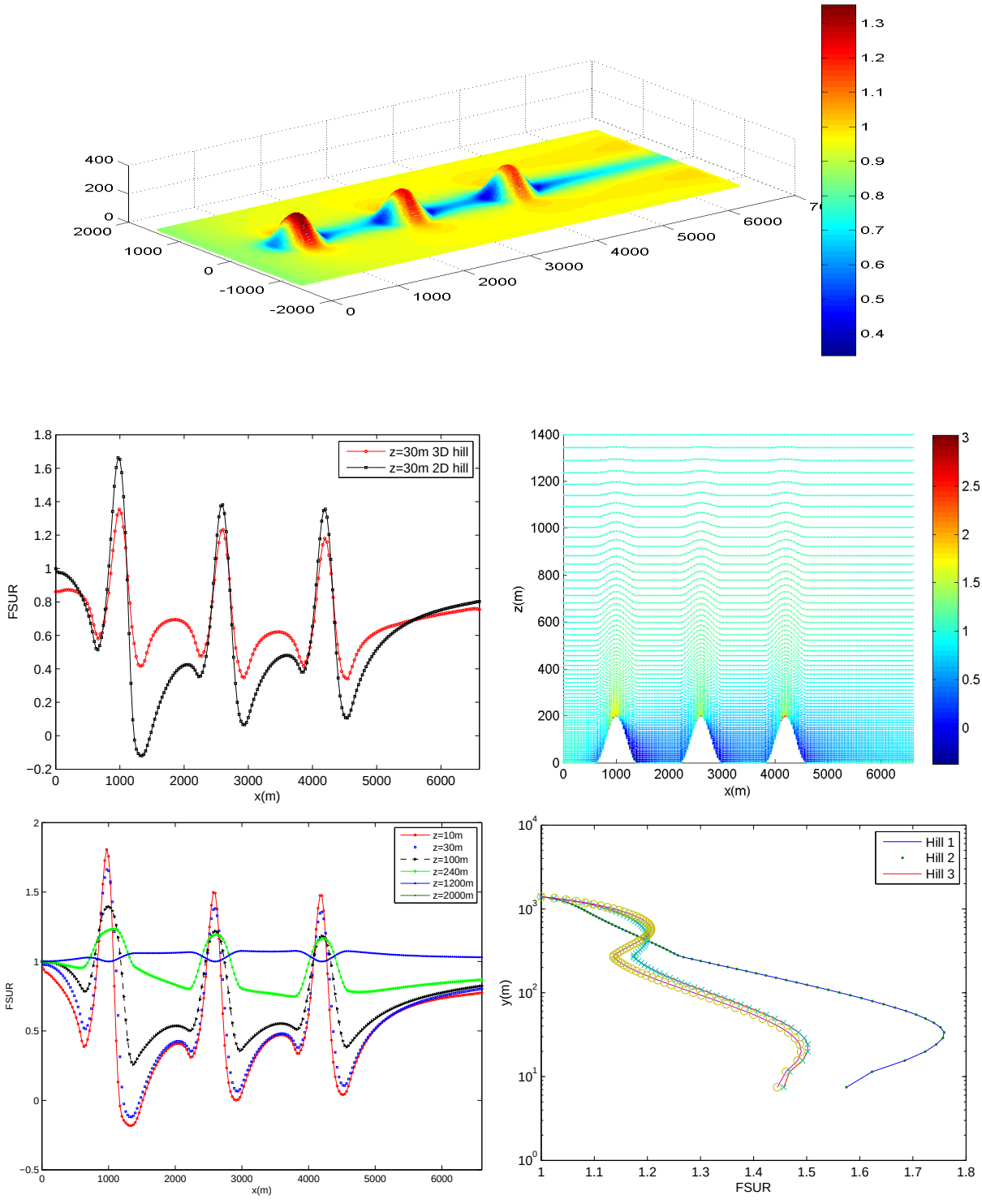


Figure 5.17: Triple steep hills FSUR color maps and line plots and comparison of 2D and 3D simulation results

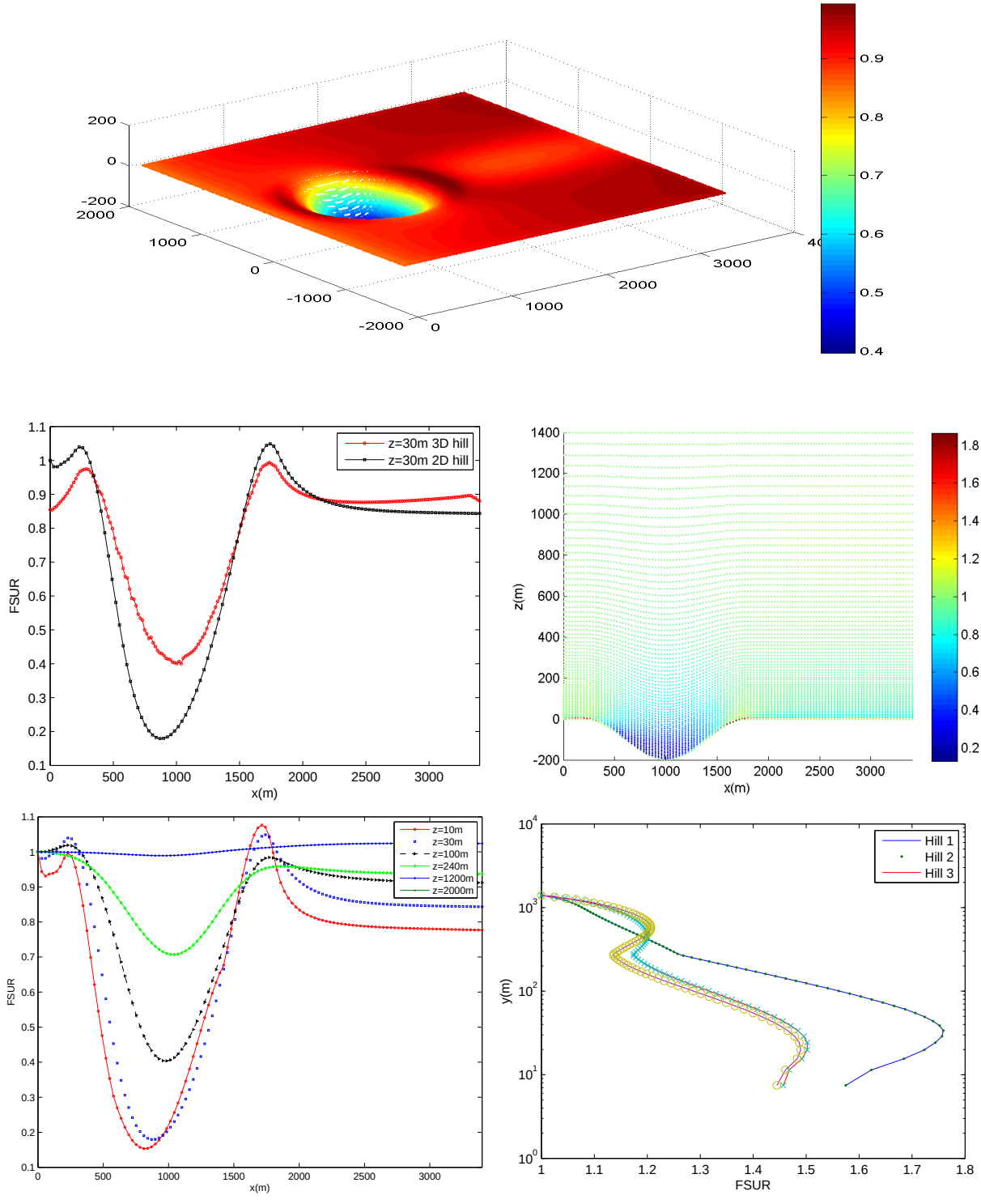


Figure 5.18: Single shallow valley FSUR color maps and line plots and comparison of 2D and 3D simulation results

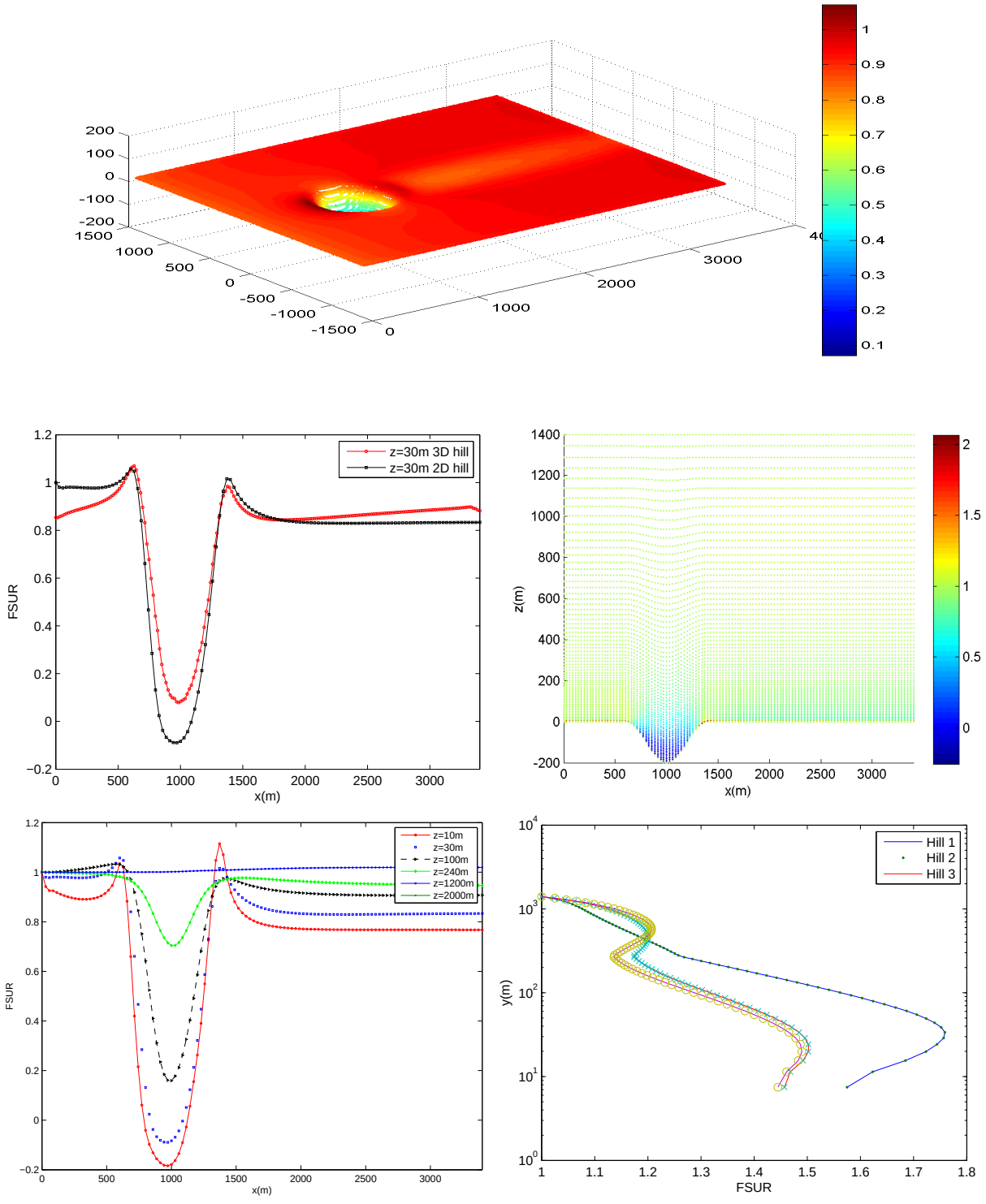


Figure 5.19: Single steep valley FSUR color maps and line plots and comparison of 2D and 3D simulation results

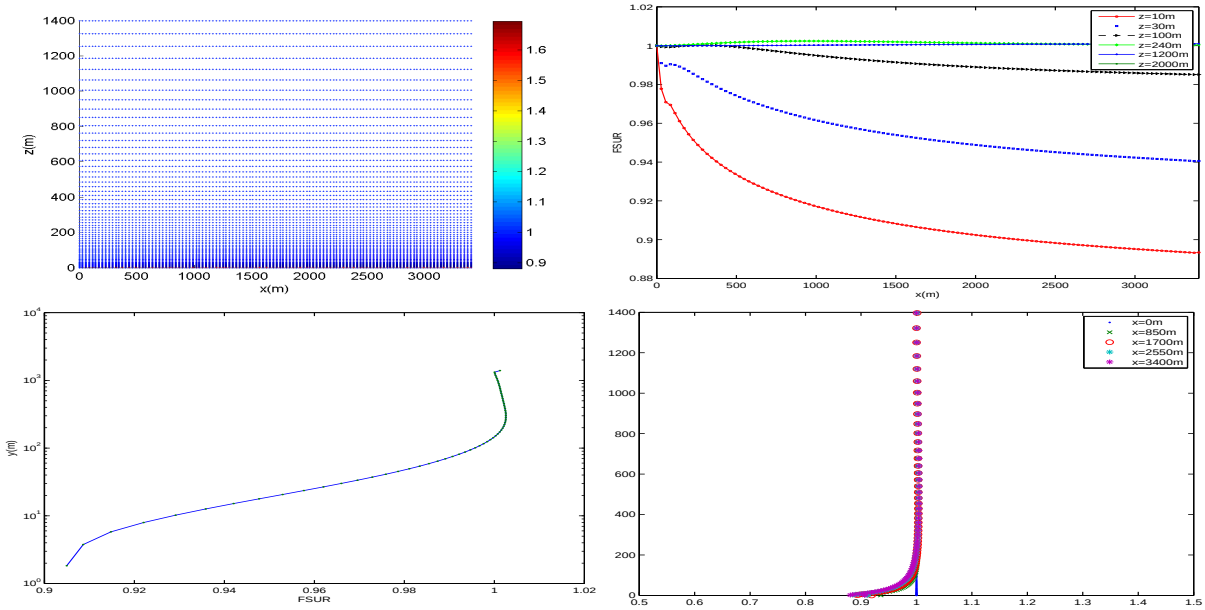


Figure 5.20: Empty domain FSUR color maps and line plots

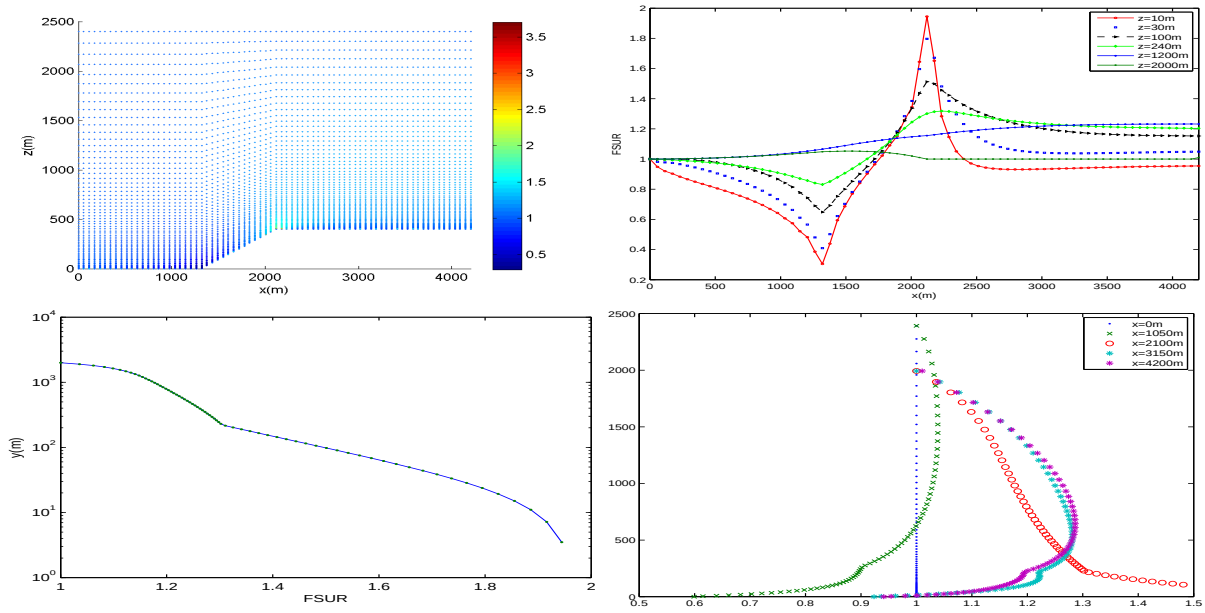


Figure 5.21: Escarpement FSUR color maps and line plots

5.2 Turbulence structure

5.2.1 Background

A fact that is usually overlooked in design codes is that turbulence intensity profiles are also significantly increased over crests of hills as much as wind speed. Some codes either ignore this fact or make suggestions for the overall turbulence intensity to be reduced. Miller & Davenport (1998) argue no allowance for reduction of turbulence intensity should be made at the very least, given the significant increase in local turbulence intensity at crest of hills as shown in Figure 5.22. The horizontal turbulence intensity increases significantly from the first to the second hill, and also the same phenomenon is observed for the vertical turbulence intensity as well. Miller & Davenport (1998) also used computational approach to evaluate speed ups using

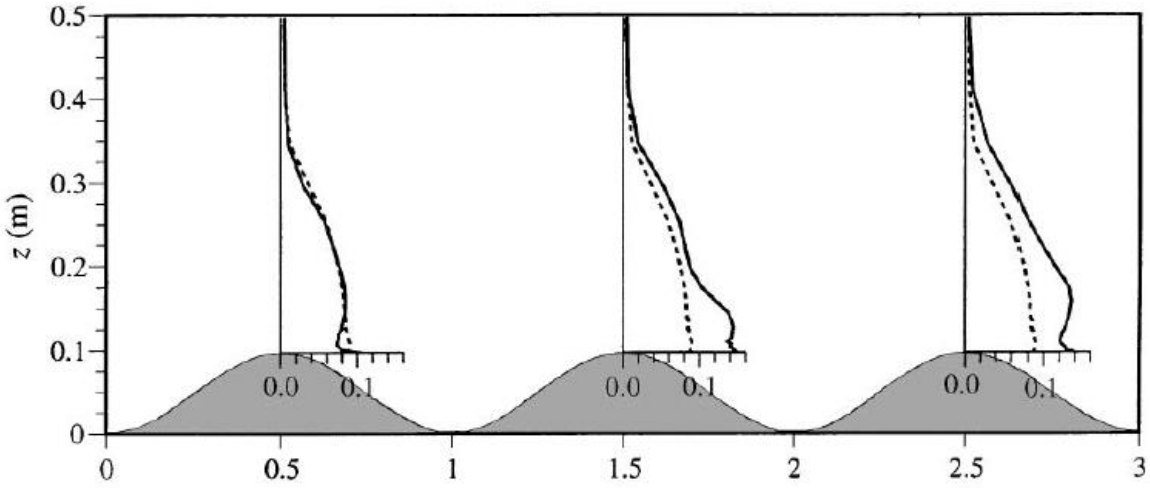


Figure 5.22: Horizontal velocity fluctuation on upstream(dotted) and crest(solid) of sinusoidal hills (Miller & Davenport 1998)

a Mixed Spectral Finite Difference (MSFD) method. The method is computationally economic and can give good predictions on speed ups but it is limited to hills with slop not more than 30%. The current work also investigates accuracy and economy of different turbulence models for general computation of turbulence intensity over complex terrain. Carpenter & Locke (1999) have also investigated flow over multiple hills using wind tunnel and CFD approaches. They concluded that CFD shows good agreement with experimental data for the mean flow quantities but the agreement for RMS fluctuation was poor. Takeshi et al. (1999) have conducted wind tunnel studies to evaluate the turbulence structure over a steep cosine-squared hill with a slope of 32^0. They observed pronounced wind speed up at the midway slope besides the one that occurs at the crest. Then the flow separates at the crest and re-attaches at the lee foot.

The variation of turbulence structure over the hill is better described from the plots of normal Reynolds stress components as shown in Fig. 5.23. This work also produces similar plots for Reynolds stress components for multiple topographic features from CFD analysis. Roughness

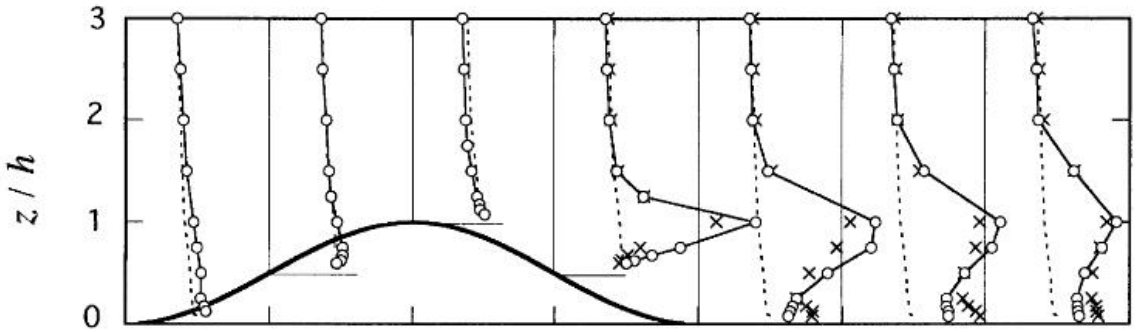


Figure 5.23: Horizontal normal stress σ_h/U_∞ profiles(Takeshi et al. 1999)

has a more severe effect on the turbulence structure than on the wind speed. Shuyang & Tetsuro (2006, 2007) have investigated this effect of roughness on wind flow over hills using wind tunnel experiments. Roughness is modeled by placing blocks over the hills. In analytical and numerical methods roughness is usually modeled using Nikurdase's approach where the roughness is assumed to be continuous and dense. Real hills are usually covered by isolated and relatively larger roughness blocks, and also roughness changes in the wind direction are common. Thus experimental investigation gives more accurate results against which analytical and numerical methods could be compared. They concluded that for low hills of up to a slope of 0.21 , the roughness conditions greatly influence the turbulence structure.

Atmospheric boundary layer simulations are usually carried out using RANS or other more economical turbulence models. Some work on the use of large eddy simulations (LES) to study turbulence structure over hills can be found in Dupont et al. (2008), Feng & Fernando (2011). In general the accuracy of LES technique and sub-grid scale models is not well studied. Feng & Fernando (2011) have tested Smagorinsky and Lagrangian dynamic SGS models against experimental results. They have found that the Smagorinsky model grossly over-predicts the size of the re-circulation bubble behind hills. Also the Smagorinky model under-predicts the speed up at the crest. The dynamic models improved the latter problem but some of the dynamic models also under-predicted the size of the recirculation bubble. The simulations are usually carried out on a model scale that reduces the Reynolds number by orders of magnitude.

5.2.2 Turbulence models

The choice of turbulence model is important for the simulation of the separated flow behind hills. When considering ABL flows, RANS models have the most appeal due to their low computational cost while still providing reasonable results close to experimentally observed results. The most commonly used RANS model is the standard k-epsilon model, which is known to have problems in adverse pressure gradient conditions. Modifications to the model to improve its performance in that regard resulted in RNG k-epsilon and Realizable k-epsilon models among others. The wall functions used can also be modified to consider effect of pressure gradient on velocity profile. Advanced turbulence models such as LES and DNS can be used to resolve the flow all the way to viscous layer. This usually requires too much computational cells to be feasible for practical flows with high Reynolds numbers. Hybrid models of RANS and LES have been proposed to get the best of the two approaches. Wall functions or RANS models can be used to model near wall flow, while the more accurate LES model is used away from the wall to resolve large scale eddies.

In both RANS and LES approaches, the flow equations are solved for averaged quantities (temporal, ensemble or spatial), while the effect of turbulence or sub-grid scales is modeled. The RANS models approximate the unknown Reynolds stress terms using turbulence viscosity hypothesis. The turbulence viscosity ν_t is determined from representative velocity and length scales of the largest energy carrying eddies.

$$\nu_t = u_* l_* \tag{5.4}$$

$$R = \nu_t |S| = \nu_t \sqrt{2\overline{S}_{ij}\overline{S}_{ij}} \tag{5.5}$$

On the other hand subgrid scale stress models of LES model the smallest unresolved scales hence the length and velocity scales are chosen to represent those smaller scales instead. Therefore selection of turbulence model in RANS is relatively more important than that for LES. Different turbulence models used in this study are briefly discussed in the following sections.

5.2.2.1 Mixing length model

The mixing length model is the simplest turbulence model that is known to give good results for simple two dimensional flows such as wakes, jets, mixing layers and boundary layers. The length scale is dependent on the type of flow. For boundary layer type flows with high Re and zero-pressure gradient, Prandtl's mixing length $l_m = \kappa y$ gives a good approximation. However

in adverse pressure gradient conditions such as the wake behind the hill, a stable boundary layer assumption is incorrect thus mixing length models do not work well.

$$l_* = l_m \tag{5.6}$$

$$u_* = l_m|S| \tag{5.7}$$

$$\nu_t = l_m^2|S| \tag{5.8}$$

The mixing length for a boundary layer can be modified to incorporate the viscous and buffer layers as well. For ABL flows with high Reynolds number the boundary layer is very thin thus the linear approximation for mixing length is acceptable.The distance from the ground surface y can be obtained by solving the following differential equations proposed by Spalding (1994).

$$\nabla.\nabla\phi = -V \tag{5.9}$$

$$y = \sqrt{\nabla\phi.\nabla\phi + 2\phi} - |\nabla\phi| \tag{5.10}$$

The boundary conditions for ϕ are Dirichlet at ground surface and Neumann elsewhere. This partial differential equation is solved only once at start up, similar to orthogonal grid generation, hence it is not as costly as solving an additional set of turbulence equations for instance.

5.2.2.2 K-epsilon models

The first improvement to the mixing length model is to calculate the velocity scale from the turbulent kinetic energy k. One equation turbulence models solve a transport energy equation for k from which velocity scale is determined.

$$u_* = ck^{1/2}l_m \tag{5.11}$$

To make the model complete, i.e. one that does not require flow dependent specification, the length scale has to be calculated from the flow as well. Two equation models such as k-epsilon and k-omega solve one additional transport equation for turbulence dissipation or similar quantity to determine time/length scales. The most commonly used two equation model in practice i.e. standard k-epsilon model is known to give very good results in many engineering

applications. However it is known to give inaccurate results in regions of high acceleration / deceleration such as flow separation points and wake regions, where turbulence production is over predicted. To address this problem many modifications to the standard model have been proposed. The simplest of which is an ad hoc modification suggested by Kato and Launder to replace one of the S's in equation 5.5 by vorticity ω.

$$R = \nu_t \sqrt{2\overline{S}_{ij}\overline{\omega}_{ij}} \tag{5.12}$$

More formal approaches to the problem have lead to different RANS models with moderate degrees of success. Renormalization group k-epsilon and Realizable k-epsilon models have been tested in this study to evaluate the re-attachment length of flow behind a single two dimensional hill.

5.2.2.3 LES models

In LES the effect of the larger eddies is explicitly solved while that of smaller scales is modeled using an eddy-viscosity approach similar to that used in RANS models. The major difference with RANS is that LES models the smallest scales that are below a certain filter width. In finite volume calculation the grid itself is usually taken as a filter for convenience. The simplest subgrid scale stress models (SGS) is that of Smagorinsky first developed for meteorological applications. The model is similar to mixing length model where the length scale is substituted by a new dimension calculated from the grid itself as shown in formulas below.

$$l_m = C_s\Delta \tag{5.13}$$

$$\Delta = \sqrt[3]{V} \tag{5.14}$$

The Smagorinsky coefficient C_S is determined experimentally to be usually between 0.1 and 0.2.The length scale at the surface of walls should be zero but the above equation for l_m gives non-zero values. A damping function can be used to reduce the length scale towards zero close to the wall. This can be achieved by integrating Prandtl mixing length to the model as follows.

$$l_m = min(C_s\Delta, \kappa y) \tag{5.15}$$

For low Reynolds number flows with thick viscous and buffer zones, Van Driest damping can be applied as follows.

$$l_m = min(C_s\Delta[1 - exp(-y^+/A^+)], \kappa y) \tag{5.16}$$

For high Reynolds number flows, LES with near wall resolution is very costly. The number of grids required is estimated to be about $Re^{1.76}$ (Pope 2000). Wall models can be used to reduce this cost. Unlike wall damping modifications that are applied to all control volumes, wall functions are applied only to the cell closest to the wall. If wall functions are used, the length scale and hence the filter width become in order of flow length scale. As a result, the number of grids becomes independent of the Reynolds number.

5.2.3 Wall models

High Reynolds number flows have thin viscous layers that necessitates use of very fine grids to resolve all near wall behavior. To reduce computational resources, wall model are usually used in practical high-Re flows. Flow quantities at the first cell nearest to the wall are directly specified to satisfy the universal log-law equation, instead of being solved. There are two approaches of wall function implementation. In the first approach, named the standard wall function, the first cell close to the wall is placed in the logarithmic region ($y^+ \geq 30$). The friction velocity u_* is then calculated iteratively from the log law equation using U_p and y_p of the first cell. Then the wall shear stress $\tau_w = \rho u_*^2$ can be directly specified as a source term in the momentum equation, or equivalently in the form of modified effective viscosity.

$$\frac{U}{u_*} = \frac{1}{\kappa}\ln(\frac{Eu_*y_p}{\nu}) \tag{5.17}$$

$$\tau_w = \rho u_* u_w = \frac{\rho u_* U_p \kappa}{\ln(Ey^+)} \tag{5.18}$$

The turbulence kinetic energy k and dissipation ϵ are also specified at the first cell closest to the wall.

$$k = \frac{u_*^2}{\sqrt{C_\mu}} \tag{5.19}$$

$$\epsilon = \frac{u_*^3}{\kappa y} \tag{5.20}$$

The above approach has problems in re-circulating flows where the friction velocity is zero, by definition, at separation and re-attachment points. To solve this problem Launder & Spalding

(1974) proposed to calculate the friction velocity from k instead of using the log-law.

$$u_* = C_\mu^{1/4} \sqrt{k} \tag{5.21}$$

The transport equation for turbulent kinetic energy is solved with a modified production term that incorporates velocity gradient term that satisfies the log law. Then turbulence dissipation ϵ is fixed at the first cell close to wall. While this method can only be used with models that have equation for turbulent kinetic energy k, the standard wall function method can be used also in LES models with no k equation. Both wall functions discussed above are not applicable in flows with adverse pressure gradient such as wake behind a hill. Advanced wall functions suitable for LES simulations are described in Eugene (2006).

5.2.4 Simulation results and discussions

5.2.4.1 Effect of turbulence models

The selection of turbulence model is important for the prediction of turbulence structure in the wake region. Comparison between the mixing length, standard k-epsilon and RNG k-epsilon and LES turbulence models is shown in Fig. 5.24. The RNG k-epsilon and standard k-epsilon models give close results in most part of the hill except in the wake where the former predicts more recirculation (lower velocity) and longer re-attachment length. On the other hand, the mixing length model and in some cases LES, under-predict the presence of recirculation zone behind the hill in most cases. It is known the mixing length model is not reliable for studying the wake structure, however its prediction of wind speed up at the crest is acceptable. The LES simulation was carried out with a wall function and Prandtl damping function, hence it can hardly be called a large eddy simulation as is commonly done. The wall models dominate all other flow behavior, and the larger eddies in the outer region does not have much influence on the recirculation zone. Resolving the flow down to the viscous layer, as in a typical LES simulation, requires a lot of computational cells. For the current simulation that has Reynolds number of $Re_h \sim O(10^8)$ based on hill height of 200m, about $O(10^{14})$ cells are required. The usual remedy for this problem is to simulate a model scale, e.g at 1:1000, that violates the Reynolds number by orders of magnitude, similar to the case in wind tunnel testing. This has been done by many researchers (Dupont et al. 2008, Feng & Fernando 2011, Iizuka & Kondo 2006, Tamura et al. 2007, Tsang et al. 2009) at a Reynolds number of about $O(10^4)$ without using wall models. The purpose of those LES simulations was to validate wind tunnel results of a scaled model, and LES has shown good agreement with the experimental results. Typically

large recirculation zones are observed at lower Reynolds number compared to the current full scale simulations that displayed smaller recirculation zones.

It was then decided to conduct LES simulations without using wall functions at model scale of about 1:10000 with H=40mm. The Reynolds number is about $Re = 12000$ similar to the case studied in Tamura et al. (2007). For the LES simulations a longer computational domain is used so that the inflow at the inlet can be specified by recycling result from a location downstream of hill. Inflow turbulence is usually generated using 'precursor' simulations for as many time steps as the actual simulation. Recycling avoids the need to store results and usually gives good results if the source plane for turbulence is placed far from the inlet. Another method is to artificially synthesize turbulence that has required spatial and temporal correlation, i.e. from inverse Fourier transform of von Karman spectrum. Simply applying random and uncorrelated correlations is not appropriate for wind engineering applications. The simulation results show large recirculation zones with long re-attachment length of about 6H as shown in Fig. 5.34. This is in agreement with the result of Tamura et al. who found it to be about 5.8H using using numerical LES simulations and about 5.4H experimentally in wind tunnels. The RNG k-epsilon model show much shorter re-attachment length of about 4H but it is still significantly larger than the re-circulation zones observed at high-Re flow with full scale dimensions.

5.2.4.1.1 Model scale LES and RANS simulations In the previous section, the effect of Reynolds number on the results of LES simulations, in particular the recirculation behind hills, is demonstrated over a test case obtained from (Tamura et al. 2007). Next all the cases of the current study are simulated at model scale of 1:10000 so that the LES simulations are feasible without the use of wall functions. The Reynodls number calculated based on the height of the hill is about $Re_h = 30000$. A RANS simulation is done in parallel using the best performing model so far, namely the RNG k-epsion model. The computational domain is extended in the longitudinal direction so that the 'recycling method' can be used for inlet turbulence generation as discussed before. The results of these simulations are shown in Figs. 5.35-5.38.

We can immediately see that both the LES and RANS simulations show larger recirculation zone for the steep hill cases at this relatively low Reynolds number of 30000. This is in accordance with the results of Tamura et al. (2007) that used a hill which has more or less the same slope as the current study's steep hills. However differences are observed between LES and RANS for the case of shallow hills in which LES shows much bigger recirculation than that of RANS simulations. The wall functions used for the RANS simulations play an important role in the accuracy of results. It is known that wall functions work well at high Reynolds numbers

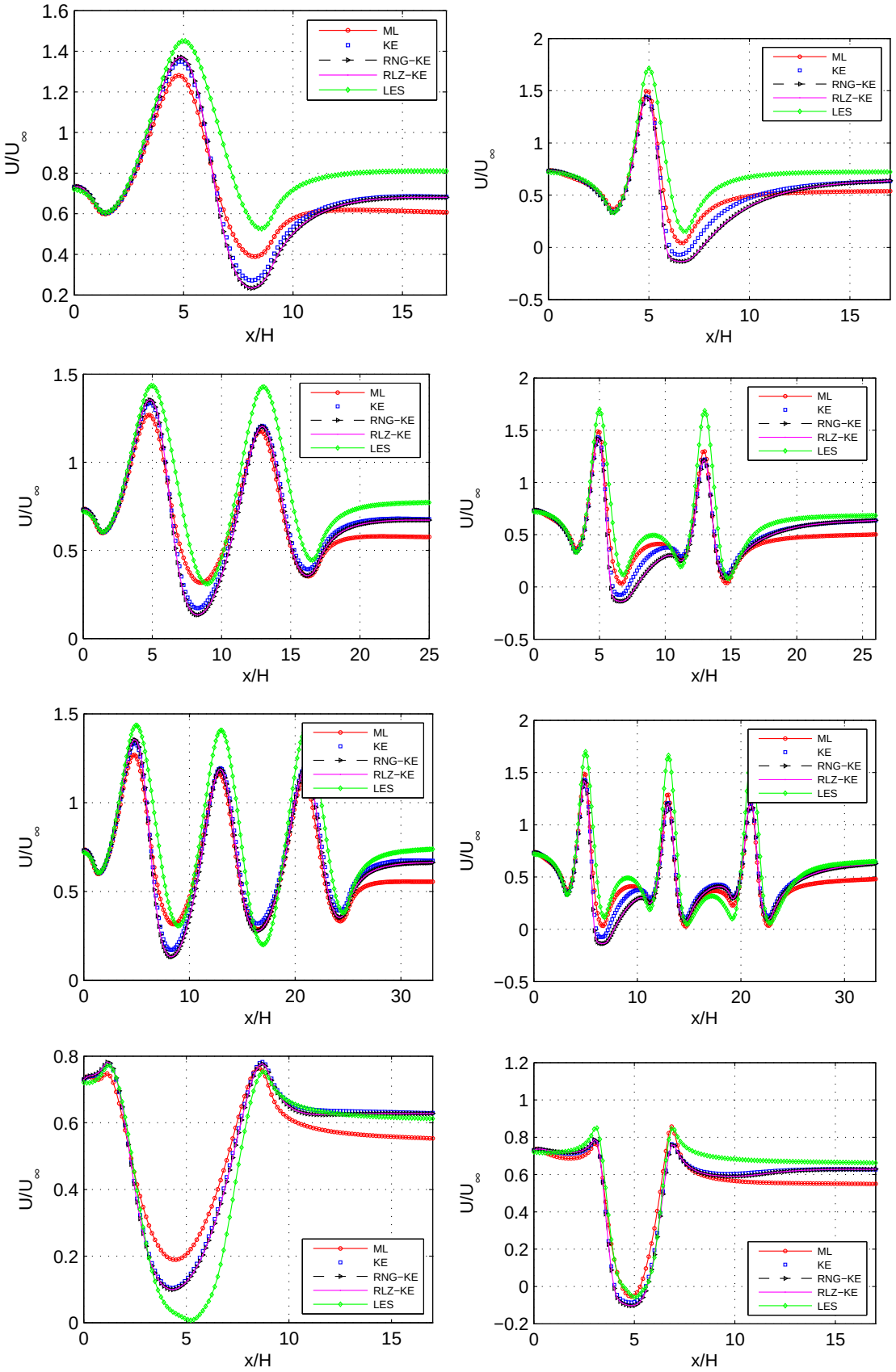


Figure 5.24: Mean horizontal velocity for shallow and steep a) isolated hill b) double hills c) triple hills d) isolated valley at 20m height from full scale simulations

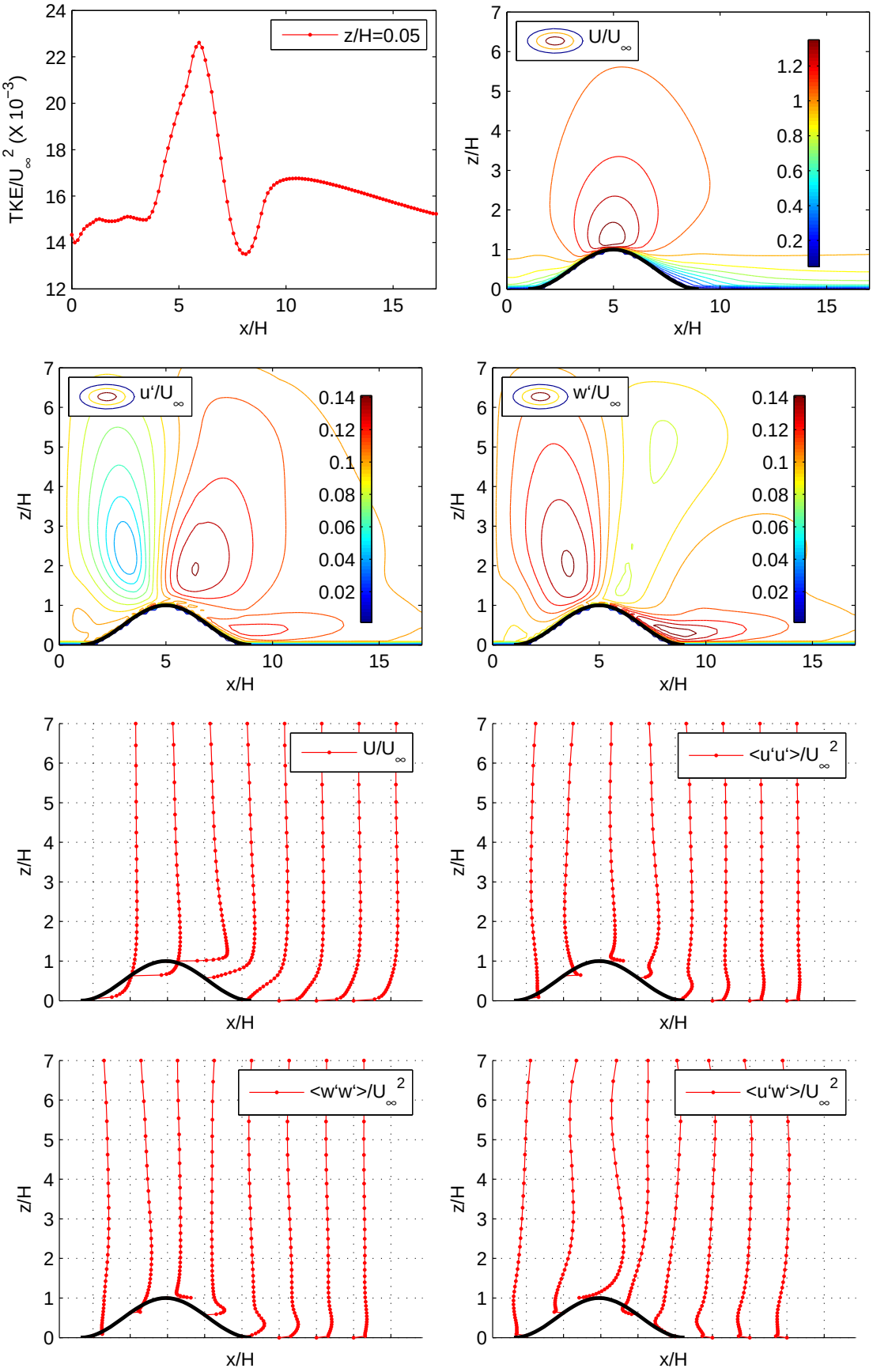


Figure 5.25: Results for single shallow hill: TKE, horizontal velocity U,fluctuations u' and w', and Reynolds stresses

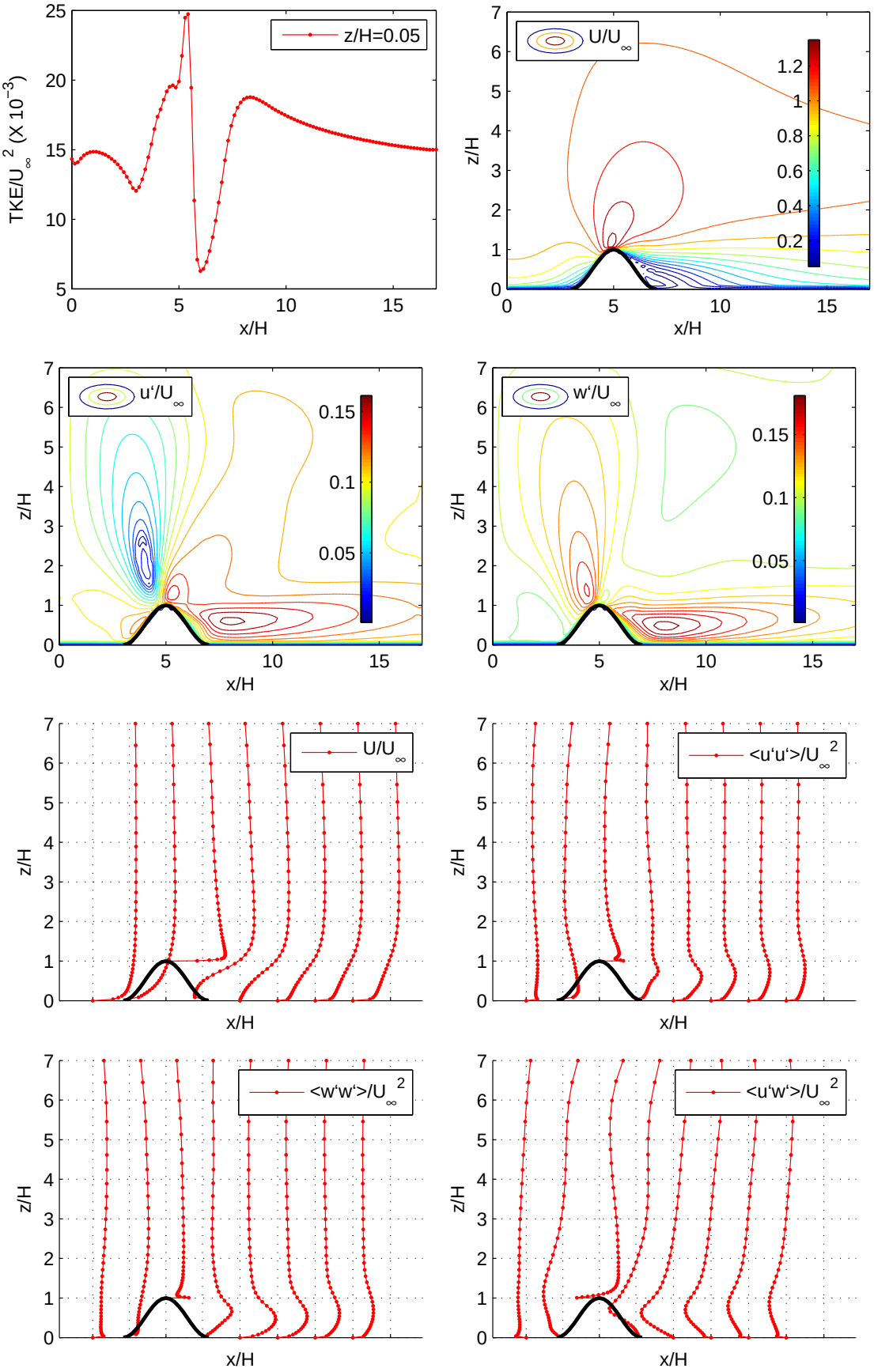


Figure 5.26: Results for single steep hill: TKE, horizontal velocity U,fluctuations u' and w', and Reynolds stresses

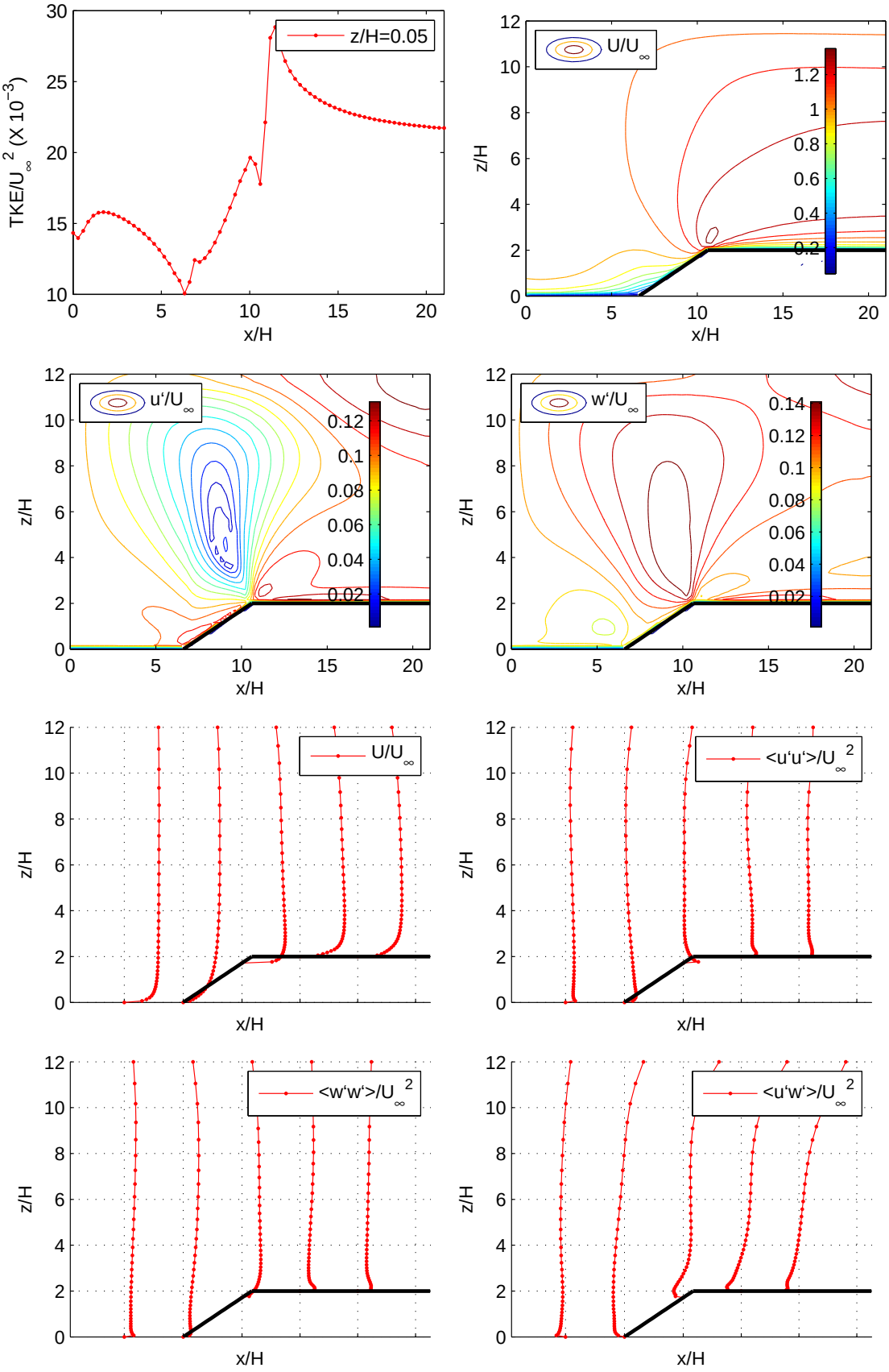


Figure 5.27: Results for escarpment: TKE, horizontal velocity U,fluctuations u' and w', and Reynolds stresses

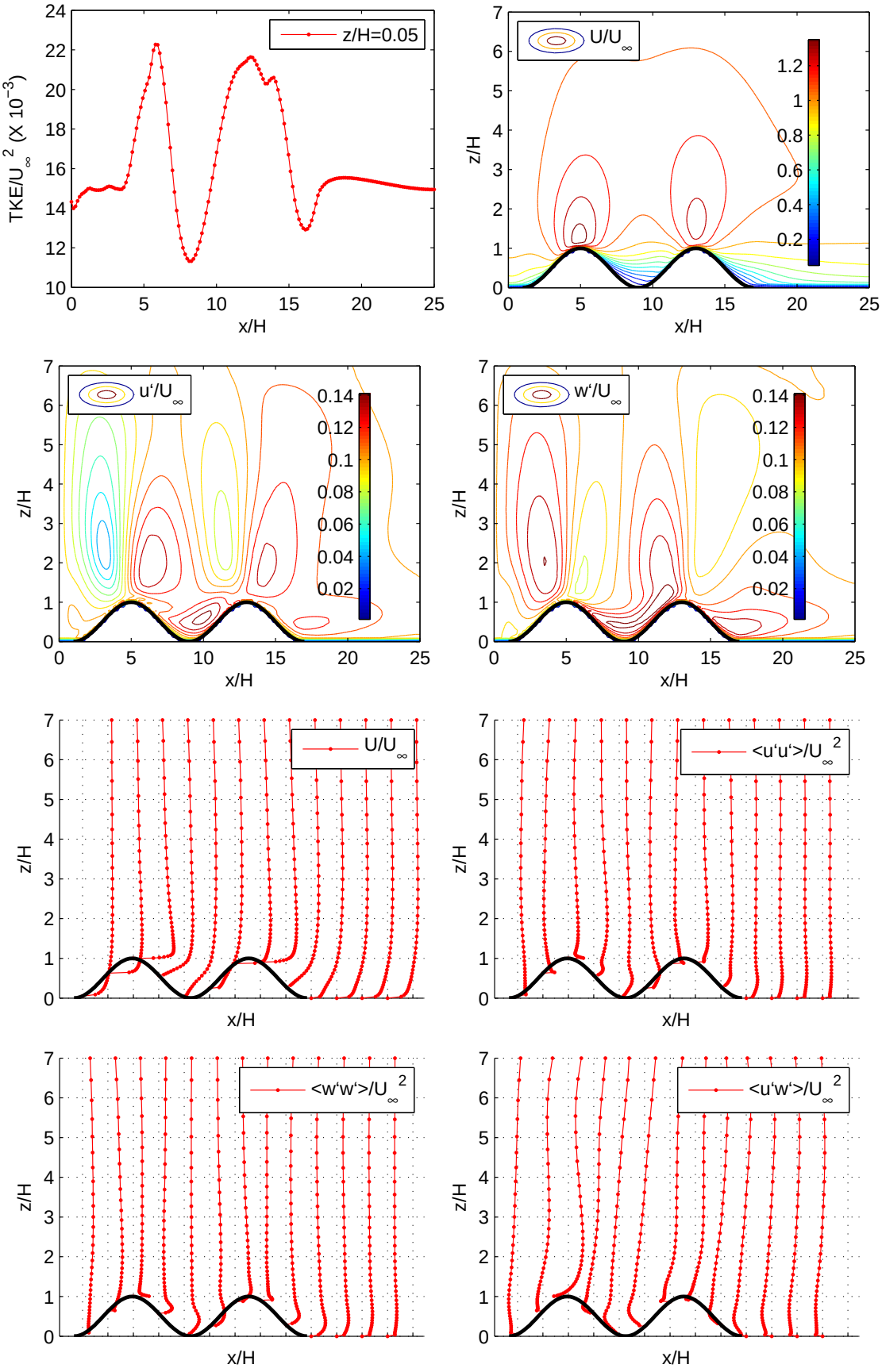


Figure 5.28: Results for double shallow hills: TKE, horizontal velocity U,fluctuations u' and w', and Reynolds stresses

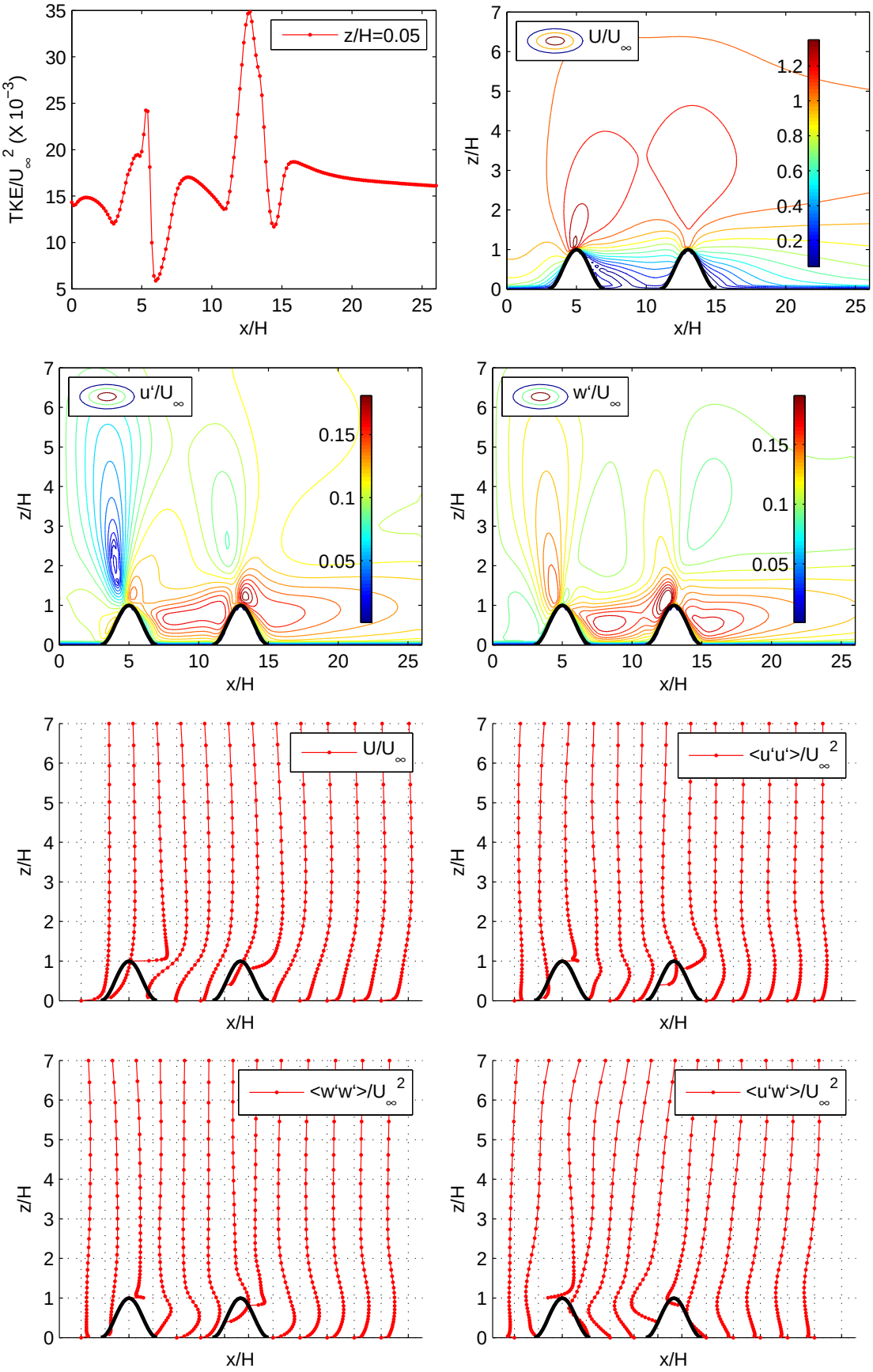


Figure 5.29: Results for double steep hills: TKE, horizontal velocity U,fluctuations u' and w', and Reynolds stresses

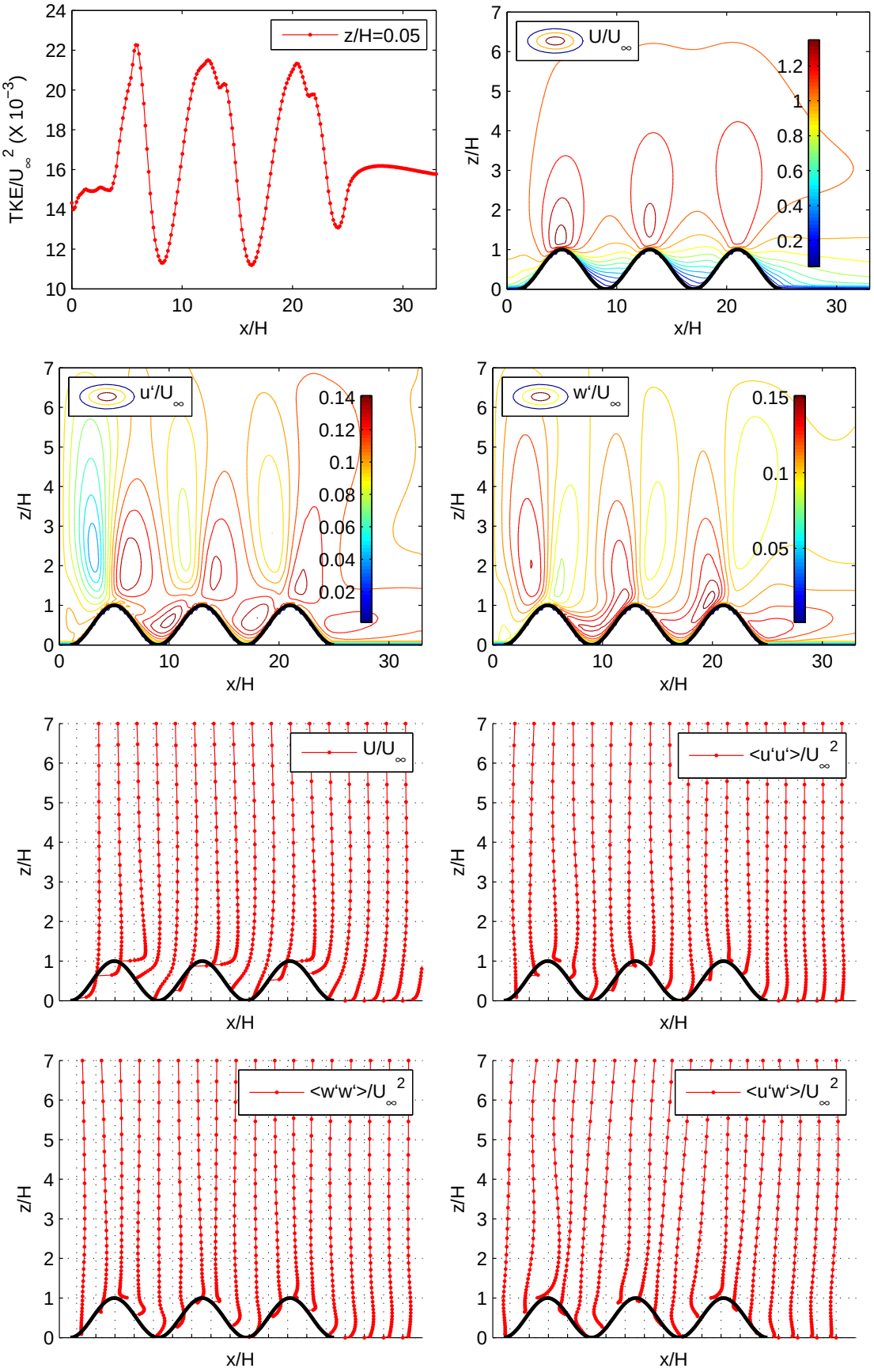


Figure 5.30: Results for triple shallow hills: TKE, horizontal velocity U,fluctuations u' and w', and Reynolds stresses

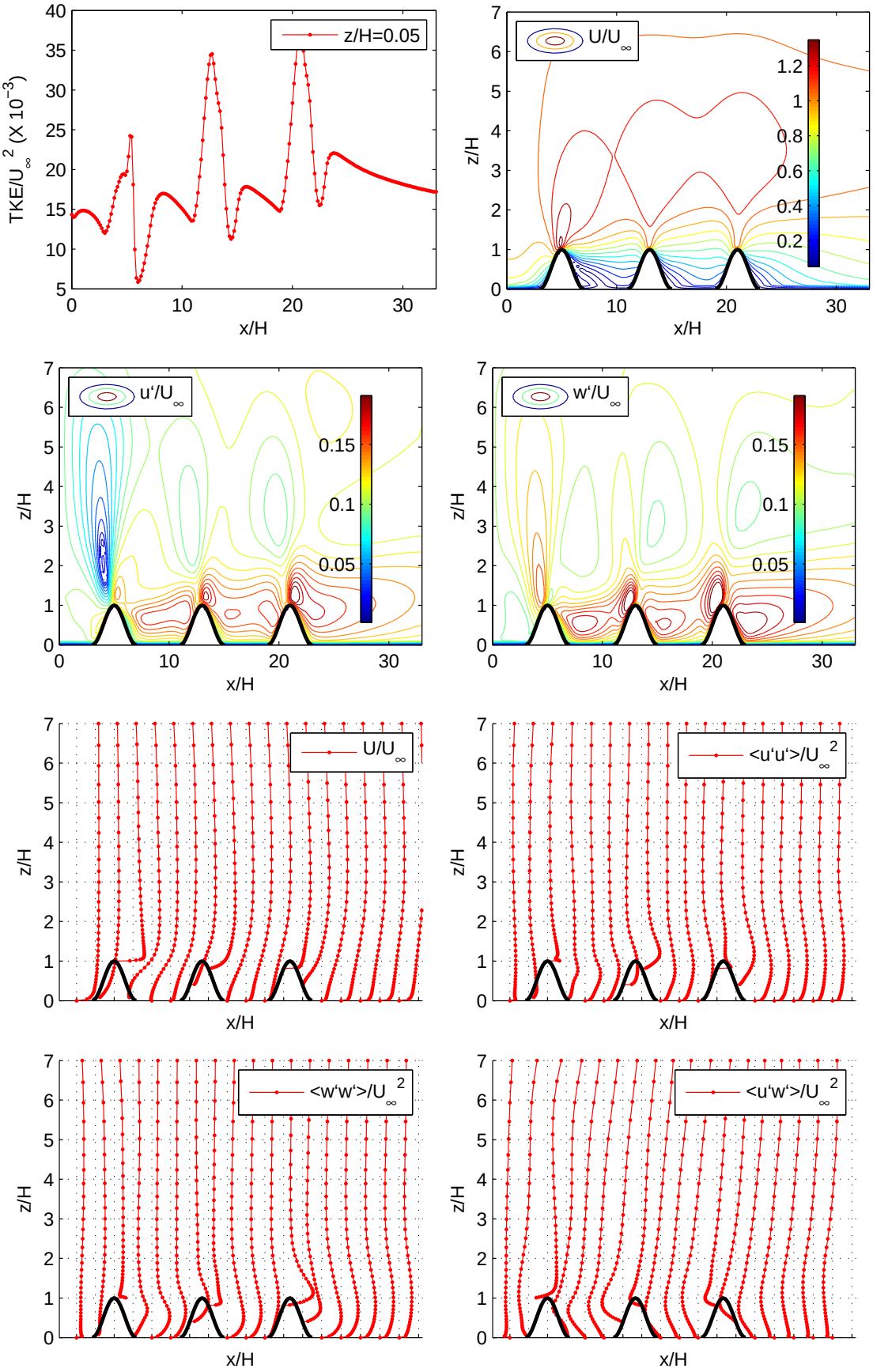


Figure 5.31: Results for triple steep hills: TKE, horizontal velocity U,fluctuations u' and w', and Reynolds stresses

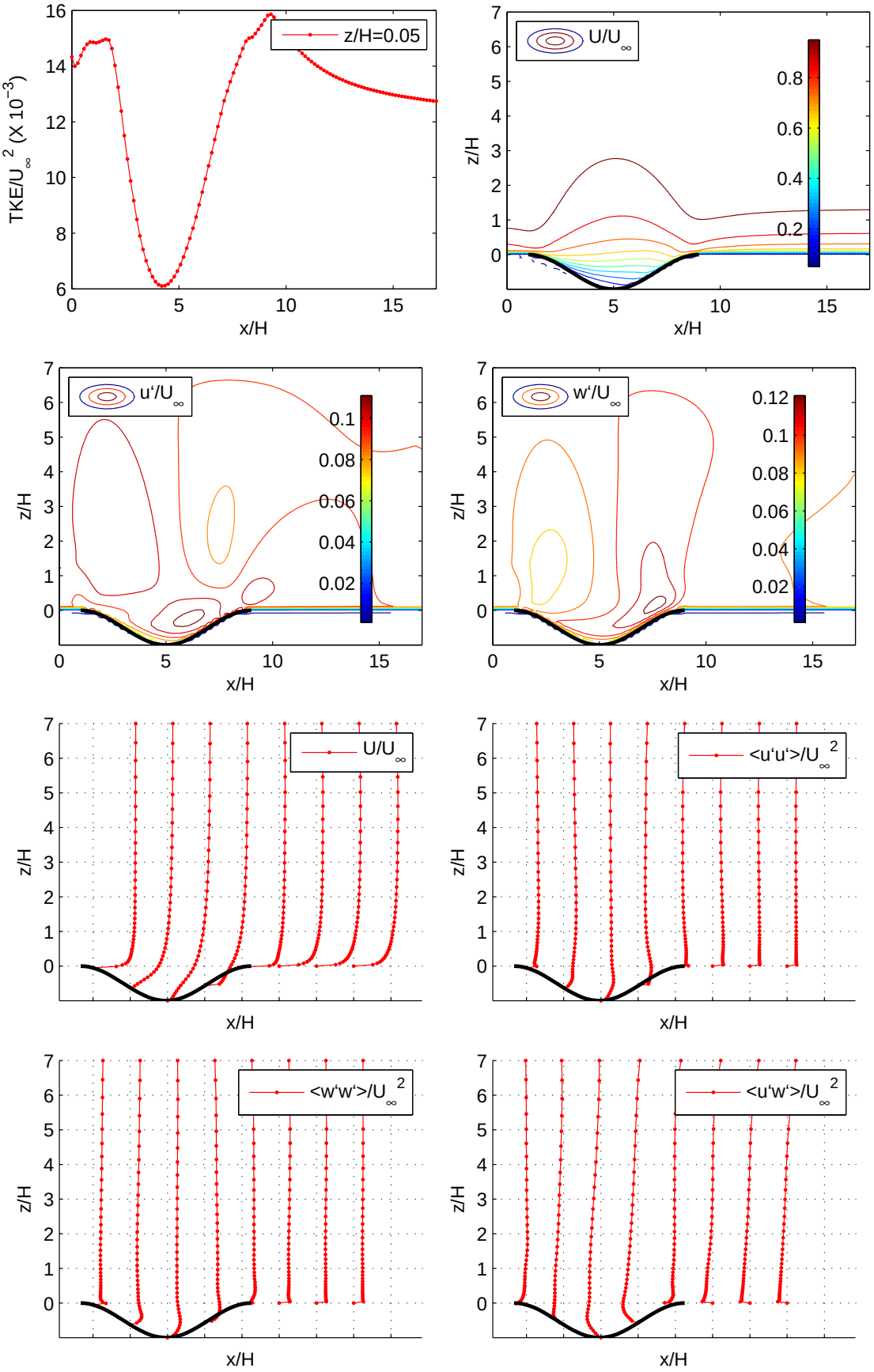


Figure 5.32: Results for single shallow valley: TKE, horizontal velocity U,fluctuations u' and w', and Reynolds stresses

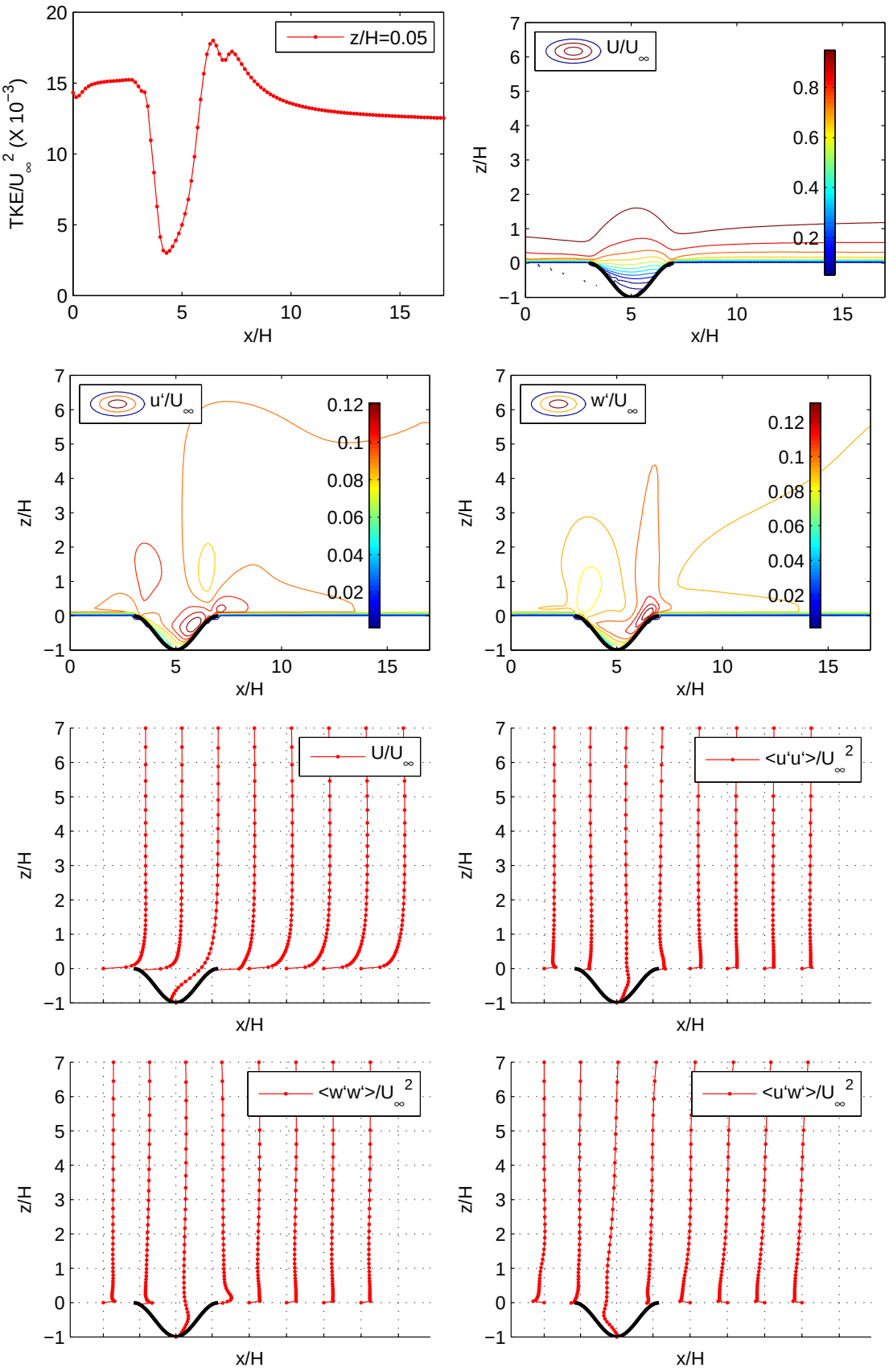
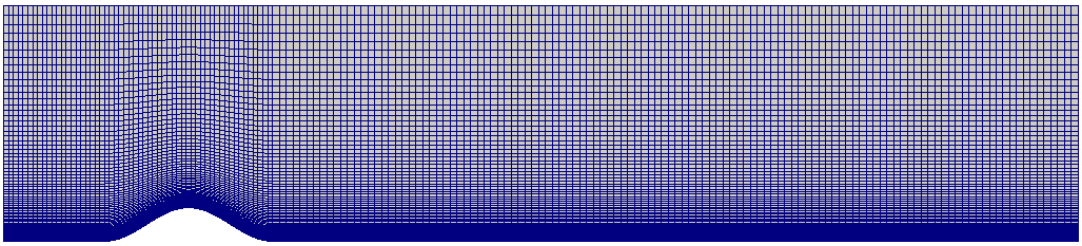
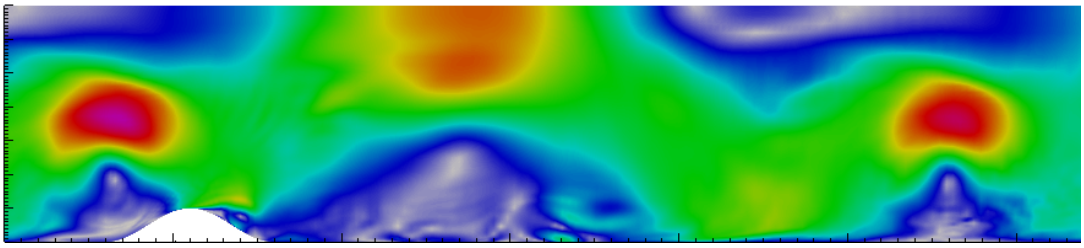


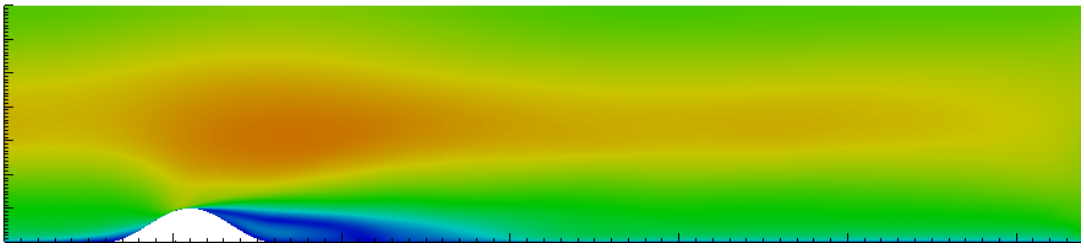
Figure 5.33: Results for single steep valley: TKE, horizontal velocity U,fluctuations u’ and w’, and Reynolds stresses



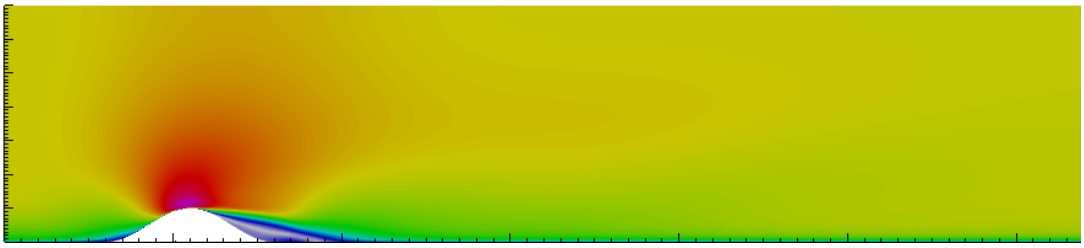
(a) Grid for the hill of Tamura et al. (2007)



(b) LES instantaneous velocity



(c) LES mean velocity



(d) RNG k-epsilon mean velocity

Figure 5.34: Horizontal velocity contour plots for model scale simulations (Re=12000) using LES and RANS models

and non-separated flows, however their accuracy decreases with Reynolds number, which in the current case is a drop by a factor of 10^4 compared to the previous full scale simulations. Griffiths & Middelton (2010) stresses the influence of the wall function on accuracy, while comparing two CFD software (RAMS and FLUENT) with regard to modeling separated flow behind hills. They also observed that better agreements are found for the steepest hill cases, similar to the findings of the current study.

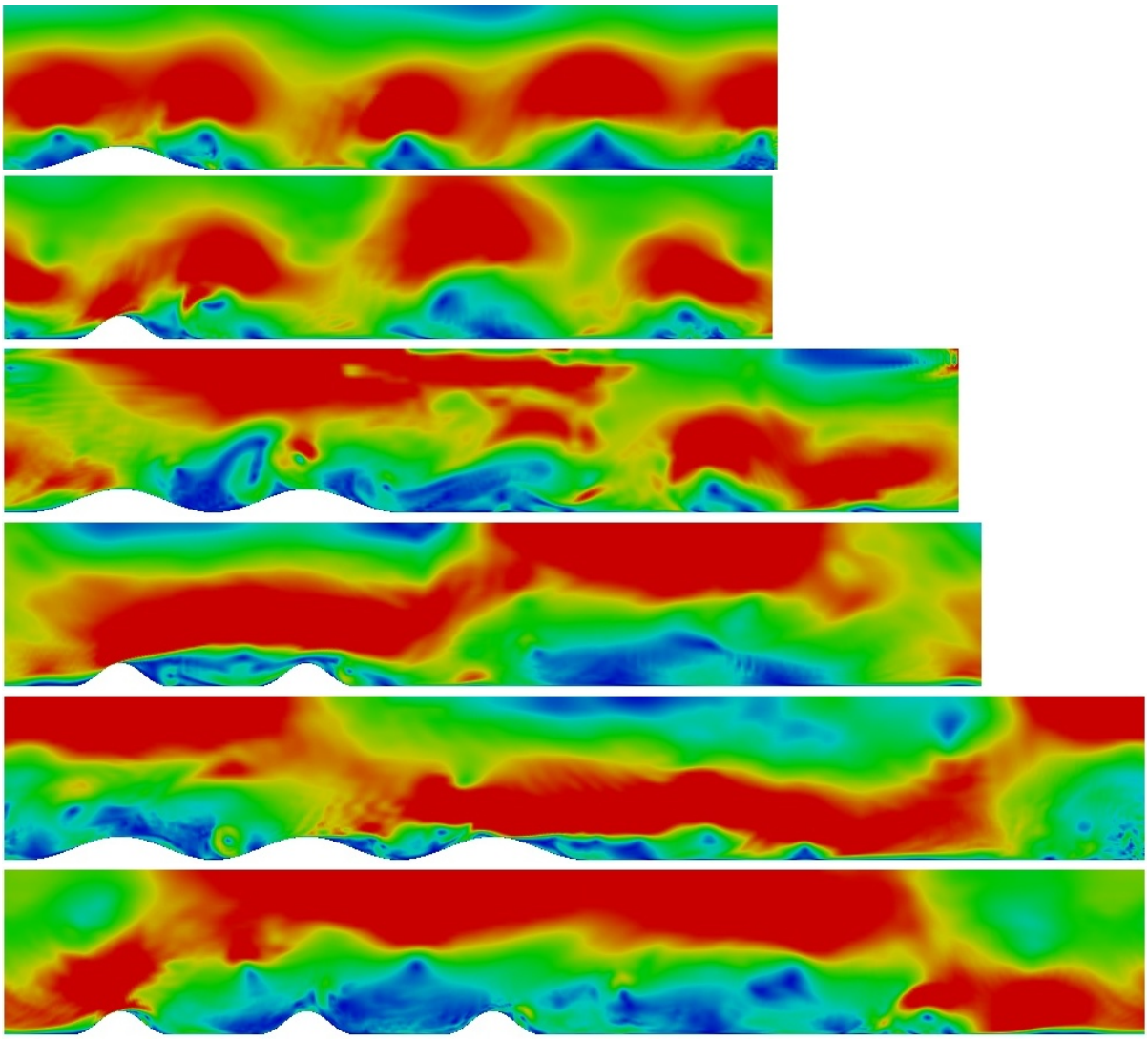


Figure 5.35: Instantaneous velocity contours of LES simulations for all the 2D hills: single shallow hill, single steep hill, double shallow hill, double steep hill, triple shallow hill, triple steep hill

5.2.4.1.2 Extracting fluctuations from simulations For LES turbulence models, the RMS fluctuations can be directly obtained, however RANS turbulence models keep only mean flow

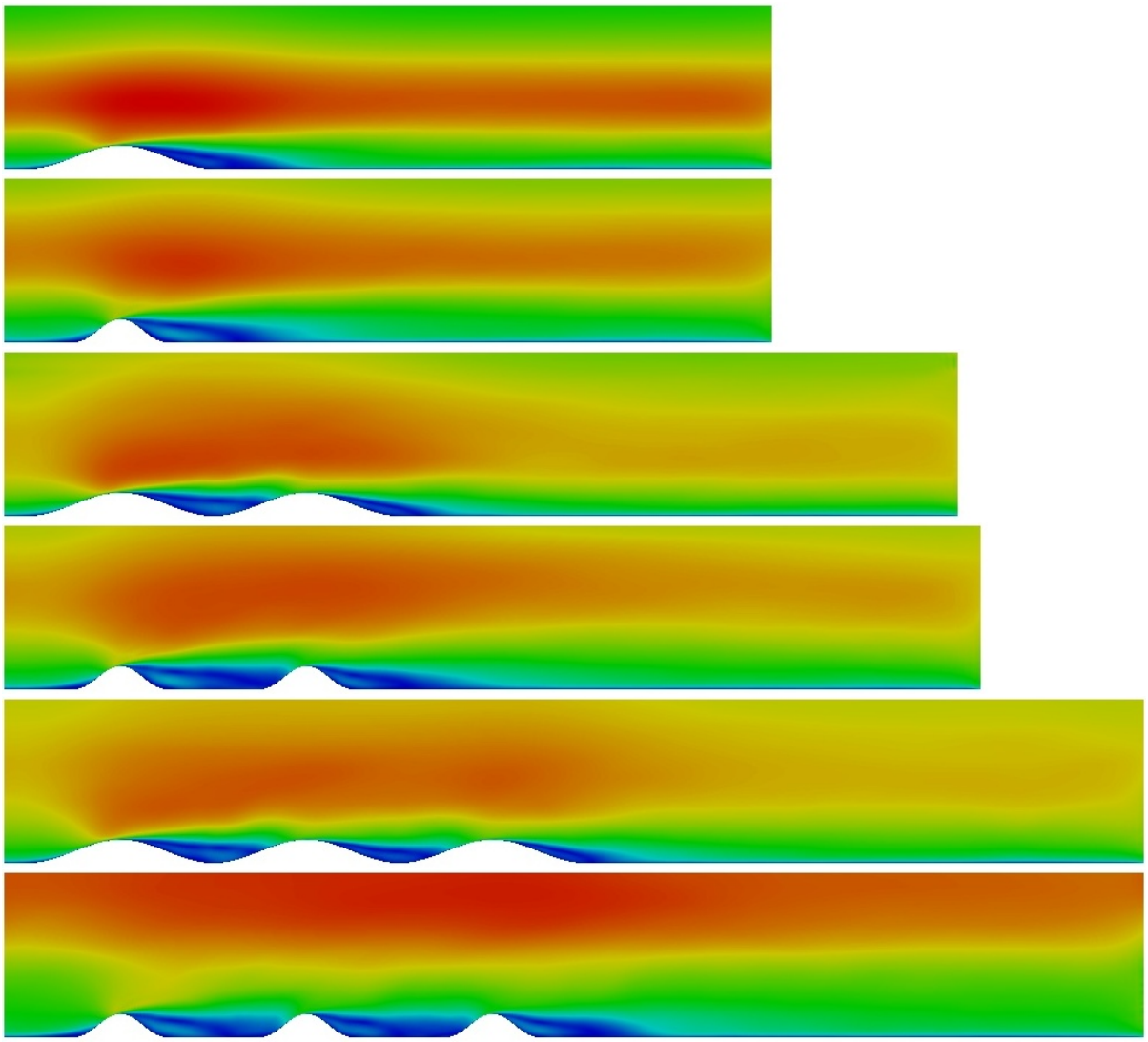


Figure 5.36: Mean velocity contours of LES simulations for all the 2D hills: single shallow hill, single steep hill, double shallow hill, double steep hill, triple shallow hill, triple steep hill

quantities so it has to be inferred from Reynolds stresses. A simple method is to assume isotropic turbulence ($u' = v' = w'$), and calculate it from TKE as $u' = \sqrt{2/3k}$. A better method is to extract it from normal Reynolds stress components ($< u'u' >$). The linear eddy viscosity assumption used in RANS models gives RMS fluctuations that are proportional to the mean straining field components. On the other hand, if the six Reynolds stress components were directly solved, as is done in Reynolds stress models (RSM), better estimates of RMS fluctuations can be obtained. The result using the RNG k-epsilon model and second approach of calculating RMS fluctuations are shown in Figs. 5.25-5.33. For a succession of multiple hills, we can observe that TKE at the crest increases significantly from the first hill to the

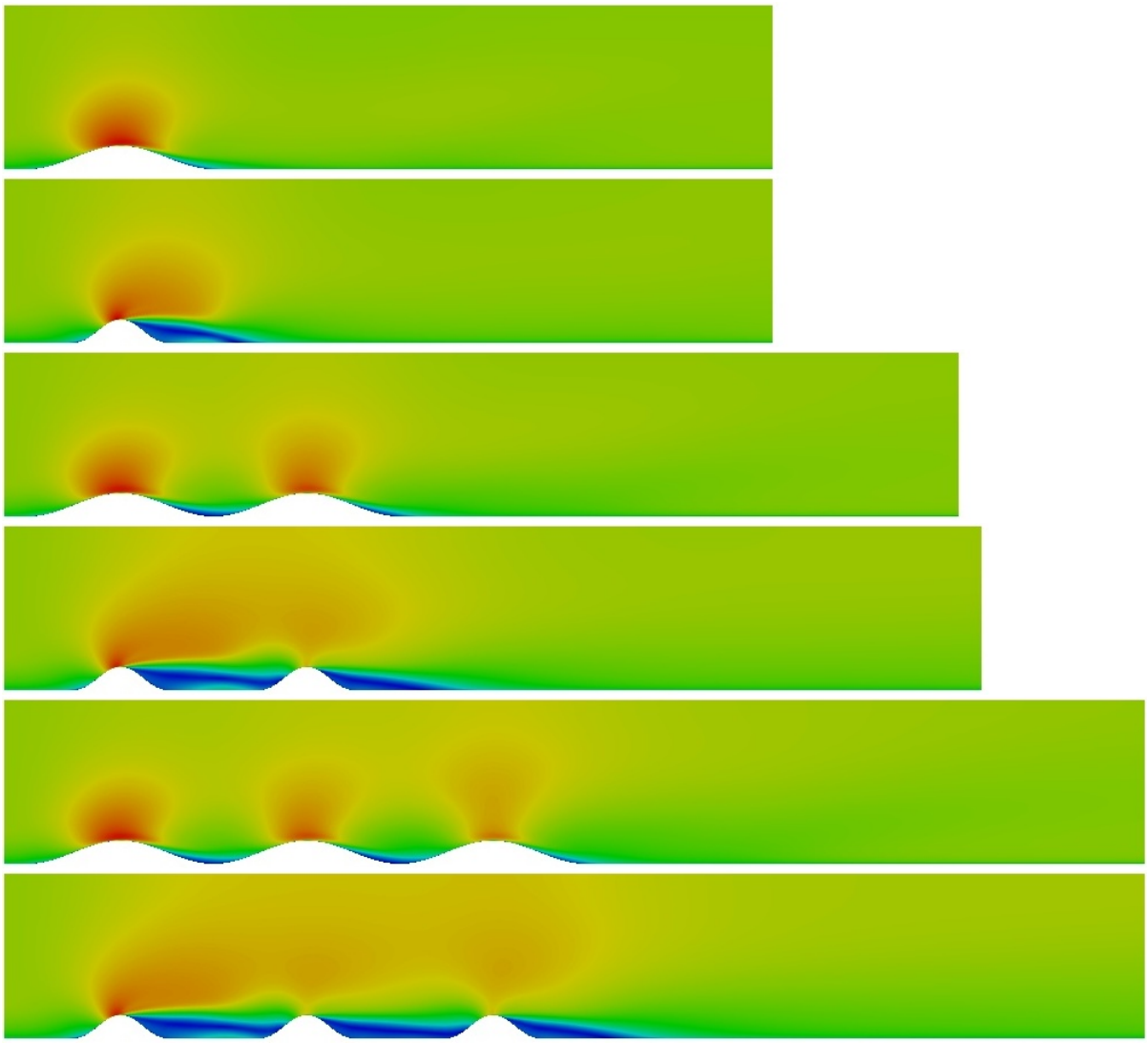


Figure 5.37: Mean velocity contours of RANS simulations for all the 2D hills: single shallow hill, single steep hill, double shallow hill, double steep hill, triple shallow hill, triple steep hill

second. The increase in TKE from the second to the third hill is not significant. The variation of Reynolds normal and shear stress components shows similar pattern to those found in literature for isolated and multiple hills.

5.2.4.2 Roughness effects

The effect of roughness on simulation results is briefly assessed. Both wind speed and turbulence structure are affected by roughness of the terrain surface (Shuyang & Tetsuro 2006, 2007). For CFD simulations, Nikurdase type equivalent sand-grain roughness (K_s) is commonly used. As noted in literature review, this method is not realistic but it is simple to im-

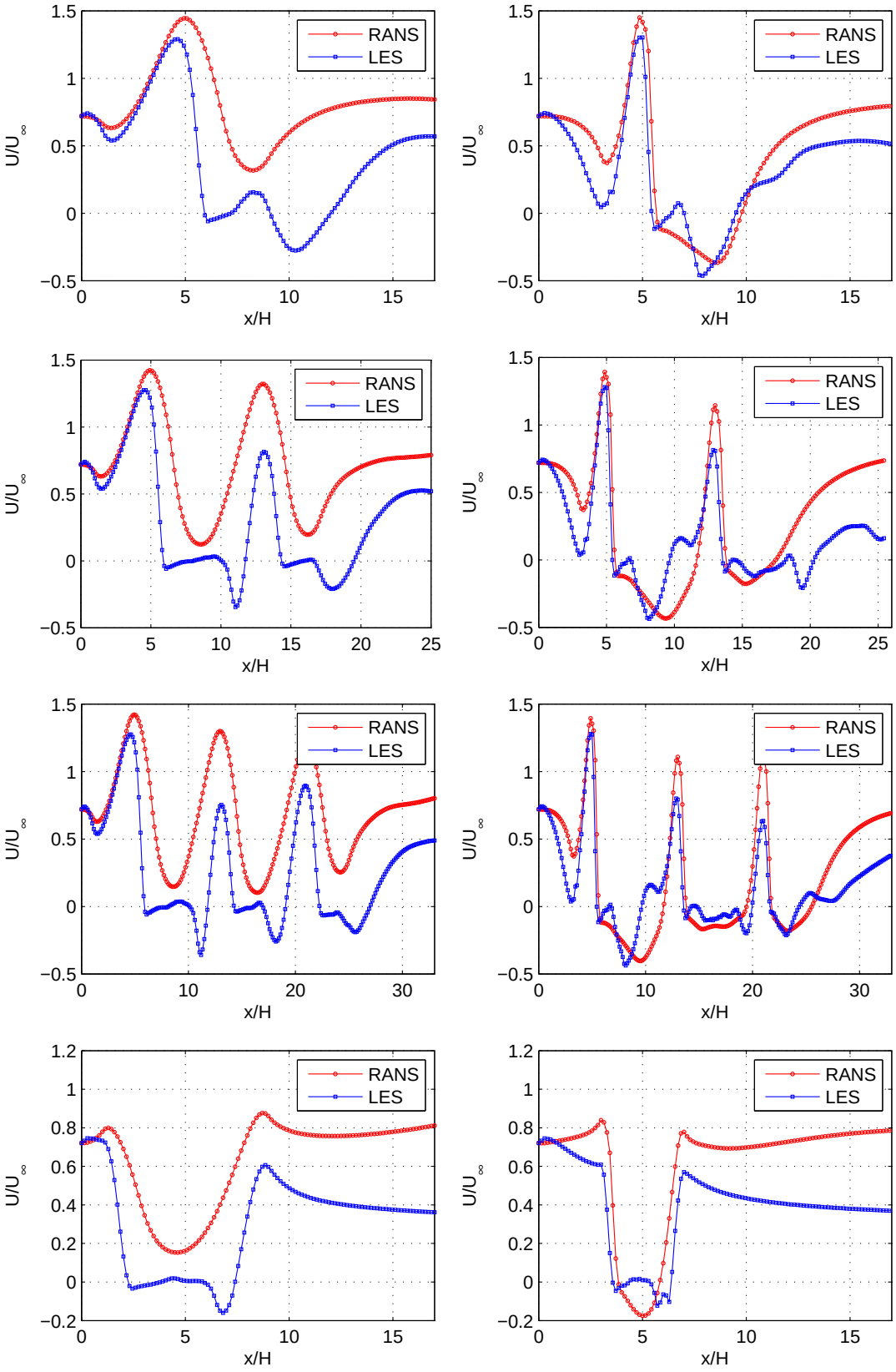


Figure 5.38: Mean horizontal velocity for shallow and steep a) isolated hill b) double hills c) triple hills d) isolated valley at 20m height from model scale simulations

plement in CFD codes.The standard wall function is modified with an additional term that is dependent on K_s to incorporate the effect of roughness. Blocken et al. (2007) discusses different issues concerning use of wall functions. The inlet velocity and turbulence intensity profiles should be compatible with the wall function used at the ground surface to avoid development of stream-wise gradients. For this study four roughness classes are considered: smooth terrain (Z_0=0.005), open (Z_0=0.024), roughly open (Z_0=0.1) and very rough (Z_0=0.42). The results for mean velocity and turbulence kinetic energy at z=10m are shown in Fig. 5.39. It can be observed that roughness affects turbulent kinetic energy more than it does wind speed. Significant TKE increases are observed on the crest and wake of the hill with each increase in roughness class.

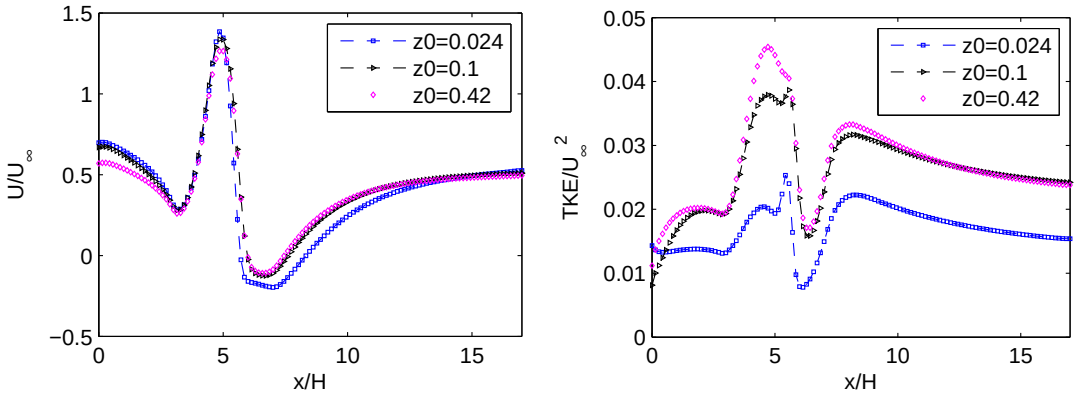


Figure 5.39: Mean velocity and TKE for different roughness lengths

5.2.4.3 Scheme sensitivity

The sensitivity of results, especially the turbulence structure behind hills, to the selected discretization schemes is briefly assessed. Simulations are carried out with different convection discretization schemes. The commonly used upwind scheme (UDS), while being very stable, is very dissipative and may perform very poorly in the re-circulation bubble. For equal number of control volumes, second order and other higher order schemes can give much better results. Higher order schemes can have higher dispersive errors and give unrealistic results with wiggles. To avoid this problem while maintaining second order, the higher order schemes are flux limited to satisfy a Total Variation Diminishing (TVD) criteria. The isolated hill problem is again analyzed for four different convection schemes applied to all terms: upwind (UDS), unbounded central difference (CDS), bounded QUICK and SUPERBEE. We can observe that the result for wind speed does not show much difference. However the TKE plot shows that the

higher order schemes show significant differences at the crest and wake. The CDS scheme, being unbounded, seem to overshoot TKE at the crest and lee foot. Therefore the use of bounded schemes is important. Martinez (2011) have done similar work but it is not clear if the significantly different result observed at the lee foot is due to the schemes being unbounded.

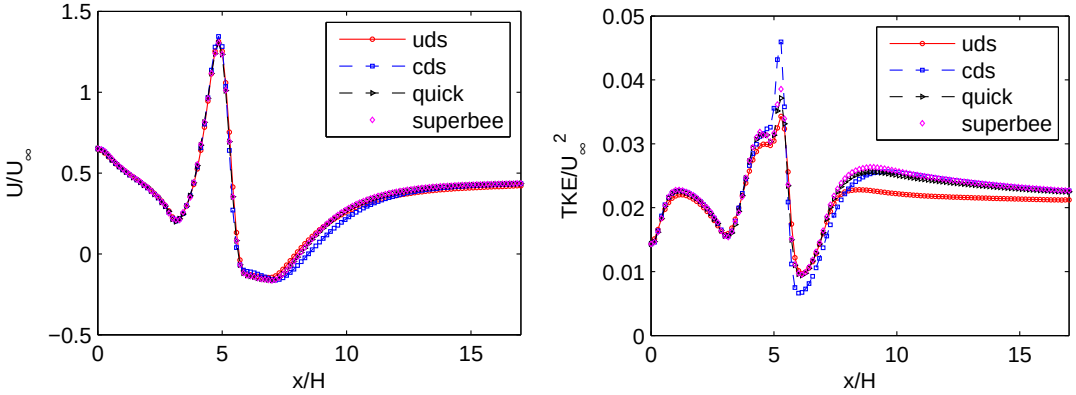


Figure 5.40: Mean velocity and TKE comparison for different convection discretization schemes

5.2.5 Conclusions

The study of turbulence structure in the wake of hills using CFD requires more effort than estimating speed up factors at the crest of hills. Significant increase in local turbulence intensity is observed at crest of hills. Also if a second hill is in the wake of another hill, the second hill shows significantly larger RMS fluctuations at its crest. From the third hill onwards, the increase in TKE is not as large. The current CFD result reinforces the statement made by Miller & Davenport (1998) that no reduction in turbulence intensity should be made at sheltered hills.

The turbulence model used for simulation play a significant role in the accuracy and economy of the simulation. The simplest model considered in this study, the mixing length model, under-predicts the size of the recirculation zone. RNG k-epsilon model gives better estimates than the standard k-epsilon model for prediction in the wake region. The LES results showed reduced recirculation zone but this is mostly due to the use of wall functions used for the current high-Re of $O(10^8)$ simulations. To have confidence in this observation, a model scale LES simulation without wall functions is carried out at Re of 10^4. The result show large re-circulation bubble with long re-attachment lengths that are in line with result from literature.

The effect of surface roughness is briefly investigated by changing sand grain roughness K_s. It is found that roughness affect both wind speed ups over the crest and RMS fluctuations

in the wake, but its effect on the RMS fluctuations is more pronounced.

5.3 Wind flow simulations on real complex terrain

So far we have considered only idealized topographic features with a smooth shape. In a real complex environment, this kind of ideal topography is rarely found. In this section we conduct simulations over a real topography and validate our results with field observations. The Askervein hill is the standard benchmark for validation of CFD code for complex terrain simulations, hence we begin the study by conducting wind flow simulations over this hill and then validate the results using field observation data of Taylor & Teunisson (1986). Then we proceed simulating an even more complex terrain to highlight the associated difficulties. The chosen complex hill has sharp edges and steep slope. Data is not available to validate the results for this case, because it was chosen randomly from USGS database for demonstration. In general lack of field observation data is a problem that plagues complex terrain studies.

The slopes of topographic features and roughness of the terrain are important factors that affect wind speed up over hills. A terrain is classified as complex when it has a steep slope leading to significant flow separation and other complex phenomenon on the leeward side. Wind codes incorporate these effects using simplified terrain models that are not representative of a real complex terrain. Besides the slope, the terrain roughness also plays a role to retard the wind near the ground.

CFD codes incorporate the effect of roughness usually through equivalent sand grain roughness (K_s) after Nikurdase who first proposed the idea. A continuous and dense roughness is assumed, and a corresponding shear stress is applied as a body force to the nearest cell to the wall. If the ground surface model consists of distinct patches with variable roughness characteristics such as grass, buildings, parking lots etc., then averaged roughness parameters K_s and C_s over the patch are specified for each patch. On the other hand, if roughness data is available in a contour format, K_s and C_s are applied to each boundary face after the computational domain is discretized. In any case the variance of surface roughness should either be explicitly modeled (for bigger obstacles) as done in the previous section or implicitly modeled through rough wall functions. An implementation for OpenFOAM for the second approach is discussed in Xabier (2009).

5.3.1 Askervein hill case study

CFD codes for micro-scale model simulations are usually validated against a well known case of wind flow over the Askervein hill. The hill has been subjected to extensive field measurements in the 1980s, making it ideal for CFD benchmarking studies in literature (Castro et al. 2003, Crasto 2007, Martinez 2011, Stangroom 2004). However it is hardly complex by the definition we gave before : nearly two dimensional, isolated ellipsoidal hill with a gentle slope varying from 12% to 25%. Contour map of the hill is shown in 5.42. Other hills may pose a challenge in meshing and modeling of variable roughness characteristics, though automatic meshing for complex terrain is a difficult task anyway.

The TU03-B data set from Taylor & Teunisson (1986) is used to validate our CFD results. The data set was acquired while the wind was blowing at 210 degree clockwise for more than three hours. The wind direction is almost perpendicular to the hill for this particular case. The hill is 116m high and has elliptical shape as shown in the contour plot of Fig. 5.42. It is located in an area where there are no major buildings or other obstacles, hence a constant roughness is assumed thereby simplifying the analysis. The value of the roughness length z_0 was also measured during the field investigation and found to range from 0.01m to 0.05m. This study used $z_0 = 0.03m$.

The surrounding is flat on the upwind side of the hill and is hilly on the down side. A reference site is located 3km south west of the hill. The wind flow is relatively un-perturbed by the surrounding at the location hence the inlet of the computational domain is placed there. The instrument towers for the field observations are located along a line passing through HT and inclined at 220^0.

5.3.1.1 Computational domain setup and grid generation

The dimension of the computational domain is set at 10km X10km X 2km around the hill center similar to that used by Stangroom (2004). A smaller height of 1km is also tested and found to be satisfactory. The hill is 116m high hence a 1km computational domain has a gap of about 9H between the hill top and the top of the domain. This is more than enough to satisfy the recommendations of Franke & Hirsch (2004) with regard to blockage effects. The blockage ratio is about 2% for a 1km boundary layer. The terrain data for the hill is available in triangulated STL file format which is in ready to use format for many commercial CFD software. Usually the problem lies in generating a grid from such complex surface model. Automatically generating a good mesh consisting of mostly hexahedral elements is still an active research area. One such meshing tool that significantly reduces involvement of the user

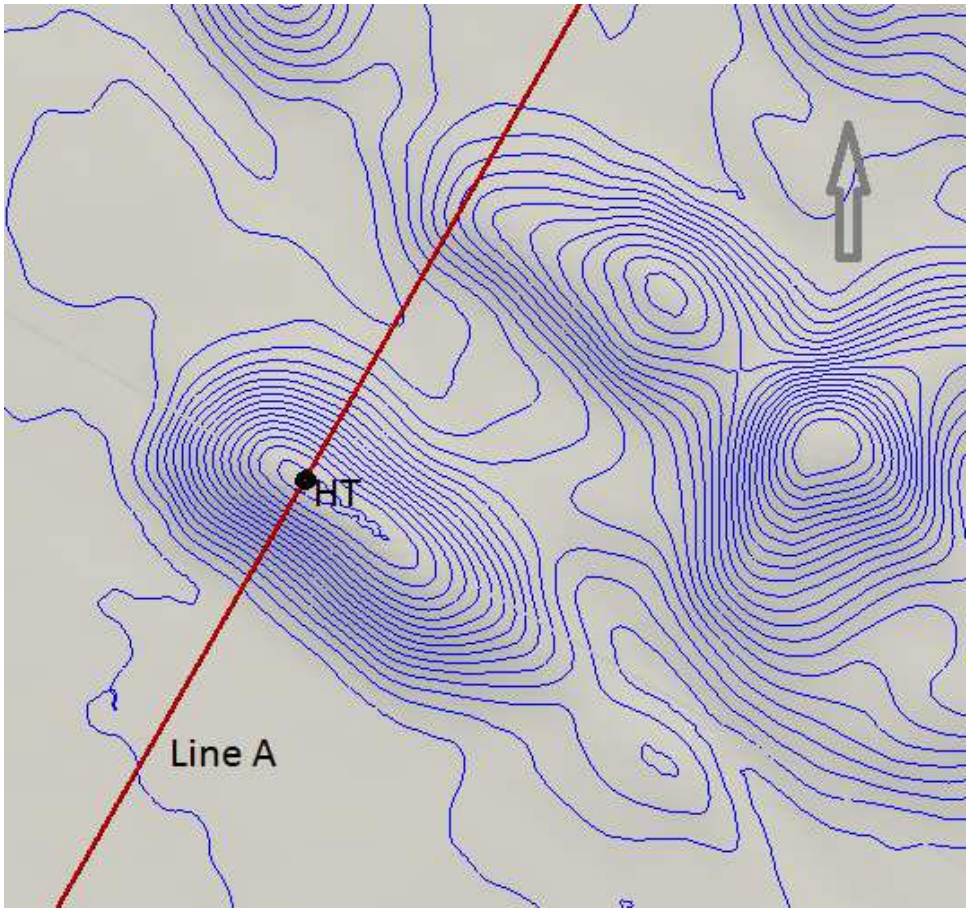


Figure 5.41: Contour map of Askervein hill showing Line-A and the hill top (HT)

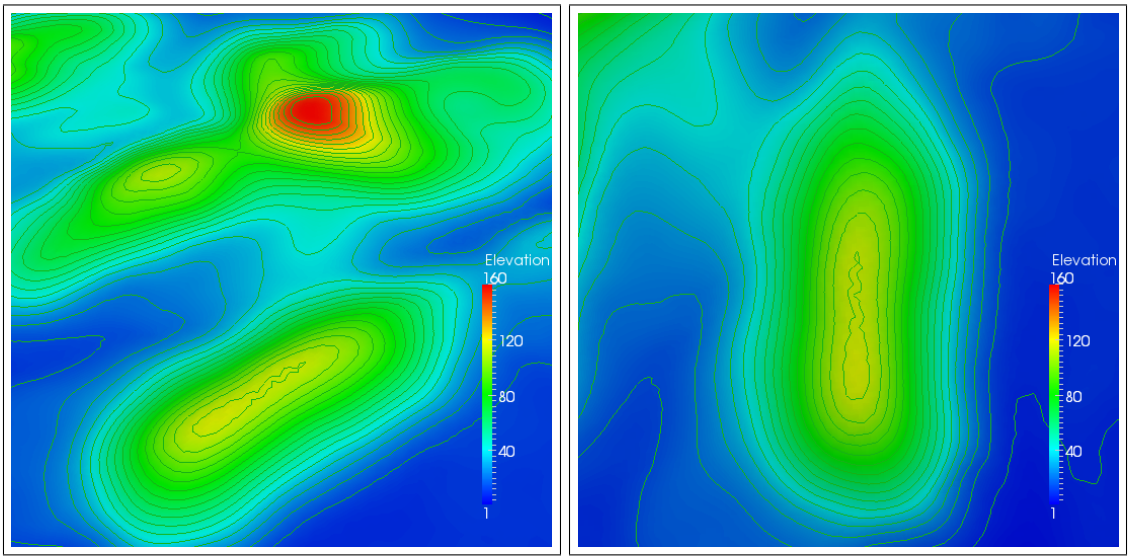


Figure 5.42: Elevation map of Askervein hill including surrounding

is snappyHexMesh (Weller et al. 1998). The meshing for Askervein hill and all other complex surfaces in this study are done by this tool. It is able to generate an unstructured mesh of mostly

hexahedral elements from a given digital surface model. A description of the steps involved and meshing refinements close to the hill is found in Martinez (2011). Mesh refinement is done around the hill in two boxes similar to the idealized isolated hill case considered in Fig. 5.10, and also seven boundary layers are added on top of the ground surface to avoid convergence problems due to skewed and other low quality cells. The main difficulty with snappyHexMesh is that it requires a background mesh with an aspect ratio of one. This requirement leads to excessively high number of cells, most of them being placed where they are not much required. It is important to place as many cells as possible in places where flow variables change very rapidly i.e. have the highest gradient.

5.3.1.2 Grid independence study

Simulations are carried out for different grid sizes to check the sensitivity of the results with the grid size. The grids used are generated in a two step manner. First a relatively coarse back background meshes of 30 X 30 X 8, 50 X 50 X 13, 100 X 100 X 24, and 120 X 120 X 32 are generated, and then refinements are applied close to the hills and the ground surface to capture the boundary layer flow better. The number of cells that snappyHexMesh produced after application of boundary layers and other refinements are 150k, 520k, 2.5million and 4million cells respectively. These numbers are significantly larger than those of corresponding background meshes, thereby highlighting the importance of mesh refinement in important regions for efficient simulation. The TU03-B data set, which was recorded along line A and while the wind was blowing at 210 degrees from north, is used for the simulations. The result of this grid independence study is shown in Fig. 5.44. Along most of the uphill slope the speed up remains the same for all grids, but starting from the crest through the wake zone significant differences are observed. This is especially true for the coarsest grid considered i.e 30 X 30. For the purpose of determining maximum wind speed up at the crest, the 50 X 50 grid gives acceptable results. However it shows some differences with 100 X 100 and 120 X 120 grids for the flow in the leeward side of the hill. Therefore we can conclude that the 100x100 case gives grid independent results for the flow over the whole length of the hill.

5.3.1.3 Different turbulence models

Another factor that has similar consequences as the grid resolution is the turbulence model. The standard k-epsilon turbulence model is commonly used for wind engineering applications, however it is known that steady RANS models give good predictions only on the upwind side. The unsteady effects on the lee side are not captured by RANS models, and it may be important

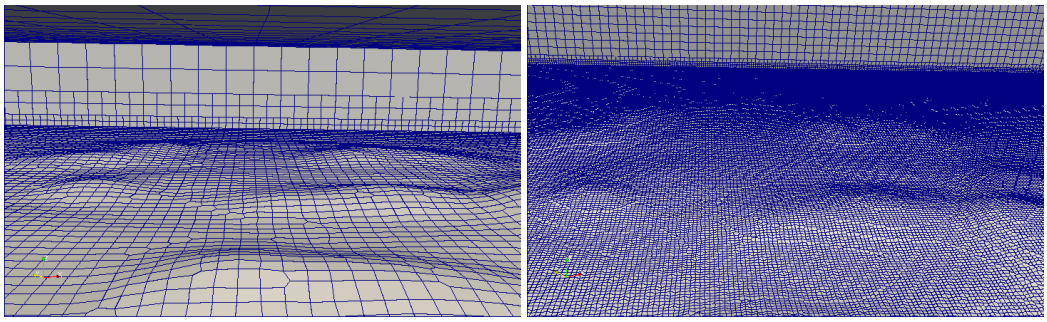


Figure 5.43: Coarse and fine meshes of Askervein hill with surrounding hills

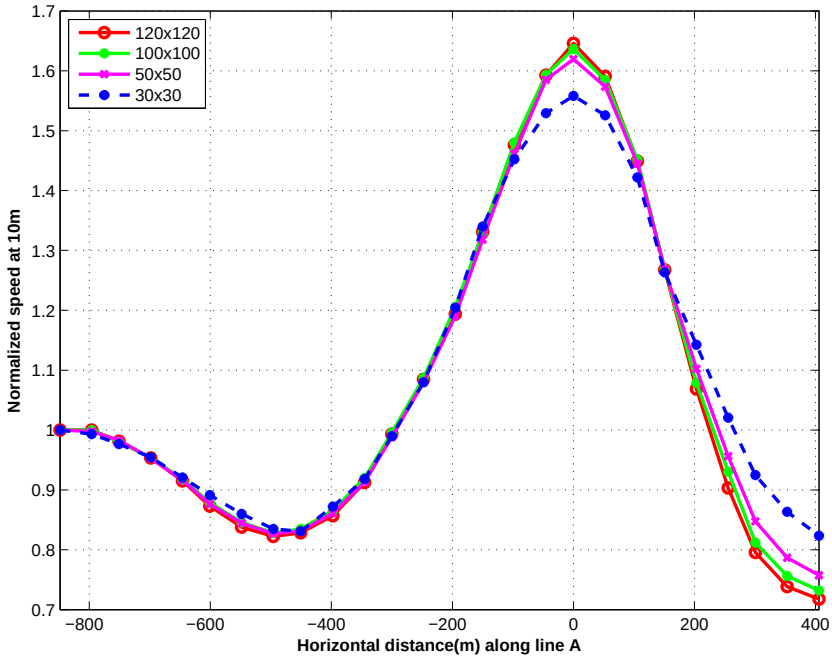


Figure 5.44: Normalized horizontal velocity for different size of grids

to use unsteady models for better understanding of re-circulation zones behind hills according to Castro et al. (2003). This study tests only RANS models even though LES simulations were attempted without success due to convergence problems. Three turbulence models namely mixing length, k-epsilon and RNG k-epsilon model are used. The results are shown in Fig. 5.45. The k-epsilon model does not show significant differences with RNG k-epsilon model in most part of the hill except at the strongest areas of re-circulation towards the bottom of the lee. The mixing length model shows differences with the above models both at the crest and the lee but in general it is not very much off as one might expect from the simplicity of the model. The reason for this observation could be that Askervein hill has a relatively gentle slope, that is favorable for linear models such as mixing length model. Nonetheless the results highlight the importance of the turbulence model to resolve the flow in the wake region. The

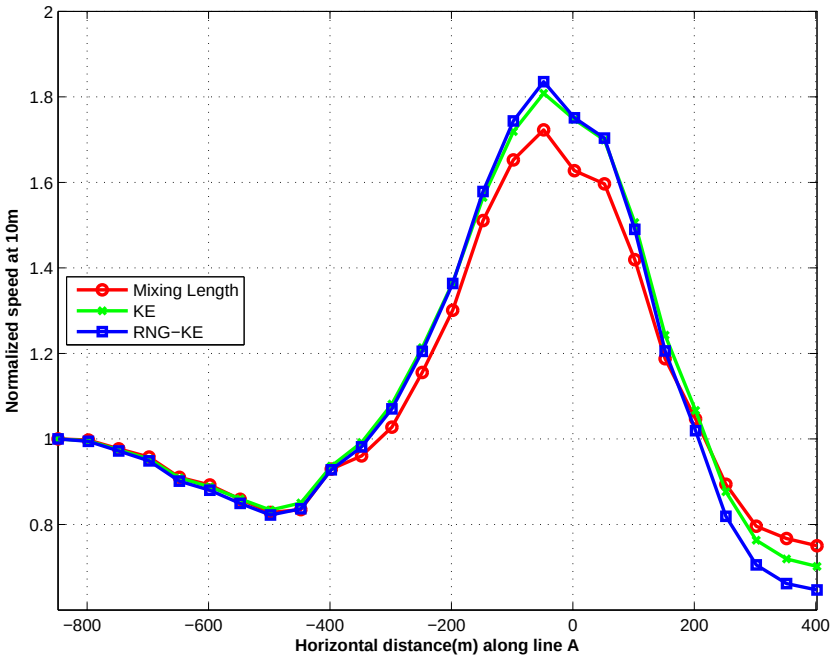


Figure 5.45: Normalized horizontal velocity for different turbulence models

grid independence study conducted in the previous section has also stressed the importance of this region.

5.3.1.4 Comparison with field measurements

The TU03-B data set along line A is used for validation of the best performing RANS model. The velocity data from the field observations is shown in Table 5.2. As we can see from Fig. 5.46, there is generally a good agreement on the upwind side of the hill, but significant differences are observed on the leeward side of the hill. This is mainly due to flow separation and unsteady flow characteristics that can not be captured with time-averaged turbulence models (Castro et al. 2003). Hence large eddy simulations may give better results. This has been investigated by many researchers including Castro et al. (2003), Crasto (2007) who concluded the superiority of LES if enough small scales are resolved and appropriate boundary conditions are used. However LES is rarely used in practice for simulation over complex terrain solely due to excessive cost of computation. In any case, our simulation results using RANS models were able to predict presence of recirculation zones to a certain extent, with the best results coming from the RNG k-epsilon turbulence model using the finest grid.

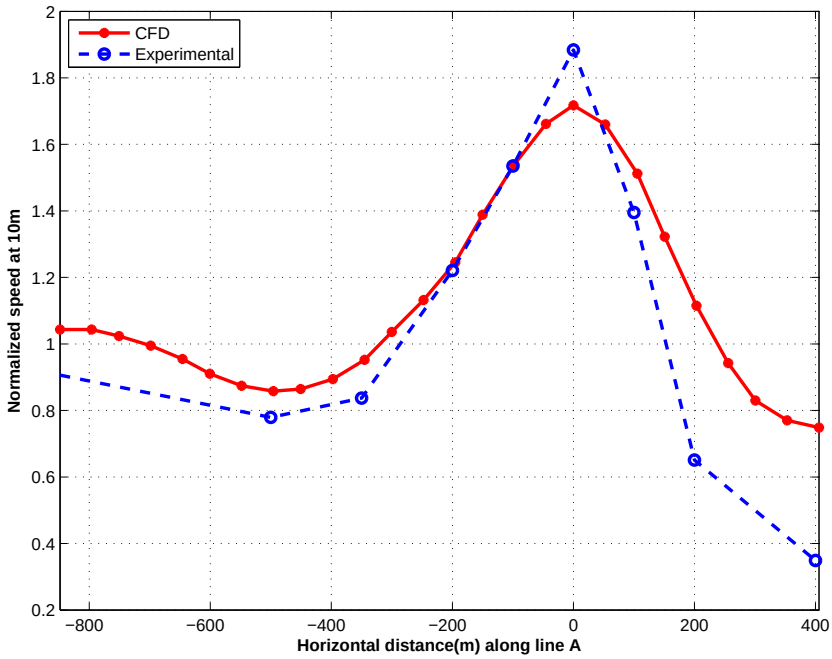


Figure 5.46: Normalized horizontal velocity comparison with field measurements

Table 5.2: Wind speed data at 10m height

Distance from HT	Wind speed (m/s)
RS	8.6
-850	7.8
-500	6.7
-350	7.2
-200	10.5
-100	13.2
HT	16.2
100	12.0
200	5.6
400	3.0

5.3.2 A second complex hill simulation

The primary reason for choosing Askervein hill is availability of field observation data for validation, but the hill can hardly be considered complex compared to other hills found elsewhere. In this section simulations are carried out on a steep and complex hill downloaded from freely available United States Geological Survey (USGS) database. Another source of terrain/built environment data, usually in point cloud format, is the International Hurricane Research Center (IHRC) that provides LiDAR data for parts of Florida. One can download terrain and roughness data for a square grid of about 5000ft X 5000ft. The point cloud data can be converted to

digital surface data (STL format) using triangulation techniques (e.g. Delaunay), after which a grid can be generated with the surface used as the base of the computational domain. The grid generation procedures is already described in the previous section of meshing idealized 3D hills, namely using snappyHexMesh, thus no additional effort is required. This is an important advantage in automatic meshing and analysis of random complex topography that would otherwise have required a lot of manual work to generate grid of acceptable quality.

The purpose of this work is not validation but to assess and demonstrate problems that may be encountered for a randomly chosen complex topography. The first difficulty arises from choosing a hill that is not surrounded too much by other topographic features. It is to be recalled that the previous case of Askervein hill is ideally placed on a flat terrain which has uniform roughness. It is not common to find such an ideally placed hill in a complex topography, therefore the boundary conditions used for a more realistic topography are usually complicated than that of Askervein's. The hill chosen for this exercise did not have such perfect conditions, but we assumed it is surrounded by a flat terrain anyway. The ground surface is cut by a horizontal plane at 1m height (shown in Fig. 5.47) to get a smooth surface and establish a zero-plane where logarithmic profiles are applied. The second problem concerns roughness that was assumed to be constant for the Askervein hill case. The current hill is in an area where vegetation and small ponds exist hence the assumption of constant roughness, as in the case of Askervein hill, is not accurate. Nonetheless, here again we make a simplifying assumption of constant roughness of $z_0 = 0.01$.

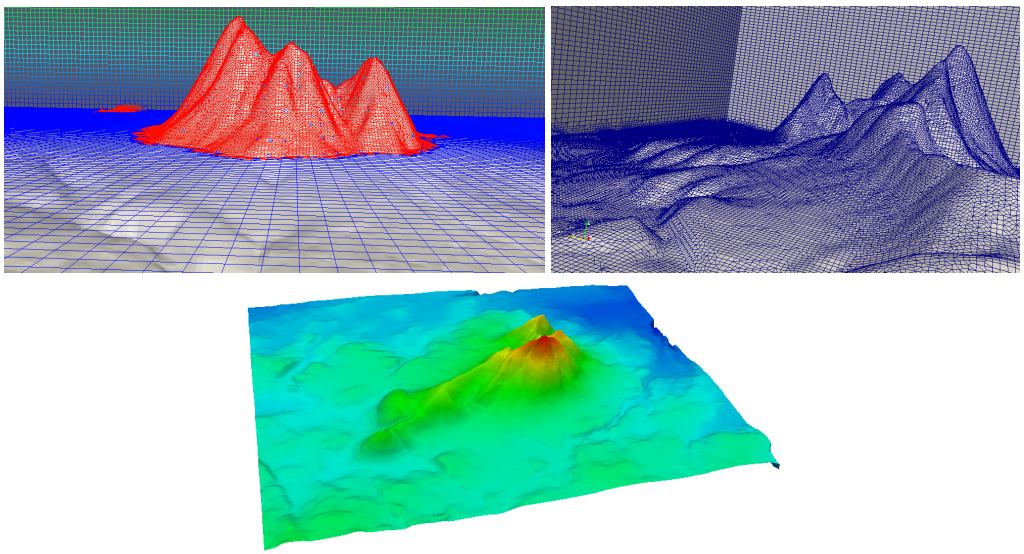
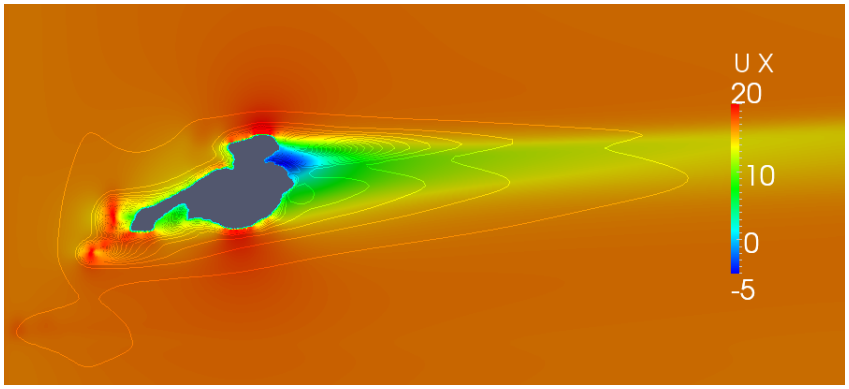
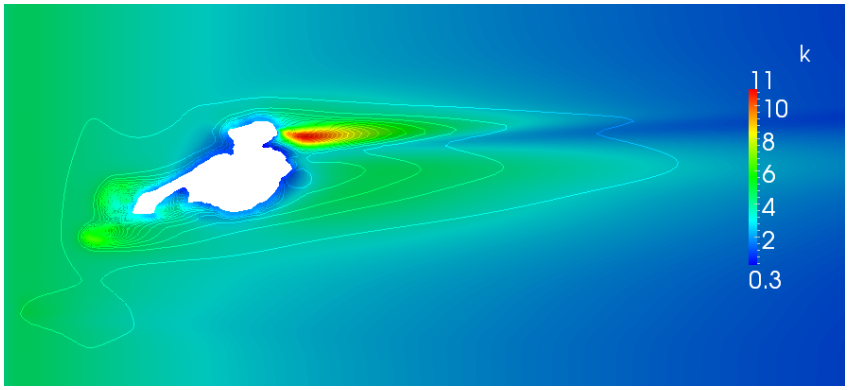


Figure 5.47: A randomly selected complex hill with sharp edges and its mesh (top), elevation contour (bottom)

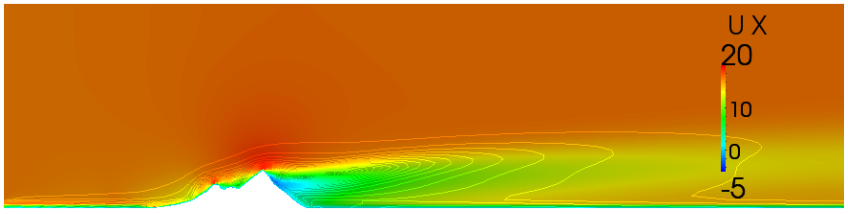
The computational domain and boundary conditions are set up following Franke & Hirsch's recommendations. Velocity and turbulence quantities are assumed at the inlet since field data is not available for comparison. Then simulations are carried out with the standard k-epsilon turbulence model. A planar section of velocity and turbulent kinetic energy contours are shown in Fig. 5.48. The flow is complex due to multiple sharp ridges, but a distinct recirculation zone, though small in size, is observed behind one of the hills. If the area is to be used for micro-sitting wind turbines it is crucial that such locations with high turbulence intensity and recirculating flow are avoided.



(a) Horizontal velocity



(b) Turbulence intensity



(c) Horizontal velocity profile

Figure 5.48: Contours of horizontal velocity and TKE

Chapter 6

Conclusions and future work

This section gives brief summary of the conclusions and findings of the current research work. The theme of the research has been numerical simulations on complex terrain and urban environment for various wind engineering purposes. The large scale nature of the project makes numerical simulations and wind tunnel testing as the most feasible investigation techniques. While field observations can potentially provide better quality data, the associated cost and the time required to carry out the investigations limit their use , except in very few cases where field observation data is required for validation purposes. The relative ease with which many numerical simulations (parametric study) can be carried out makes them suitable at least in the preliminary stage of investigations of micro-siting studies and similar purposes. Atmospheric boundary layer flow is affected by presence of topographic features such as hills, escarpments and valleys as well as surface roughness characteristics. This research has investigated these two factors separately in Chapter 4 and Chapter 5 using Computational Fluid Dynamics (CFD) methodology.

The current research has three main themes pursued in Chapter 3 to Chapter 5. The findings, conclusions and unique contributions in each chapter are briefly summarized in the following sections.

6.1 High performance CFD code

This research work started from development of an in-house CFD software tailored for Atmospheric Boundary Layer (ABL) simulations, through which the author has gained expertise on the components of CFD software. The final outcome is a 7300 lines of CFD code that has all the necessary features for ABL simulation over a complex topography. The program is a three

dimensional finite volume code and uses polyhedral elements as building blocks. The program is parallelized using Message Passing Interface (MPI) to run on a cluster of processors such as the Shared Hierarchial Academic Research Computing Network (SHARCNET) cluster at Western. It can also utilize the latest technology in high performance computing such as Graphic Processing Units (GPUs). Different turbulence models suitable for ABL simulation are implemented and tested for suitability of complex simulations. The turbulence models implemented include linear mixing-length model, many Reynolds Averaged Navier-Stokes (RANS) models including k-epsilon and RNG k-epsilon, and the Smagornisky Large Eddy Simulation (LES) model. The code is validated against well known benchmark cases including problems specific to wind engineering.

Contribution:
A common problem regarding large scale simulations is scaling of algorithms on supercomputers. CFD is usually parallelized using domain decomposition strategies in which a processor is responsible for a part of the terrain and information (pressure and velocity) is exchanged at the interfaces. Usually the parallelization scheme is done with blocking calls, where every processor is forced to synchronize at the end of each iteration. As far as the author's knowledge goes, industry standard CFD software such as Fluent and OpenFOAM suffer from scaling issues related to synchronized communication. This work has investigated a unique approach of asynchronous communication where processors can be at different stages of solution, and exchange of pressure and velocity is not necessarily forced. This has the potential to avoid bottlenecks incurred by synchronization calls, and allow for scaling on larger cluster of computers.

6.2 Effect of roughness

Chapter 4 have investigated the effect of roughness alone on wind characteristics. First simplified models with roughness blocks of different arrangement, similar to the case in a boundary layer wind tunnel, are simulated using the developed software. The simulation results are compared against empirical formulas that use average frontal and planar area density ratios. Then the problem of multiple roughness patches on the upstream side of a building is investigated. This work has investigated 3D explicit modeling of roughness elements. First a Virtual Boundary Layer Wind Tunnel (V-BLWT) is simulated by replicating all the roughness features such as spires, barrier and roughness blocks to examine the effect of each element. Then spires and

barrier are dropped in the latter simulations with the blocks as the only roughness features and a boundary layer profile applied at the inlet. This setup is used to evaluate the effect of multiple roughness features on wind profile using many test setups found in literature. The results are compared with Boundary Layer Wind Tunnel (BLWT) data and existing wind speed models. Furthermore for roughness elements that are arranged in a regular manner, the inherent symmetry is exploited to reduce the computational domain significantly. The results obtained are almost the same as that of the full virtual wind tunnel simulation. Finally a semi idealized urban environment is simulated and results validated against existing BLWT tests of the model, with which good agreements are obtained. Finally a complex models that is representative of real built environment is simulated.

Contribution:
Surface roughness is usually incorporated into CFD codes using an equivalent sand grain roughness concept. However this approach can not be used when the roughness elements are large due to conflicting requirements as outlined in Blocken et al. (2007). This work has investigated one of the approaches suggested in that work, namely explicit modeling of roughness elements for sub-urban and urban surfaces. The effect of multiple roughness patches is investigated using this approach and the results obtained are found to be in good agreement with existing wind speed models. A second contribution is an extensive use of virtual boundary layer wind tunnel to conduct simulations on multiple array of blocks. Even though it is found later that a simplified model with a single row of obstacles is enough for the purpose at hand, the virtual wind tunnel approach can be used for more complex cases. The progressive approach taken to investigate roughness effects starting from the simplest case of empty domain to a real built environment is unique.

6.3 Effect of topographic features

Topographic features such as hills, valleys and escarpments significantly modify ABL flow. While recommendations for idealized models (isolated and symmetrical) can be found in building codes and standards, complex cases are not covered well. This work extends original work done by Bitsuamlak et al. (2004) using 2D simulations on multiple topographic features to a more elaborate 3D CFD simulations. Many turbulence models, ranging from the simplest mixing-length to LES models, have been investigated . In general, the 2D simulation overestimate the speed up over crests of the hill and also has larger recirculation bubbles compared to

their 3D counterparts. Fractional Speed Up Ratio (FSUR) comparisons with results available in literature show good agreement. The next phase of the work involved conducting simulation on a real complex topography with field measurements that are used validate simulation results. The Askervein hill is selected for this study because of availability of validation data and a relatively simple digital surface model. Parametric studies are conducted for different resolutions of grid, different turbulence models and dimension of the computational domain. The results obtained show good agreement with field measurements concerning wind speed up over the upstream side, but distinct differences are observed in the wake of the hill. This is attributed to weakness of RANS turbulence models that are not able to capture the unsteady effects in recirculation zones. On upstream side of the hill, the simplest turbulence model gives comparable results with the more complex RANS models. This is mainly because of the gentle slope of the Askervein hill where linear models have the potential to perform as well as more complex turbulence models.

Contribution:

Effect of 3D orography on wind flow has been investigated using CFD simulations over idealized and real complex terrain models. Jackson (1981) first analyzed flow over an isolated hill analytically using linearized forms the governing equations. Their model is applicable only to low hills where recirculation zones are absent. In light of this original work, the current work has investigated the simplest turbulence model for non-linear CFD, i.e. mixing length model, along with along with more complex RANS and LES turbulence models. The mixing length model have been found to give good predictions of speed up over the crest and upstream side of hills. The simulations carried out in this study are done at full scale dimensions where the hill height is $H = 200m$. In literature, model scale simulations at 1:100 to 1:10000 are usually carried out. This is mainly because these simulations are meant for validating corresponding wind tunnel tests at the same scale. This work has shown that there can be a significant Reynolds number effect for the LES simulations. LES simulations on an isolated hill carried out at Reynolds number (Re) = $O(10^4)$ show significantly larger recirculation bubble compared to one carried out at full scale dimensions with wall functions and also to results obtained from RANS models as well.

6.4 Future work

Potential improvements and possible extensions of the present study are:

1. The current software can be run on homogeneous cluster of CPUs or a single GPU. Its capability can be extended to heterogeneous accelerator based many-core architecture. This will allow simulations of ever bigger atmospheric problems with better accuracy on current/future generation High Performance Computing (HPC) clusters.

2. A neutrally stratified atmospheric boundary layer (dry atmosphere) is assumed for all the micro-scale simulations conducted in this study. For domain sizes of $\sim 10km$, hydrostatic assumptions does not hold anymore. Therfore future research which considers stability of the atmosphere can be conducted to include non-hydrostatic effects due to density (temperature) variations.

3. The effect of Coriolis force has also been neglected but it is known that flow in the Ekman layer ($> 100m$) and above is significantly affected by it depending on the location of orographic features (via Rosby number) and altitude above which FSUR is computed. Thus investigation of Coriolis effects on ABL flows over topography is a potential extension of the current study.

4. While the Askervein hill is used to validate the case of flow over a real complex terrain, validation data for the corresponding real built environment case, namely the Downtown Miami case, was not available. This is a problem that plagues such studies in general, thus in the future simulations over a built environment of which field measurements are available can be conducted for validation.

References

Abdi, D., Levin, S. & Bitsuamlak, G. (2009), 'Application of an artificial neural network model for boundary layer wind tunnel profile development', *11th Americas conference on wind Engineering* .

Arnfield, A. (2003), 'Two decades of urban climate research: a review of turbulence, exchanges of energy and water, and the urban heat island', *Internation Journal of Climatology* **23**, 1 – 26.

Arya, S., Capuano, M. & Fagen, L. (1987), 'Some fluid modeling studies of flow and dispersion over two-dimensional low hills', *Atmospheric Environment - Part A General topics* **21**(4), 753 – 764.

Ayotte, K. & Hughes, D. (2004), 'Observations of boundary-layer wind-tunnel flow over isolated ridges of varying steepness and roughness', *Boundary Layer Metereology* **112**(3), 525 – 56.

Bitsuamlak, G. (2004), Evaluating the effect of topographic elements on wind flow: A combined numerical simulation neural network approach, PhD thesis, Concordia University.

Bitsuamlak, G. (2006), 'Application of computational wind engineering: A practical perspective', *Third National Conference in Wind Enginering* .

Bitsuamlak, G., Dagnew, A. & Chowdhury, G. (2010), 'Computational assessment of blockage and wind simulator proximity effects for a new full-scale testing facility', *Wind and Structures* **13**(1), 21 – 36.

Bitsuamlak, G., Stathopoulos, T. & Bedard, C. (2004), 'Numerical evaluation of wind flow over complex terrain: review', *Journal of Aerospace Engineering* **17**(4), 135 – 145.

Bitsuamlak, G., Stathopoulos, T. & Bedard, C. (2006), 'Effects of upstream two dimensional hills on design wind loads: a computational appraoch', *Wind and Structures* **9**(1), 37 – 38.

Blocken, B. & Carmeliet, J. (2004), 'Pedestrian wind environment around buildings: Literature review and practical examples', *Journal of Thermal Env. and Building Physics* **28**(2), 107 – 159.

Blocken, B., Janssen, W. & Hooff, T. (2011), 'CFD simulation for pedestrian wind comfort and wind safety in urban areas: General decision framework and case study for the eindhoven university campus', *Environmental Modelling and Software* **28**, 15 – 34.

Blocken, B., Stathopaulos, T. & Carmeliet, J. (2007), 'CFD simulation of the atmospheric boundary layer: wall function problems', *Journal of Wind Engineering and Ind. Aero.* **41**(2), 238 – 252.

Blocken, B., Stathopaulos, T., Carmeliet, J. & Hensen, J. (2009), 'Application of CFD in building performance simulation for the outdoor environment', *Eleventh International IBPSA Conference* **4**(2), 157 – 184.

Botella, O. & Peyret, R. (1998), 'Benchmark results for the lid driven cavity flow', *Computers and Fluids* **27**(4), 421 – 433.

Bradley, E. (1968), 'A micrometeorological study of velocity profiles and surface drag in the region modified by a change in surface roughness', *Journal of the Royal Meteorological Society* **94**, 361–379.

Cao, S. & Tamura, T. (2007), 'Effects of roughness blocks on atmospheric boundary layer flow over a two-dimensional low hill with/without sudden roughness change', *Journal of Wind Engineering and Ind. Aero.* **95**(8), 679 – 695.

Carpenter, P. & Locke, N. (1999), 'Investigation of wind speeds over multiple two-dimensional hills', *Journal of Wind Engineering and Ind. Aero.* **83**(1 - 3), 109 – 120.

Castro, F., Palma, J. & Silvia, A. (2003), 'Simulation of the askervein flow. part 1: Reynolds averaged navier-stokes equations(k-e turbulence model)', *Boundary Layer Meteorology* **107**(3), 501.

CEDVAL-LES (2011), 'Compilation of experimental data for validation of microscale dispersion models: www.mi.uni-hamburg.de/cedval-les-v.6332.0.html', *Meterological Institute, University of Hamburg, Germany* .

Chazan, D. & Miranker, W. (1969), 'Chaotic relaxation', *Linear Algebra and its Applications* **2**, 199–222.

Chung, J. & Bienkiewicz, B. (2004), 'Numerical simulation of flow past 2d hill and valley', *Wind and Structures* **7**(1), 1–12.

Coceal, O. & Belcher, S. (2005), 'Mean winds through an inhomogeneous canopy', *Boundary Layer Meteorology* **115**(1), 47 – 68.

Corrigan, A., Fernando, C. & Lohner (2009), 'Running unstructured grid based solvers on modern graphics hardware', *19th AIAA Computational Fluid Dynamics* .

Counihan, J. (1971), 'Wind tunnel determination of the roughness length as a function of the fetch and roughness density of three dimensional roughness elements', *Atmoshpheric Environment* **5**(8), 637 – 642.

Crasto, G. (2007), Numerical simulation of the atmospheric boundary layer, PhD thesis, Universita degli Studi di Cagliari.

Dagnew, A. & Bitsuamlak, G. (2013), 'Computational evaluation of wind loads on buildings: a review', *Wind and Structures* **16**(6), 629 660.

Davenport, A., Grimmond, C., Oke, T. & Wieringa, J. (2000), 'Estimating the roughness of cites and sheltered country', *Proceedings of the 12th American Meteorological Society Conference On Applied Climatology* .

Deaves, D. (1981), 'Computation of wind flow over changes in surface roughness', *Journal of Wind Engineering and Ind. Aero.* **7**(1), 65 – 94.

Deaves, D. & Harris, R. (1978), *A mathematical model of the structure of strong winds*, Report. Construction Industry Research and Information Association, CIRIA report 76.

Dupont, S., Brunet, Y. & Finnigan, J. (2008), 'Large eddy simulation of turbulent flow over a forested hill: Validation and coherent structure identification', *Journal of the Royal Meteorological Society* **134**(636), 1911 1929.

EPA (1987), 'On-site meteorological program guidance for regulatory modeling applications', *USEPA, OAQPS, Research Triangle Park, North Carolina, EPA-450/4-87-013* .

ESDU-82026 (1993), *Strong winds in the atmospheric boundary layer, Part 1: hourly-mean wind speeds.*, Engineering Science Data Unit.

ESDU-84030 (1993), *Longituidinal turbulence intensities over terrain with multiple roughness changes.*, Engineering Science Data Unit.

Eugene, D. (2006), The potential of large eddy simulation for the modelling of wall bounded flows, PhD thesis, Imperial College of Science.

Feng, W. & Fernando, P. (2011), 'Large eddy simulation of stably stratified flow over a steep hill', *Boundary Layer Metereology* **138**(3), 367 – 384.

Ferreira, A. D., Lopes, A. M. G., Viegas, D. X. & Sousa, A. C. M. (1995), 'Experimental and numerical simulation of flow around two-dimensional hills', *Journal of Wind Engineering and Ind. Aero.* **54**(55), 173 – 181.

Ferziger, J. & Peric, M. (2001), *Computational methods for fluid dynamics*, third edn, Springer-Verlag, Berlin.

Finnigan, J. (1988), *Air flow over complex terrain*, Springer Verlag.

Finnigan, J., Raupach, M., Bradley, E. & Aldis, G. (1990), 'A wind tunnel study of turbulent flow over a two-dimensional ridge', *Boundary Layer Metereology* **50**(1 - 4), 277– 317.

Franke, J. & Hirsch, C. (2004), 'Recommendations on the use of CFD in wind engineering', *Proceedings of International Conference in Urban Wind Engineering and Building Aerodynamics* .

Fredrick, B. & Marc, S. (2010), 'Ghost cell pattern', *Proceedings of the 2010 Workshop on Parallel Programming Patterns* **4**.

Gardner, A. (2004), A full-scale investigation of roughness lengths in inhomogeneous terrain and a comparison of wind prediction models for transitional flow regimes, PhD thesis, Texas Tech University.

Garrat, J. (1989), 'The internal boundary layer - a review', *Boundary Layer Meteorology* **50**(1 - 4), 171 – 203.

Glanville, M. & Kwok, K. (1997), 'Measurement of topographic multipliers and flow separation from a steep escarpment. Part II. Model scale measurement', *Journal of Wind Engineering and Ind. Aero.* **69**(71), 893 902.

Gong, W. & Ibbetson, A. (1989), 'A wind tunnel study of turbulent flow over model hills', *Boundary Layer Metreology* **49**(1 - 2), 113 148.

Goyon, O. (1996), 'High reynolds number solutions of navier-stokes equations using incremental unknowns', *Computer Methods in App. Mech. and Engg.* **130**(3 - 4), 319 335.

Grant, A. & Mason, P. (1990), 'Observation of boundary layer structure over complex terrain', *Journal of the Royal Meteorological Society* **116**, 159 – 186.

Griffiths, A. & Middelton, J. (2010), 'Simulations of separated flow over two-dimensional hills', *Journal of Wind Engineering and Ind. Aero.* **98**(3), 155 – 160.

Grimmond, C. & Oke, T. (1999), 'Aerodynamic properties of urban areas derived from analysis of surface form', *Journal of Applied Metreology* **38**, 1262 – 1292.

Gropp, W., Lusk, E. & Skjellum, A. (1999), *Using MPI: Portable Parallel Programming with the Message Passing Interface,* 2nd edition, MIT Press, Cambridge, MA.

Hall, D., Macdonald, R., Walker, S. & Spanton, A. (1996), 'Measurements of dispersion within simulated urban arrays: A small scale wind tunnel study', *BRE Client Report CR 178/96* .

Hansen, F. (1993), 'Surface roughness lengths', *ARL Technical Report U. S. Army, White Sands Missile Range, NM 88002-5501.* .

Hargreeves, D. & Wright, N. (2007), 'On the use of k-epsilon model in commercial cfd software to model the atmospheric boundary layer', *Journal of Wind Engineering and Ind. Aero.* **95**, 355 – 369.

Hertwig, D., Efthimiou, G., Bartzis, J. & Leitl, B. (2012), 'Cfd-rans model validation of turbulent flow in a semi-idealized urban canopy', *Journal of Wind Engineering and Ind. Aero.* **111**, 61 – 72.

Horsfield, J., Chan, C. & Denoon, R. (2002), 'Towards sustainable development through innovative engineering', *Housing Conference, Wanchai,Hong Kong* .

Iizuka, S. & Kondo, H. (2006), 'Large eddy simulations of turbulent flow over complex terrain using modified static eddy viscosity models', *Atmospheric Envirnoment* **40**(5), 925 – 935.

Irwin, H. (1979), 'Design and use of spires for natural wind simulation', *National Research Council Canada, National Aeronautical Establishment, Report LTR-LA-233* .

Ishihara, T., Hibi, K. & Oikawa, S. (1999), 'A wind tunnel study of turbulent flow over a three-dimensional steep hill', *Journal of Wind Eng. and Ind. Aero* **83**(1 - 4), 95 – 107.

Jackson, P. (1981), 'On the displacemtn height in the velocity profile', *Jounral of Fluid Mechanics* **111**, 15 – 25.

Jackson, P. & Hunt, J. (1975), 'Turbulent wind flow over a low hill', *Jounral of Royal Metreology Society* **101**, 929 – 955.

Jasak, H. (1996), Error analysis and estimation for the finite Volume method with applications to fluid flows, PhD thesis, Imperial College of Science.

Jasak, H., Jemcov, A. & Tukovic, Z. (2007), 'A c++ library for complex physics simulations', *International Workshop on Coupled Methods in Numerical Dynamics* .

Jimenez, J. (2004), 'Turbulent flow over rough walls', *Annual Rev. Fluid Mechanics* **36**, 173 – 196.

Julien, C. & Senocak, I. (2009), 'CUDA Implementation of a Navier-Stokes Solver on Multi-GPU Desktop Platforms for Incompressible Flows', *47th AIAA Aerospace Sciences Meeting and Exhibit* .

Kato, M. & Launder, B. (1993), 'The modeling of turbulent flow around stationary and vibrating square cylinders', *Proc. 9th Symposium on Turbulent Shear Flows, Kyoto* pp. 10.4.1 – 10.4.6.

Kose, D. & Dick, E. (2010), 'Prediction of pressure distribution on a cubical building with implicit les', *Journal of Wind Engineering and Ind. Aero.* **115**(10 - 11), 628 649.

Launder, B., Reece, G. & Rodi, W. (1975), 'Progress in the development of reynolds-stess turbulent closure', *Journal of Fluid Mechanics* **68**(3), 537 – 566.

Launder, B. & Spalding, D. (1974), 'The numerical computation of turbulent flows', *Computational Methods App. Mech. Engg* **3**(2), 269 – 289.

Lee, S., Lim, H. & Park, K. (2002), 'Wind flow over sinusoidal hilly obstacles located in a uniform flow', *Wind and Structures* **5**(6), 515–526.

Letchford, C., Gardner, A., Howard, R. & Schroeder, J. (2001), 'A comparison of wind predcition models for transitional wind flow regimes using full scale hurricane data', *Journal of Wind Engineering and Ind. Aero.* **89**(10), 925 – 945.

Lettau, H. (1969), 'Note on aerodynamic roughness parameter estimation on the basis of roughness element description', *Journal of Applied Metereology* **8**, 828 – 833.

Lim, H., Thomas, T. & Castro, I. (2009), 'Flow around a cube in a turbulent boundary layer', *Journal of Wind Engineering and Ind. Aero.* **97**(2), 96–109.

Lo, A. (1990), 'On the determination of zero-plane displacement height and roughness length for flow over forest canopies', *Boundary Layer Metereology* **51**(3), 225 – 268.

Lubitz, W. & White, B. (2007), 'Wind-tunnel and field investigation of the effect of local wind direction on speed-up over hills', *Journal of Wind Engineering and Ind. Aero.* **95**(8), 639 – 661.

MacDonald, R., Griffiths, R. & Hall, D. (1998), 'An improved method for the estimation of surface roughness of obstacle arrays', *Atmospheric Environment* **32**(11), 1857 – 1864.

Martinez, B. (2011), Wind resource in complex terrain with openfoam, Master's thesis, Technical University of Denmark.

Mason, P. & Sykes, R. (1979), 'Flow over an isolated hill of moderate slope', *Journal of the Royal Meteorological Society* pp. 383 – 395.

Maurizi, A., Palma, M. & Castro, J. (1998), 'Numerical simulation of the atmospheric flow in a mountainous region of the north of portugal', *Journal of Wind Engineering and Ind. Aero.* **74 - 76**, 219 – 228.

Meuer, H. (2013), 'Top 500 supercomputers'.
URL: *http://www.top500.org/*

Miles, S. & Westbury, P. (2003), 'Practical tools for wind engineering in the built environment', *The QNET-CFD network newsletter* **2**.

Miller, C. & Davenport, A. (1998), 'Guidelines for the calculation of wind speed ups in complex terrain', *Journal of Wind Engineering and Ind. Aero.* **74 - 76**, 189 – 197.

Monteith, J. (1965), 'Evaporation and environment', *Symp Soc Expl Biol* pp. 205 – 234.

Nicholas, J. (1997), 'The deaves and harris abl model applied to heterogeneous terrain', *Journal of Wind Engineering and Ind. Aero.* **66**(3), 197 – 214.

Oke, T. (1998), 'Street design and urban canopy layer climate', *Energy Building* **103**(1 - 3), 103 – 113.

OpenFOAM (2013), 'Openfoam, the open source cfd toolbox'.
URL: *http://www.openfoam.com/*

O'Sullivan, J., Archer, R. & Flay, R. (2011), 'Consistent boundary conditions for flows within the atmospheric boundary layer', *Journal of Wind Engineering and Ind. Aero.* **99**(9), 66 67.

Panofsky, H. & Dutton, J. (1984), *Atmospheric turbulence*, first edn, John Willey.

Patankar, S. (1980), *Numerical heat transfer and fluid flow*, first edn, Hemisphere Publishing Corporation.

Peterson, R. (1994), 'A wind tunnel evaluation of methods for estimating roughness length at industrial facilities', *Atmospheric Environment* **31**(1), 45 – 57.

Pope, S. (2000), *Turbulent flows*, first edn, Cambridge University Press.

Rasheed, A. (2010), 'On the effects of complex urban geometries in meso-scale modeling', *The Fifth International Symposium on Computational Wind Engineering (CWE2010)* .

Rasoulli, A. (2010), Experimental and numerical modelling of wind flow over complex topography, PhD thesis, University of Western Ontario.

Rasoulli, A. & Hangan, H. (2013), 'Microscale computational fluid dynamics simulaton for wind mapping over complex topography terrains', *Journal of Solar Energy Engineering* **135**(4), 1–18.

Raupach, M. (1992), 'Drag and drag partitions on rough surfaces', *Boundary Layer Meterology* **60**(4), 375 – 395.

Raupach, M., Antonia, R. & Rajagopalan, S. (1991), 'Rough wall turbulent boundary layers', *Applied Mechanics Review* **44**(1), 1 – 25.

Richards, P. & Hoxey, R. (1993), 'Appropriate boundary conditions for computational wind engineering models using the k-epsilon turbulence model.', *Journal of Wind Engineering and Ind. Aero.* **46**, 145 – 153.

Richardson, L. (1922), *Weather prediction by numerical process*, 1st edn, Cambridge Univ. Press.

Selvama, P. & Landrus, K. (2010), 'Gpu computing for wind engineering', *The Fifth International Symposium on Computational Wind Engineering (CWE2010)* .

Shuyang, C. & Tetsuro, T. (2006), 'Experimental study on roughness effects on turbulent boundary layer flow over a two-dimensional steep hill', *Journal of Wind Engineering and Ind. Aero.* **94**(1), 1 – 19.

Shuyang, C. & Tetsuro, T. (2007), 'Effects of roughness blocks on atmospheric boundary layer flow over a two dimensional low hill with and without sudden roughness change', *Journal of Wind Engineering and Ind. Aero.* **95**(9), 679 695.

Snyder, W. & Britter, R. (1987), 'A wind tunnel study of the flow structure and dispersion from sources upwind of three-dimensional hills', *Atompsheric Environment - Part A General Topics* **21**(4), 735 – 751.

Spalding, D. (1994), 'Calculation of turbulent heat transfer in cluttered spaces', *10th International Heat Transfer Conference* .

Stangroom, P. (2004), CFD modeling of wind flows over terrain, PhD thesis, University of Nottingham.

Stull, R. (1988), *An introduction to boundary layer metereology*, first edn, Kluwer Academic.

Takeshi, I. & Hibi, K. (2002), 'Numerical study of turbulent wake flow behind a three-dimensional steep hill', *Wind and Structures* **5**(2-4), 317–328.

Takeshi, I., Kazuki, H. & Susumu, O. (1999), 'A wind tunnel study of turbulent flow over a three dimensional steep hill', *Journal of Wind Engineering and Ind. Aero.* **83**, 95 – 103.

Tamura, T., Okuno, A. & Sugio, Y. (2007), 'LES analysis of turbulent boundary layer over 3d steep hill covered with vegetation', *Journal of Wind Engineering and Ind. Aero.* **95**(9 - 11), 1463 – 1475.

Taylor, P. & Teunisson, H. (1986), 'The askervein hill project: 0verview and background data.', *Boundary Layer Meteorology* **39**(1-2), 15 – 39.

Theurer, W. (1993), Dispersion of ground level emissions in complex built-up areas, PhD thesis, University of Karlsruhe.

Thom, A. (1972), 'Momentum, mass and heat exchange of vegetation', *Journal of the Royal Meteorological Society* **98**, 124 – 134.

Tsang, C., Kwok, K., Hitchcock, P. & Hui, D. (2009), 'Numerical study of turbulent wake flow behind a three-dimensional steep hill', *Wind and Structures* **5**(2-4), 317–328.

Uchida, T. & Ohya, Y. (2008), 'Micro-siting technique for wind turbine generators by using large-eddy simulation', *Journal of Wind Engineering and Ind. Aero.* **96**(10-11), 2121 2138.

Wang, K. & Stathopoulos, T. (2007*a*), 'Exposure model for wind loading of buildings', *Journal of Wind Engineering and Ind. Aero.* **95**(9-11), 1511–1525.

Wang, K. & Stathopoulos, T. (2007*b*), Modeling Terrain Effects and Application to the Wind Loading of Low Buildings, PhD thesis, Concordia University.

Weller, H., Tabor, G., Jasak, H. & Fureby, C. (1998), 'A tensorial approach to computational continum mechanics using object oriented techniques', *Computers in Physics* **12**(6), 620 – 631.

Weng, W., Taylor, P. & Walmsley, J. (2000), 'Guidelines for airflow over complex terrain: model developments', *Journal of Wind Engineering and Ind. Aero.* **83**(2-3), 169 – 186.

Wieringa, J. (1992), 'Updating the davenport roughness classification', *Journal of Wind Engineering and Ind. Aero.* **41**(1-3), 357 368.

Wieringa, J. (1993), 'Representative roughness parameters for homogeneous terrain', *Boundary Layer Meteorology* **63**, 323 – 363.

Wright, N. & Easom, G. (2003), 'Non-linear k-e turbulence model results for flow over a building at full scale', *Applied Math. Model* **27**(12), 1013 – 1033.

Xabier, P. (2009), Modelling of wind flow over complex terrain using openfoam, Master's thesis, University of Guvle.

Zaki, S., Hagishma, A., Tanimoto, J. & Ikegaya, N. (2010), 'Wind tunnel measurement of aerodynamic parameters of urban building arrays with random geometries', *The Fifth International Symposium on Computational Wind Engineering (CWE2010)* .

Appendix A

Plots of wind speed model

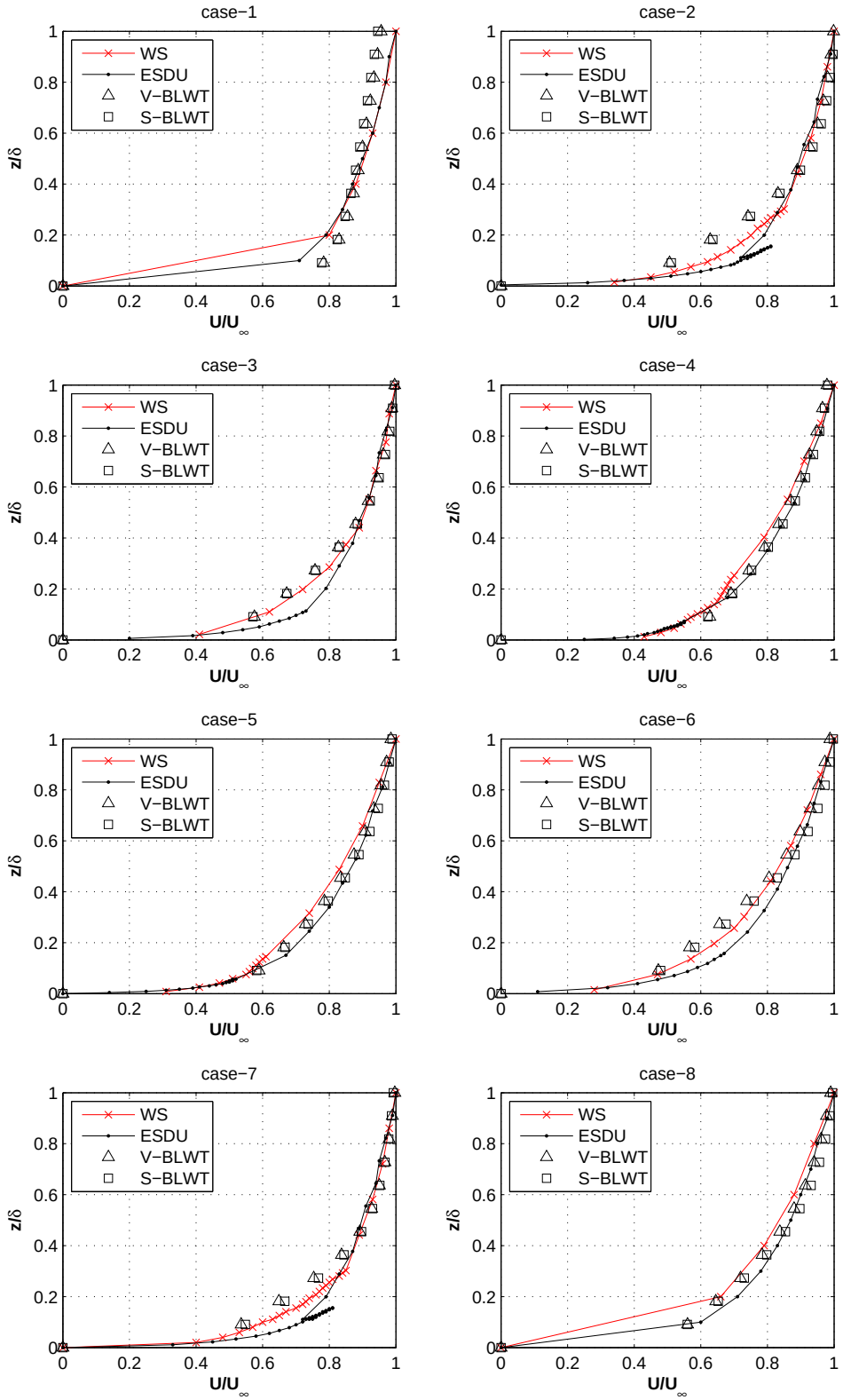


Figure A.1: Horizontal velocity comparison of CFD with existing models for cases 1-8

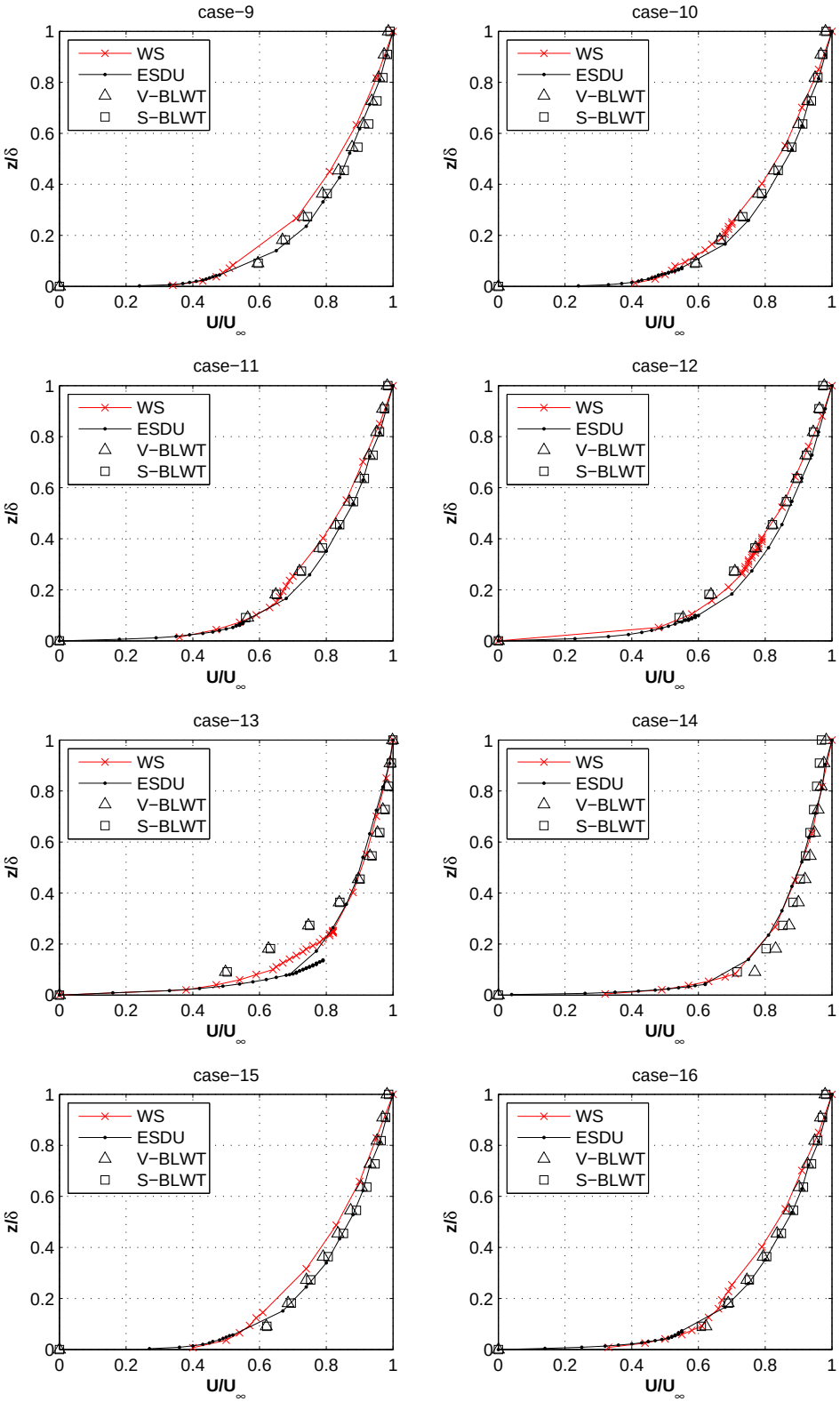


Figure A.2: Horizontal velocity comparison of CFD with existing models for cases 9-16

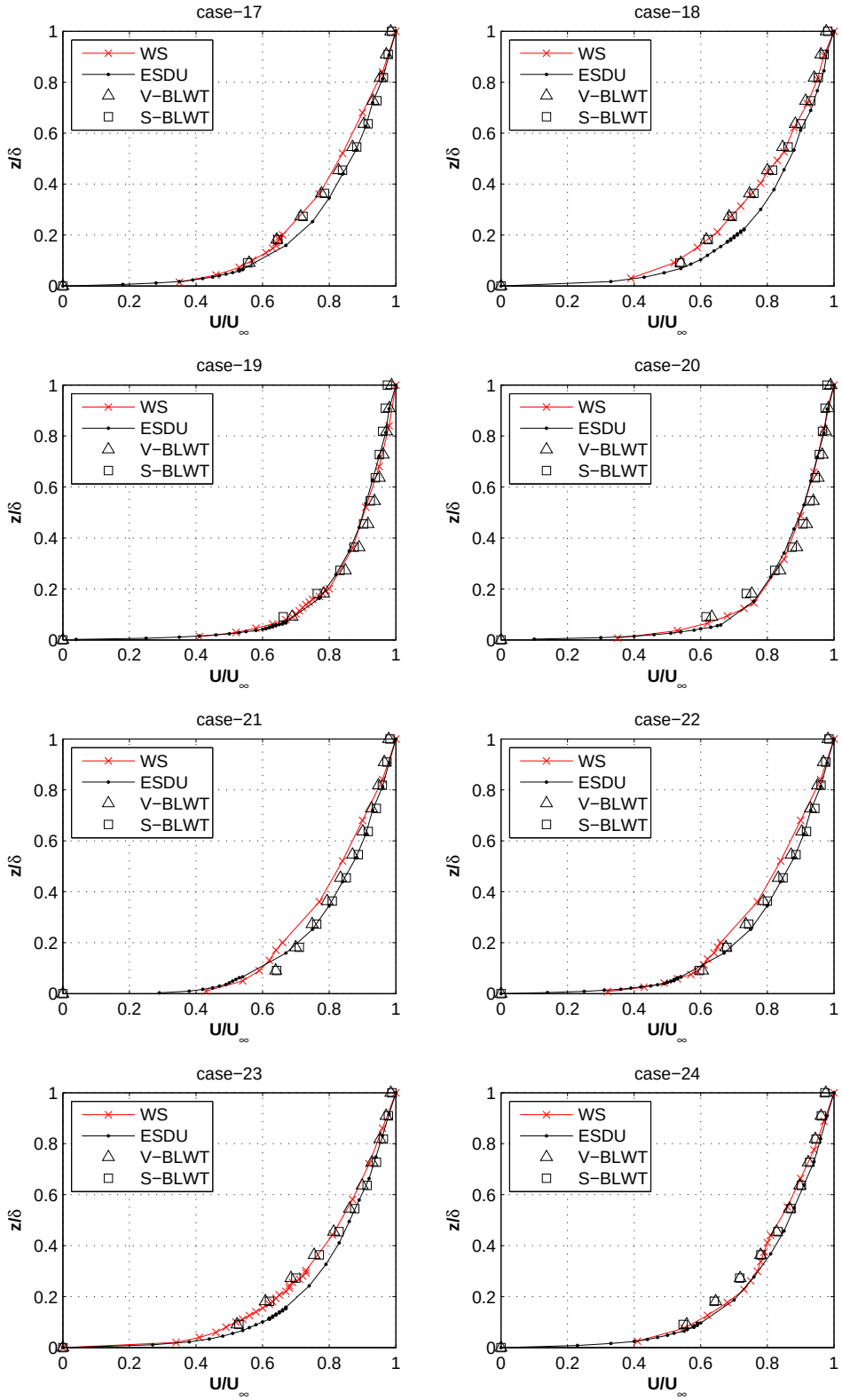


Figure A.3: Horizontal velocity comparison of CFD with existing models for cases 17-24

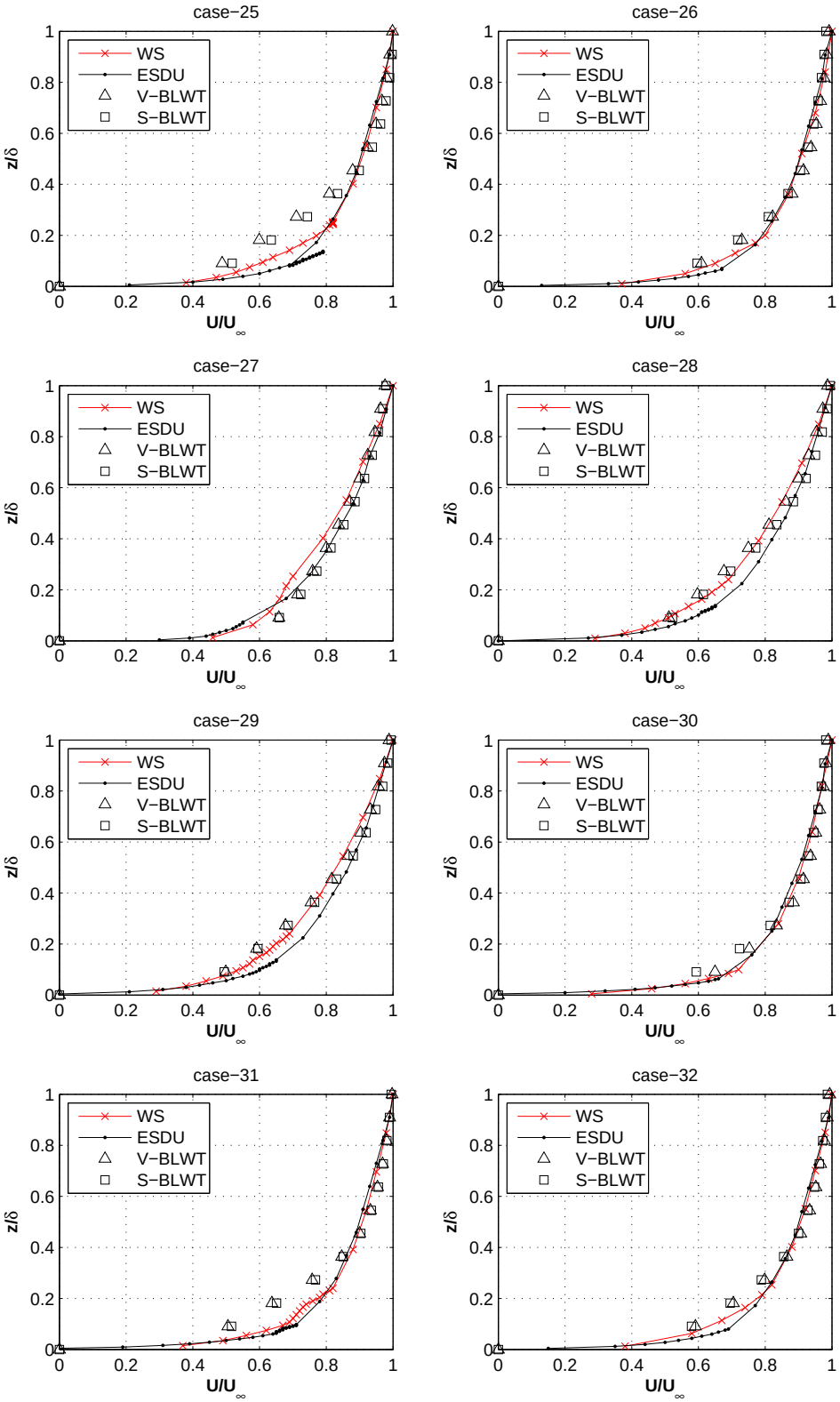


Figure A.4: Horizontal velocity comparison of CFD with existing models for cases 25-32

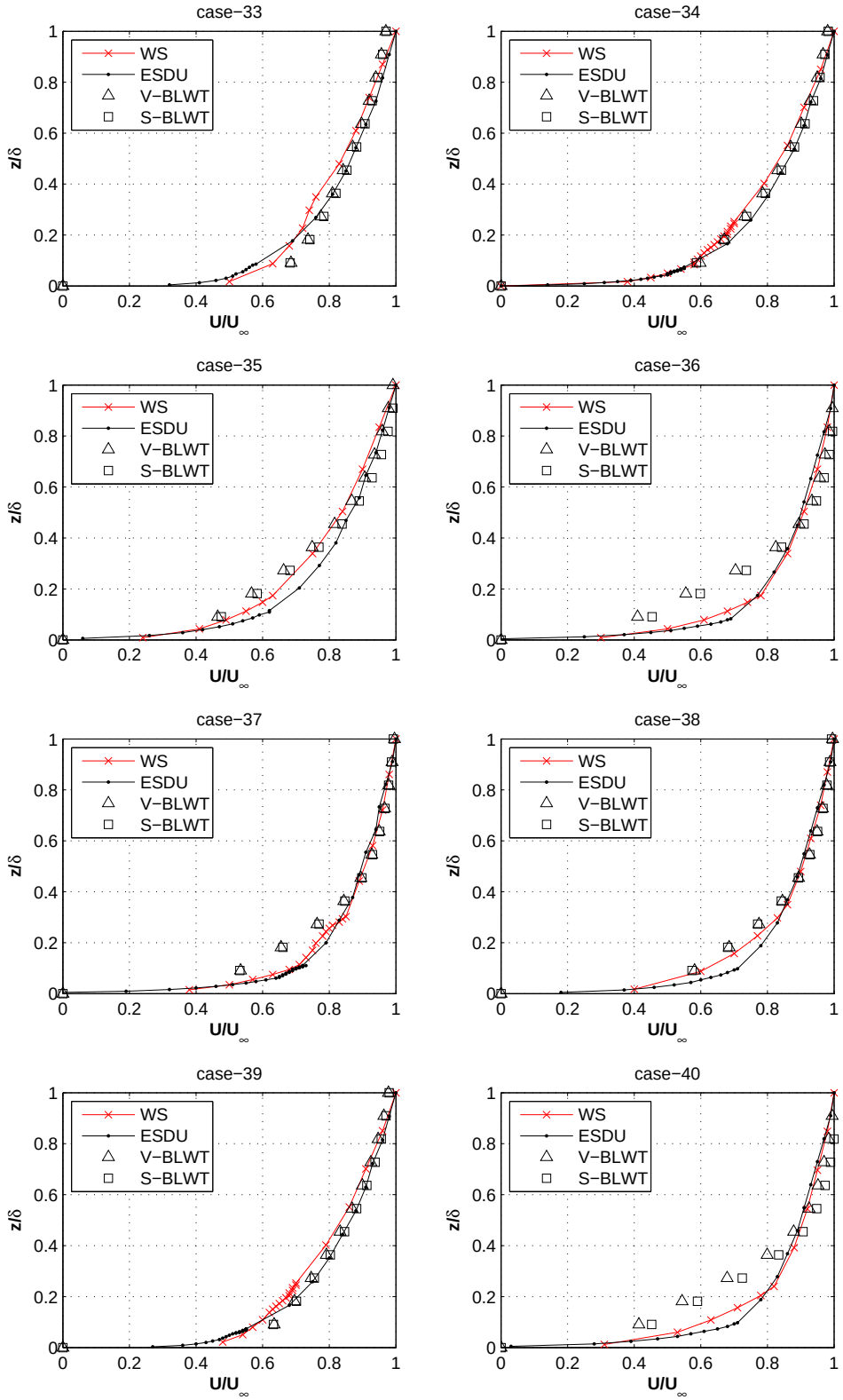


Figure A.5: Horizontal velocity comparison of CFD with existing models for cases 33-40

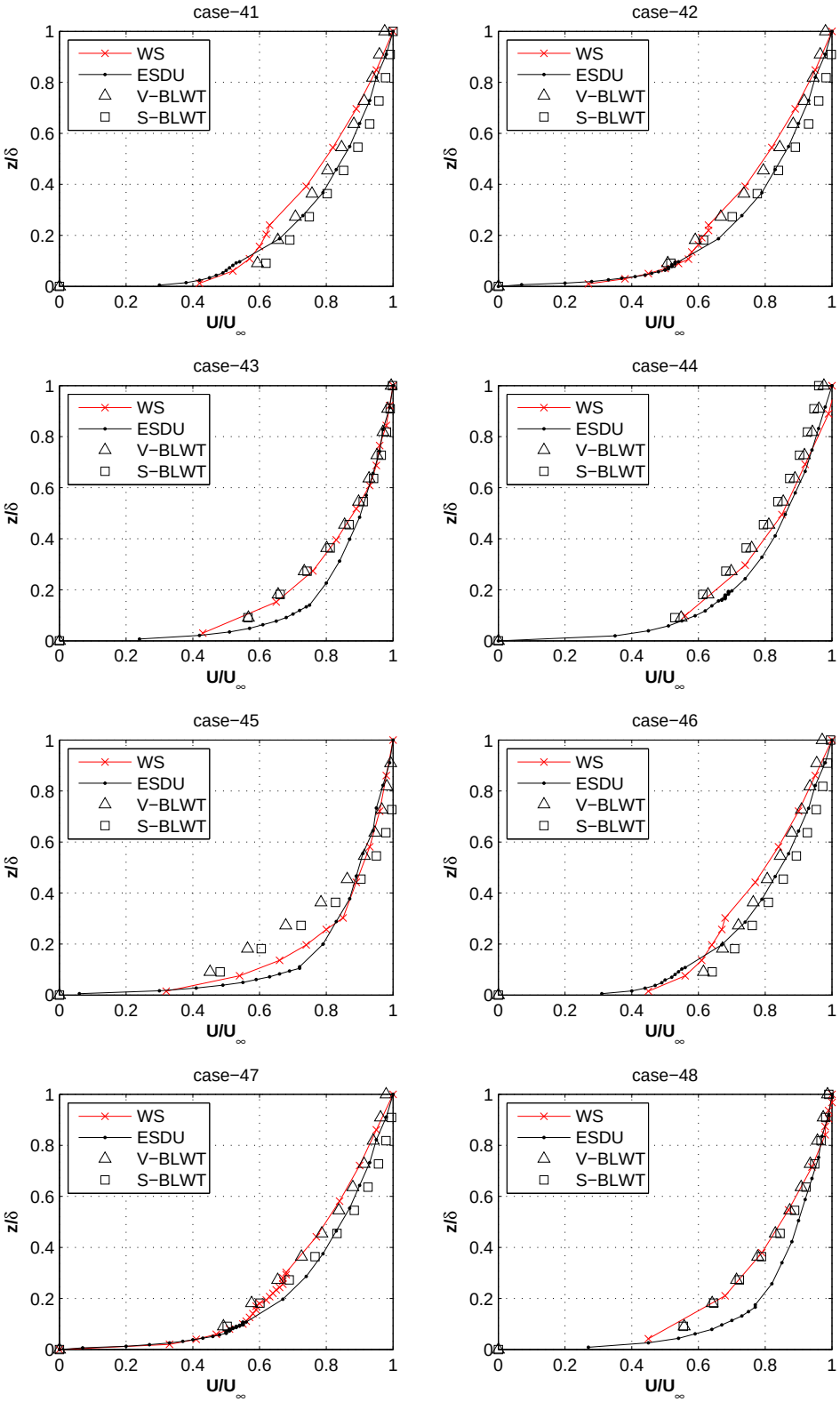


Figure A.6: Horizontal velocity comparison of CFD with existing models for cases 41-48

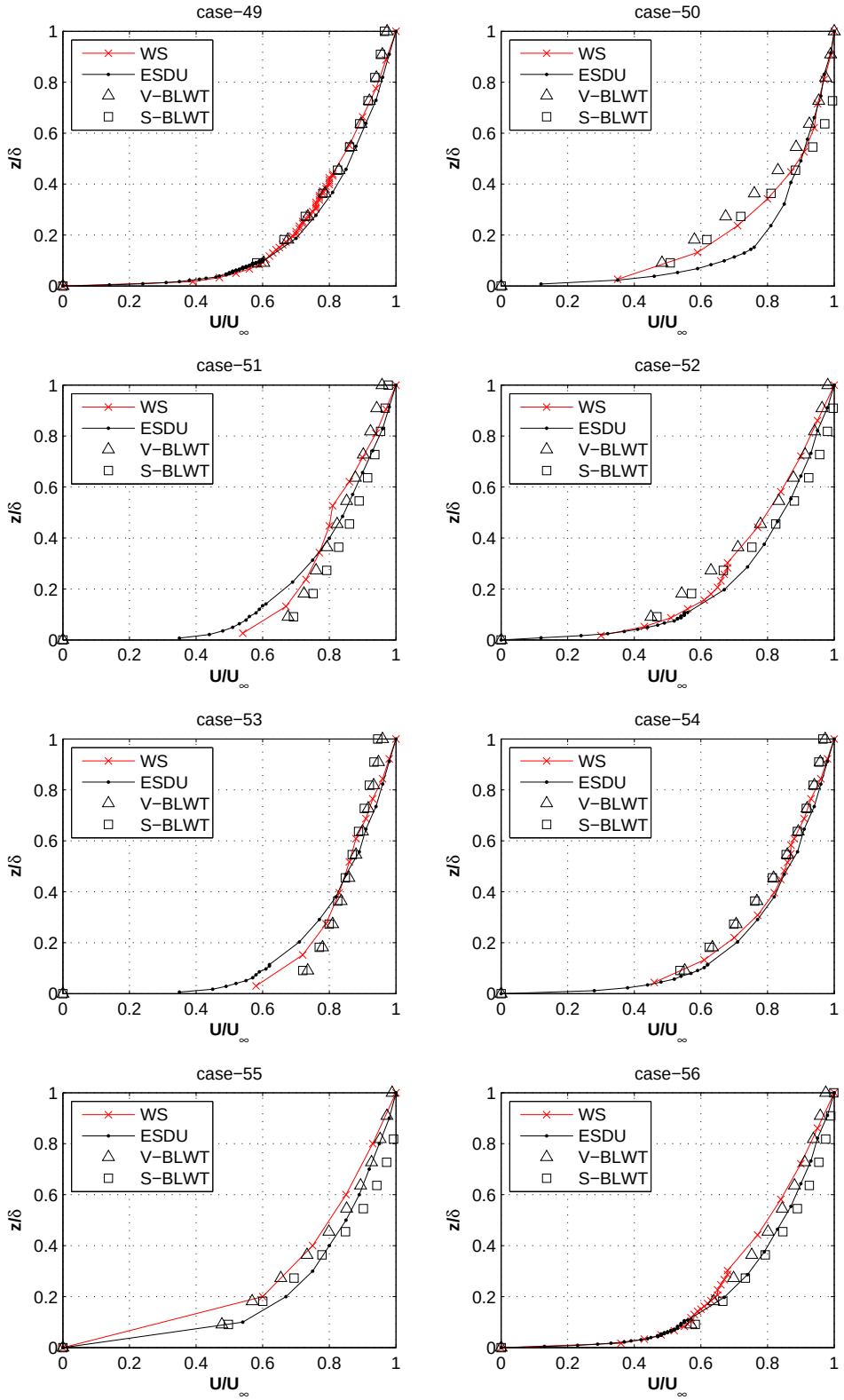


Figure A.7: Horizontal velocity comparison of CFD with existing models for cases 49-56

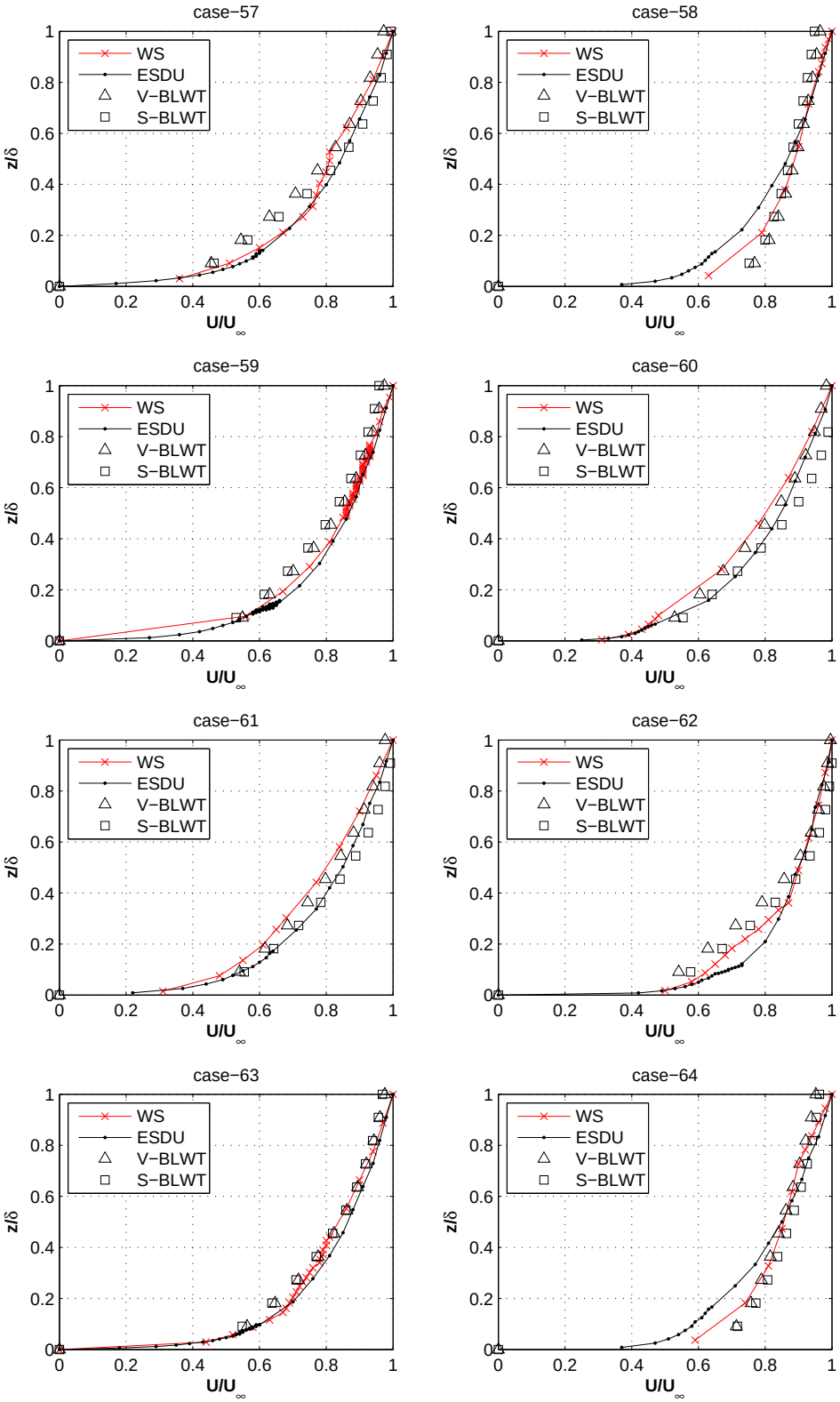


Figure A.8: Horizontal velocity comparison of CFD with existing models for cases 57-64

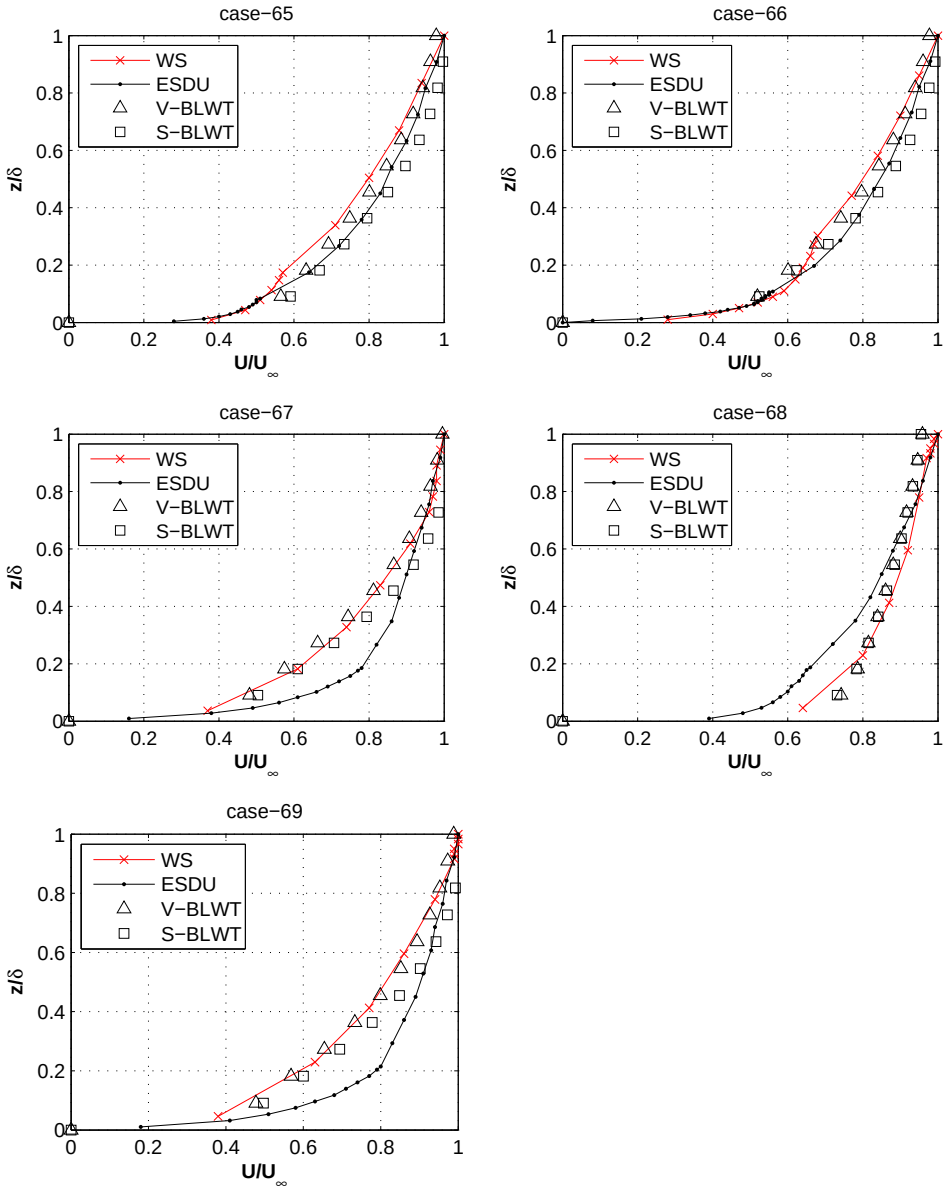


Figure A.9: Horizontal velocity comparison of CFD with existing models for cases 65-69

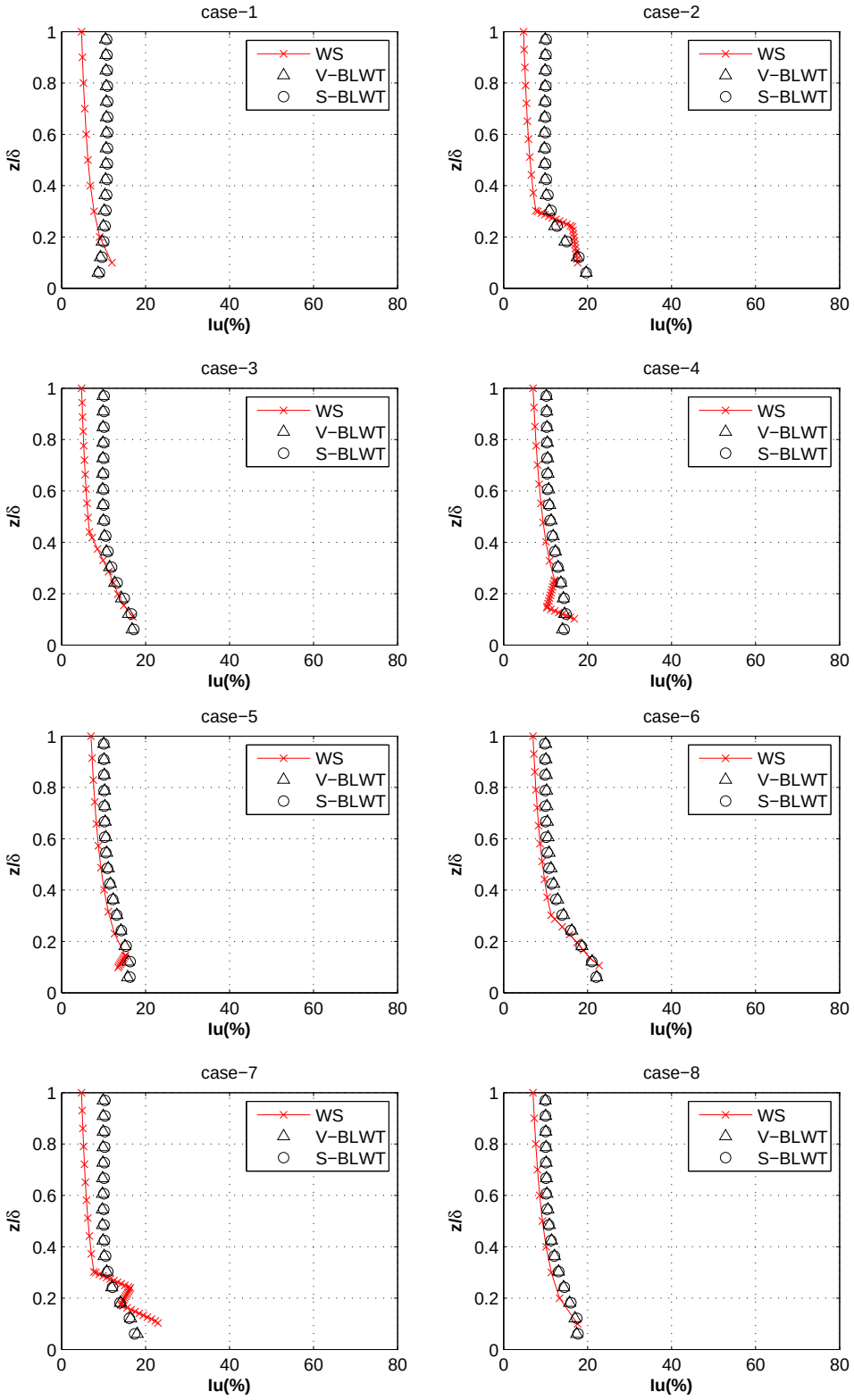


Figure A.10: Turbulence intenisy comparison of CFD with existing models for cases 1-8

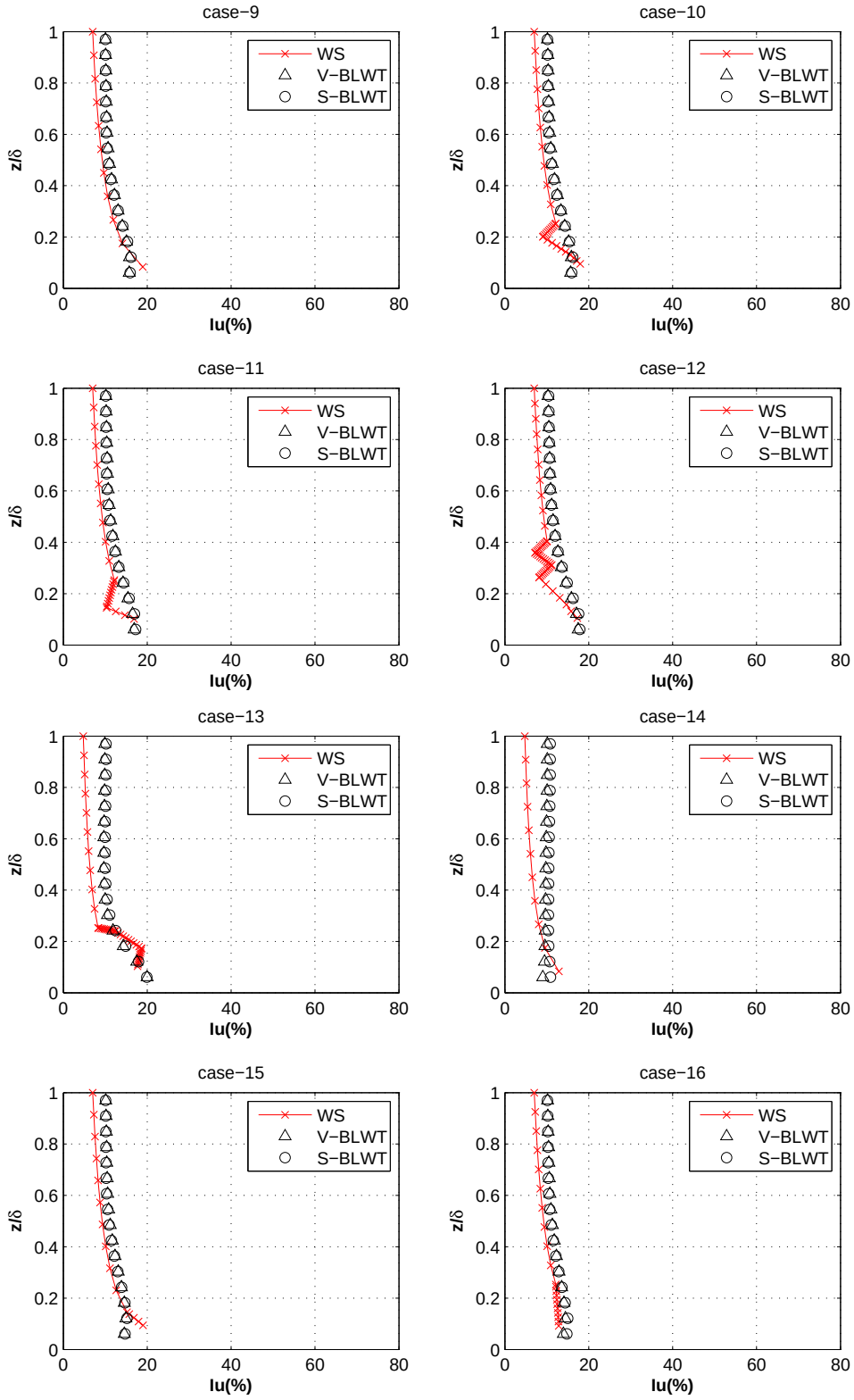


Figure A.11: Turbulence intenisy comparison of CFD with existing models for cases 9-16

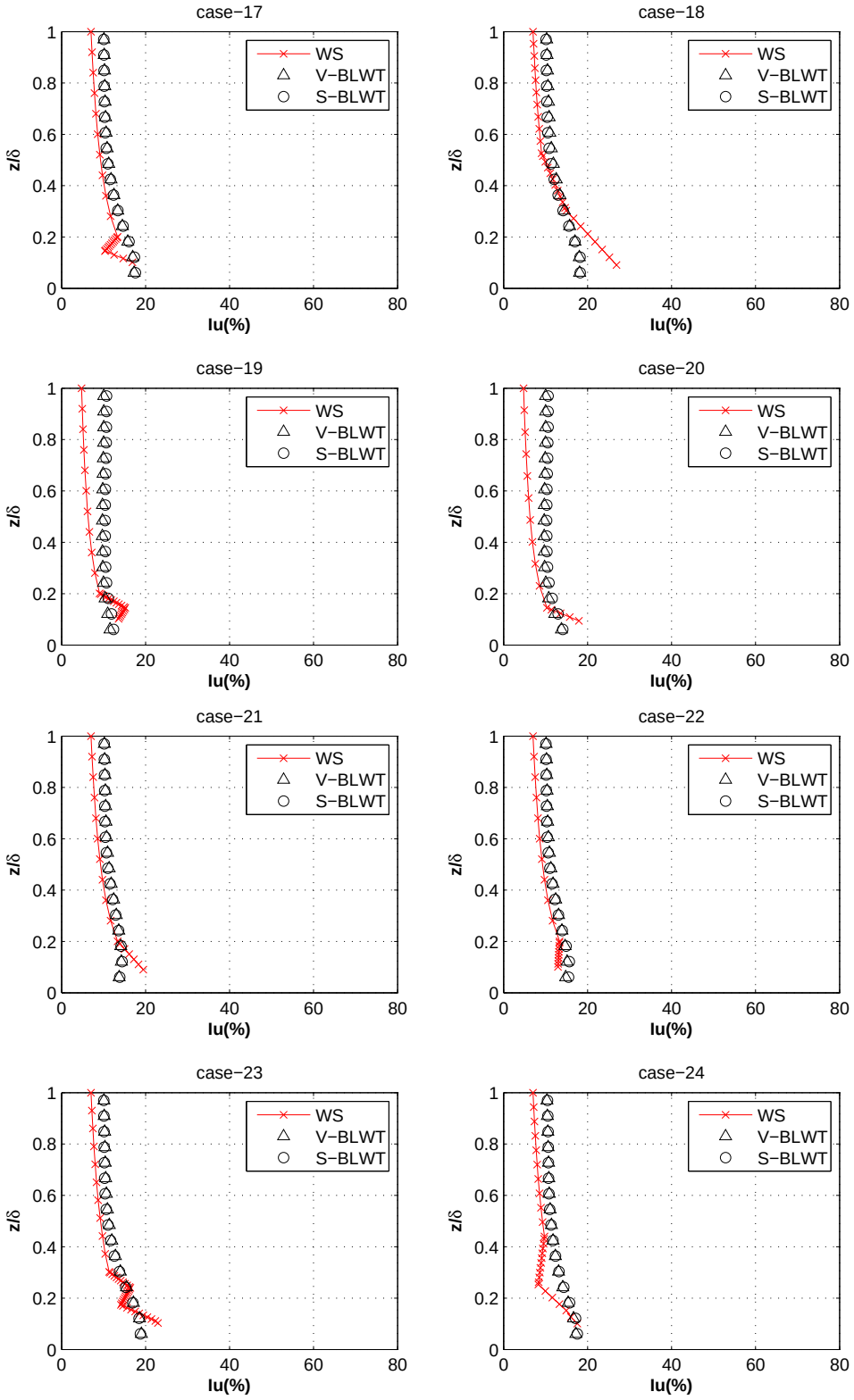


Figure A.12: Turbulence intenisy comparison of CFD with existing models for cases 17-24

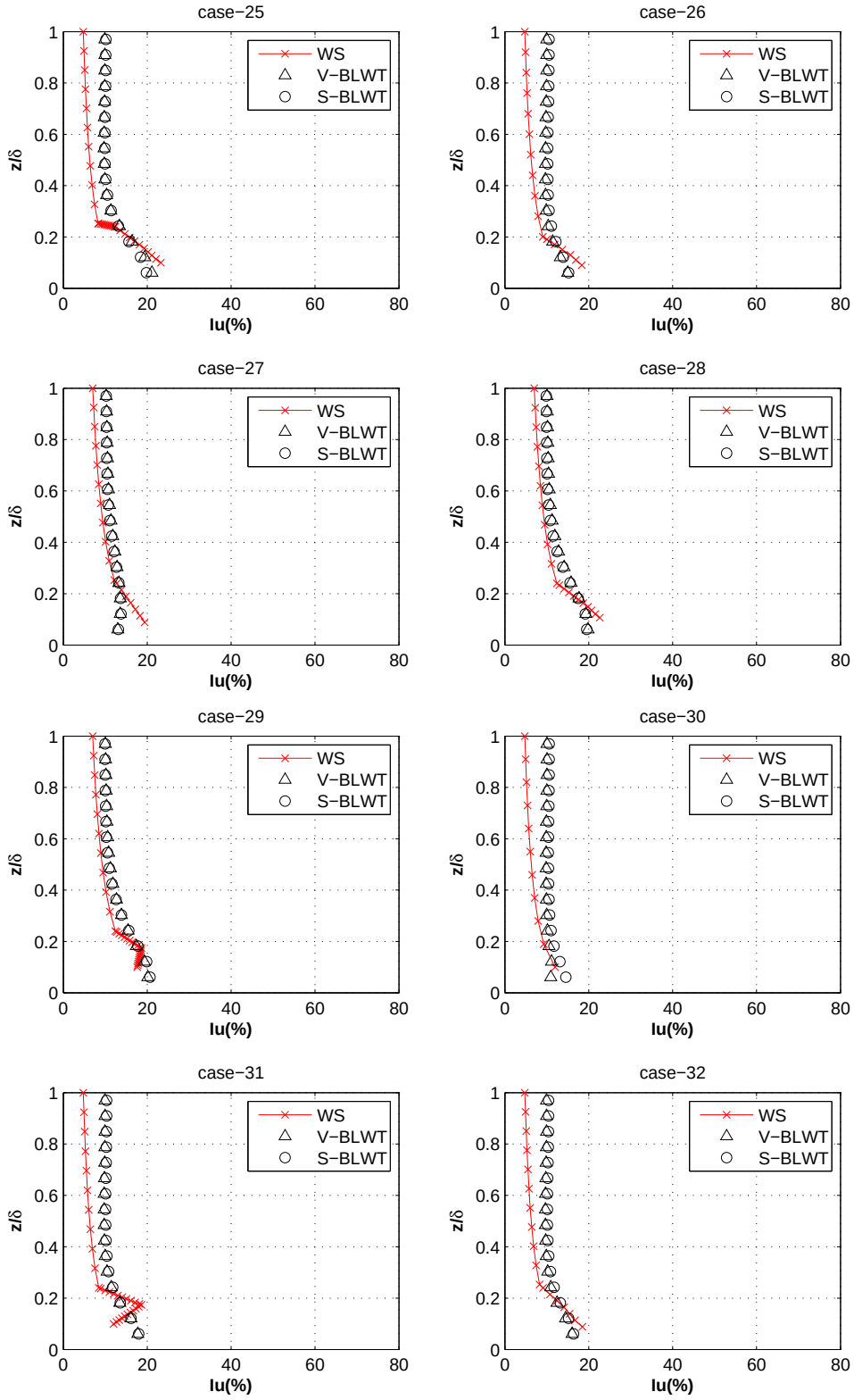


Figure A.13: Turbulence intenisy comparison of CFD with existing models for cases 25-32

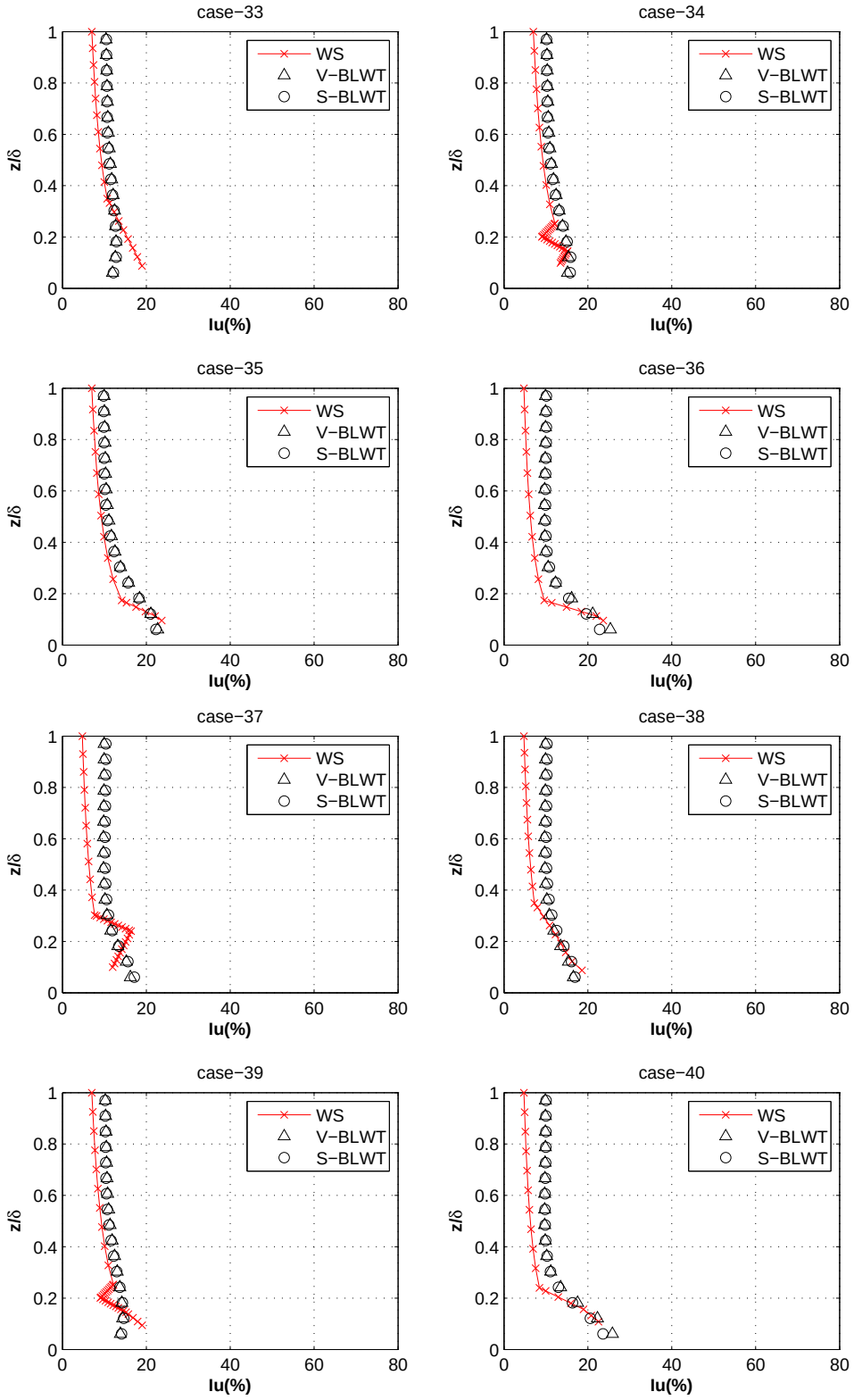


Figure A.14: Turbulence intenisy comparison of CFD with existing models for cases 33-40

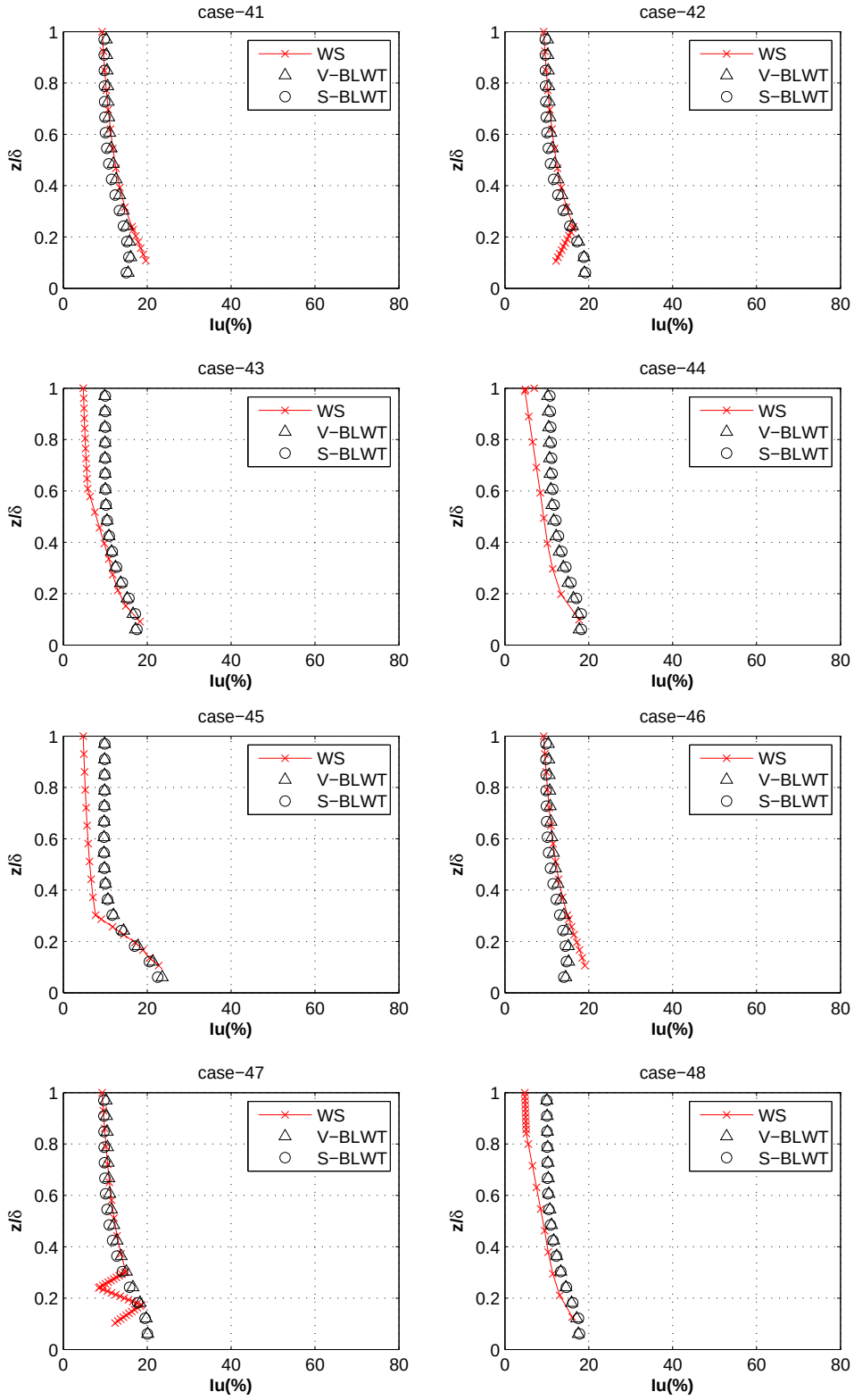


Figure A.15: Turbulence intenisy comparison of CFD with existing models for cases 41-48

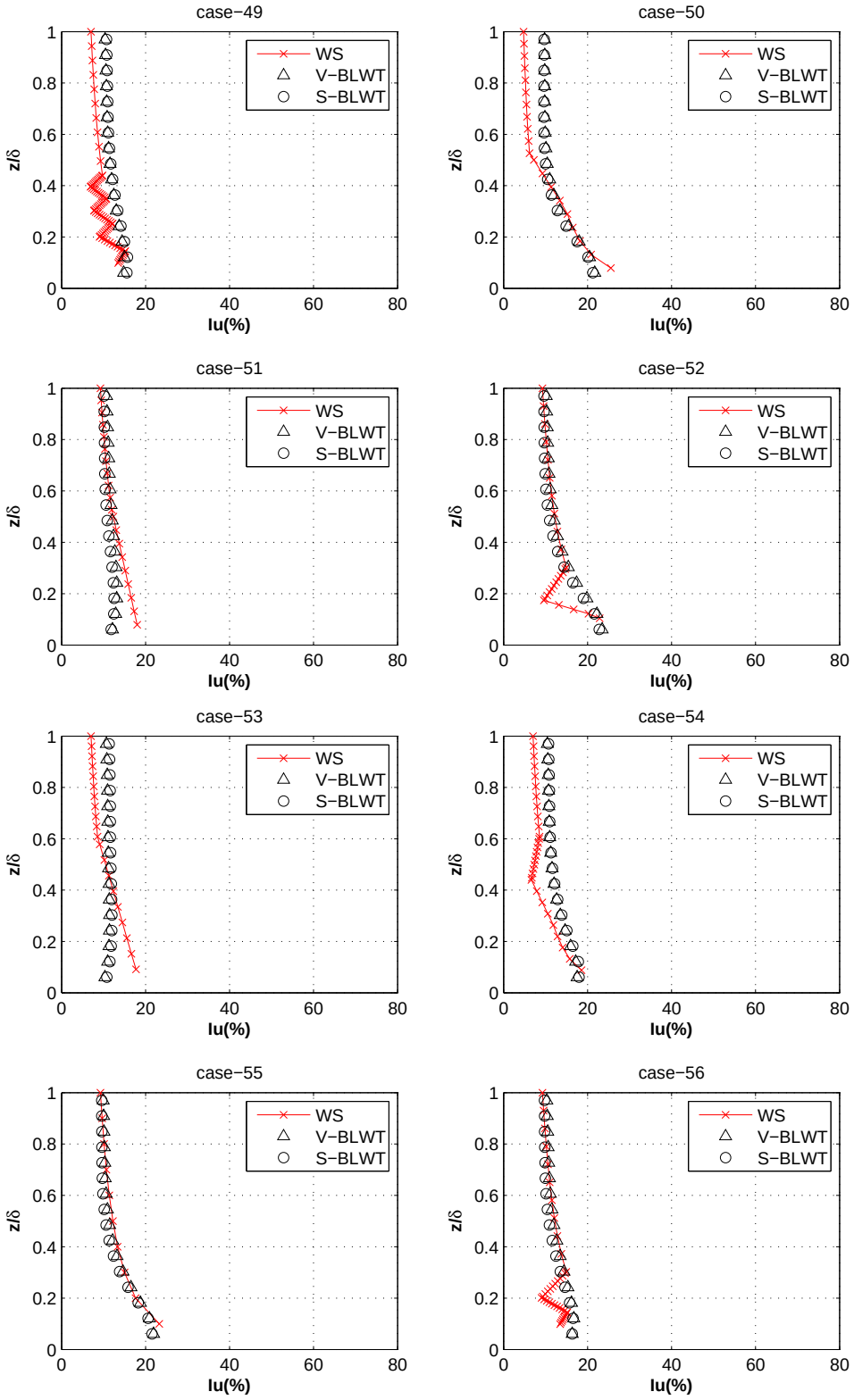


Figure A.16: Turbulence intenisy comparison of CFD with existing models for cases 49-56

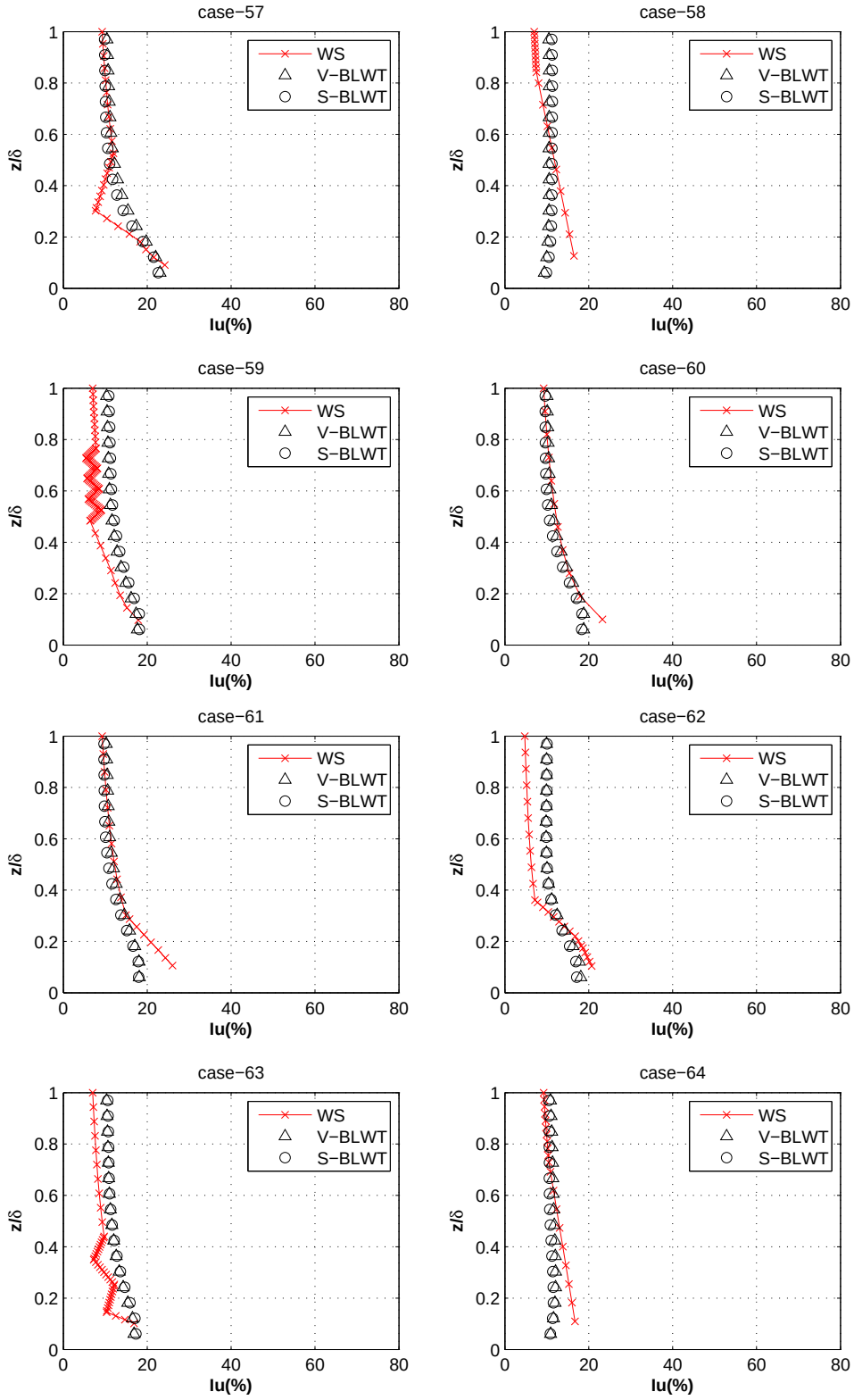


Figure A.17: Turbulence intenisy comparison of CFD with existing models for cases 57-64

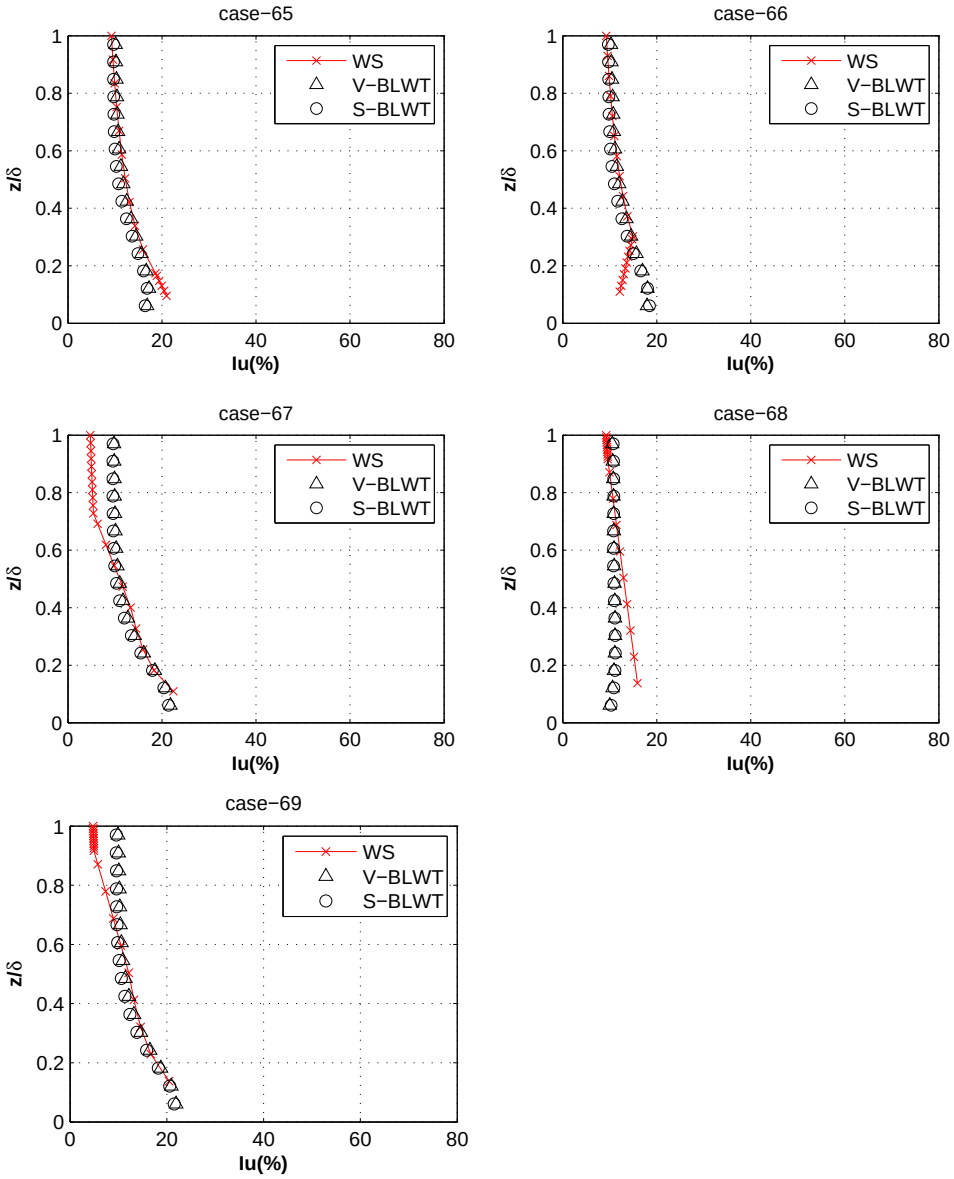
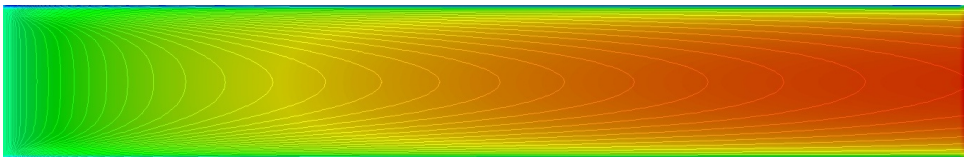
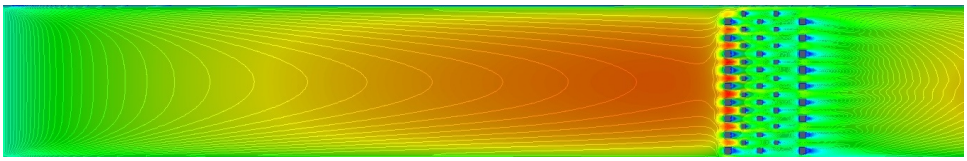


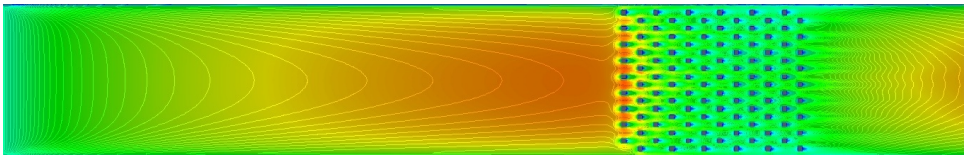
Figure A.18: Turbulence intenisy comparison of CFD with existing models for cases 65-69



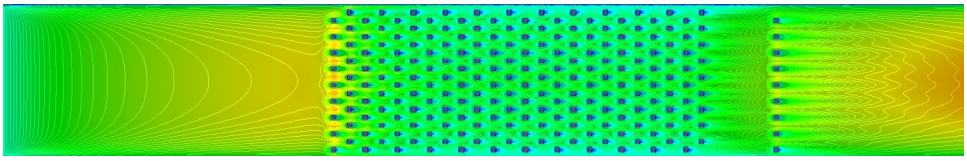
(a) BLWT case-1



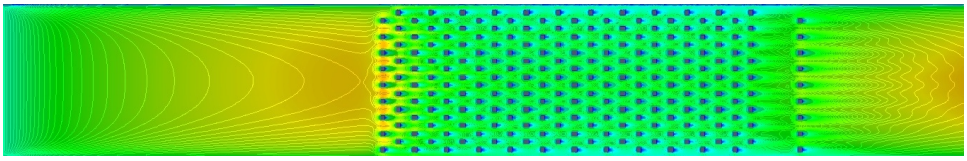
(b) BLWT case-2



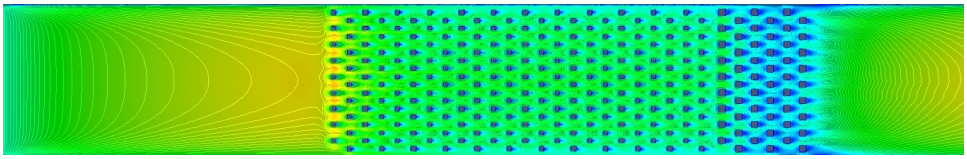
(c) BLWT case-3



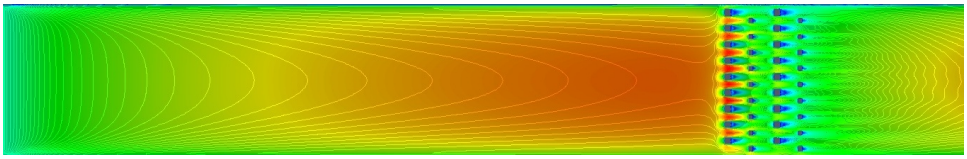
(d) BLWT case-4



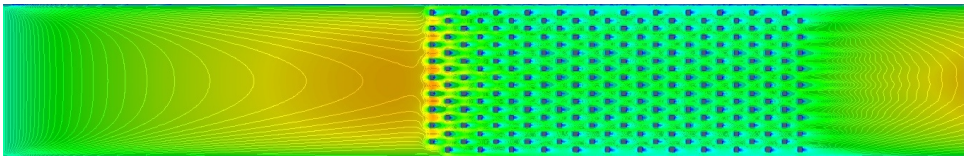
(e) BLWT case-5



(f) BLWT case-6

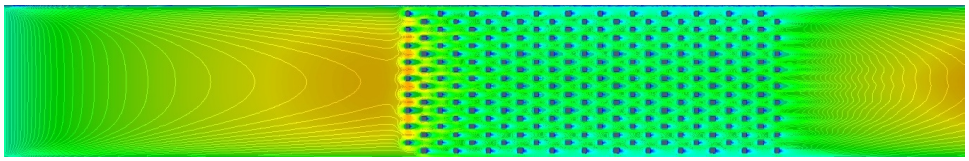


(g) BLWT case-7

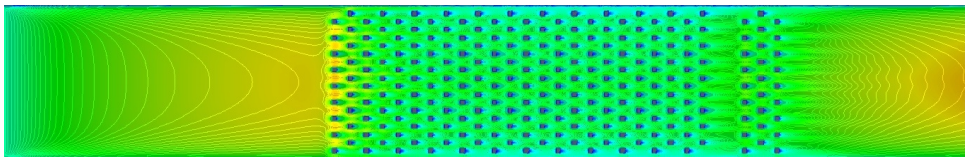


(h) BLWT case-8

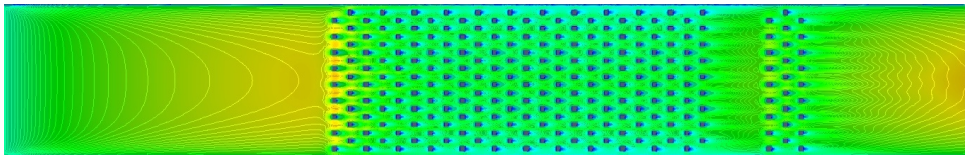
Figure A.19: Horizontal velocity contour for V-BLWT configuration of cases 1-8



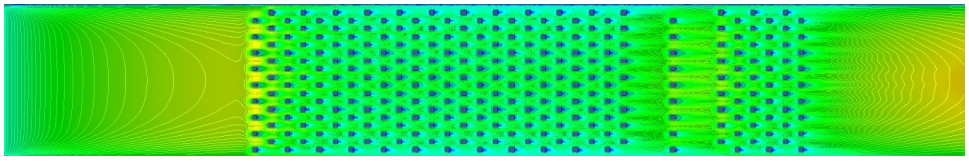
(a) BLWT case-9



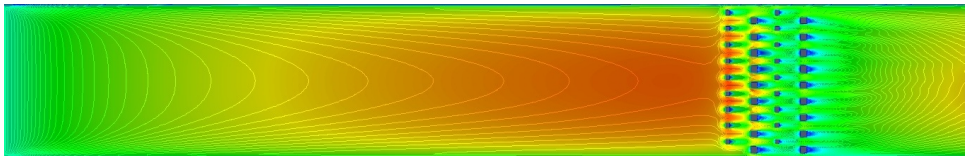
(b) BLWT case-10



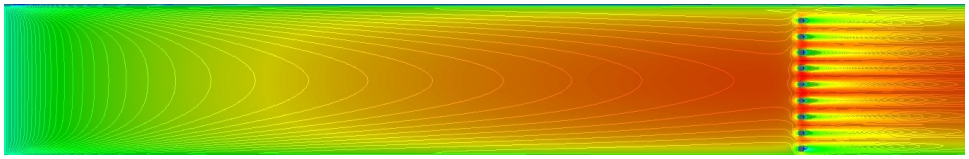
(c) BLWT case-11



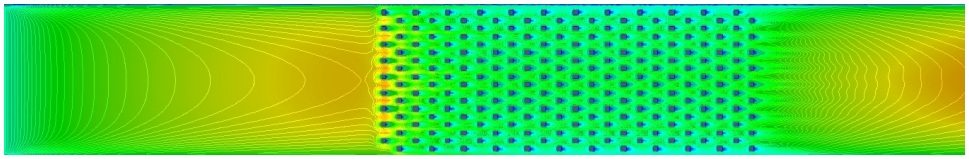
(d) BLWT case-12



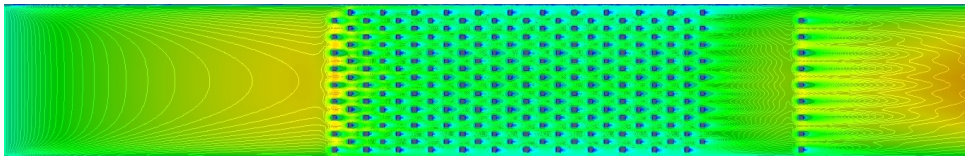
(e) BLWT case-13



(f) BLWT case-14

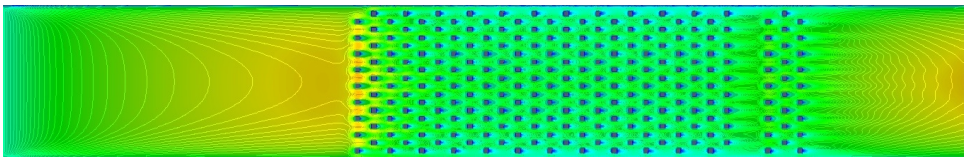


(g) BLWT case-15

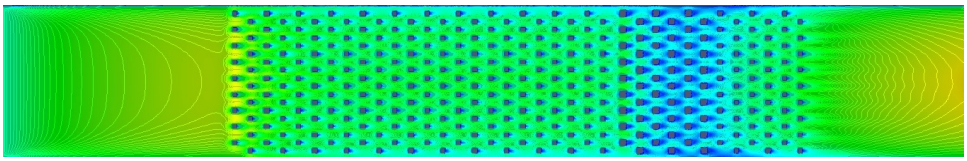


(h) BLWT case-16

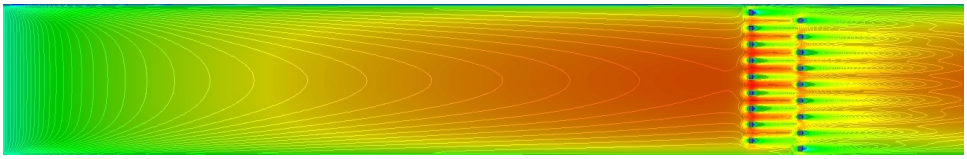
Figure A.20: Horizontal velocity contour for V-BLWT configuration of cases 9-16



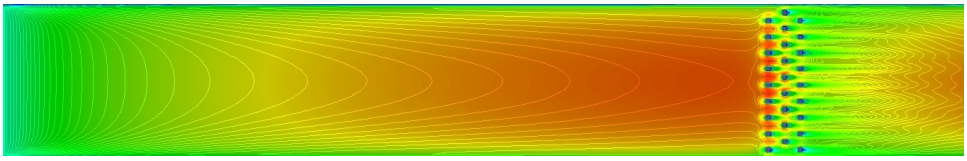
(a) BLWT case-17



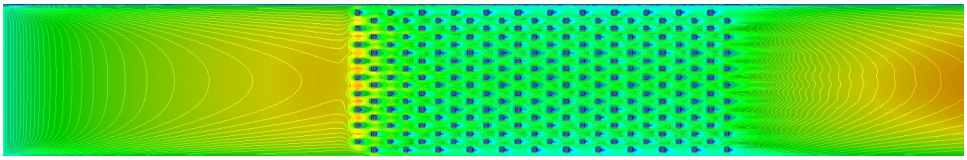
(b) BLWT case-18



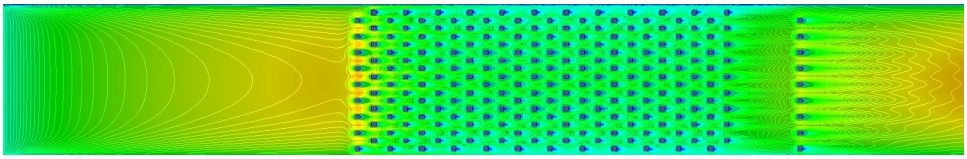
(c) BLWT case-19



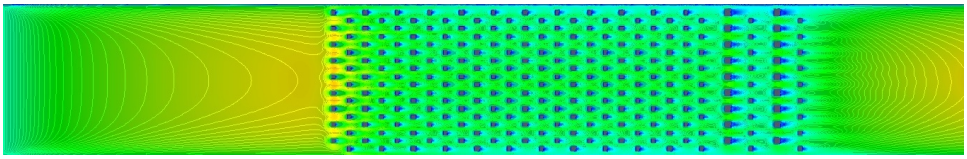
(d) BLWT case-20



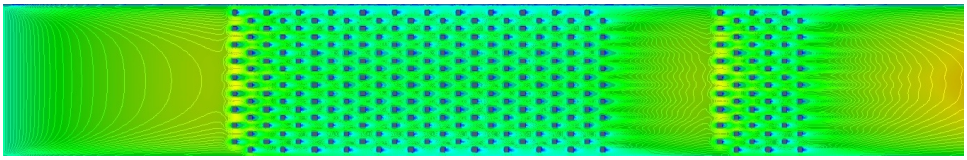
(e) BLWT case-21



(f) BLWT case-22

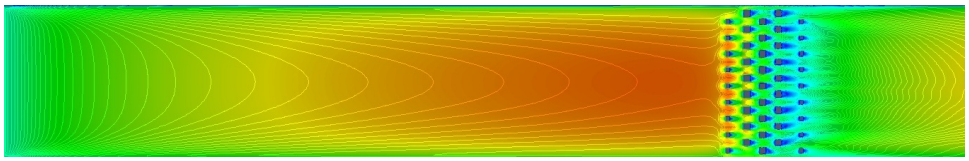


(g) BLWT case-23

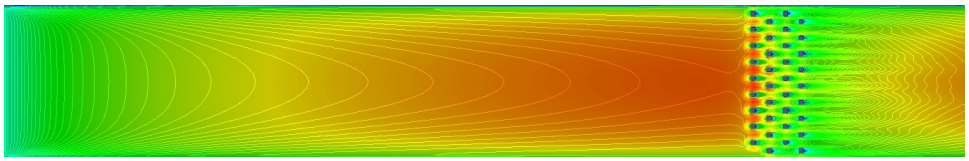


(h) BLWT case-24

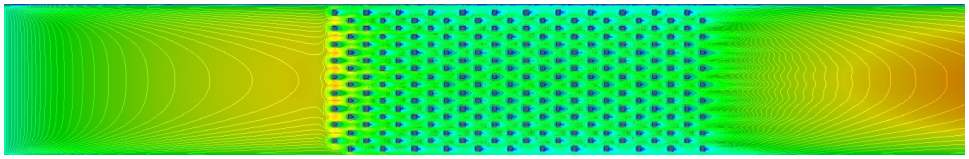
Figure A.21: Horizontal velocity contour for V-BLWT configuration of cases 17-24



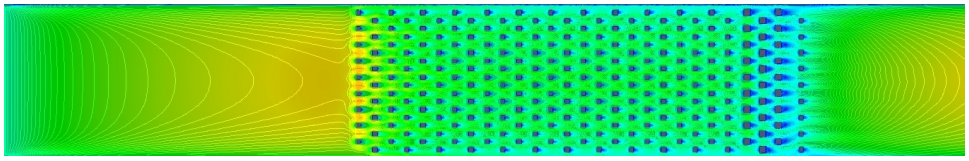
(a) BLWT case-25



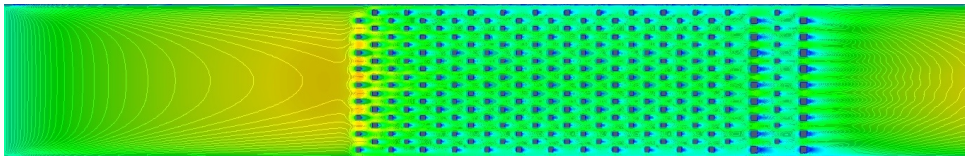
(b) BLWT case-26



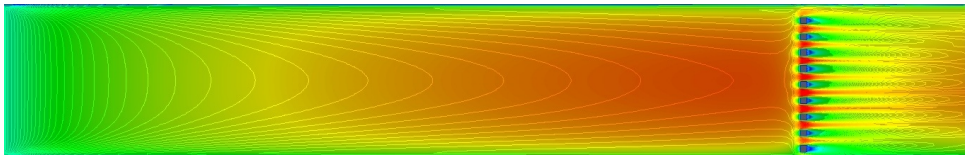
(c) BLWT case-27



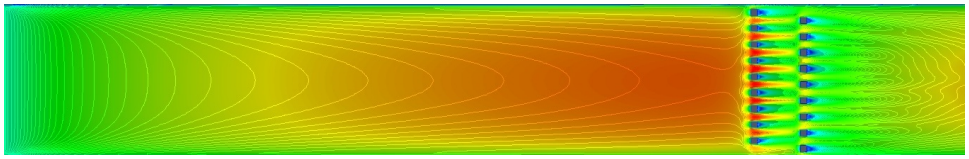
(d) BLWT case-28



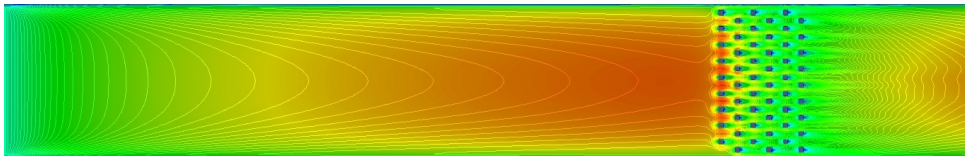
(e) BLWT case-29



(f) BLWT case-30

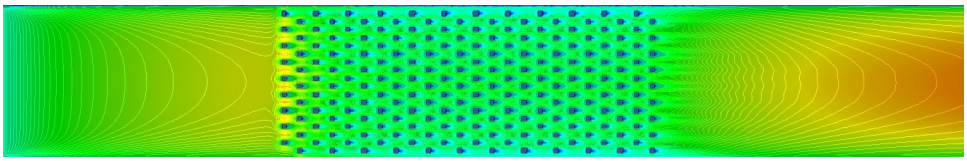


(g) BLWT case-31

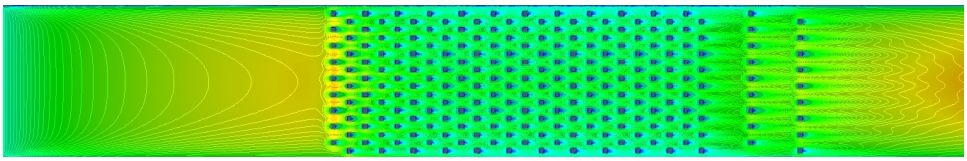


(h) BLWT case-32

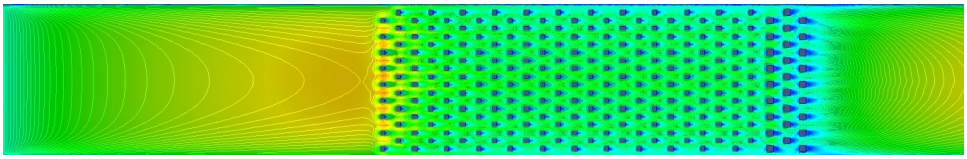
Figure A.22: Horizontal velocity contour for V-BLWT configuration of cases 25-32



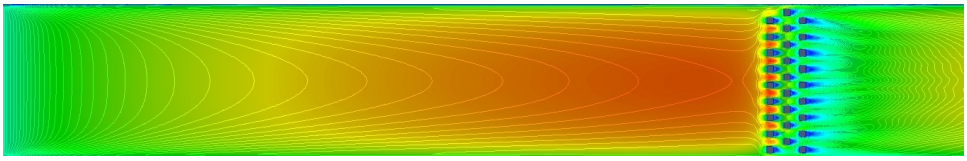
(a) BLWT case-33



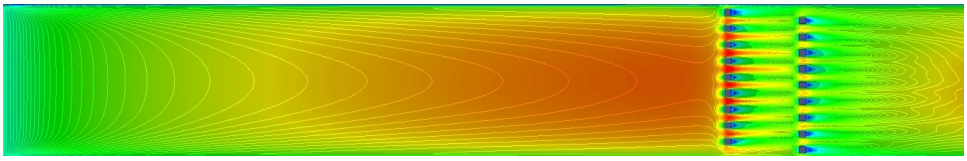
(b) BLWT case-34



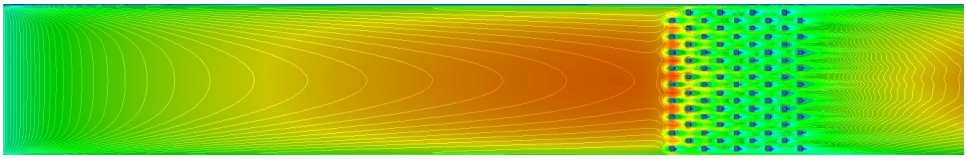
(c) BLWT case-35



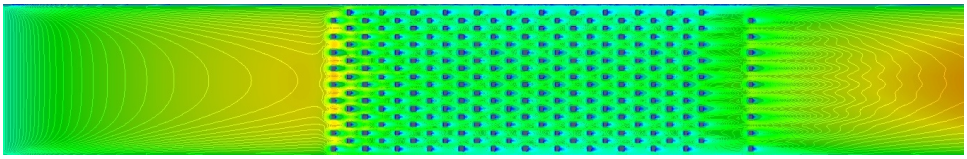
(d) BLWT case-36



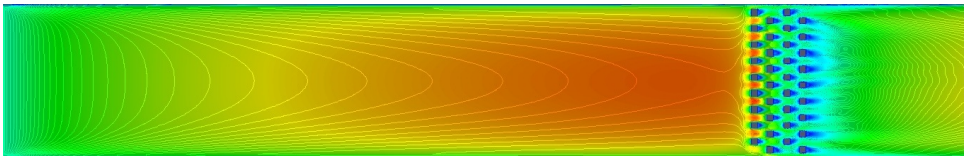
(e) BLWT case-37



(f) BLWT case-38

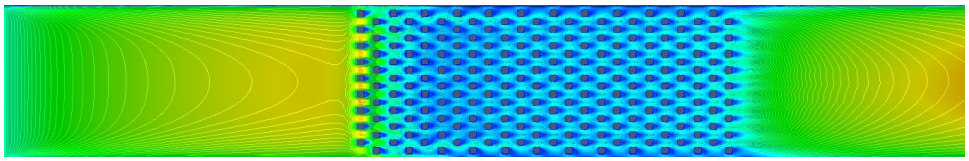


(g) BLWT case-39

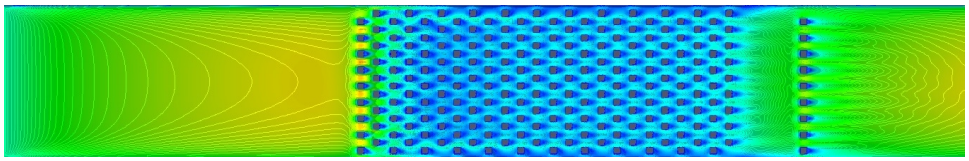


(h) BLWT case-40

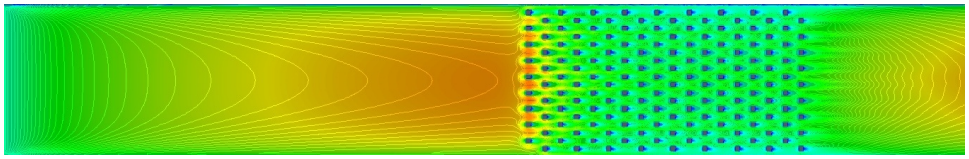
Figure A.23: Horizontal velocity contour for V-BLWT configuration of cases 33-40



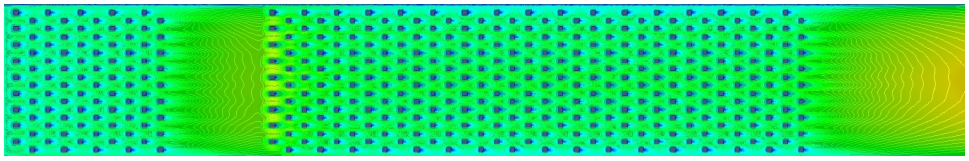
(a) BLWT case-41



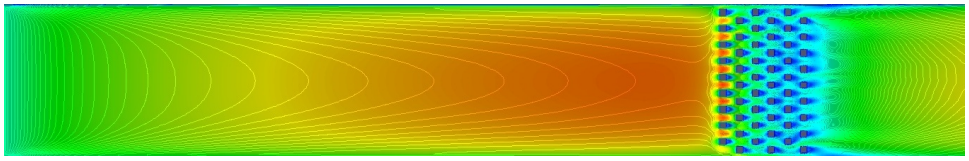
(b) BLWT case-42



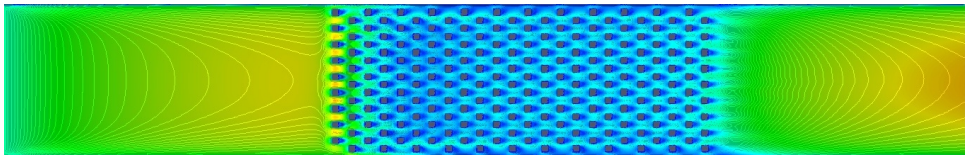
(c) BLWT case-43



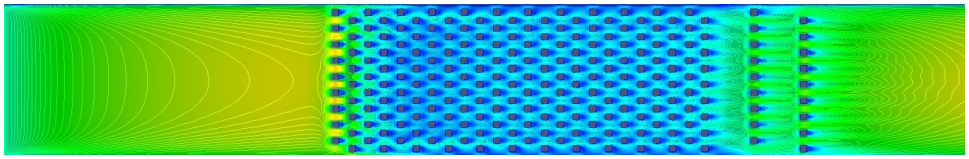
(d) BLWT case-44



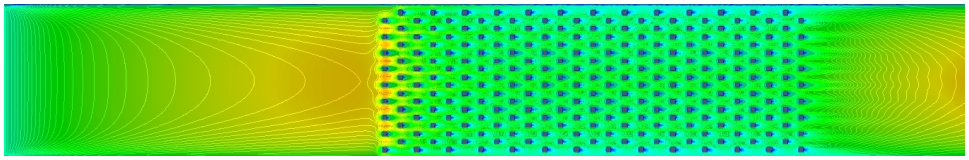
(e) BLWT case-45



(f) BLWT case-46

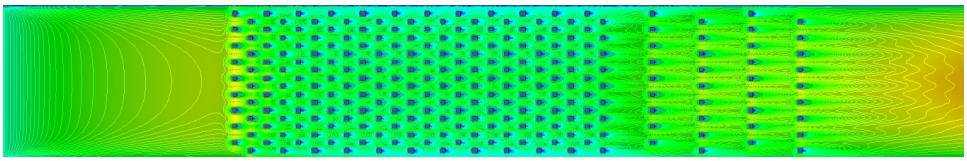


(g) BLWT case-47

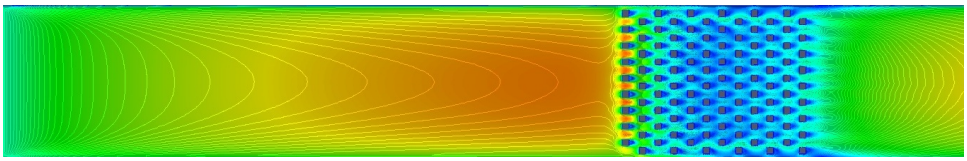


(h) BLWT case-48

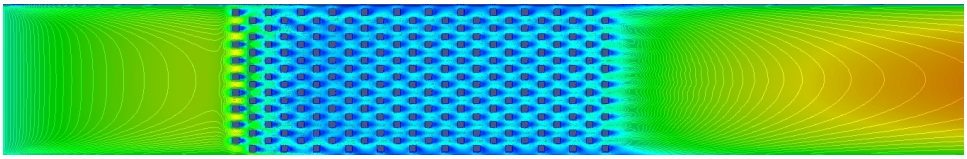
Figure A.24: Horizontal velocity contour for V-BLWT configuration of cases 41-48



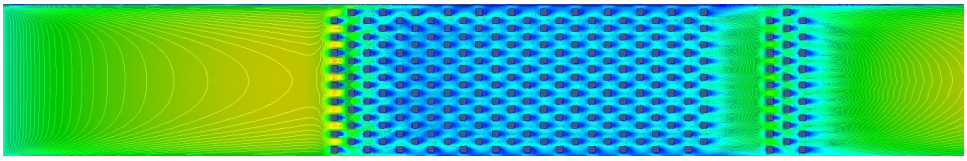
(a) BLWT case-49



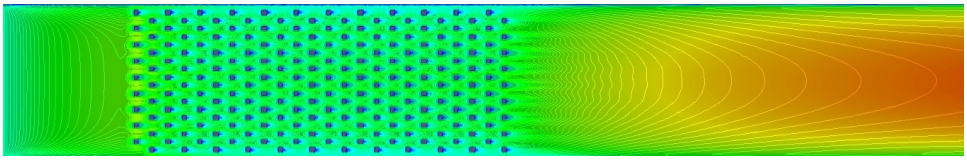
(b) BLWT case-50



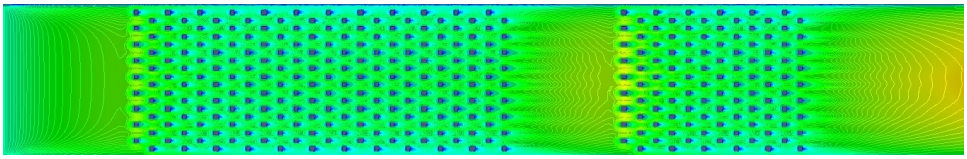
(c) BLWT case-51



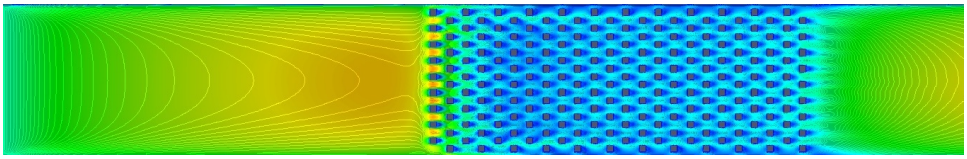
(d) BLWT case-52



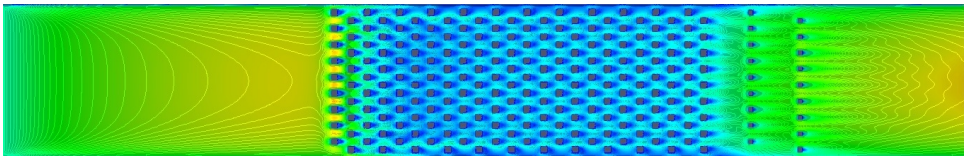
(e) BLWT case-53



(f) BLWT case-54

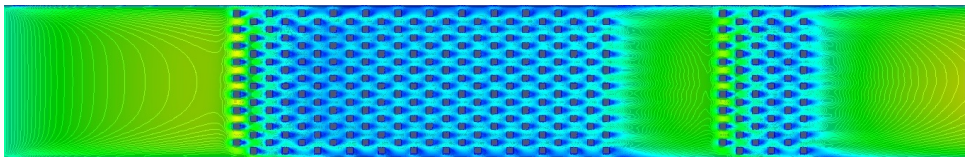


(g) BLWT case-55

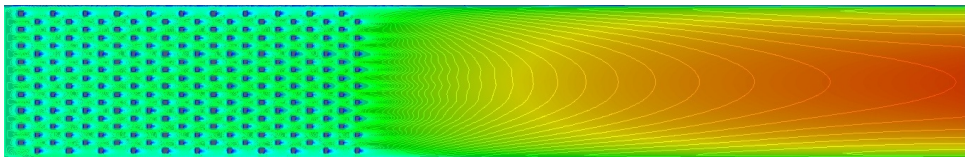


(h) BLWT case-56

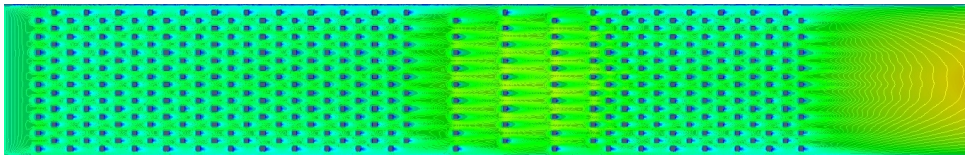
Figure A.25: Horizontal velocity contour for V-BLWT configuration of cases 49-56



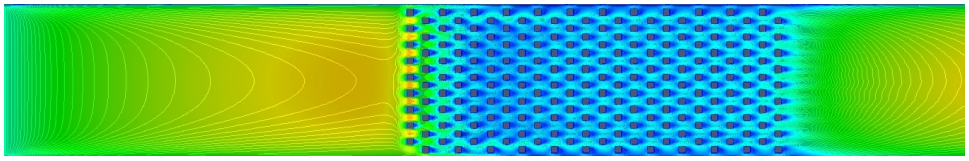
(a) BLWT case-57



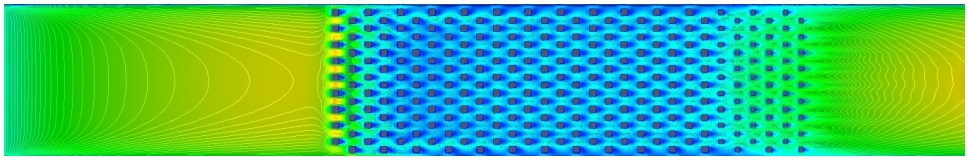
(b) BLWT case-58



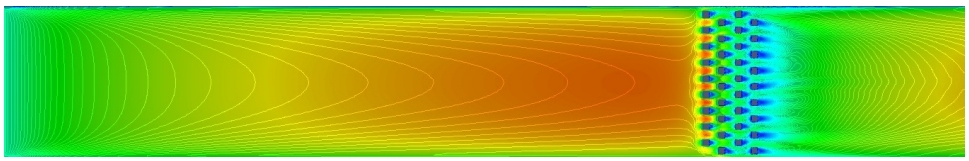
(c) BLWT case-59



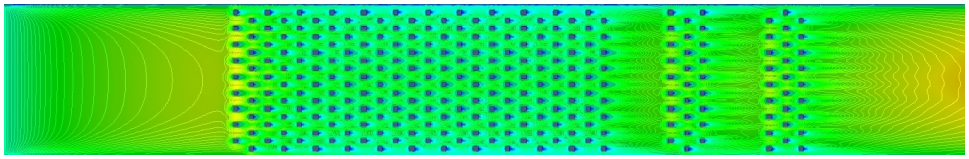
(d) BLWT case-60



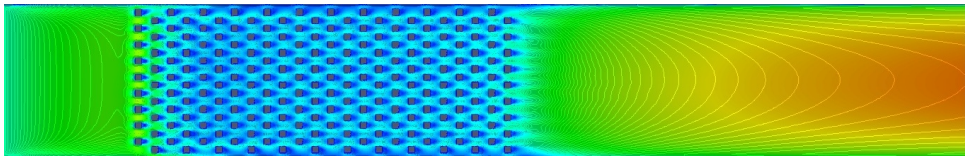
(e) BLWT case-61



(f) BLWT case-62

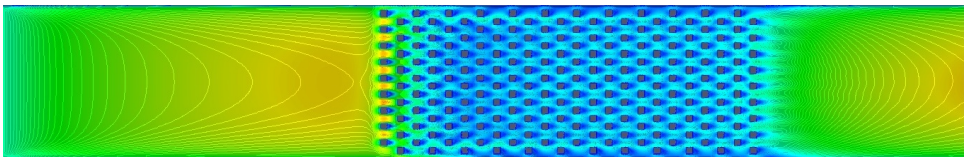


(g) BLWT case-63

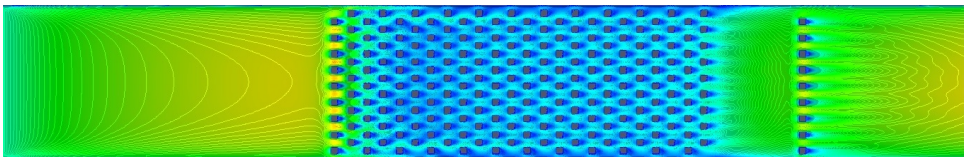


(h) BLWT case-64

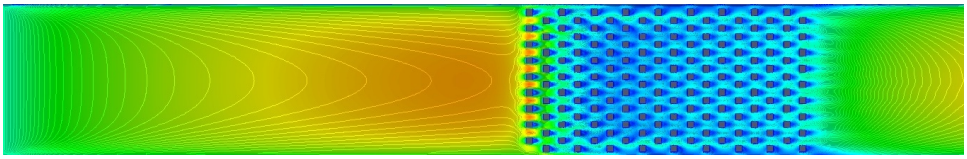
Figure A.26: Horizontal velocity contour for V-BLWT configuration of cases 57-64



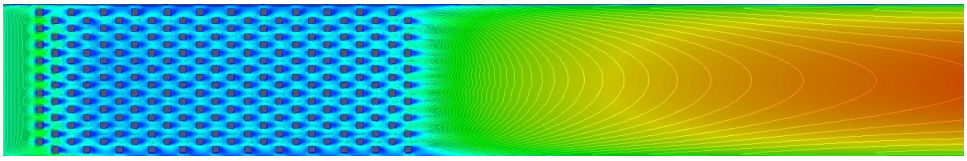
(a) BLWT case-65



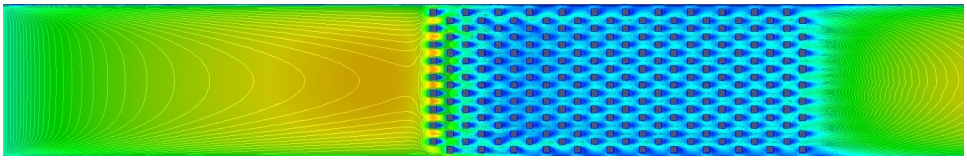
(b) BLWT case-66



(c) BLWT case-67



(d) BLWT case-68



(e) BLWT case-69

Figure A.27: Horizontal velocity contour for V-BLWT configuration of cases 65-69

Appendix B

Artificial neural network source code

```
/*
Multilayer perception method using backpropagation
*/
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>
#include<time.h>

/*
Neuron activation functions
*/
__inline double activation(double x) {
 return (1 / (1 + exp(-x)));
}
__inline double derivative(double x) {
 return x * (1 - x);
}
/*
weight
*/
struct WEIGHT {
 double val;
 double inc;
 WEIGHT() {
  val = (2.0 * rand()) / RAND_MAX - 1.0;
  inc = 0.0;
 }
};
/*
neuron
*/
typedef struct NEURON {
 double in;
 double out;
 double delta;
 WEIGHT* weight;
 NEURON() {
  in = 0.0;
  out = 0.0;
```

```
 delta = 0.0;
    weight = NULL;
 }
 void malloc(int sz) {
  weight = new WEIGHT[sz];
 }
 void free() {
  delete[] weight;
 }
} *PNEURON;

/*
Artifical neural network class
*/
class ANN {
public:
 int n_layers;
 int* n_neurons;
    PNEURON* neurons;
 double momentum;
 double step;

public:
    ANN(int,const int*,double,double);
 ~ANN();
 void feed_forward(double*);
 void back_propagate(double*);
 double mse(double*);
};


ANN :: ANN(int nlayers,const int* nneurons,double a,double b) {
   register int i,j;

   //momentum and step size
   momentum = a;
   step = b;

   //set number of layers
   n_layers = nlayers;

```

```
83    //allocate layers
84    n_neurons = new int[n_layers];
85    for(i = 0;i < n_layers;i++)
86     n_neurons[i] = nneurons[i];
87
88    //allocate neurons
89    neurons = new PNEURON[n_layers];
90    for(i = 0;i < n_layers;i++) {
91     neurons[i] = new NEURON[n_neurons[i]];
92     for(j = 0;j < n_neurons[i];j++) {
93      neurons[i][j].malloc(n_neurons[i - 1] + 1);
94     }
95    }
96 }
97
98
99 ANN :: ~ANN() {
100  register int i,j;
101
102     //number of neurons in each layer
103  delete[] n_neurons;
104
105     //all neurons
106  for(i = 0;i < n_layers;i++) {
107   for(j = 0;j < n_neurons[i];j++)
108    neurons[i][j].free();
109   delete[] neurons[i];
110  }
111  delete[] neurons;
112 }
113
114 void ANN::feed_forward(double* input) {
115  register int i,j,k;
116  double sum;
117
118  //input layer
119  for(i = 0;i < n_neurons[0];i++) {
120   neurons[0][i].in = input[i];
121   neurons[0][i].out = input[i];
122  }
123
```

```
124  //other layers
125  for(i = 1;i < n_layers;i++) {
126   for(j = 0;j < n_neurons[i];j++) {
127    sum = 0;
128    for(k = 0;k < n_neurons[i - 1];k++) {
129     sum += neurons[i][j].weight[k].val * neurons[i - 1][k].out;
130    }
131    neurons[i][j].in = sum;
132    neurons[i][j].out = activation(sum);
133   }
134  }
135 }
136 void ANN::back_propagate(double* target) {
137  register int i,j,k;
138  double sum;
139  PNEURON pneuron;
140 
141  //output layer
142  for(i = 0;i < n_neurons[n_layers - 1];i++) {
143   sum = target[i] - neurons[n_layers - 1][i].out;
144   neurons[n_layers - 1][i].delta = derivative(neurons[n_layers - 1][i].out) * sum;
145  }
146 
147  //other layers
148  for(i = n_layers - 2; i > 0; i--) {
149   for(j = 0;j < n_neurons[i]; j++) {
150    sum = 0;
151    for(k = 0;k < n_neurons[i + 1]; k++){
152     sum += neurons[i + 1][k].weight[j].val * neurons[i + 1][k].delta;
153    }
154    neurons[i][j].delta = derivative(neurons[i][j].out) * sum;
155   }
156  }
157 
158  //modify weights
159  for(i = 1;i < n_layers;i++){
160   for(j = 0;j < n_neurons[i]; j++){
161    pneuron = &neurons[i][j];
162    for(k = 0;k <= n_neurons[i - 1]; k++){
163     pneuron->weight[k].val += momentum * pneuron->weight[k].inc;
164     pneuron->weight[k].inc = step * pneuron->delta * neurons[i - 1][k].out;
```

```
    pneuron->weight[k].val += pneuron->weight[k].inc;
   }
  }
 }
}

//calculate mean square error
double ANN::mse(double *target) {
 double mse = 0;
 for(int i = 0;i < n_neurons[n_layers - 1];i++){
  mse += pow(target[i] - neurons[n_layers - 1][i].out, 2);
 }
 return (mse / 2);
}

/*
constants to be modified
depending on file input.
*/
static int NINPUT = 4;
static int NOUTPUT = 1;
static int NTOTAL = NINPUT + NOUTPUT;
static const double INERTIA = 0.1;
static const double STEP = 0.9;
static float TOLERANCE = 0.0003f;
static const int MAX_ITER = 6144;//(1 << 14);
static const int NLAYERS = 4;
static int NNEURONS[] = {
 NINPUT,
 4,
 4,
 NOUTPUT
};
char in_name[26] = "input.txt";
const char out_name[] = "out.xls";
/*
main
*/
void main() {
 double**data, *pdata;
 double val,val_avg;
```

```
206 float valf;
207 int i,j,count,SIZE;
208 int training_start = 0;//2 * 51;
209
210 //time
211 srand((unsigned)(time(NULL)));
212 clock_t start,end;
213 start=clock();
214
215   //open file
216 FILE* pf = fopen(in_name,"r");
217
218 //determine size and allocate space
219 count = 0;
220 while(fscanf(pf,"%f",&valf) != EOF) count++;
221 SIZE = count / NTOTAL;
222 data = new double*[SIZE];
223 for(i = 0;i < SIZE;i++)
224  data[i] = new double[NTOTAL];
225
226 //read data
227 fseek(pf,0L,SEEK_SET );
228 count = 0;
229 while(fscanf(pf,"%f",&valf) != EOF) {
230  data[count / NTOTAL][count % NTOTAL] = valf;
231  count++;
232 }
233
234
235 //close file
236 fclose(pf);
237
238   //create ANN
239 ANN *bp = new ANN(NLAYERS,&NNEURONS[0],INERTIA,STEP);
240
241   //train
242 printf("\nTraining....\n");
243
244 for(i = 0;i < MAX_ITER;i++) {
245  val_avg = 0;
246
```

```
247  //loop through training data
248  for(j = training_start;j < SIZE;j++) {
249
250   pdata = data[j];
251
252         bp->feed_forward(pdata);
253   bp->back_propagate(pdata + NINPUT);
254
255   //converged?
256   val = bp->mse(pdata + NINPUT);
257   val_avg += val;
258  }
259  val_avg /= SIZE;
260
261  //display
262  if(i % 16 == 0) {
263   printf("Iteration %d : MSE = %e\t\t\t\r",i,val_avg);
264   if(i % 512 == 0) printf("\n");
265  }
266  //failed
267  if(val_avg <= TOLERANCE) {
268   printf("\nSuccess in %d iterations.\nMSE = %e\n",i,val_avg);
269   break;
270  }
271 }
272
273
274 //predict
275 bool show = false;
276 pf = fopen(out_name,"w");
277
278 printf("\nPrediction....\n");
279
280 //for(i = 0;i < training_start;i++) {
281 for(i = 0;i < SIZE;i++) {
282  pdata = data[i];
283
284  bp->feed_forward(pdata);
285
286  for(j = 0;j < NTOTAL;j++) {
287   if(show) {
```

```
288     if(j == NINPUT) printf("| ");
289     printf("%f ",pdata[j]);
290    }
291    if(j == NINPUT) fprintf(pf,"\t");
292            fprintf(pf,"%f\t",pdata[j]);
293   }
294
295        if(show) printf("| ");
296   fprintf(pf,"\t");
297
298   for(j = 0;j < bp->n_neurons[bp->n_layers - 1];j++) {
299    if(show) printf("%f ",bp->neurons[bp->n_layers - 1][j].out);
300            fprintf(pf,"%f\t",bp->neurons[bp->n_layers - 1][j].out);
301   }
302   if(show) printf("\n");
303   fprintf(pf,"\n");
304  }
305  fclose(pf);
306
307  //end
308  end = clock();
309  printf("\nTime elapsed = %.2f sec\n",(end - start) / 1000.0f);
310  printf("Done !\n");
311 }
```

Appendix C

CFD program

C.1 Brief information on usage

The CFD code is compiled into three separate programs for the following purposes

1. Grid generation: **mesh** Grid can be imported from FlUENT ascii format (.msh)

```
./mesh -i test.msh -o test
```

   This will import the grid file *test.msh* and save the result in the file *test*. The second option, useful for simple hexahedral grid generation, is to use the built-in grid generator

```
./mesh block.txt >grid
```

   This will generate grid according to specification in *block.txt* and save the result in the file*grid*. The specification file basically consists of the *vertices*, *faces* and *control volumes* of the computational domain.

2. Solvers: **solver** There are currently five solvers implemented as described in Chapter 3

   (a) PISO - for Navier-Stokes equations

   (b) Walldist - Wall distance solver

   (c) Diffusion - Heat diffusion solver

   (d) Transport - Transport equation solver

   (e) Potential - Potential flow solver

   The type of solver and settings for the solvers can be specified in a separate file *controls.txt* and passed to the solver program

```
./solver controls.txt
```

The contents of *controls.txt* look like the following

```
general
{
#######################################
#  mesh file name and solver type
#######################################
    solver                  piso
    mesh                    grid
#######################################
#  Fluid properties specific to solver
#######################################
    rho                     1
    viscosity               0.1
#############################
#  Time increment
#############################
    state                   STEADY
    start_step              0
    end_step                1
    write_interval          1
    dt                      0.1
#############################
#  Discretization schemes
#############################
    convection_scheme       UDS
    interpolation_scheme    CDS
    nonortho_scheme         OVER_RELAXED
    time_scheme_factor      1
    blend_factor            0.2
#############################
#  Solver options
#############################
    method                  PCG
    tolerance               1e-5
    max_iterations          6400
    SOR_omega               1.7
    ghost_exchange          BLOCKED
    parallel_method         BLOCKED
```

```
############################
#  Probe locations
############################
    probe 0 {
    }
}
#####################################
#  PISO options
#####################################
piso
{
    turbulence_model         KE
    velocity_UR              1
    pressure_UR              1
    k_UR                     1
    x_UR                     1
    n_PISO                   1
    n_ORTHO                  0
}
diffusion
{
    DT                       1
    t_UR                     0.7
}
transport
{
    DT                       0.1
    t_UR                     1
}
turbulence
{
    k_UR                     0.5
    x_UR                     0.5
}
```

3. Pre/Post processing; **prepare** This tool is used for pre and post processing results such as generating VTK file of results for viewing with paraview or similar tool, taking samples at specified probe points, preparing to run program in parallel via static domain decomposition, merging decomposed results in to one etc...

```
./prepare prepare.txt -vtk -start 0 -end 5
```

This converts the results into VTK file format for time steps between 0 and 5. The contents of *prepare.txt* look like

```
general
{
mesh  grid
decompose 3 {2 1 1}
fields  5 { U p k e emu}
probe         8 {
0 0.5 20
28.571 0.5 20
57.143 0.5 20
85.714 0.5 20
114.29 0.5 20
142.86 0.5 20
171.43 0.5 20
200 0.5 20
}
}
```

C.2 Source code

```
#ifndef __TENSOR_H
#define __TENSOR_H

#ifdef _MSC_VER
#   pragma warning (disable: 4996)
#endif

#include <fstream>
#include <iostream>
#include <cmath>

#define __DOUBLE

#ifdef _MSC_VER
# define FORCEINLINE __forceinline
#else
# define FORCEINLINE __inline
#endif

/*******************
 * Int is unsigned
 ******************/
typedef unsigned int Int;

/*************************
 * scalars
 ************************/
#if defined __DOUBLE
# define Scalar double
#else
# define Scalar float
#endif

/* Arthimetic operators are defined via compound assignment*/
#define Operator(T,$)            \
 friend FORCEINLINE T operator $ (const T& p,const T& q) {  \
  T r = p;             \
  r $##= q;             \
  return r;             \
```

```
 }
/* Default operator overloads for scalars vs others*/
#define OpS($)              \
 template<class T>            \
 FORCEINLINE T operator $ (const T& p,const Scalar& q) {   \
  T r = p;             \
  r $##= q;             \
  return r;             \
 }
#define COp($)              \
 template<class T>            \
 FORCEINLINE T operator $ (const Scalar& p,const T& q) {   \
  T r = q;             \
  r $##= p;             \
  return r;             \
 }
#define NCOp($)              \
 template<class T>            \
 FORCEINLINE T operator $ (const Scalar& p,const T& q) {   \
  T r = p;             \
  r $##= q;             \
  return r;             \
 }
OpS(*);
OpS(/);
COp(+);
COp(*);
NCOp(/);
NCOp(-);
#undef OpS
#undef COp
#undef NCOp
/*other scalar operations*/
FORCEINLINE Scalar mag(const Scalar& p) {
 return fabs(p);
}
FORCEINLINE Scalar sdiv(const Scalar& p,const Scalar& q) {
 return p ? (p / q) : 0;
}
FORCEINLINE Scalar max(const Scalar& p,const Scalar& q) {
 return (p >= q) ? p : q;
```

```
}
FORCEINLINE Scalar min(const Scalar& p,const Scalar& q) {
 return (p <= q) ? p : q;
}
/*********************************
 * loop unroller for tensors
 *********************************/
template <int N>
struct Unroll {
 /*macro*/
#define Op(name,$)                \
 static FORCEINLINE void name(Scalar* p,const Scalar* q) {     \
  *p $ *q;                \
  Unroll<N - 1>::name(p + 1,q + 1);          \
 }
#define SOp(name,$)                \
 static FORCEINLINE void name(Scalar* p,const Scalar q) {     \
  *p $ q;                 \
  Unroll<N - 1>::name(p + 1,q);           \
 }
#define Fp(name,$)                                                    \
 static FORCEINLINE void name(Scalar* r,const Scalar* p,const Scalar* q) { \
  *r = ::$(*p,*q);               \
  Unroll<N - 1>::name(r + 1,p + 1,q + 1);            \
 }
#define Fp1(name,$)                                                   \
 static FORCEINLINE void name(Scalar* r,const Scalar* p,const Scalar q) { \
  *r = ::$(*p,q);               \
  Unroll<N - 1>::name(r + 1,p + 1,q);          \
 }
#define Fp2(name,$)                                                   \
 static FORCEINLINE void name(Scalar* r,const Scalar* p) {     \
  *r = ::$(*p);                \
  Unroll<N - 1>::name(r + 1,p + 1);              \
 }
 /*special*/
 static FORCEINLINE Scalar dot(const Scalar* p,const Scalar* q) {
  return (*p) * (*q) + Unroll<N - 1>::dot(p + 1,q + 1);
 }
 /*define ops*/
 Op(equ,=);
```

```
122  Op(neg,=-);
123  Op(inc,+=);
124  Op(dec,-=);
125  Op(mul,*=);
126  Op(div,/=);
127  SOp(equ,=);
128  SOp(neg,=-);
129  SOp(inc,+=);
130  SOp(dec,-=);
131  SOp(mul,*=);
132  SOp(div,/=);
133  Fp(sdiv,sdiv);
134     /*from math.h*/
135  Fp2(acos,acos);
136  Fp2(asin,asin);
137  Fp2(atan,atan);
138  Fp(atan2,atan2);
139  Fp2(ceil,ceil);
140  Fp2(cos,cos);
141  Fp2(cosh,cosh);
142  Fp2(exp,exp);
143  Fp2(fabs,fabs);
144  Fp2(floor,floor);
145  Fp2(log,log);
146  Fp2(log10,log10);
147  Fp1(pow,pow);
148  Fp2(sin,sin);
149  Fp2(sinh,sinh);
150  Fp2(sqrt,sqrt);
151  Fp2(tan,tan);
152  Fp2(tanh,tanh);
153  Fp(min,min);
154  Fp(max,max);
155 #undef Op
156 #undef SOp
157 #undef Fp
158 #undef Fp1
159 #undef Fp2
160 };
161
162 template <>
```

```
163 struct Unroll<0> {
164  /*macro*/
165 #define Op(name)                                                    \
166  static FORCEINLINE void name(Scalar* p,const Scalar* q) {}
167 #define SOp(name)                                                   \
168  static FORCEINLINE void name(Scalar* p,const Scalar q) {}
169 #define Fp(name)                                                    \
170  static FORCEINLINE void name(Scalar* r,const Scalar* p,const Scalar* q) {}
171 #define Fp1(name)                                                   \
172  static FORCEINLINE void name(Scalar* r,const Scalar* p,const Scalar q) {}
173 #define Fp2(name)                                                   \
174  static FORCEINLINE void name(Scalar* r,const Scalar* p) {}
175  /*special*/
176  static FORCEINLINE Scalar dot(const Scalar* p,const Scalar* q) {return 0;}
177  /*define ops*/
178  Op(equ);
179  Op(neg);
180  Op(inc);
181  Op(dec);
182  Op(mul);
183  Op(div);
184  SOp(equ);
185  SOp(neg);
186  SOp(inc);
187  SOp(dec);
188  SOp(mul);
189  SOp(div);
190  Fp(sdiv);
191  /*from math.h*/
192  Fp2(acos);
193  Fp2(asin);
194  Fp2(atan);
195  Fp(atan2);
196  Fp2(ceil);
197  Fp2(cos);
198  Fp2(cosh);
199  Fp2(exp);
200  Fp2(fabs);
201  Fp2(floor);
202  Fp2(log);
203  Fp2(log10);
```

```
204  Fp1(pow);
205  Fp2(sin);
206  Fp2(sinh);
207  Fp2(sqrt);
208  Fp2(tan);
209  Fp2(tanh);
210  Fp(min);
211  Fp(max);
212 #undef Op
213 #undef SOp
214 #undef Fp
215 #undef Fp1
216 #undef Fp2
217 };
218
219 /****************************************
220  * Template Tensor class
221  ****************************************/
222 template <Int SIZE>
223 class TTensor {
224 public:
225  Scalar P[SIZE];
226 public:
227     /*c'tors*/
228  TTensor() {
229  }
230  TTensor(const TTensor& p) {
231   *this = p;
232  }
233  explicit TTensor(const Scalar& p) {
234   Unroll<SIZE>::equ(P,p);
235  }
236  TTensor(const Scalar& xx,const Scalar& yy,const Scalar& zz) {
237   P[0] = xx;
238   P[1] = yy;
239   P[2] = zz;
240   Unroll<SIZE - 3>::equ(&P[3],Scalar(0));
241  }
242  /*accessors*/
243  Scalar& operator [] (Int i) {
244   return P[i];
```

```
 }
 const Scalar& operator [] (Int i) const {
  return P[i];
 }
 /*unary ops*/
 TTensor operator - () {
  TTensor r;
  Unroll<SIZE>::neg(r.P,P);
  return r;
 }
 friend Scalar operator & (const TTensor& p,const TTensor& q) {
  Scalar r = Unroll<SIZE>::dot(p.P,q.P);
  if(SIZE == 6) r += Unroll<3>::dot(&p.P[3],&q.P[3]);
  return r;
 }

 /*unrolled operations*/
#define Op(name,$)              \
 TTensor& operator $(const TTensor& q) {       \
  Unroll<SIZE>::name(P,q.P);          \
  return *this;             \
 }
#define SOp(name,$)              \
 TTensor& operator $(const Scalar& q) {       \
  Unroll<SIZE>::name(P,q);          \
  return *this;             \
 }
#define Fp(name)              \
 friend TTensor name(const TTensor& p,const TTensor& s) {  \
  TTensor r;              \
  Unroll<SIZE>::name(r.P,p.P,s.P);       \
  return r;             \
 }
#define Fp1(name)              \
 friend TTensor name(const TTensor& p,const Scalar& s) {   \
  TTensor r;              \
  Unroll<SIZE>::name(r.P,p.P,s);        \
  return r;             \
 }
#define Fp2(name)              \
 friend TTensor name(const TTensor& p) {       \
```

```
286  TTensor r;              \
287  Unroll<SIZE>::name(r.P,p.P);        \
288  return r;               \
289 }
290    /*define ops*/
291 Op(equ,=);
292 Op(inc,+=);
293 Op(dec,-=);
294 Op(mul,*=);
295 Op(div,/=);
296 SOp(equ,=);
297 SOp(inc,+=);
298 SOp(dec,-=);
299 SOp(mul,*=);
300 SOp(div,/=);
301 Fp(sdiv);
302 /*from math.h*/
303 Fp2(acos);
304 Fp2(asin);
305 Fp2(atan);
306 Fp(atan2);
307 Fp2(ceil);
308 Fp2(cos);
309 Fp2(cosh);
310 Fp2(exp);
311 Fp2(fabs);
312 Fp2(floor);
313 Fp2(log);
314 Fp2(log10);
315 Fp1(pow);
316 Fp2(sin);
317 Fp2(sinh);
318 Fp2(sqrt);
319 Fp2(tan);
320 Fp2(tanh);
321 Fp(min);
322 Fp(max);
323 #undef Op
324 #undef SOp
325 #undef Fp
326 #undef Fp1
```

```
#undef Fp2
 Operator(TTensor,+);
 Operator(TTensor,-);
 Operator(TTensor,*);
 Operator(TTensor,/);
 /*others*/
 friend Scalar magSq(const TTensor& p) {
  return (p & p);
 }
 friend Scalar mag(const TTensor& p) {
  return sqrt(magSq(p));
 }
 friend TTensor unit(const TTensor& p) {
  TTensor r = p;
  Scalar mg = mag(r);
  r /= mg;
  return r;
 }
 friend Scalar tr(const TTensor& p) {
  return p[0] + p[1] + p[2];
 }
 friend TTensor dev(const TTensor& p,const Scalar factor = 1.) {
  TTensor r = p;
  Scalar t = tr(p) * factor / 3;
  r[0] -= t;
  r[1] -= t;
  r[2] -= t;
  return r;
 }
 friend TTensor hyd(const TTensor& p,const Scalar factor = 1.) {
  TTensor r(1,1,1);
  Scalar t = tr(p) * factor / 3;
  r[0] = t;
  r[1] = t;
  r[2] = t;
  return r;
 }
 /*IO*/
 friend std::ostream& operator << (std::ostream& os, const TTensor<SIZE>& p) {
  for(Int i = 0;i < SIZE;i++)
   os << p[i] << " ";
```

```
368   return os;
369  }
370  friend std::istream& operator >> (std::istream& is, TTensor<SIZE>& p) {
371   for(Int i = 0;i < SIZE;i++)
372    is >> p[i];
373   return is;
374  }
375 };
376 /*typedef tensors*/
377 typedef TTensor<3> Vector;
378 typedef TTensor<6> STensor;
379 typedef TTensor<9> Tensor;
380 
381 /*Tensor operations*/
382 Vector operator ^ (const Vector& p,const Vector& q);
383 STensor mul(const Vector& p);
384 Tensor mul(const Vector& p,const Vector& q);
385 Tensor mul(const Tensor& p,const Tensor& q);
386 STensor mul(const STensor& p,const STensor& q);
387 Vector dot(const Vector&,const Tensor&);
388 Vector dot(const Vector&,const STensor&);
389 STensor sym(const Tensor& p);
390 Tensor skw(const Tensor& p);
391 Tensor trn(const Tensor& p);
392 Vector rotate(const Vector& v,const Vector& N,const Scalar& theta);
393 
394 /*constants*/
395 namespace Constants {
396  enum {
397   XX, YY , ZZ , XY, YZ , XZ , YX, ZY , ZX
398  };
399  const Int MAX_INT = Int(1 << 31);
400  const Scalar PI = Scalar(3.14159265358979323846264);
401  const Scalar E = Scalar(2.71828182845904523536028);
402  const Scalar MachineEpsilon = (sizeof(Scalar) == 4) ? Scalar(1e-8) : Scalar(1e-15);
403  const Vector I_V = Vector(1,1,1);
404  const Tensor I_T = Tensor(1,1,1);
405  const STensor I_ST = STensor(1,1,1);
406 }
407 
408 FORCEINLINE bool equal(const Scalar& p,const Scalar& q) {
```

```
409  return mag(p - q) <= (Constants::MachineEpsilon * pow(10.0,double(sizeof(Scalar))));
410 }
411 FORCEINLINE bool equal(const Vector& p,const Vector& q) {
412  return (equal(p[0],q[0]) && equal(p[1],q[1]) && equal(p[2],q[2]));
413 }
414 /*for symmetry boundary condition*/
415 FORCEINLINE Scalar sym(const Scalar& p,const Vector& n) {
416  return p;
417 }
418 FORCEINLINE Vector sym(const Vector& p,const Vector& n) {
419  Vector en = unit(n);
420  STensor A = Constants::I_ST - mul(en);
421  Vector r = dot(p,A);
422  Scalar magR = mag(r);
423  if(equal(magR,Scalar(0)))
424   return r;
425  return r * (mag(p) / magR);
426 }
427 FORCEINLINE STensor sym(const STensor& p,const Vector& n) {
428  Vector en = unit(n);
429  STensor A = Constants::I_ST - mul(en);
430  STensor r = mul(mul(A,p),A);
431  Scalar magR = mag(r);
432  if(equal(magR,Scalar(0)))
433   return r;
434  return r * (mag(p) / magR);
435 }
436 FORCEINLINE Tensor sym(const Tensor& p,const Vector& n) {
437  Vector en = unit(n);
438  Tensor A = Constants::I_T - mul(en,en);
439  Tensor r = mul(mul(A,p),A);
440  Scalar magR = mag(r);
441  if(equal(magR,Scalar(0)))
442   return r;
443  return r * (mag(p) / magR);
444 }
445 /*
446  * Blending
447  */
448 template <class T>
449 T Interpolate_face (Scalar r,Scalar s, T x00, T x01, T x10,
```

```
450   T x11, T xr0, T xr1, T x0s, T x1s
451   ) {
452
453   T result =
454     - ( 1.0 - r ) * ( 1.0 - s ) * x00
455     + ( 1.0 - r )           * x0s
456     - ( 1.0 - r ) *      s  * x01
457     +             ( 1.0 - s ) * xr0
458     +                    s  * xr1
459     -        r   * ( 1.0 - s ) * x10
460     +        r               * x1s
461     -        r   *        s  * x11;
462
463   return result;
464 }
465 template <class T>
466 T Interpolate_cell ( Scalar r, Scalar s, Scalar t,
467   T x000, T x001, T x010, T x011,
468   T x100, T x101, T x110, T x111,
469   T xr00, T xr01, T xr10, T xr11,
470   T x0s0, T x0s1, T x1s0, T x1s1,
471   T x00t, T x01t, T x10t, T x11t,
472   T x0st, T x1st, T xr0t, T xr1t, T xrs0, T xrs1
473   ) {
474
475  T result =
476    ( 1.0 - r ) * ( 1.0 - s ) * ( 1.0 - t ) * x000
477       - ( 1.0 - r ) * ( 1.0 - s )         * x00t
478       + ( 1.0 - r ) * ( 1.0 - s ) *    t  * x001
479       - ( 1.0 - r )           * ( 1.0 - t ) * x0s0
480       + ( 1.0 - r )                       * x0st
481       - ( 1.0 - r )           *        t  * x0s1
482       + ( 1.0 - r ) *      s  * ( 1.0 - t ) * x010
483       - ( 1.0 - r ) *      s              * x01t
484       + ( 1.0 - r ) *      s  *        t  * x011
485       -             ( 1.0 - s ) * ( 1.0 - t ) * xr00
486       +             ( 1.0 - s )           * xr0t
487       -             ( 1.0 - s ) *      t  * xr01
488       +                         ( 1.0 - t ) * xrs0
489       +                               t  * xrs1
490       -                   s  * ( 1.0 - t ) * xr10
```

```
491       +                   s              * xr1t
492       -                   s   *        t   * xr11
493       +        r   * ( 1.0 - s ) * ( 1.0 - t ) * x100
494       -        r   * ( 1.0 - s )           * x10t
495       +        r   * ( 1.0 - s ) *      t   * x101
496       -        r               * ( 1.0 - t ) * x1s0
497       +        r                             * x1st
498       -        r               *        t   * x1s1
499       +        r   *        s   * ( 1.0 - t ) * x110
500       -        r   *        s              * x11t
501       +        r   *        s   *        t   * x111;
502
503   return result;
504 }
505
506 /*iterator loops*/
507 #define forEach(field,i)        \
508  for(register Int i = 0;i < (field).size();i++)
509
510 #define forEachRev(field,i)        \
511  for(register int i = (field).size() - 1;i >= 0;i--)
512
513 #define forEachS(field,i,strt)       \
514  for(register Int i = strt;i < (field).size();i++)
515
516 #define forEachSRev(field,i,strt)      \
517  for(register int i = (field).size() - 1;i >= strt;i--)
518
519 #define forEachIt(cont,field,it)      \
520  for(cont::iterator it = (field).begin();   \
521   it != (field).end();++it)
522
523 /*
524  * end
525  */
526 #endif
527 #include "tensor.h"
528
529 using namespace Constants;
530
531 Vector operator ^ (const Vector& p,const Vector& q) {
```

```
 Vector r;
 r[XX] = p[YY] * q[ZZ] - p[ZZ] * q[YY];
 r[YY] = p[ZZ] * q[XX] - p[XX] * q[ZZ];
 r[ZZ] = p[XX] * q[YY] - p[YY] * q[XX];
 return r;
}
Tensor mul(const Vector& p,const Vector& q) {
 Tensor r;
 r[XX] = p[XX] * q[XX];
 r[YY] = p[YY] * q[YY];
 r[ZZ] = p[ZZ] * q[ZZ];

 r[XY] = p[XX] * q[YY];
 r[YZ] = p[YY] * q[ZZ];
 r[XZ] = p[XX] * q[ZZ];

 r[YX] = p[YY] * q[XX];
 r[ZY] = p[ZZ] * q[YY];
 r[ZX] = p[ZZ] * q[XX];
 return r;
}
STensor mul(const Vector& p) {
 STensor r;
 r[XX] = p[XX] * p[XX];
 r[YY] = p[YY] * p[YY];
 r[ZZ] = p[ZZ] * p[ZZ];

 r[XY] = p[XX] * p[YY];
 r[YZ] = p[YY] * p[ZZ];
 r[XZ] = p[XX] * p[ZZ];
 return r;
}
Tensor mul(const Tensor& p,const Tensor& q) {
 Tensor r;
 r[XX] = p[XX] * q[XX] + p[XY] * q[YX] + p[XZ] * q[ZX];
 r[XY] = p[XX] * q[XY] + p[XY] * q[YY] + p[XZ] * q[ZY];
 r[XZ] = p[XX] * q[XZ] + p[XY] * q[YZ] + p[XZ] * q[ZZ];

 r[YX] = p[YX] * q[XX] + p[YY] * q[YX] + p[YZ] * q[ZX];
 r[YY] = p[YX] * q[XY] + p[YY] * q[YY] + p[YZ] * q[ZY];
 r[YZ] = p[YX] * q[XZ] + p[YY] * q[YZ] + p[YZ] * q[ZZ];
```

```
573
574  r[ZX] = p[ZX] * q[XX] + p[ZY] * q[YX] + p[ZZ] * q[ZX];
575  r[ZY] = p[ZX] * q[XY] + p[ZY] * q[YY] + p[ZZ] * q[ZY];
576  r[ZZ] = p[ZX] * q[XZ] + p[ZY] * q[YZ] + p[ZZ] * q[ZZ];
577
578  return r;
579 }
580 STensor mul(const STensor& p,const STensor& q) {
581  STensor r;
582  r[XX] = p[XX] * q[XX] + p[XY] * q[XY] + p[XZ] * q[XZ];
583  r[XY] = p[XX] * q[XY] + p[XY] * q[YY] + p[XZ] * q[YZ];
584  r[XZ] = p[XX] * q[XZ] + p[XY] * q[YZ] + p[XZ] * q[ZZ];
585
586  r[YY] = p[XY] * q[XY] + p[YY] * q[YY] + p[YZ] * q[YZ];
587  r[YZ] = p[XY] * q[XZ] + p[YY] * q[YZ] + p[YZ] * q[ZZ];
588  r[ZZ] = p[XZ] * q[XZ] + p[YZ] * q[YZ] + p[ZZ] * q[ZZ];
589
590  return r;
591 }
592 Vector dot(const Vector& p,const Tensor& q) {
593  Vector r;
594  r[XX] = q[XX] * p[XX] + q[XY] * p[YY] + q[XZ] * p[ZZ];
595  r[YY] = q[YX] * p[XX] + q[YY] * p[YY] + q[YZ] * p[ZZ];
596  r[ZZ] = q[ZX] * p[XX] + q[ZY] * p[YY] + q[ZZ] * p[ZZ];
597  return r;
598 }
599 Vector dot(const Vector& p,const STensor& q) {
600  Vector r;
601  r[XX] = q[XX] * p[XX] + q[XY] * p[YY] + q[XZ] * p[ZZ];
602  r[YY] = q[XY] * p[XX] + q[YY] * p[YY] + q[YZ] * p[ZZ];
603  r[ZZ] = q[XZ] * p[XX] + q[YZ] * p[YY] + q[ZZ] * p[ZZ];
604  return r;
605 }
606 STensor sym(const Tensor& p) {
607  STensor r;
608  r[XX] = p[XX];
609  r[YY] = p[YY];
610  r[ZZ] = p[ZZ];
611
612  r[XY] = (p[XY] + p[YX]) / 2;
613  r[YZ] = (p[YZ] + p[ZY]) / 2;
```

```
614  r[XZ] = (p[XZ] + p[ZX]) / 2;
615  return r;
616 }
617 Tensor skw(const Tensor& p) {
618  Tensor r;
619  r[XX] = 0;
620  r[YY] = 0;
621  r[ZZ] = 0;
622
623  r[XY] = (p[XY] - p[YX]) / 2;
624  r[YZ] = (p[YZ] - p[ZY]) / 2;
625  r[XZ] = (p[XZ] - p[ZX]) / 2;
626
627  r[YX] = (p[YX] - p[XY]) / 2;
628  r[ZY] = (p[ZY] - p[YZ]) / 2;
629  r[ZX] = (p[ZX] - p[XZ]) / 2;
630  return r;
631 }
632 Tensor trn(const Tensor& p) {
633  Tensor r = p;
634  r[XX] = p[XX];
635  r[YY] = p[YY];
636  r[ZZ] = p[ZZ];
637
638  r[XY] = p[YX];
639  r[YZ] = p[ZY];
640  r[XZ] = p[ZX];
641
642  r[YX] = p[XY];
643  r[ZY] = p[YZ];
644  r[ZX] = p[XZ];
645  return r;
646 }
647 Vector rotate(const Vector& v,const Vector& N,const Scalar& theta) {
648  Vector r;
649  Scalar sum = v & N;
650  Scalar cost = cos(theta), sint = sin(theta);
651  r[XX] = N[XX] * sum * (1 - cost) + v[XX] * cost + (-N[ZZ] * v[YY] + N[YY] * v[ZZ]) *
        sint;
652  r[YY] = N[YY] * sum * (1 - cost) + v[YY] * cost + (+N[ZZ] * v[XX] - N[XX] * v[ZZ]) *
        sint;
```

```
653  r[ZZ] = N[ZZ] * sum * (1 - cost) + v[ZZ] * cost + (-N[YY] * v[XX] + N[XX] * v[YY]) *
         sint;
654  return r;
655 }
656 #ifndef __FIELD_H
657 #define __FIELD_H
658
659 #include <list>
660 #include <sstream>
661 #include "mesh.h"
662 #include "mp.h"
663
664 /*******************************************************************************
665  *                            Control parameters
666  *******************************************************************************/
667 namespace Controls {
668
669  enum Scheme{
670   CDS,UDS,HYBRID,BLENDED,LUD,CDSS,MUSCL,QUICK,
671   VANLEER,VANALBADA,MINMOD,SUPERBEE,SWEBY,QUICKL,UMIST,
672   DDS,FROMM
673  };
674  enum NonOrthoScheme {
675   NONE,MINIMUM, ORTHOGONAL, OVER_RELAXED
676  };
677  enum TimeScheme {
678   EULER, SECOND_ORDER
679  };
680  enum Solvers {
681   JACOBI, SOR, PCG
682  };
683  enum Preconditioners {
684   NOP,DIAG,SORP,DILU
685  };
686  enum CommMethod {
687   BLOCKED, ASYNCHRONOUS
688  };
689  enum State {
690   STEADY, TRANSIENT
691  };
692
```

```
693  extern Scheme convection_scheme;
694  extern Int TVDbruner;
695  extern Scheme interpolation_scheme;
696  extern NonOrthoScheme nonortho_scheme;
697  extern TimeScheme time_scheme;
698  extern Solvers Solver;
699  extern Preconditioners Preconditioner;
700  extern CommMethod ghost_exchange;
701  extern CommMethod parallel_method;
702  extern State state;
703
704  extern Scalar SOR_omega;
705  extern Scalar tolerance;
706  extern Scalar blend_factor;
707  extern Scalar time_scheme_factor;
708  extern Scalar dt;
709
710  extern Int max_iterations;
711  extern Int write_interval;
712  extern Int start_step;
713  extern Int end_step;
714  extern Int n_deferred;
715  extern Int save_average;
716 }
717
718 namespace {
719
720  enum ACCESS {
721   NO = 0, READ = 1, WRITE = 2,READWRITE = 3,STOREPREV = 4
722  };
723
724 }
725 /* ****************************************************************************
726  *                  Field variables defined on mesh
727  * ****************************************************************************/
728 template <class type,ENTITY entity>
729 class MeshField {
730 private:
731  type*      P;
732  int        allocated;
733  static Int SIZE;
```

```
734 public:
735  ACCESS     access;
736  Int        fIndex;
737  std::string fName;
738 
739  /*common*/
740  static const Int TYPE_SIZE = sizeof(type) / sizeof(Scalar);
741  static std::list<MeshField*> fields_;
742  static std::list<type*> mem_;
743 
744    /*constructors*/
745  MeshField(const char* str = "", ACCESS a = NO) :
746    P(0),allocated(0),access(a),fName(str) {
747   construct(str,a);
748  }
749  MeshField(const MeshField& p) : allocated(0) {
750   allocate();
751   forEach(*this,i)
752    P[i] = p[i];
753  }
754  MeshField(const type& p) : allocated(0) {
755   allocate();
756   forEach(*this,i)
757    P[i] = p;
758  }
759  explicit MeshField(const bool) : allocated(0) {
760  }
761  /*allocators*/
762  void allocate() {
763   if(mem_.empty()) {
764    switch(entity) {
765     case CELL: SIZE = Mesh::gCells.size(); break;
766     case FACET: SIZE = Mesh::gFacets.size(); break;
767     case VERTEX: SIZE = Mesh::gVertices.size(); break;
768    }
769    P = new type[SIZE];
770   } else {
771    P = mem_.front();
772    mem_.pop_front();
773   }
774   allocated = 1;
```

```
775 }
776 void allocate(std::vector<type>& q) {
777  switch(entity) {
778    case CELL: SIZE = Mesh::gCells.size(); break;
779    case FACET: SIZE = Mesh::gFacets.size(); break;
780    case VERTEX: SIZE = Mesh::gVertices.size(); break;
781  }
782  P = &q[0];
783  allocated = 0;
784 }
785 void construct(const char* str = "", ACCESS a = NO) {
786  access = a;
787  fName = str;
788  if(Mesh::gCells.size())
789   allocate();
790  fIndex = Util::hash_function(str);
791  if(fIndex)
792   fields_.push_back(this);
793 }
794 /*d'tor re-cycles memory */
795 ~MeshField() {
796  if(allocated && !Util::Terminated) {
797   mem_.push_front(P);
798   if(fIndex)
799    fields_.remove(this);
800  }
801 }
802
803 /*static functions*/
804 void readInternal(std::istream&);
805 void read(Int step);
806 void write(Int step);
807
808 /*accessors*/
809 Int size() const {
810  return SIZE;
811 }
812 type& operator [] (Int i) const {
813  return P[i];
814 }
815 /*unary ops*/
```

```
816  MeshField operator - () {
817   MeshField r;
818   forEach(*this,i)
819    r[i] = -P[i];
820   return r;
821  }
822  friend MeshField<Scalar,entity> operator & (const MeshField& p,const MeshField& q) {
823   MeshField<Scalar,entity> r;
824   forEach(r,i)
825    r[i] = p[i] & q[i];
826   return r;
827  }
828  /*unrolled operations*/
829 #define Op($)              \
830  MeshField& operator $(const MeshField& q) {      \
831   forEach(*this,i)           \
832    P[i] $ q[i];           \
833   return *this;             \
834  }
835 #define SOp($)               \
836  MeshField& operator $(const Scalar& q) {      \
837   forEach(*this,i)           \
838    P[i] $ q;            \
839   return *this;             \
840  }
841 #define Fp(name)             \
842  friend MeshField name(const MeshField& p,const MeshField& s) { \
843   MeshField r;             \
844   forEach(r,i)             \
845    r[i] = name(p[i],s[i]);         \
846   return r;              \
847  }
848 #define Fp1(name)              \
849  friend MeshField name(const MeshField& p,const Scalar& s) {  \
850   MeshField r;             \
851   forEach(r,i)             \
852    r[i] = name(p[i],s);          \
853   return r;              \
854  }
855 #define Fp2(name)              \
856  friend MeshField name(const MeshField& p) {      \
```

```
857  MeshField r;            \
858  forEach(r,i)            \
859   r[i] = name(p[i]);          \
860  return r;             \
861 }
862    /*define ops*/
863 Op(=);
864 Op(+=);
865 Op(-=);
866 Op(*=);
867 Op(/=);
868 SOp(=);
869 SOp(+=);
870 SOp(-=);
871 SOp(*=);
872 SOp(/=);
873 Fp(sdiv);
874 /*from math.h*/
875 Fp2(acos);
876 Fp2(asin);
877 Fp2(atan);
878 Fp(atan2);
879 Fp2(ceil);
880 Fp2(cos);
881 Fp2(cosh);
882 Fp2(exp);
883 Fp2(fabs);
884 Fp2(floor);
885 Fp2(log);
886 Fp2(log10);
887 Fp1(pow);
888 Fp2(sin);
889 Fp2(sinh);
890 Fp2(sqrt);
891 Fp2(tan);
892 Fp2(tanh);
893 Fp(min);
894 Fp(max);
895 /*additional*/
896 Fp2(unit);
897 #undef Op
```

```
898 #undef SOp
899 #undef Fp
900 #undef Fp1
901 #undef Fp2
902  Operator(MeshField,+);
903  Operator(MeshField,-);
904  Operator(MeshField,*);
905  Operator(MeshField,/);
906  /*friend ops*/
907  friend MeshField<Scalar,entity> mag(const MeshField& p) {
908   MeshField<Scalar,entity> r;
909   forEach(r,i)
910    r[i] = mag(p[i]);
911   return r;
912  }
913  friend MeshField dev(const MeshField& p,const Scalar factor = 1.) {
914   MeshField r;
915   forEach(r,i)
916    r[i] = dev(p[i],factor);
917   return r;
918  }
919  friend MeshField hyd(const MeshField& p,const Scalar factor = 1.) {
920   MeshField r;
921   forEach(r,i)
922    r[i] = hyd(p[i],factor);
923   return r;
924  }
925  /*relax*/
926  void Relax(const MeshField& po,Scalar UR) {
927   forEach(*this,i)
928    P[i] = po[i] + (P[i] - po[i]) * UR;
929  }
930  /*read/write all fields*/
931  static void readAll(Int step) {
932   forEachIt(typename std::list<MeshField*>, fields_, it) {
933    if((*it)->access & READ)
934     (*it)->read(step);
935   }
936  }
937  static void writeAll(Int step) {
938   forEachIt(typename std::list<MeshField*>, fields_, it) {
```

```
939   if((*it)->access & WRITE)
940    (*it)->write(step);
941  }
942 }
943 static int count_writable() {
944  int count = 0;
945  forEachIt(typename std::list<MeshField*>, fields_, it) {
946   if((*it)->access & WRITE)
947    count++;
948  }
949  return count;
950 }
951 static void writeVtkCellAll(std::ostream& os) {
952  MeshField<type,CELL>* pf;
953  forEachIt(typename std::list<MeshField*>, fields_, it) {
954   pf = *it;
955   if(pf->access & WRITE) {
956    os << pf->fName <<" "<< TYPE_SIZE <<" "
957     << Mesh::gBCellsStart << " float" << std::endl;
958    for(Int i = 0;i < Mesh::gBCellsStart;i++)
959     os << (*pf)[i] << std::endl;
960    os << std::endl;
961   }
962  }
963 }
964 static void writeVtkVertexAll(std::ostream& os) {
965  MeshField<type,VERTEX> vf;
966  forEachIt(typename std::list<MeshField*>, fields_, it) {
967   if((*it)->access & WRITE) {
968    vf = cds(cds(*(*it)));
969    os << (*it)->fName <<" "<< TYPE_SIZE <<" "
970     << vf.size() << " float" << std::endl;
971    forEach(vf,i)
972     os << vf[i] << std::endl;
973    os << std::endl;
974   }
975  }
976 }
977 /*interpolation*/
978 typedef std::list< MeshField<type,VERTEX> > vertexFieldsType;
979 static vertexFieldsType* vf_fields_;
```

```
980 static void interpolateVertexAll() {
981  vf_fields_ = new vertexFieldsType;
982  vf_fields_->clear();
983  MeshField<type,VERTEX> vf;
984  forEachIt(typename std::list<MeshField*>, fields_, it) {
985   if((*it)->access & WRITE) {
986    vf = cds(cds(*(*it)));
987    vf_fields_->push_back(vf);
988   }
989  }
990 }
991 /*Store previous values*/
992 MeshField* tstore;
993 void initStore() {
994  tstore = new MeshField[2];
995  access = ACCESS(int(access) | STOREPREV);
996  updateStore();
997 }
998 void updateStore() {
999  tstore[1] = tstore[0];
1000  tstore[0] = *this;
1001 }
1002 /*Time history*/
1003 static std::vector<std::ofstream*> tseries;
1004 static std::vector<MeshField*> tavgs;
1005 static std::vector<MeshField*> tstds;
1006
1007 static void initTimeSeries() {
1008  MeshField<type,CELL>* pf;
1009  int sz = fields_.size();
1010  forEachIt(typename std::list<MeshField*>, fields_, it) {
1011   pf = *it;
1012   if(pf->access & WRITE) {
1013    if(Mesh::probeCells.size()) {
1014     std::string name = pf->fName + "i";
1015     std::ofstream* of = new std::ofstream(name.c_str());
1016     tseries.push_back(of);
1017    }
1018    if(Controls::save_average) {
1019     std::string name;
1020     name = pf->fName + "avg";
```

```
1021     MeshField* avg = new MeshField(name.c_str(),READWRITE);
1022     tavgs.push_back(avg);
1023     name = pf->fName + "std";
1024     MeshField* std = new MeshField(name.c_str(),READWRITE);
1025     tstds.push_back(std);
1026    }
1027   }
1028  }
1029 }
1030 static void updateTimeSeries(int i) {
1031  int count = 0;
1032  MeshField<type,CELL>* pf;
1033  forEachIt(typename std::list<MeshField*>, fields_, it) {
1034   pf = *it;
1035   if(pf->access & WRITE) {
1036    if(Mesh::probeCells.size()) {
1037     std::ofstream& of = *tseries[count];
1038     of << i << " ";
1039     forEach(Mesh::probeCells,j)
1040      of << (*pf)[Mesh::probeCells[j]] << " ";
1041     of << endl;
1042    }
1043    if(Controls::save_average) {
1044     MeshField& avg = *tavgs[count];
1045     avg += (*pf);
1046     MeshField& std = *tstds[count];
1047     std += (*pf) * (*pf);
1048     count++;
1049    }
1050   }
1051   if(pf->access & STOREPREV) {
1052    pf->updateStore();
1053   }
1054  }
1055 }
1056 /*IO*/
1057 friend std::ostream& operator << (std::ostream& os, const MeshField& p) {
1058  forEach(p,i)
1059   os << p[i] << std::endl;
1060  os << std::endl;
1061  return os;
```

```
 }
 friend std::istream& operator >> (std::istream& is, MeshField& p) {
  forEach(p,i)
   is >> p[i];
  return is;
 }
};
#define forEachField(X) { \
 ScalarCellField::X; \
    VectorCellField::X; \
 STensorCellField::X; \
 TensorCellField::X; \
}
/***********************************
 *  Specific tensor operations
 ***********************************/
/* Default operator overload for scalar fields*/
#define Op(name,F,S)                    \
 template<class T,ENTITY E>                 \
 MeshField<T,E> name(const MeshField<F,E>& p,const MeshField<S,E>& q) {      \
  MeshField<T,E> r;                  \
  forEach(r,i)                    \
   r[i] = name(p[i],q[i]);                \
  return r;                      \
 }
Op(operator *,Scalar,T);
Op(operator /,Scalar,T);
Op(operator *,T,Scalar);
Op(operator /,T,Scalar);
#undef Op
/*multiply*/
template <ENTITY E>
MeshField<Tensor,E> mul(const MeshField<Vector,E>& p,const MeshField<Vector,E>& q) {
 MeshField<Tensor,E> r;
 forEach(r,i)
  r[i] = mul(p[i],q[i]);
 return r;
}
template <ENTITY E>
inline MeshField<Vector,E> mul(const MeshField<Vector,E>& p,const MeshField<Scalar,E>&
     q) {
```

```
1102  return p * q;
1103 }
1104 template <class T,ENTITY E>
1105 MeshField<T,E> mul(const MeshField<T,E>& p,const MeshField<T,E>& q) {
1106  MeshField<T,E> r;
1107  forEach(r,i)
1108   r[i] = mul(p[i],q[i]);
1109  return r;
1110 }
1111 /*dot*/
1112 template <ENTITY E,Int SIZE>
1113 MeshField<Vector,E> dot(const MeshField<TTensor<SIZE>,E>& p,const MeshField<Vector,E>&
      q) {
1114  MeshField<Vector,E> r;
1115  forEach(r,i)
1116   r[i] = dot(q[i],p[i]);
1117  return r;
1118 }
1119 template <ENTITY E>
1120 inline MeshField<Scalar,E> dot(const MeshField<Vector,E>& p,const MeshField<Vector,E>&
      q) {
1121  return p & q;
1122 }
1123 /*symmetric & skew-symmetric*/
1124 template <ENTITY E>
1125 MeshField<STensor,E> sym(const MeshField<Tensor,E>& p) {
1126  MeshField<STensor,E> r;
1127  forEach(r,i)
1128   r[i] = sym(p[i]);
1129  return r;
1130 }
1131 template <ENTITY E>
1132 MeshField<Tensor,E> skw(const MeshField<Tensor,E>& p) {
1133  MeshField<Tensor,E> r;
1134  forEach(r,i)
1135   r[i] = skw(p[i]);
1136  return r;
1137 }
1138 /*transpose*/
1139 template <ENTITY E>
1140 MeshField<Tensor,E> trn(const MeshField<Tensor,E>& p) {
```

```
 MeshField<Tensor,E> r;
 forEach(r,i)
  r[i] = trn(p[i]);
 return r;
}
/* ********************************************
 *  Input - output operations
 * ********************************************/
template <class T,ENTITY E>
void MeshField<T,E>::readInternal(std::istream& is) {
 using namespace Mesh;
 /*size*/
 char c;
 int size;
 std::string str;
 is >> str >> size;

 /*internal field*/
 if((c = Util::nextc(is)) && isalpha(c)) {
  T value = T(0);
  is >> str;
  if(str == "uniform")
   is >> value;
  *this = value;
 } else {
  char symbol;
  is >> size >> symbol;
  for(int i = 0;i < size;i++) {
   is >> (*this)[i];
  }
  is >> symbol;
 }
}
template <class T,ENTITY E>
void MeshField<T,E>::read(Int step) {
 using namespace Mesh;

 /*open*/
 std::stringstream path;
 path << fName << step;
 std::ifstream is(path.str().c_str());
```

```
1182  if(is.fail())
1183   return;
1184
1185  /*start reading*/
1186  std::cout << "Reading " << fName
1187    << step << std::endl;
1188  std::cout.flush();
1189
1190  /*internal*/
1191  readInternal(is);
1192
1193  /*boundary*/
1194  char c;
1195  BCondition<T>* bc;
1196  while((c = Util::nextc(is)) && isalpha(c)) {
1197   bc = new BCondition<T>(this->fName);
1198   is >> *bc;
1199   AllBConditions.push_back(bc);
1200  }
1201
1202  /*update BCs*/
1203  updateExplicitBCs(*this,true,true);
1204 }
1205 template <class T,ENTITY E>
1206 void MeshField<T,E>::write(Int step) {
1207  using namespace Mesh;
1208
1209  /*open*/
1210  std::stringstream path;
1211  path << fName << step;
1212  std::ofstream of(path.str().c_str());
1213
1214  /*size*/
1215  of << "size " << sizeof(T) / sizeof(Scalar) << std::endl;
1216
1217  /*internal field*/
1218  of << gBCellsStart << std::endl;
1219  of << "{" << std::endl;
1220  for(Int i = 0;i < gBCellsStart;i++)
1221   of << (*this)[i] << std::endl;
1222  of << "}" << std::endl;
```

```
1223
1224  /*boundary field*/
1225  BasicBCondition* bbc;
1226  BCondition<T>* bc;
1227  forEach(AllBConditions,i) {
1228   bbc = AllBConditions[i];
1229   if(bbc->fIndex == this->fIndex) {
1230    bc = static_cast<BCondition<T>*> (bbc);
1231    of << *bc << std::endl;
1232   }
1233  }
1234 }
1235
1236 /*static variables*/
1237 template <class T,ENTITY E>
1238 std::list<MeshField<T,E>*> MeshField<T,E>::fields_;
1239
1240 template <class T,ENTITY E>
1241 std::list<T*> MeshField<T,E>::mem_;
1242
1243 template <class T,ENTITY E>
1244 Int MeshField<T,E>::SIZE;
1245
1246 template <class T,ENTITY E>
1247 std::vector<std::ofstream*> MeshField<T,E>::tseries;
1248
1249 template <class T,ENTITY E>
1250 std::vector<MeshField<T,E>*> MeshField<T,E>::tavgs;
1251
1252 template <class T,ENTITY E>
1253 std::vector<MeshField<T,E>*> MeshField<T,E>::tstds;
1254
1255 template <class T,ENTITY E>
1256 typename MeshField<T,E>::vertexFieldsType* MeshField<T,E>::vf_fields_;
1257 /* typedefs */
1258 typedef MeshField<Scalar,CELL> ScalarCellField;
1259 typedef MeshField<Scalar,FACET> ScalarFacetField;
1260 typedef MeshField<Scalar,VERTEX> ScalarVertexField;
1261 typedef MeshField<Vector,CELL> VectorCellField;
1262 typedef MeshField<Vector,FACET> VectorFacetField;
1263 typedef MeshField<Vector,VERTEX> VectorVertexField;
```

```
1264 typedef MeshField<Tensor,CELL> TensorCellField;
1265 typedef MeshField<Tensor,FACET> TensorFacetField;
1266 typedef MeshField<Tensor,VERTEX> TensorVertexField;
1267 typedef MeshField<STensor,CELL> STensorCellField;
1268 typedef MeshField<STensor,FACET> STensorFacetField;
1269 typedef MeshField<STensor,VERTEX> STensorVertexField;
1270
1271 /* ************************************
1272  * global mesh fields
1273  * ************************************/
1274 namespace Mesh {
1275  extern VectorVertexField vC;
1276  extern VectorFacetField fC;
1277  extern VectorCellField cC;
1278  extern VectorFacetField fN;
1279  extern ScalarCellField cV;
1280  extern ScalarFacetField fI;
1281  extern ScalarCellField yWall;
1282
1283  void  initGeomMeshFields(bool = true);
1284  void  write_fields(Int);
1285  void  read_fields(Int);
1286  void  calc_walldist(Int,Int = 1);
1287 }
1288 /*******************************************************************************
1289  *                   matrix class defined on mesh
1290  *******************************************************************************/
1291 template <class type>
1292 struct MeshMatrix {
1293  MeshField<type,CELL>* cF;
1294  MeshField<type,CELL> Su;
1295  ScalarCellField ap;
1296  ScalarFacetField an[2];
1297  Int flags;
1298  enum FLAG {
1299   SYMMETRIC = 1
1300  };
1301  /*c'tors*/
1302  MeshMatrix() {
1303   cF = 0;
1304   flags = 0;
```

```
1305 }
1306 MeshMatrix(const MeshMatrix& p) {
1307  cF = p.cF;
1308  flags = p.flags;
1309  ap = p.ap;
1310  an[0] = p.an[0];
1311  an[1] = p.an[1];
1312  Su = p.Su;
1313 }
1314 MeshMatrix(const MeshField<type,CELL>& p) {
1315  cF = 0;
1316  flags = SYMMETRIC;
1317  ap = Scalar(0);
1318  an[0] = Scalar(0);
1319  an[1] = Scalar(0);
1320  Su = p;
1321 }
1322    /*operators*/
1323 MeshMatrix operator - () {
1324  MeshMatrix r;
1325  r.cF = cF;
1326  r.flags = flags;
1327  r.ap = -ap;
1328  r.an[0] = -an[0];
1329  r.an[1] = -an[1];
1330  r.Su = -Su;
1331  return r;
1332 }
1333 MeshMatrix& operator = (const MeshMatrix& q) {
1334  cF = q.cF;
1335  flags = q.flags;
1336  ap = q.ap;
1337  an[0] = q.an[0];
1338  an[1] = q.an[1];
1339  Su = q.Su;
1340  return *this;
1341 }
1342 MeshMatrix& operator += (const MeshMatrix& q) {
1343  flags &= q.flags;
1344  ap += q.ap;
1345  an[0] += q.an[0];
```

```
1346  an[1] += q.an[1];
1347  Su += q.Su;
1348  return *this;
1349 }
1350 MeshMatrix& operator -= (const MeshMatrix& q) {
1351  flags &= q.flags;
1352  ap -= q.ap;
1353  an[0] -= q.an[0];
1354  an[1] -= q.an[1];
1355  Su -= q.Su;
1356  return *this;
1357 }
1358 MeshMatrix& operator *= (const Scalar& q) {
1359  ap *= q;
1360  an[0] *= q;
1361  an[1] *= q;
1362  Su *= q;
1363  return *this;
1364 }
1365 MeshMatrix& operator /= (const Scalar& q) {
1366  ap /= q;
1367  an[0] /= q;
1368  an[1] /= q;
1369  Su /= q;
1370  return *this;
1371 }
1372 /*binary ops*/
1373 Operator(MeshMatrix,+);
1374 Operator(MeshMatrix,-);
1375 /*is equal to*/
1376 friend MeshMatrix operator == (const MeshMatrix& p,const MeshMatrix& q) {
1377  MeshMatrix r = p;
1378  r -= q;
1379  return r;
1380 }
1381 /*relax*/
1382 void Relax(Scalar UR) {
1383  ap /= UR;
1384  Su += (*cF) * ap * (1 - UR);
1385 }
1386 /*Fix*/
```

```
1387  void Fix(Int c,type value) {
1388   /*diagonal fix*/
1389   ap[c] = 10e30;
1390   Su[c] = value * 10e30;
1391  }
1392  /*IO*/
1393  friend std::ostream& operator << (std::ostream& os, const MeshMatrix& p) {
1394   os << p.ap << std::endl << std::endl;
1395   os << p.an[0] << std::endl << std::endl;
1396   os << p.an[1] << std::endl << std::endl;
1397   os << p.Su << std::endl << std::endl;
1398   return os;
1399  }
1400  friend std::istream& operator >> (std::istream& is, MeshMatrix& p) {
1401   is >> p.ap;
1402   is >> p.an[0];
1403   is >> p.an[1];
1404   is >> p.Su;
1405   return is;
1406  }
1407 };
1408
1409 /*typedefs*/
1410 typedef MeshMatrix<Scalar> ScalarMeshMatrix;
1411 typedef MeshMatrix<Vector> VectorMeshMatrix;
1412 typedef MeshMatrix<Tensor> TensorMeshMatrix;
1413 typedef MeshMatrix<STensor> STensorMeshMatrix;
1414
1415 /* ************************************
1416  * Implicit boundary conditions
1417  * ************************************/
1418  template <class T>
1419  void applyImplicitBCs(const MeshMatrix<T>& M) {
1420   using namespace Mesh;
1421   MeshField<T,CELL>& cF = *M.cF;
1422   BasicBCondition* bbc;
1423   BCondition<T>* bc;
1424
1425   /*boundary conditions*/
1426   forEach(AllBConditions,i) {
1427    bbc = AllBConditions[i];
```

```
1428    if(bbc->fIndex == cF.fIndex) {
1429     if(bbc->cIndex == NEUMANN ||
1430      bbc->cIndex == SYMMETRY)
1431      ;
1432     else continue;
1433
1434     bc = static_cast<BCondition<T>*> (bbc);
1435     Int sz = bc->bdry->size();
1436     if(sz == 0) continue;
1437
1438     for(Int j = 0;j < sz;j++) {
1439      Int k = (*bc->bdry)[j];
1440      Int c1 = gFO[k];
1441      Int c2 = gFN[k];
1442      if(bc->cIndex == NEUMANN) {
1443       Vector dv = cC[c2] - cC[c1];
1444       M.ap[c1] -= M.an[1][k];
1445       M.Su[c1] += M.an[1][k] * (bc->value * mag(dv));
1446       M.an[1][k] = 0;
1447      } else if(bc->cIndex == ROBIN) {
1448       Vector dv = cC[c2] - cC[c1];
1449       M.ap[c1] -= (1 - bc->shape) * M.an[1][k];
1450       M.Su[c1] += M.an[1][k] * (bc->shape * bc->value +
1451        (1 - bc->shape) * bc->tvalue * mag(dv));
1452       M.an[1][k] = 0;
1453      } else if(bc->cIndex == SYMMETRY) {
1454       M.ap[c1] -= M.an[1][k];
1455       M.Su[c1] += M.an[1][k] * (sym(cF[c1],fN[k]) - cF[c1]);
1456       M.an[1][k] = 0;
1457      }
1458     }
1459    }
1460   }
1461  }
1462 /* ************************************
1463  * Explicit boundary conditions
1464  * ************************************/
1465 template<class T,ENTITY E>
1466 void updateExplicitBCs(const MeshField<T,E>& cF,
1467        bool update_ghost = false,
1468        bool update_fixed = false
```

```
1469        ) {
1470 using namespace Mesh;
1471 BasicBCondition* bbc;
1472 BCondition<T>* bc;
1473 Scalar z = Scalar(0),
1474     zmin = Scalar(0),
1475     zmax = Scalar(0),
1476     zR = Scalar(0);
1477 Vector C(0);
1478
1479 /*boundary conditions*/
1480 forEach(AllBConditions,i) {
1481  bbc = AllBConditions[i];
1482  if(bbc->fIndex == cF.fIndex) {
1483   if(bbc->cIndex == GHOST)
1484    continue;
1485
1486   bc = static_cast<BCondition<T>*> (bbc);
1487   Int sz = bc->bdry->size();
1488   if(sz == 0) continue;
1489
1490   if(update_fixed) {
1491    if(bc->cIndex == DIRICHLET ||
1492     bc->cIndex == POWER ||
1493     bc->cIndex == LOG ||
1494     bc->cIndex == PARABOLIC ||
1495     bc->cIndex == INVERSE
1496     ) {
1497      Int ci,j;
1498      Scalar r;
1499      if(bc->zMax > 0) {
1500       zmin = bc->zMin;
1501       zmax = bc->zMax;
1502       zR = zmax - zmin;
1503      } else {
1504       zmin = Scalar(10e30);
1505       zmax = -Scalar(10e30);
1506       C = Vector(0);
1507       for(j = 0;j < sz;j++) {
1508        Facet& f = gFacets[j];
1509        forEach(f,k) {
```

```
1510        z = (vC[f[k]] & bc->dir);
1511        if(z < zmin)
1512         zmin = z;
1513        if(z > zmax)
1514         zmax = z;
1515       }
1516       C += fC[j];
1517      }
1518      C /= Scalar(sz);
1519      zR = zmax - zmin;
1520
1521      if(bc->cIndex == PARABOLIC) {
1522       ci = gFN[(*bc->bdry)[0]];
1523       zR = magSq(cC[ci] - C);
1524       for(j = 1;j < sz;j++) {
1525        ci = gFN[(*bc->bdry)[0]];
1526        r = magSq(cC[ci] - C);
1527        if(r < zR) zR = r;
1528       }
1529      }
1530     }
1531    }
1532   }
1533   for(Int j = 0;j < sz;j++) {
1534    Int k = (*bc->bdry)[j];
1535    Int c1 = gFO[k];
1536    Int c2 = gFN[k];
1537    if(bc->cIndex == NEUMANN) {
1538     Vector dv = cC[c2] - cC[c1];
1539     cF[c2] = cF[c1] + bc->value * mag(dv);
1540    } else if(bc->cIndex == ROBIN) {
1541     Vector dv = cC[c2] - cC[c1];
1542     cF[c2] = bc->shape * bc->value +
1543      (1 - bc->shape) * (cF[c1] + bc->tvalue * mag(dv));
1544    } else if(bc->cIndex == SYMMETRY) {
1545     cF[c2] = sym(cF[c1],fN[k]);
1546    } else if(bc->cIndex == CYCLIC) {
1547     Int c22;
1548     if(j < sz / 2)
1549      c22 = gFO[(*bc->bdry)[j + sz/2]];
1550     else
```

```
1551      c22 = gFO[(*bc->bdry)[j - sz/2]];
1552     cF[c2] = cF[c22];
1553    } else {
1554     if(update_fixed) {
1555      T v(0);
1556      z = (cC[c2] & bc->dir) - zmin;
1557      if(bc->cIndex == DIRICHLET) {
1558       v = bc->value;
1559      } else if(bc->cIndex == POWER) {
1560       if(z < 0) z = 0;
1561       if(z > zR) v = bc->value;
1562       else v = bc->value * pow(z / zR,bc->shape);
1563      } else if(bc->cIndex == LOG) {
1564       if(z < 0) z = 0;
1565       if(z > zR) v = bc->value;
1566       else v = bc->value * (log(1 + z / bc->shape) / log(1 + zR / bc->shape));
1567      } else if(bc->cIndex == PARABOLIC) {
1568       z = magSq(cC[c2] - C);
1569       v = bc->value * (z / zR);
1570      } else if(bc->cIndex == INVERSE) {
1571       v = bc->value / (z + bc->shape);
1572      }
1573      if(!bc->first && !equal(mag(bc->tvalue),0)) {
1574       T meanTI = v * (bc->tvalue * pow (z / zR,-bc->tshape));
1575       Scalar rFactor = 4 * ((rand() / Scalar(RAND_MAX)) - 0.5);
1576       v += ((cF[c2] - v) * 0.9 + (meanTI * rFactor) * 0.1);
1577      }
1578      bc->fixed[j] = cF[c2] = v;
1579     } else {
1580      cF[c2] = bc->fixed[j];
1581     }
1582    }
1583   }
1584   bc->first = false;
1585  }
1586 }
1587 /*ghost cells*/
1588 if(update_ghost && gInterMesh.size()) {
1589  exchange_ghost(&cF[0]);
1590 }
1591}
```

```
1592 /* ************************************
1593  * Fill boundary from internal values
1594  * ************************************/
1595 template<class T,ENTITY E>
1596 void fillBCs(const MeshField<T,E>& cF,
1597     bool update_ghost = false) {
1598  /*neumann update*/
1599  using namespace Mesh;
1600  forEachS(cF,i,gBCellsStart)
1601   cF[i] = cF[gFO[gCells[i][0]]];
1602  /*ghost cells*/
1603  if(update_ghost && gInterMesh.size()) {
1604   exchange_ghost(&cF[0]);
1605  }
1606 }
1607 /************************************
1608  * Exchange ghost cell information
1609  ************************************/
1610 template <class T>
1611 void exchange_ghost(T* P) {
1612  using namespace Mesh;
1613  /*blocked exchange*/
1614  if(Controls::ghost_exchange == Controls::BLOCKED) {
1615   MeshField<T,CELL> buffer;
1616   forEach(gInterMesh,i) {
1617    interBoundary& b = gInterMesh[i];
1618    IntVector& f = *(b.f);
1619    if(b.from < b.to) {
1620     //send
1621     forEach(f,j)
1622      buffer[j] = P[gFO[f[j]]];
1623     MP::send(&buffer[0],f.size(),b.to,MP::FIELD);
1624     //recieve
1625     MP::recieve(&buffer[0],f.size(),b.to,MP::FIELD);
1626     forEach(f,j)
1627      P[gFN[f[j]]] = buffer[j];
1628    } else {
1629     //recieve
1630     MP::recieve(&buffer[0],f.size(),b.to,MP::FIELD);
1631     forEach(f,j)
1632      P[gFN[f[j]]] = buffer[j];
```

```
1633      //send
1634      forEach(f,j)
1635       buffer[j] = P[gFO[f[j]]];
1636      MP::send(&buffer[0],f.size(),b.to,MP::FIELD);
1637     }
1638    }
1639      /*Asynchronous exchange*/
1640   } else {
1641    MeshField<T,CELL> sendbuf,recvbuf;
1642    std::vector<MP::REQUEST> request(2 * gInterMesh.size(),0);
1643    Int rcount = 0;
1644    //fill send buffer
1645    forEach(gInterMesh,i) {
1646     interBoundary& b = gInterMesh[i];
1647     IntVector& f = *(b.f);
1648     forEach(f,j)
1649      sendbuf[b.buffer_index + j] = P[gFO[f[j]]];
1650    }
1651
1652    forEach(gInterMesh,i) {
1653     interBoundary& b = gInterMesh[i];
1654     //non-blocking send/recive
1655     MP::isend(&sendbuf[b.buffer_index],b.f->size(),
1656      b.to,MP::FIELD,&request[rcount]);
1657     rcount++;
1658     MP::irecieve(&recvbuf[b.buffer_index],b.f->size(),
1659      b.to,MP::FIELD,&request[rcount]);
1660     rcount++;
1661    }
1662    //wait
1663    MP::waitall(rcount,&request[0]);
1664    //recieve buffer
1665    forEach(gInterMesh,i) {
1666     interBoundary& b = gInterMesh[i];
1667     IntVector& f = *(b.f);
1668     forEach(f,j)
1669      P[gFN[f[j]]] = recvbuf[b.buffer_index + j];
1670    }
1671   }
1672   /*end*/
1673  }
```

```
1674
1675 /* ******************************
1676  * matrix - vector product p * q
1677  * ******************************/
1678 template <class T>
1679 MeshField<T,CELL> operator * (const MeshMatrix<T>& p,const MeshField<T,CELL>& q) {
1680  using namespace Mesh;
1681  MeshField<T,CELL> r;
1682  Int c1,c2;
1683  r = q * p.ap;
1684  forEach(gFacets,f) {
1685   c1 = gFO[f];
1686   c2 = gFN[f];
1687   r[c1] -= q[c2] * p.an[1][f];
1688   r[c2] -= q[c1] * p.an[0][f];
1689  }
1690  return r;
1691 }
1692 /*matrix transopose - vector product pT * q */
1693 template <class T>
1694 MeshField<T,CELL> operator ^ (const MeshMatrix<T>& p,const MeshField<T,CELL>& q) {
1695  using namespace Mesh;
1696  MeshField<T,CELL> r;
1697  Int c1,c2;
1698  r = q * p.ap;
1699  forEach(gFacets,f) {
1700   c1 = gFO[f];
1701   c2 = gFN[f];
1702   r[c2] -= q[c1] * p.an[1][f];
1703   r[c1] -= q[c2] * p.an[0][f];
1704  }
1705  return r;
1706 }
1707 /* calculate RHS sum */
1708 template <class T>
1709 MeshField<T,CELL> getRHS(const MeshMatrix<T>& p) {
1710  using namespace Mesh;
1711  MeshField<T,CELL> r;
1712  Int c1,c2;
1713  r = p.Su;
1714  forEach(gFacets,f) {
```

```
1715   c1 = gFO[f];
1716   c2 = gFN[f];
1717   r[c1] += (*p.cF)[c2] * p.an[1][f];
1718   r[c2] += (*p.cF)[c1] * p.an[0][f];
1719  }
1720  return r;
1721 }
1722
1723 /* ******************************
1724  * Interpolate field operations
1725  * ******************************/
1726 /*central difference*/
1727 template<class type>
1728 MeshField<type,FACET> cds(const MeshField<type,CELL>& cF) {
1729  using namespace Mesh;
1730  MeshField<type,FACET> fF;
1731  forEach(fF,i) {
1732   fF[i] = (cF[gFO[i]] * (fI[i])) + (cF[gFN[i]] * (1 - fI[i]));
1733  }
1734  return fF;
1735 }
1736 /*upwind*/
1737 template<class type>
1738 MeshField<type,FACET> uds(const MeshField<type,CELL>& cF,const ScalarFacetField& flux)
      {
1739  using namespace Mesh;
1740  MeshField<type,FACET> fF;
1741  forEach(fF,i) {
1742   if(flux[i] >= 0) fF[i] = cF[gFO[i]];
1743   else fF[i] = cF[gFN[i]];
1744  }
1745  return fF;
1746 }
1747 /*facet data to vertex data */
1748 template<class type>
1749 MeshField<type,VERTEX> cds(const MeshField<type,FACET>& fF) {
1750  using namespace Mesh;
1751  std::vector<Scalar> cnt;
1752  MeshField<type,VERTEX> vF;
1753  cnt.assign(vF.size(),Scalar(0));
1754  Scalar dist;
```

```
1755
1756  vF = type(0);
1757  forEach(fF,i) {
1758   Facet& f = gFacets[i];
1759   if(gFN[i] < gBCellsStart) {
1760    forEach(f,j) {
1761     dist = 1.f / magSq(gVertices[f[j]] - fC[i]);
1762     vF[f[j]] += (fF[i] * dist);
1763     cnt[f[j]] += dist;
1764    }
1765   } else {
1766    forEach(f,j) {
1767     vF[f[j]] += Scalar(10e30) * fF[i];
1768     cnt[f[j]] += Scalar(10e30);
1769    }
1770   }
1771  }
1772  forEach(vF,i) {
1773   vF[i] /= cnt[i];
1774   if(mag(vF[i]) < Constants::MachineEpsilon)
1775    vF[i] = type(0);
1776  }
1777  return vF;
1778 }
1779 /* ************************
1780  * Integrate field operation
1781  * ************************/
1782 template<class type>
1783 MeshField<type,CELL> sum(const MeshField<type,FACET>& fF) {
1784  using namespace Mesh;
1785  MeshField<type,CELL> cF;
1786  cF = type(0);
1787  forEach(fF,i) {
1788   cF[gFO[i]] += fF[i];
1789   cF[gFN[i]] -= fF[i];
1790  }
1791  return cF;
1792 }
1793 /*****************************************************************
1794  * Gradient field operation.
1795  *   gradV(p) = Sum_f ( fN * p)
```

```
1796  *   grad(p) = gradV(p) / V
1797  * gradV(p) is integrated over the volume so it can be used directly in
1798  * finite volume equations just like div,lap,ddt,src etc...
1799  * grad(p) returns per-unit volume gradient at the centre.
1800  ******************************************************************************/
1801
1802 /*Explicit*/
1803 inline VectorCellField gradV(const ScalarFacetField& p) {
1804  return sum(mul(Mesh::fN,p));
1805 }
1806 inline VectorCellField gradV(const ScalarCellField& p) {
1807  return gradV(cds(p));
1808 }
1809 inline TensorCellField gradV(const VectorFacetField& p) {
1810  return sum(mul(Mesh::fN,p));
1811 }
1812 inline TensorCellField gradV(const VectorCellField& p) {
1813  return gradV(cds(p));
1814 }
1815
1816 /*Explicit*/
1817 inline VectorCellField grad(const ScalarFacetField& p) {
1818  VectorCellField f = gradV(p) / Mesh::cV;
1819  fillBCs(f,true);
1820  return f;
1821 }
1822 inline VectorCellField grad(const ScalarCellField& p) {
1823  return grad(cds(p));
1824 }
1825 inline TensorCellField grad(const VectorFacetField& p) {
1826  TensorCellField f = gradV(p) / Mesh::cV;
1827  fillBCs(f,true);
1828  return f;
1829 }
1830 inline TensorCellField grad(const VectorCellField& p) {
1831  return grad(cds(p));
1832 }
1833
1834 /* *********************************************
1835  * Laplacian field operation
1836  * *********************************************/
```

```
1837
1838 /*Implicit*/
1839 template<class type>
1840 MeshMatrix<type> lap(MeshField<type,CELL>& cF,const ScalarFacetField& mu) {
1841  using namespace Controls;
1842  using namespace Mesh;
1843  MeshMatrix<type> m;
1844  VectorFacetField K;
1845  Vector dv;
1846  Int c1,c2;
1847  Scalar D = 0;
1848  /*clear*/
1849  m.cF = &cF;
1850  m.flags |= m.SYMMETRIC;
1851  m.Su = type(0);
1852  m.ap = Scalar(0);
1853  forEach(mu,i) {
1854   c1 = gFO[i];
1855   c2 = gFN[i];
1856   dv = cC[c2] - cC[c1];
1857   /*diffusivity coefficient*/
1858   if(nonortho_scheme == NONE) {
1859    D = mag(fN[i]) / mag(dv);
1860   } else {
1861    if(nonortho_scheme == OVER_RELAXED) {
1862     D = ((fN[i] & fN[i]) / (fN[i] & dv));
1863    } else if(nonortho_scheme == MINIMUM) {
1864     D = ((fN[i] & dv) / (dv & dv));
1865    } else if(nonortho_scheme == ORTHOGONAL) {
1866     D = sqrt((fN[i] & fN[i]) / (dv & dv));
1867    }
1868    K[i] = fN[i] - D * dv;
1869   }
1870   /*coefficients*/
1871   m.an[0][i] = D * mu[i];
1872   m.an[1][i] = D * mu[i];
1873   m.ap[c1] += m.an[0][i];
1874   m.ap[c2] += m.an[1][i];
1875  }
1876  /*non-orthogonality handled through deferred correction*/
1877  if(nonortho_scheme != NONE) {
```

```
1878   MeshField<type,FACET> r = dot(cds(grad(cF)),K);
1879   type res;
1880   forEach(mu,i) {
1881    c1 = gFO[i];
1882    c2 = gFN[i];
1883    res = m.an[0][i] * (cF[c2] - cF[c1]);
1884    if(mag(r[i]) > Scalar(0.5) * mag(res))
1885     r[i] = Scalar(0.5) * res;
1886   }
1887   m.Su = sum(r);
1888  }
1889  /*end*/
1890  return m;
1891 }
1892
1893 template<class type>
1894 inline MeshMatrix<type> lap(MeshField<type,CELL>& cF,const ScalarCellField& mu) {
1895  return lap(cF,cds(mu));
1896 }
1897
1898 /* ****************************************************
1899  * Divergence field operation
1900  * ****************************************************/
1901 /*face flux*/
1902 inline ScalarFacetField flx(const VectorFacetField& p) {
1903  return dot(p,Mesh::fN);
1904 }
1905 inline ScalarFacetField flx(const VectorCellField& p) {
1906  return flx(cds(p));
1907 }
1908 inline VectorFacetField flx(const TensorFacetField& p) {
1909  return dot(p,Mesh::fN);
1910 }
1911 inline VectorFacetField flx(const TensorCellField& p) {
1912  return flx(cds(p));
1913 }
1914 /* Explicit */
1915 inline ScalarCellField div(const VectorFacetField& p) {
1916  return sum(flx(p));
1917 }
1918 inline ScalarCellField div(const VectorCellField& p) {
```

```
1919  return sum(flx(p));
1920 }
1921 inline VectorCellField div(const TensorFacetField& p) {
1922  return sum(flx(p));
1923 }
1924 inline VectorCellField div(const TensorCellField& p) {
1925  return sum(flx(p));
1926 }
1927 /* Implicit */
1928 template<class type>
1929 MeshMatrix<type> div(MeshField<type,CELL>& cF,const ScalarFacetField& flux,const
        ScalarFacetField& mu) {
1930  using namespace Controls;
1931  using namespace Mesh;
1932  MeshMatrix<type> m;
1933  Scalar F,G;
1934  m.cF = &cF;
1935  m.flags = 0;
1936  m.Su = type(0);
1937  m.ap = Scalar(0);
1938
1939  /*Implicit convection schemes*/
1940  bool isImplicit = (
1941   convection_scheme == CDS ||
1942   convection_scheme == UDS ||
1943   convection_scheme == BLENDED ||
1944   convection_scheme == HYBRID );
1945
1946  if(isImplicit) {
1947   ScalarFacetField gamma;
1948   if(convection_scheme == CDS)
1949    gamma = Scalar(1);
1950   else if(convection_scheme == UDS)
1951    gamma = Scalar(0);
1952   else if(convection_scheme == BLENDED)
1953    gamma = Scalar(blend_factor);
1954   else if(convection_scheme == HYBRID) {
1955    Scalar D;
1956    Vector dv;
1957    forEach(gFacets,j) {
1958     /*calc D - uncorrected */
```

```
1959      dv = cC[gFN[j]] - cC[gFO[j]];
1960      D = (mag(fN[j]) / mag(dv)) * mu[j];
1961      /*compare F and D */
1962      F = flux[j];
1963      if(F < 0) {
1964       if(-F * fI[j] > D) gamma[j] = 0;
1965       else gamma[j] = 1;
1966      } else {
1967       if(F * (1 - fI[j]) > D) gamma[j] = 0;
1968       else gamma[j] = 1;
1969      }
1970     }
1971    }
1972    forEach(flux,i) {
1973     F = flux[i];
1974     G = gamma[i];
1975     m.an[0][i] = ((G) * (-F * ( fI[i] )) + (1 - G) * (-max( F,0)));
1976     m.an[1][i] = ((G) * ( F * (1 - fI[i])) + (1 - G) * (-max(-F,0)));
1977     m.ap[gFO[i]] += m.an[0][i];
1978     m.ap[gFN[i]] += m.an[1][i];
1979    }
1980   /*deferred correction*/
1981   } else {
1982    forEach(flux,i) {
1983     F = flux[i];
1984     m.an[0][i] = -max( F,0);
1985     m.an[1][i] = -max(-F,0);
1986     m.ap[gFO[i]] += m.an[0][i];
1987     m.ap[gFN[i]] += m.an[1][i];
1988    }
1989
1990    MeshField<type,FACET> corr;
1991    if(convection_scheme == CDSS) {
1992     corr = cds(cF) - uds(cF,flux);
1993    } else if(convection_scheme == LUD) {
1994     VectorFacetField R = fC - uds(cC,flux);
1995     corr = dot(uds(grad(cF),flux),R);
1996    } else if(convection_scheme == MUSCL) {
1997     VectorFacetField R = fC - uds(cC,flux);
1998     corr = ( blend_factor ) * (cds(cF) - uds(cF,flux));
1999     corr += (1 - blend_factor) * (dot(uds(grad(cF),flux),R));
```

```
} else {
 /*
 TVD schemes
 ~~~~~~~~~~~
 Reference:
  M.S Darwish and F Moukalled "TVD schemes for unstructured grids"
  Versteeg and Malaskara
 Description:
  phi = phiU + psi(r) * [(phiD - phiC) * (1 - fi)]
 Schemes
  psi(r) = 0 =>UDS
  psi(r) = 1 =>CDS
 R is calculated as ratio of upwind and downwind gradient
  r = phiDC / phiCU
 Further modification to unstructured grid to better fit LUD scheme
  r = (phiDC / phiCU) * (fi / (1 - fi))
 */
 /*calculate r*/
 MeshField<type,FACET> q,r,phiDC,phiCU;
 ScalarFacetField uFI;
 {
  ScalarFacetField nflux = Scalar(0)-flux;
  phiDC = uds(cF,nflux) - uds(cF,flux);
  forEach(phiDC,i) {
   if(flux[i] >= 0) G = fI[i];
   else G = 1 - fI[i];
   uFI[i] = G;
  }
  /*Bruner's or Darwish way of calculating r*/
  if(TVDbruner) {
   VectorFacetField R = fC - uds(cC,flux);
   phiCU = 2 * (dot(uds(grad(cF),flux),R));
  } else {
   VectorFacetField R = uds(cC,nflux) - uds(cC,flux);
   phiCU = 2 * (dot(uds(grad(cF),flux),R)) - phiDC;
  }
  /*end*/
 }
 r = (phiCU / phiDC) * (uFI / (1 - uFI));
 forEach(phiDC,i) {
  if(equal(phiDC[i] * (1 - uFI[i]),type(0)))
```

```
2041      r[i] = type(0);
2042    }
2043    /*TVD schemes*/
2044    if(convection_scheme == VANLEER) {
2045     q = (r+fabs(r)) / (1+r);
2046    } else if(convection_scheme == VANALBADA) {
2047     q = (r+r*r) / (1+r*r);
2048    } else if(convection_scheme == MINMOD) {
2049     q = max(type(0),min(r,type(1)));
2050    } else if(convection_scheme == SUPERBEE) {
2051     q = max(min(r,type(2)),min(2*r,type(1)));
2052     q = max(q,type(0));
2053    } else if(convection_scheme == SWEBY) {
2054     Scalar beta = 2;
2055     q = max(min(r,type(beta)),min(beta*r,type(1)));
2056     q = max(q,type(0));
2057    } else if(convection_scheme == QUICKL) {
2058     q = min(2*r,(3+r)/4);
2059     q = min(q,type(2));
2060     q = max(q,type(0));
2061    } else if(convection_scheme == UMIST) {
2062     q = min(2*r,(3+r)/4);
2063     q = min(q,(1+3*r)/4);
2064     q = min(q,type(2));
2065     q = max(q,type(0));
2066    } else if(convection_scheme == QUICK) {
2067     q = (3+r)/4;
2068    } else if(convection_scheme == DDS) {
2069     q = 2;
2070    } else if(convection_scheme == FROMM) {
2071     q = (1+r)/2;
2072    }
2073    corr = q * phiDC * (1 - uFI);
2074    /*end*/
2075   }
2076   m.Su = sum(flux * corr);
2077  }
2078  return m;
2079 }
2080
2081 template<class type,ENTITY E>
```

```
2082 inline MeshMatrix<type> div(MeshField<type,CELL>& cF,const MeshField<Vector,E>& rhoU,
         const ScalarFacetField& mu) {
2083  return div(cF,div(rhoU),mu);
2084 }
2085 
2086 /* *******************************
2087  * Temporal derivative
2088  * *******************************/
2089 template<class type>
2090 MeshMatrix<type> ddt(MeshField<type,CELL>& cF,const ScalarCellField& rho) {
2091  MeshMatrix<type> m;
2092  m.cF = &cF;
2093  m.flags |= m.SYMMETRIC;
2094  if(Controls::time_scheme == Controls::EULER || !(cF.access & STOREPREV)) {
2095   if(Controls::time_scheme != Controls::EULER) cF.initStore();
2096   m.ap = (Mesh::cV * rho) / -Controls::dt;
2097   m.Su = cF * m.ap;
2098  } else if(Controls::time_scheme == Controls::SECOND_ORDER) {
2099   m.ap = (1.5 * Mesh::cV * rho) / -Controls::dt;
2100   m.Su = ((4.0 * cF - cF.tstore[1]) / 3.0) * m.ap;
2101  }
2102  m.an[0] = Scalar(0);
2103  m.an[1] = Scalar(0);
2104  return m;
2105 }
2106 template<class type>
2107 MeshMatrix<type> ddt2(MeshField<type,CELL>& cF,const ScalarCellField& rho) {
2108  MeshMatrix<type> m;
2109  m.cF = &cF;
2110  m.flags |= m.SYMMETRIC;
2111  if(!(cF.access & STOREPREV)) cF.initStore();
2112  m.ap = (Mesh::cV * rho) / -(Controls::dt * Controls::dt);
2113  m.Su = (2.0 * cF - cF.tstore[1]) * m.ap;
2114  m.an[0] = Scalar(0);
2115  m.an[1] = Scalar(0);
2116  return m;
2117 }
2118 /* *******************************
2119  * Linearized source term
2120  * *******************************/
2121 template<class type>
```

```
2122 MeshMatrix<type> src(MeshField<type,CELL>& cF,const ScalarCellField& Sc,const
         ScalarCellField Sp) {
2123  MeshMatrix<type> m;
2124  m.cF = &cF;
2125  m.flags |= m.SYMMETRIC;
2126  m.ap = -(Sp * Mesh::cV);
2127  m.an[0] = Scalar(0);
2128  m.an[1] = Scalar(0);
2129  m.Su = (Sc * Mesh::cV);
2130  return m;
2131 }
2132 /* ************************************
2133  *   CSR - compressed sparse row format
2134  *       * Used for on GPU computation
2135  *       * Propably for AMG too
2136  * ************************************/
2137 template <class T>
2138 class CSRMatrix {
2139 public:
2140  std::vector<Int> rows;
2141  std::vector<Int> cols;
2142  std::vector<Scalar> an;
2143  std::vector<Scalar> anT;
2144  std::vector<T> cF;
2145  std::vector<T> Su;
2146 public:
2147  template <class T1>
2148  CSRMatrix(const MeshMatrix<T1>& A) {
2149   using namespace Mesh;
2150   const Int N = A.ap.size();
2151   const Int NN = A.ap.size() +
2152              A.an[0].size() +
2153         A.an[1].size();
2154   register Int i,j,f;
2155
2156   /*resize*/
2157   cF.resize(N);
2158   Su.resize(N);
2159   rows.reserve(N + 1);
2160   cols.reserve(NN);
2161   an.reserve(NN);
```

```
2162  anT.reserve(NN);
2163
2164       /*source term*/
2165  for(i = 0;i < N;i++) {
2166   Su[i] = A.Su[i];
2167   cF[i] = (*A.cF)[i];
2168  }
2169
2170  /*fill matrix in CSR format.Diagonal element
2171    is always at the start of a row */
2172  Int cn = 0;
2173  for(i = 0;i < N;i++) {
2174   Cell& c = gCells[i];
2175
2176   rows.push_back(cn);
2177
2178   an.push_back(A.ap[i]);
2179   anT.push_back(A.ap[i]);
2180   cols.push_back(i);
2181   cn++;
2182
2183   forEach(c,j) {
2184    f = c[j];
2185    if(i == gFO[f]) {
2186     an.push_back(A.an[1][f]);
2187     anT.push_back(A.an[0][f]);
2188     cols.push_back(gFN[f]);
2189     cn++;
2190    } else {
2191     an.push_back(A.an[0][f]);
2192     anT.push_back(A.an[1][f]);
2193     cols.push_back(gFO[f]);
2194     cn++;
2195    }
2196   }
2197  }
2198  /*push extra row*/
2199  rows.push_back(cn);
2200 }
2201 /*IO*/
2202 friend std::ostream& operator << (std::ostream& os, const CSRMatrix& p) {
```

```
2203   os << p.rows << std::endl;
2204   os << p.cols << std::endl;
2205   os << p.an << std::endl;
2206   os << p.Su << std::endl;
2207   return os;
2208  }
2209  friend std::istream& operator >> (std::istream& is, CSRMatrix& p) {
2210   is >> p.rows;
2211   is >> p.cols;
2212   is >> p.an;
2213   is >> p.Su;
2214   return is;
2215  }
2216  /*end*/
2217 };
2218 /* ********************
2219  *       End
2220  * ********************/
2221 #endif
2222 #include "field.h"
2223
2224 using namespace std;
2225
2226 namespace Mesh {
2227  VectorVertexField vC;
2228  VectorFacetField fC;
2229  VectorCellField cC;
2230  VectorFacetField fN;
2231  ScalarCellField cV;
2232  ScalarFacetField fI;
2233  ScalarCellField yWall(false);
2234 }
2235 namespace Controls {
2236  Scheme convection_scheme = HYBRID;
2237  Int TVDbruner = 0;
2238  Scheme interpolation_scheme = CDS;
2239  NonOrthoScheme nonortho_scheme = OVER_RELAXED;
2240  TimeScheme time_scheme = EULER;
2241  Scalar time_scheme_factor = 1;
2242  Scalar blend_factor = Scalar(0.2);
2243  Scalar tolerance = Scalar(1e-5f);
```

```
 Scalar dt = Scalar(.1);
 Scalar SOR_omega = Scalar(1.7);
 Solvers Solver = PCG;
 Preconditioners Preconditioner = SORP;
 State state = STEADY;
 Int max_iterations = 500;
 Int write_interval = 20;
 Int start_step = 0;
 Int end_step = 2;
 Int n_deferred = 0;
 Int save_average = 0;
 CommMethod ghost_exchange = BLOCKED;
 CommMethod parallel_method = BLOCKED;
}

/*
 * Initialize geometric mesh fields
 */
void Mesh::initGeomMeshFields(bool remove_empty) {
 /*initialize mesh*/
 addBoundaryCells();
 calcGeometry();
 /* remove empty faces*/
 if(remove_empty) {
  Boundaries::iterator it = gBoundaries.find("delete");
  if(it != gBoundaries.end()) {
   removeBoundary(gBoundaries["delete"]);
   gBoundaries.erase(it);
  }
 }
 /*erase interior and empty boundaries*/
 for(Boundaries::iterator it = gBoundaries.begin();
              it != gBoundaries.end();) {
  if(it->second.size() <= 0 ||
   it->first.find("interior") != std::string::npos
   ) {
    gBoundaries.erase(it++);
  } else ++it;
 }
 /* Allocate fields*/
 vC.allocate(gVertices);
```

```
 fC.allocate(_fC);
 cC.allocate(_cC);
 fN.allocate(_fN);
 cV.allocate(_cV);
 fI.allocate();
 /* Facet interpolation factor to the owner of the face.
  * Neighbor takes (1 - f) */
 exchange_ghost(&cV[0]);
 exchange_ghost(&cC[0]);
 forEach(gFacets,i) {
  Int c1 = gFO[i];
  Int c2 = gFN[i];
  Scalar s1 = mag(cC[c1] - fC[i]);
  Scalar s2 = mag(cC[c2] - fC[i]);
  fI[i] = 1.f - s1 / (s1 + s2);
 }
 /*Construct wall distance field*/
 {
  yWall.construct("yWall");
  yWall = Scalar(0);
  /*boundary*/
  BCondition<Scalar>* bc;
  forEachIt(Boundaries,gBoundaries,it) {
   string bname = it->first;
   bc = new BCondition<Scalar>(yWall.fName);
   bc->bname = bname;
   if(bname.find("WALL") != std::string::npos) {
    bc->cname = "DIRICHLET";
    bc->value = Scalar(0);
   } else if(bname.find("interMesh") != std::string::npos) {
   } else {
    bc->cname = "NEUMANN";
    bc->value = Scalar(0);
   }
   bc->init_indices();
   AllBConditions.push_back(bc);
  }
  updateExplicitBCs(yWall,true,true);
 }
}
/*
```

```
2326  * Read/Write
2327  */
2328 void Mesh::write_fields(Int step) {
2329  forEachField(writeAll(step));
2330 }
2331 void Mesh::read_fields(Int step) {
2332  forEachField(readAll(step));
2333 }
2334 void Mesh::enroll(Util::ParamList& params) {
2335  using namespace Controls;
2336  using namespace Util;
2337 
2338  params.enroll("max_iterations",&max_iterations);
2339  params.enroll("write_interval",&write_interval);
2340  params.enroll("start_step",&start_step);
2341  params.enroll("end_step",&end_step);
2342  params.enroll("n_deferred",&n_deferred);
2343 
2344     params.enroll("blend_factor",&blend_factor);
2345  params.enroll("tolerance",&tolerance);
2346  params.enroll("dt",&dt);
2347  params.enroll("SOR_omega",&SOR_omega);
2348  params.enroll("time_scheme_factor",&time_scheme_factor);
2349 
2350  params.enroll("probe",&Mesh::probePoints);
2351 
2352  Option* op;
2353  op = new Option(&convection_scheme,17,
2354   "CDS","UDS","HYBRID","BLENDED","LUD","CDSS","MUSCL","QUICK",
2355   "VANLEER","VANALBADA","MINMOD","SUPERBEE","SWEBY","QUICKL","UMIST",
2356   "DDS","FROMM");
2357  params.enroll("convection_scheme",op);
2358  op = new BoolOption(&TVDbruner);
2359  params.enroll("tvd_bruner",op);
2360  op = new Option(&interpolation_scheme,2,"CDS","UDS");
2361  params.enroll("interpolation_scheme",op);
2362  op = new Option(&nonortho_scheme,4,"NONE","MINIMUM","ORTHOGONAL","OVER_RELAXED");
2363  params.enroll("nonortho_scheme",op);
2364  op = new Option(&time_scheme,2,"EULER","SECOND_ORDER");
2365  params.enroll("time_scheme",op);
2366  op = new Option(&Solver,3,"JACOBI","SOR","PCG");
```

```
 params.enroll("method",op);
 op = new Option(&Preconditioner,4,"NONE","DIAG","SOR","DILU");
 params.enroll("preconditioner",op);
 op = new Option(&state,2,"STEADY","TRANSIENT");
 params.enroll("state",op);
 op = new Option(&ghost_exchange,2,"BLOCKED","ASYNCHRONOUS");
 params.enroll("ghost_exchange",op);
 op = new Option(&parallel_method,2,"BLOCKED","ASYNCHRONOUS");
 params.enroll("parallel_method",op);
 op = new Util::BoolOption(&save_average);
 params.enroll("average",op);
}
#ifndef __HEX_MESH_H
#define __HEX_MESH_H

#include "mesh.h"

enum {
 LINEAR, GEOMETRIC, WALL, MIXED
};
enum {
 NONE = 0,ARC,COSINE,QUAD
};

struct Edge {
 int type;
 Scalar theta;
 Scalar L;
 Vector N;
 Vertex v[8];
 Edge() {
  type = NONE;
 }
};

struct MergeObject {
 Vertices vb;
 Facets  fb;
};

void hexMesh(Int* n,Scalar* s,Int* type,Vector* vp,Edge* edges,Mesh::MeshObject& mo);
```

```
2408 void merge(Mesh::MeshObject&,MergeObject&,Mesh::MeshObject&);
2409 void remove_duplicate(Mesh::MeshObject&);
2410 void merge(Mesh::MeshObject&,MergeObject&);
2411
2412 #endif
2413 #ifndef __MESH_H
2414 #define __MESH_H
2415
2416 #include <string>
2417 #include <vector>
2418 #include <map>
2419 #include "tensor.h"
2420 #include "util.h"
2421
2422 /*Index by ID instead of pointers */
2423 typedef std::vector<Int>  IntVector;
2424
2425 /*our basic building blocks */
2426 enum ENTITY {
2427  CELL, FACET, VERTEX
2428 };
2429
2430 /*typdefs*/
2431 typedef Vector       Vertex;
2432 typedef IntVector    Facet;
2433 typedef IntVector    Cell;
2434
2435 typedef std::vector<Vertex> Vertices;
2436 typedef std::vector<Facet> Facets;
2437 typedef std::vector<Cell> Cells;
2438 typedef std::map<std::string,IntVector> Boundaries;
2439
2440 /*global mesh*/
2441 namespace Mesh {
2442  struct interBoundary {
2443   IntVector* f;
2444   Int from;
2445   Int to;
2446   Int buffer_index;
2447  };
2448  struct MeshObject {
```

```
2449  /*vertices , facets and cells */
2450  Vertices v;
2451  Facets  f;
2452  Cells   c;
2453  /*other info*/
2454  std::string name;
2455  Boundaries bdry;
2456  IntVector fo;
2457  IntVector fn;
2458  std::vector<interBoundary> interMesh;
2459  /*start of boundary cells,facets & vertices*/
2460  Int     nv;
2461  Int     nf;
2462  Int     nc;
2463  /*funcs*/
2464  void write(std::ostream& os);
2465 };
2466
2467 extern std::vector<Vector> _fC;
2468 extern std::vector<Vector> _cC;
2469 extern std::vector<Vector> _fN;
2470 extern std::vector<Scalar> _cV;
2471 extern std::vector<bool> _reversed;
2472
2473 extern MeshObject    gMesh;
2474 extern std::string&  gMeshName;
2475 extern Vertices&  gVertices;
2476 extern Facets&   gFacets;
2477 extern Cells&   gCells;
2478 extern Boundaries&   gBoundaries;
2479 extern IntVector&    gFO;
2480 extern IntVector&    gFN;
2481 extern Int&          gBCellsStart;
2482 extern std::vector<interBoundary>& gInterMesh;
2483 extern Vertices      probePoints;
2484 extern IntVector     probeCells;
2485
2486 bool faceInBoundary(Int);
2487 void addBoundaryCells();
2488 void calcGeometry();
2489 void removeBoundary(IntVector&);
```

```
2490  void readMesh();
2491  void enroll(Util::ParamList& params);
2492  Int  findNearestCell(const Vector& v);
2493  Int  findNearestFace(const Vector& v);
2494  void getProbeCells(IntVector&);
2495  void getProbeFaces(IntVector&);
2496 }
2497 /*
2498  * Model for flow close to the wall (Law of the wall).
2499  *   1 -> Viscous layer
2500  *   2 -> Buffer layer
2501  *   3 -> Log-law layer
2502  * The wall function model is modified for rough surfaces
2503  * using Cebecci and Bradshaw formulae.
2504  */
2505 struct LawOfWall {
2506  Scalar E;
2507  Scalar kappa;
2508  Scalar ks;
2509  Scalar cks;
2510
2511  Scalar yLog;
2512
2513  LawOfWall() :
2514   E(9.8),
2515   kappa(0.41),
2516   ks(0),
2517   cks(0.5)
2518  {
2519   init();
2520  }
2521  void init() {
2522   yLog = 11.3f;
2523   for(Int i = 0;i < 20;i++)
2524    yLog = log(E * yLog) / kappa;
2525  }
2526  Scalar getUstar(Scalar nu,Scalar U,Scalar y) {
2527   Scalar a = kappa * U * y / nu;
2528   Scalar yp = a;
2529   for(Int i = 0;i < 10;i++)
2530    yp = (a + yp) / (1 + log(E * yp));
```

```
2531   Scalar ustar = yp * nu / y;
2532   return ustar;
2533  }
2534  Scalar getUp(Scalar ustar,Scalar nu,Scalar yp) {
2535   Scalar up,dB;
2536   Scalar ksPlus = (ustar * ks) / nu;
2537   if(ksPlus < 2.25) {
2538    dB = 0;
2539   } else if(ksPlus < 90) {
2540    dB = (1 / kappa) * log((ksPlus - 2.25) / 87.75 + cks * ksPlus)
2541                * sin(0.4258 * (log(ksPlus) - 0.811));
2542   } else {
2543    dB = (1 / kappa) * log(1 + cks * ksPlus);
2544   }
2545   if(yp > yLog) up = log(E * yp) / kappa - dB;
2546   else          up = yp;
2547   return up;
2548  }
2549  void write(std::ostream& os) const {
2550   os << "\tE " << E << std::endl;
2551   os << "\tkappa " << kappa << std::endl;
2552   os << "\tks " << ks << std::endl;
2553   os << "\tcks " << cks << std::endl;
2554  }
2555  bool read(std::istream& is,std::string str) {
2556   using namespace Util;
2557   if(!compare(str,"E")) {
2558    is >> E;
2559   } else if(!compare(str,"kappa")) {
2560    is >> kappa;
2561   } else if(!compare(str,"ks")) {
2562    is >> ks;
2563   } else if(!compare(str,"cks")) {
2564    is >> cks;
2565   } else
2566    return false;
2567   return true;
2568  }
2569 };
2570 /*Boundary condition types*/
2571 namespace Mesh {
```

```
2572  const Int DIRICHLET = Util::hash_function("DIRICHLET");
2573  const Int NEUMANN   = Util::hash_function("NEUMANN");
2574  const Int ROBIN     = Util::hash_function("ROBIN");
2575  const Int SYMMETRY  = Util::hash_function("SYMMETRY");
2576     const Int CYCLIC    = Util::hash_function("CYCLIC");
2577  const Int GHOST     = Util::hash_function("GHOST");
2578  const Int POWER     = Util::hash_function("POWER");
2579  const Int LOG       = Util::hash_function("LOG");
2580     const Int PARABOLIC = Util::hash_function("PARABOLIC");
2581  const Int INVERSE   = Util::hash_function("INVERSE");
2582  const Int ROUGHWALL = Util::hash_function("ROUGHWALL");
2583 }
2584 struct BasicBCondition {
2585  IntVector* bdry;
2586  Int    fIndex;
2587  Int    cIndex;
2588  std::string cname;
2589  std::string bname;
2590  std::string fname;
2591  LawOfWall low;
2592 };
2593 template <class type>
2594 struct BCondition : public BasicBCondition {
2595  type  value;
2596  Scalar shape;
2597  type  tvalue;
2598  Scalar tshape;
2599  Scalar zMin;
2600  Scalar zMax;
2601  Vector dir;
2602  bool  first;
2603  bool  read;
2604  std::vector<type> fixed;
2605
2606  BCondition(std::string tfname) {
2607   fname = tfname;
2608   reset();
2609  }
2610  void reset() {
2611   value = tvalue = type(0);
2612   shape = tshape = zMin = zMax = Scalar(0);
```

```
2613   dir = Vector(0,0,1);
2614  }
2615  void init_indices() {
2616   bdry = &Mesh::gBoundaries[bname];
2617   fixed.resize(bdry->size());
2618   first = true;
2619   read = false;
2620   fIndex = Util::hash_function(fname);
2621   cIndex = Util::hash_function(cname);
2622  }
2623 };
2624 /*IO*/
2625 template <class type>
2626 std::ostream& operator << (std::ostream& os, const BCondition<type>& p) {
2627  os << p.bname << "\n{\n";
2628  os << "\ttype " << p.cname << std::endl;
2629  if(!equal(mag(p.value),Scalar(0)))
2630   os << "\tvalue " << p.value << std::endl;
2631  if(!equal(p.shape,Scalar(0)))
2632   os << "\tshape " << p.shape << std::endl;
2633  if(!equal(mag(p.tvalue),Scalar(0)))
2634   os << "\ttvalue " << p.tvalue << std::endl;
2635  if(!equal(p.tshape,Scalar(0)))
2636   os << "\ttshape " << p.tshape << std::endl;
2637  if(!equal(p.dir,Vector(0,0,1)))
2638   os << "\tdir " << p.dir << std::endl;
2639  if(p.zMax > 0) {
2640   os << "\tzMin " << p.zMin << std::endl;
2641   os << "\tzMax " << p.zMax << std::endl;
2642  }
2643  if(p.read) {
2644   os << "\tfixed " << p.fixed << std::endl;
2645  }
2646  if(p.cIndex == Mesh::ROUGHWALL)
2647   p.low.write(os);
2648  os << "}\n";
2649  return os;
2650 }
2651 template <class type>
2652 std::istream& operator >> (std::istream& is, BCondition<type>& p) {
2653  using namespace Util;
```

```
2654  std::string str;
2655  char c;
2656
2657  p.reset();
2658  is >> p.bname >> c;
2659
2660  while(c = Util::nextc(is)) {
2661   if(c == '}') {
2662    is >> c;
2663    break;
2664   }
2665   is >> str;
2666   if(!compare(str,"type")) {
2667    is >> p.cname;
2668   } else if(!compare(str,"value")) {
2669    is >> p.value;
2670   } else if(!compare(str,"shape")) {
2671    is >> p.shape;
2672   } else if(!compare(str,"tvalue")) {
2673    is >> p.tvalue;
2674   } else if(!compare(str,"tshape")) {
2675    is >> p.tshape;
2676   } else if(!compare(str,"dir")) {
2677    is >> p.dir;
2678   } else if(!compare(str,"zMin")) {
2679    is >> p.zMin;
2680   } else if(!compare(str,"zMax")) {
2681    is >> p.zMax;
2682   } else if(!compare(str,"fixed")) {
2683    is >> p.fixed;
2684    p.read = true;
2685   } else if(p.low.read(is,str)) {
2686   }
2687  }
2688
2689  p.init_indices();
2690  p.low.init();
2691  return is;
2692 }
2693 /*list of all BCS*/
2694 namespace Mesh {
```

```
 extern std::vector<BasicBCondition*> AllBConditions;
}
#endif
#ifndef __MSH_MESH_H
#define __MSH_MESH_H

#include "mesh.h"

void readMshMesh(std::istream& is,Mesh::MeshObject& mo);
void writeMshMesh(std::ostream& os,Mesh::MeshObject& mo);

#endif
#include "hexMesh.h"
#include <cmath>

using namespace Mesh;

Vector center(const Vector& v1,const Vector& v2,const Vector& v3) {
 Vector v12 = v1 - v2;
 Vector v13 = v1 - v3;
 Vector v23 = v2 - v3;
 Scalar d = 2 * magSq(v12 ^ v23);
 Scalar a = magSq(v23) * (v12 & v13) / d;
 Scalar b = magSq(v13) * (-v12 & v23) / d;
 Scalar c = magSq(v12) * (v13 & v23) / d;
 return a * v1 + b * v2 + c * v3;
}

void ADDV(int w,Scalar m,Vector* vp,Edge* edges,Vector* vd) {
 Edge& e = edges[w];
 if(e.type == NONE) {
  vd[w] = (1 - m) * e.v[0] + (m) * e.v[1];
 } else if(e.type == ARC) {
  vd[w] = rotate(e.v[0] - e.v[3],e.N,e.theta * m) + e.v[3];
 } else if(e.type == COSINE) {
  vd[w] = (1 - m) * e.v[0] + (m) * e.v[1] +
   pow(cos(3.1416 * (m - 0.5)),2) * e.N;
 } else if(e.type == QUAD) {
  vd[w] = (1 - m) * e.v[0] + (m) * e.v[1] +
   (4 * m * (1 - m)) * e.N;
 }
```

```
}
void hexMesh(Int* n,Scalar* s,Int* type,Vector* vp,Edge* edges,MeshObject& mo) {
 Int i,j,k,m;

 /*for wall division set twice
 number of divisions requested*/
 for(j = 0;j < 3;j++) {
  bool found = false;
  for(i = j;i < 12;i+=3) {
   if(type[i] == WALL) {
    if((n[j] % 2) && (n[j] != 1)) {
     found = true;
     break;
    }
   }
  }
  if(found) {
   n[j]++;
   for(i = j;i < 12;i+=3) {
    s[i] = 1 / s[i];
   }
  }
 }

 /*calculate scale*/
 Scalar* sc[12];
 for(i = 0;i < 12;i++) {
  Int nt = n[i / 4];
  sc[i] = new Scalar[nt + 1];
  if(type[i] == WALL)
   s[i] = pow(s[i],Scalar(1./(nt / 2.)));
  else
   s[i] = pow(s[i],Scalar(1./nt));
 }
 for(i = 0;i < 12;i++) {
  Int nt = n[i / 4];
  Scalar r = s[i];
  if(nt == 1) {
   sc[i][0] = 0;
   sc[i][1] = 1;
  } else {
```

```
2777   if(type[i] == WALL)
2778    nt /= 2;
2779   for(j = 0;j <= nt;j++) {
2780    if(equal(r,Scalar(1)))
2781     sc[i][j] = Scalar(j) / (nt);
2782    else
2783     sc[i][j] = (1 - pow(r,Scalar(j))) / (1 - pow(s[i],Scalar(nt)));
2784   }
2785   if(type[i] == WALL) {
2786    for(j = 0;j <= nt;j++)
2787     sc[i][j] /= 2;
2788    for(j = 0;j <= nt;j++)
2789     sc[i][j + nt] = Scalar(1.0) - sc[i][nt - j];
2790   }
2791  }
2792 }
2793 for(i = 0;i < 12;i++) {
2794  Edge& e = edges[i];
2795  if(e.type == ARC) {
2796   Vector C = center(e.v[0],e.v[1],e.v[2]);
2797   Vector r1 = e.v[0] - C;
2798   Vector r2 = e.v[1] - C;
2799   e.theta = acos((r1 & r2) / (mag(r1) * mag(r2)));
2800   e.v[3] = C;
2801   e.N = (e.v[2] - e.v[0]) ^ (e.v[1] - e.v[0]);
2802   e.N = unit(e.N);
2803  } else if(e.type == COSINE || e.type == QUAD) {
2804   Vector mid = (e.v[1] + e.v[0]) / 2;
2805   e.N = e.v[2] - mid;
2806   e.L = mag(mid - e.v[0]) / 2;
2807  }
2808 }
2809    /*variables*/
2810 Int nx = n[0] + 1 , ny = n[1] + 1 , nz = n[2] + 1;
2811 const Int B1 = (nx - 0) * (ny - 1) * (nz - 1);
2812 const Int B2 = (nx - 1) * (ny - 0) * (nz - 1);
2813 const Int B3 = (nx - 1) * (ny - 1) * (nz - 0);
2814 IntVector VI(nx * ny * nz,0);
2815 IntVector FI(B1 + B2 + B3, 0);
2816
2817 /*vertices*/
```

```
2818  Vertex v,v1,v2,vd[12],vf[6];
2819  Scalar rx,ry,rz;
2820
2821 #define I0(i,j,k) (i * ny * nz + j * nz + k)
2822
2823 #define ADDF(w,rr,rs,i00,i01,i10,i11,ir0,ir1,i0s,i1s) { \
2824  vf[w] = Interpolate_face(       \
2825    rr,rs,          \
2826    vp[i00],vp[i01],vp[i10],vp[i11],   \
2827    vd[ir0],vd[ir1],vd[i0s],vd[i1s]);   \
2828 }
2829
2830 #define ADDC() {          \
2831  v = Interpolate_cell(        \
2832    rx,ry,rz,         \
2833    vp[0],vp[4],vp[3],vp[7],     \
2834    vp[1],vp[5],vp[2],vp[6],     \
2835    vd[0],vd[3],vd[1],vd[2],     \
2836    vd[4],vd[7],vd[5],vd[6],     \
2837    vd[8],vd[11],vd[9],vd[10],     \
2838    vf[4],vf[5],vf[2],vf[3],vf[0],vf[1]);  \
2839 }
2840
2841 #define ADD() {                              \
2842  ADDV(0,sc[0][i],vp,edges,vd);     \
2843  ADDV(1,sc[1][i],vp,edges,vd);     \
2844  ADDV(2,sc[2][i],vp,edges,vd);     \
2845  ADDV(3,sc[3][i],vp,edges,vd);     \
2846  ADDV(4,sc[4][j],vp,edges,vd);     \
2847  ADDV(5,sc[5][j],vp,edges,vd);     \
2848  ADDV(6,sc[6][j],vp,edges,vd);     \
2849  ADDV(7,sc[7][j],vp,edges,vd);     \
2850  ADDV(8,sc[8][k],vp,edges,vd);     \
2851  ADDV(9,sc[9][k],vp,edges,vd);     \
2852  ADDV(10,sc[10][k],vp,edges,vd);     \
2853  ADDV(11,sc[11][k],vp,edges,vd);     \
2854  rx = i / Scalar(nx - 1);      \
2855  ry = j / Scalar(ny - 1);      \
2856  rz = k / Scalar(nz - 1);      \
2857  ADDF(0, rx,ry, 0,3,1,2, 0,1,4,5);    \
2858  ADDF(1, rx,ry, 4,7,5,6, 3,2,7,6);    \
```

```
 ADDF(2, rx,rz, 0,4,1,5, 0,3,8,9);    \
 ADDF(3, rx,rz, 3,7,2,6, 1,2,11,10);    \
 ADDF(4, ry,rz, 0,4,3,7, 4,7,8,11);    \
 ADDF(5, ry,rz, 1,5,2,6, 5,6,9,10);    \
 ADDC();           \
};

 /*interior*/
 for(j = 1;j < ny - 1;j++) {
  for(i = 1;i < nx - 1;i++) {
   for(k = 1;k < nz - 1;k++) {
    ADD();
    mo.v.push_back(v);
    VI[I0(i,j,k)] = mo.v.size() - 1;
   }
  }
 }
 mo.nv = mo.v.size();

 /*boundaries*/
 for(i = 0;i < nx; i += (nx - 1)) {
  for(j = 0;j < ny;j++) {
   for(k = 0;k < nz;k++) {
    ADD();
    mo.v.push_back(v);
    VI[I0(i,j,k)] = mo.v.size() - 1;
   }
  }
 }
 for(j = 0;j < ny; j += (ny - 1)) {
  for(i = 1;i < nx - 1;i++) {
   for(k = 0;k < nz;k++) {
    ADD();
    mo.v.push_back(v);
    VI[I0(i,j,k)] = mo.v.size() - 1;
   }
  }
 }
 for(k = 0;k < nz; k += (nz - 1)) {
  for(i = 1;i < nx - 1;i++) {
   for(j = 1;j < ny - 1;j++) {
```

```
    ADD();
    mo.v.push_back(v);
    VI[I0(i,j,k)] = mo.v.size() - 1;
   }
  }
 }
 /*end*/
#undef ADD
#undef ADDF
#undef ADDE

 delete[] sc[0];
 delete[] sc[1];
 delete[] sc[2];

 /*faces*/
#define I1(i,j,k)  (i * (ny - 1) * (nz - 1) + j * (nz - 1) + k)
#define I2(i,j,k)  (i * (ny - 0) * (nz - 1) + j * (nz - 1) + k + B1)
#define I3(i,j,k)  (i * (ny - 1) * (nz - 0) + j * (nz - 0) + k + B1 + B2)

#define ADD(a1,a2,a3,a4) {       \
 Facet f;          \
 m = I0(i,j,k);         \
 f.push_back(VI[a1]);       \
 f.push_back(VI[a2]);       \
 f.push_back(VI[a3]);       \
 f.push_back(VI[a4]);       \
 mo.f.push_back(f);        \
};

 /*interior*/
 for(i = 1;i < nx - 1;i++) {
  for(j = 0;j < ny - 1;j++) {
   for(k = 0;k < nz - 1;k++) {
    ADD(m,m + nz,m + nz + 1,m + 1);
    FI[I1(i,j,k)] = mo.f.size() - 1;
   }
  }
 }
 for(i = 0;i < nx - 1;i++) {
  for(j = 1;j < ny - 1;j++) {
```

```
2941    for(k = 0;k < nz - 1;k++) {
2942     ADD(m,m + 1,m + ny * nz + 1,m + ny * nz);
2943     FI[I2(i,j,k)] = mo.f.size() - 1;
2944    }
2945   }
2946  }
2947  for(i = 0;i < nx - 1;i++) {
2948   for(j = 0;j < ny - 1;j++) {
2949    for(k = 1;k < nz - 1;k++) {
2950     ADD(m, m + ny * nz,m + ny * nz + nz, m + nz);
2951     FI[I3(i,j,k)] = mo.f.size() - 1;
2952    }
2953   }
2954  }
2955  mo.nf = mo.f.size();
2956  /*boundaries*/
2957  for(i = 0;i < nx; i += (nx - 1)) {
2958   for(j = 0;j < ny - 1;j++) {
2959    for(k = 0;k < nz - 1;k++) {
2960     ADD(m,m + nz,m + nz + 1,m + 1);
2961     FI[I1(i,j,k)] = mo.f.size() - 1;
2962    }
2963   }
2964  }
2965  for(j = 0;j < ny;j += (ny - 1)) {
2966   for(i = 0;i < nx - 1;i++) {
2967    for(k = 0;k < nz - 1;k++) {
2968     ADD(m,m + 1,m + ny * nz + 1,m + ny * nz);
2969     FI[I2(i,j,k)] = mo.f.size() - 1;
2970    }
2971   }
2972  }
2973  for(k = 0;k < nz; k += (nz - 1)) {
2974   for(i = 0;i < nx - 1;i++) {
2975    for(j = 0;j < ny - 1;j++) {
2976     ADD(m, m + ny * nz,m + ny * nz + nz, m + nz);
2977     FI[I3(i,j,k)] = mo.f.size() - 1;
2978    }
2979   }
2980  }
2981  /*end*/
```

```
2982 #undef ADD
2983
2984  /*cells*/
2985  for(i = 0;i < nx - 1;i++) {
2986   for(j = 0;j < ny - 1;j++) {
2987    for(k = 0;k < nz - 1;k++) {
2988     Cell c;
2989     m = I1(i,j,k);
2990     c.push_back(FI[m]);
2991     c.push_back(FI[m + (ny - 1) * (nz - 1)]);
2992
2993     m = I2(i,j,k);;
2994     c.push_back(FI[m]);
2995     c.push_back(FI[m + (nz - 1)]);
2996
2997     m = I3(i,j,k);;
2998     c.push_back(FI[m]);
2999     c.push_back(FI[m + 1]);
3000
3001     mo.c.push_back(c);
3002    }
3003   }
3004  }
3005  mo.nc = mo.c.size();
3006 #undef I0
3007 #undef I1
3008 #undef I2
3009 #undef I3
3010  /*remove duplicates*/
3011  int deformed = 0;
3012  for(i = 0;i < 8;i++) {
3013   for(j = i + 1;j < 8;j++) {
3014    if(equal(vp[i],vp[j])) {
3015     deformed = 1;
3016     break;
3017    }
3018   }
3019  }
3020  if(deformed)
3021   remove_duplicate(mo);
3022  /*end*/
```

```
3023 }
3024
3025 /*remove duplicate*/
3026 void remove_duplicate(Mesh::MeshObject& p) {
3027  Int i,j,sz,corr;
3028  int count;
3029  /*vertices*/
3030  sz = p.v.size();
3031  corr = 0;
3032  std::vector<int> dup(sz,0);
3033  for(i = 0;i < sz;i++) {
3034   for(j = sz - 1;j >= i + 1;j--) {
3035    if(equal(p.v[i],p.v[j])) {
3036     dup[i] = -int(j);
3037     if(i < p.nv) corr++;
3038     break;
3039    }
3040   }
3041  }
3042  p.nv -= corr;
3043  //remove duplicate vertices
3044  {
3045   Vertices vt(p.v.begin(), p.v.end());
3046   p.v.clear();
3047   count = 0;
3048   for(i = 0;i < sz;i++) {
3049    if(!dup[i]) {
3050     p.v.push_back(vt[i]);
3051        dup[i] = count++;
3052    }
3053   }
3054   for(i = 0;i < sz;i++) {
3055    if(dup[i] < 0)
3056     dup[i] = dup[-dup[i]];
3057   }
3058  }
3059  /*faces*/
3060  sz = p.f.size();
3061  for(i = 0;i < sz;i++) {
3062   Facet& f = p.f[i];
3063   forEach(f,j)
```

```
3064   f[j] = dup[f[j]];
3065  }
3066  dup.clear();
3067  dup.assign(sz,0);
3068  count = 0;
3069  corr = 0;
3070  for(i = 0;i < sz;i++) {
3071   Facet& f = p.f[i];
3072   forEach(f,j) {
3073    forEachS(f,k,j+1) {
3074     if(f[j] == f[k]) {
3075      f.erase(f.begin() + k);
3076      k--;
3077     }
3078    }
3079   }
3080   if(f.size() < 3) {
3081    dup[i] = -1;
3082    if(i < p.nf) corr++;
3083   } else {
3084    dup[i] = count;
3085    count++;
3086   }
3087  }
3088  p.nf -= corr;
3089  //remove deformed faces
3090  {
3091   Facets ft(p.f.begin(), p.f.end());
3092   p.f.clear();
3093   for(i = 0;i < sz;i++) {
3094    if(dup[i] >= 0) p.f.push_back(ft[i]);
3095   }
3096  }
3097  /*cells*/
3098  sz = p.c.size();
3099  for(i = 0;i < sz;i++) {
3100   Cell& c = p.c[i];
3101   forEach(c,j) {
3102    if(dup[c[j]] < 0) {
3103     c.erase(c.begin() + j);
3104     j--;
```

```
3105    } else
3106     c[j] = dup[c[j]];
3107   }
3108  }
3109 }
3110 /*Merge meshes*/
3111 #define MAXNUM 1073741824
3112
3113 void merge(MeshObject& m1,MergeObject& b,MeshObject& m2) {
3114  Int i,j,found,s0,s1,s2,s3;
3115
3116     //vertices
3117  {
3118   s0 = m1.v.size();
3119   s1 = m2.nv;
3120   s2 = m2.v.size();
3121   s3 = b.vb.size();
3122   m1.v.insert(m1.v.end(),m2.v.begin(),m2.v.begin() + s1);
3123
3124   IntVector locv(s2 - s1,MAXNUM);
3125   for(i = s1;i < s2;i++) {
3126    found = 0;
3127    for(j = 0;j < s3;j++) {
3128     if(equal(m2.v[i],b.vb[j])) {
3129      locv[i - s1] += j;
3130      found = 1;
3131      break;
3132     }
3133    }
3134    if(!found) {
3135     b.vb.push_back(m2.v[i]);
3136     locv[i - s1] += b.vb.size() - 1;
3137    }
3138   }
3139   forEach(m2.f,i) {
3140    Facet& ft = m2.f[i];
3141    forEach(ft,j) {
3142     if(ft[j] >= s1) {
3143      ft[j] = locv[ft[j] - s1];
3144     } else {
3145      ft[j] += s0;
```

```
3146     }
3147    }
3148   }
3149  }
3150  //faces
3151  {
3152   s0 = m1.f.size();
3153   s1 = m2.nf;
3154   s2 = m2.f.size();
3155   s3 = b.fb.size();
3156   m1.f.insert(m1.f.end(),m2.f.begin(),m2.f.begin() + s1);
3157
3158   IntVector index0(s3,0),index1(s2 - s1,0);
3159   Int count = 0;
3160   b.fb.reserve(s3 + s2 - s1);
3161   for(j = 0;j < s3;j++) {
3162    found = 0;
3163    for(i = s1;i < s2;i++) {
3164     if(!index1[i - s1] && equal(m2.f[i],b.fb[j])) {
3165
3166      m1.f.push_back(b.fb[j]);
3167      index0[j]    = m1.f.size() - 1;
3168      index1[i - s1] = m1.f.size() - 1;
3169
3170      found = 1;
3171      break;
3172     }
3173    }
3174    if(!found) {
3175     index0[j] = MAXNUM + count;
3176     b.fb[count] = b.fb[j];
3177     count++;
3178    }
3179   }
3180   for(i = s1;i < s2;i++) {
3181    if(!index1[i - s1]) {
3182     index1[i - s1] = MAXNUM + count;
3183
3184     if(count >= s3) b.fb.push_back(m2.f[i]);
3185     else b.fb[count] = m2.f[i];
3186     count++;
```

```
3187    }
3188   }
3189   b.fb.resize(count);
3190
3191   forEach(m1.c,i) {
3192    Cell& ct = m1.c[i];
3193    forEach(ct,j) {
3194     if(ct[j] >= MAXNUM) {
3195      ct[j] = index0[ct[j] - MAXNUM];
3196     }
3197    }
3198   }
3199   forEach(m2.c,i) {
3200    Cell& ct = m2.c[i];
3201    forEach(ct,j) {
3202     if(ct[j] >= s1) {
3203      ct[j] = index1[ct[j] - s1];
3204     } else {
3205      ct[j] += s0;
3206     }
3207    }
3208   }
3209  }
3210  //cells
3211  {
3212   m1.c.insert(m1.c.end(),m2.c.begin(),m2.c.end());
3213  }
3214 }
3215 void merge(Mesh::MeshObject& m,MergeObject& b) {
3216  m.nv = m.v.size();
3217  m.nf = m.f.size();
3218  m.nc = m.c.size();
3219
3220  m.v.insert(m.v.end(),b.vb.begin(),b.vb.end());
3221  m.f.insert(m.f.end(),b.fb.begin(),b.fb.end());
3222  forEach(m.f,i) {
3223   Facet& ft = m.f[i];
3224   forEach(ft,j) {
3225    if(ft[j] >= MAXNUM) {
3226     ft[j] -= MAXNUM;
3227     ft[j] += m.nv;
```

```
   }
  }
 }
 forEach(m.c,i) {
  Cell& ct = m.c[i];
  forEach(ct,j) {
   if(ct[j] >= MAXNUM) {
    ct[j] -= MAXNUM;
    ct[j] += m.nf;
   }
  }
 }
}

#undef MAXNUM
#include <cstring>
#include "mesh.h"
#include "hexMesh.h"
#include "mshMesh.h"

using namespace std;

/*boundary*/
struct Bdry {
 string name;
 IntVector index;
 /*point in polygon*
 int pnpoly(Vertices keys,Vertex C) {
  Vector ki,kj;
  int i, j, nvert = index.size(), c = 0;
  for (i = 0, j = nvert-1; i < nvert; j = i++) {
   ki = keys[index[i]];
   kj = keys[index[j]];
   if ( ((ki[1]>C[1]) != (kj[1]>C[1])) &&
    (C[0] < (kj[0]-ki[0]) * (C[1]-ki[1]) /
    (kj[1]-ki[1]) + ki[0]) )
    c = !c;
  }
  return c;
 }*/
};
```

```
3269
3270 /*generate mesh*/
3271 int main(int argc,char* argv[]) {
3272  using namespace Mesh;
3273  using namespace Util;
3274  Vertices keys;
3275  vector<Bdry> Bdrys;
3276  MergeObject bMerge;
3277  string str;
3278  string default_name;
3279  char* i_file_name = argv[1];
3280  char* e_file_name = 0;
3281  bool Import = false;
3282  bool Export = false;
3283  char c;
3284
3285  /*command line arguments*/
3286  for(int i = 1;i < argc;i++) {
3287   if(!strcmp(argv[i],"-i")) {
3288    i++;
3289    Import = true;
3290    i_file_name = argv[i];
3291   } else if(!strcmp(argv[i],"-o")) {
3292    i++;
3293    Export = true;
3294    e_file_name = argv[i];
3295   }
3296  }
3297
3298  /*export to msh file format*/
3299  if(Export) {
3300   ofstream output(e_file_name);
3301   if(Import) str = i_file_name;
3302   else str = "grid";
3303   Mesh::gMeshName = str;
3304   Mesh::readMesh();
3305   Mesh::addBoundaryCells();
3306   Mesh::calcGeometry();
3307
3308   output << hex;
3309   writeMshMesh(output,gMesh);
```

```
3310  output << dec;
3311  return 0;
3312 }
3313
3314 /*input stream*/
3315 ifstream input(i_file_name);
3316
3317 /*import*/
3318 if(Import) {
3319  input >> hex;
3320  readMshMesh(input,gMesh);
3321  input >> dec;
3322
3323  gMesh.write(cout);
3324  return 0;
3325 }
3326
3327 /*read key points*/
3328 if(Util::nextc(input))
3329  input >> keys;
3330
3331 while((c = Util::nextc(input)) != 0) {
3332  char symbol;
3333  if(isdigit(c)) {
3334   /*read indices to keys*/
3335   IntVector index;
3336   input >> index;
3337
3338   Vertices v(index.size(),Vector(0));
3339   forEach(v,i)
3340    v[i] = keys[index[i]];
3341
3342   IntVector n;
3343   Int type;
3344   vector<Scalar> s(12,Scalar(1));
3345   vector<Int> t(12);
3346
3347   input >> str;
3348   if(!compare(str,"linear")) {
3349    input >> n;
3350    type = LINEAR;
```

```
3351    t.assign(12,type);
3352   } else {
3353    if(!compare(str,"geometric")) type = GEOMETRIC;
3354    else if(!compare(str,"wall")) type = WALL;
3355    else if(!compare(str,"mixed")) type = MIXED;
3356    else return 1;
3357
3358    //read divisions
3359    vector<Scalar> ts(s);
3360    vector<Int> tt(t);
3361
3362    Int sz;
3363    input >> n;
3364    input >> sz >> symbol;
3365
3366    if(type == MIXED) {
3367     for(Int i = 0;i < sz ;i++) {
3368      input >> symbol;
3369      switch(symbol) {
3370       case 'l':
3371       case 'L':
3372        type = LINEAR;
3373        break;
3374       case 'g':
3375       case 'G':
3376        type = GEOMETRIC;
3377        break;
3378       case 'w':
3379       case 'W':
3380        type = WALL;
3381        break;
3382      }
3383      tt[i] = type;
3384      input >> ts[i];
3385     }
3386    } else {
3387     for(Int i = 0;i < sz ;i++)
3388      input >> ts[i];
3389     tt.assign(12,type);
3390    }
3391
```

```
3392     input >> symbol;
3393
3394     //assign to each side
3395     Int r = 12 / sz;
3396     for(Int i = 0;i < sz;i++) {
3397      for(Int j = 0;j < r;j++) {
3398       if(i * r + j < 12) {
3399        s[i * r + j] = ts[i];
3400        t[i * r + j] = tt[i];
3401       }
3402      }
3403     }
3404    }
3405
3406    //curved edges
3407    static const int sides[12][2] = {
3408     {0,1}, {3,2}, {7,6}, {4,5},
3409     {0,3}, {1,2}, {5,6}, {4,7},
3410     {0,4}, {1,5}, {2,6}, {3,7}
3411    };
3412    vector<Edge> edges(12);
3413    for(Int i = 0;i < 12;i++) {
3414     edges[i].v[0] = v[sides[i][0]];
3415     edges[i].v[1] = v[sides[i][1]];
3416    }
3417
3418    if((c = Util::nextc(input)) && (c == 'e')) {
3419     Int sz,side,key;
3420     input >> str;
3421     if(!compare(str,"edges")) {
3422      input >> sz >> symbol;
3423      for(Int i = 0;i < sz;i++) {
3424       input >> str >> side >> key;
3425       Edge& e = edges[side];
3426       e.v[2] = keys[key];
3427       if(!compare(str,"arc")) {
3428        e.type = ARC;
3429       } else if(!compare(str,"cosine")) {
3430        e.type = COSINE;
3431       } else if(!compare(str,"quad")) {
3432        e.type = QUAD;
```

```
3433      } else {
3434       e.type = NONE;
3435      }
3436     }
3437     input >> symbol;
3438    } else {
3439     Bdry b;
3440     b.name = str;
3441     while((c = Util::nextc(input)) && isdigit(c)) {
3442      input >> b.index;
3443      Bdrys.push_back(b);
3444     }
3445    }
3446   }
3447
3448   //generate mesh
3449   MeshObject mo;
3450   hexMesh(&n[0],&s[0],&t[0],&v[0],&edges[0],mo);
3451   merge(gMesh,bMerge,mo);
3452  } else {
3453   /*read boundaries*/
3454   Bdry b;
3455   input >> b.name;
3456   if(b.name == "default") {
3457    input >> default_name;
3458   } else {
3459    while((c = Util::nextc(input)) && isdigit(c)) {
3460     input >> b.index;
3461     Bdrys.push_back(b);
3462    }
3463   }
3464  }
3465 }
3466 /*merge boundary & internals*/
3467 merge(gMesh,bMerge);
3468
3469 /*boundaries*/
3470 forEach(Bdrys,i) {
3471  IntVector list;
3472  IntVector& b = Bdrys[i].index;
3473  Vector N = (keys[b[1]] - keys[b[0]]) ^ (keys[b[2]] - keys[b[0]]);
```

```
3474  N /= mag(N);
3475  forEachS(gFacets,j,gMesh.nf) {
3476   Facet& f = gFacets[j];
3477   Vector N1 = ((gVertices[f[1]] - gVertices[f[0]])
3478    ^ (gVertices[f[2]] - gVertices[f[0]]));
3479   N1 /= mag(N1);
3480   Vector H = (gVertices[f[0]] - keys[b[0]]);
3481   Scalar d = mag(N ^ N1);
3482   Scalar d2 = sqrt(mag(N & H));
3483   if(d <= 10e-4 && d2 <= 10e-4) {
3484    /*
3485    Vector C(0);
3486    forEach(f,m)
3487     C += gVertices[f[m]];
3488    C /= Scalar(f.size());
3489    if(Bdrys[i].pnpoly(keys,C))
3490     */
3491     list.push_back(j);
3492   }
3493  }
3494  if(!list.empty()) {
3495   IntVector& gB = gBoundaries[Bdrys[i].name.c_str()];
3496   IntVector::iterator it = find(gB.begin(),gB.end(),list[0]);
3497   if(it == gB.end()) {
3498    forEach(list,j)
3499     gB.push_back(list[j]);
3500   }
3501  }
3502 }
3503 /*default specified*/
3504 if(!default_name.empty()) {
3505  IntVector& gB = gBoundaries[default_name.c_str()];
3506  forEachS(gFacets,i,gMesh.nf) {
3507   if(!faceInBoundary(i)) {
3508    gB.push_back(i);
3509   }
3510  }
3511 }
3512 /*write it*/
3513 gMesh.write(cout);
3514 return 0;
```

```
3515 }
3516 #include "mesh.h"
3517
3518 using namespace std;
3519
3520 /*global mesh*/
3521 namespace Mesh {
3522  MeshObject     gMesh;
3523  std::string&   gMeshName = gMesh.name;
3524  Vertices&      gVertices = gMesh.v;
3525  Facets&        gFacets  = gMesh.f;
3526  Cells&         gCells   = gMesh.c;
3527  Boundaries&    gBoundaries = gMesh.bdry;
3528  IntVector&     gFO = gMesh.fo;
3529  IntVector&     gFN = gMesh.fn;
3530  Int&           gBCellsStart = gMesh.nc;
3531  vector<BasicBCondition*> AllBConditions;
3532  std::vector<interBoundary>& gInterMesh = gMesh.interMesh;
3533  Vertices       probePoints;
3534  IntVector      probeCells;
3535
3536  std::vector<Vector> _fC;
3537  std::vector<Vector> _cC;
3538  std::vector<Vector> _fN;
3539  std::vector<Scalar> _cV;
3540  std::vector<bool> _reversed;
3541 }
3542
3543 /*read mesh*/
3544 void Mesh::readMesh() {
3545  cout << "Reading mesh :" << endl;
3546  ifstream is(gMeshName.c_str());
3547  is >> hex;
3548  is >> gVertices;
3549  cout << " \t" << gVertices.size() << " vertices" << endl;
3550  is >> gFacets;
3551  cout << " \t" << gFacets.size() << " facets" << endl;
3552  is >> gCells;
3553  cout << " \t" << gCells.size() << " cells" << endl;
3554  cout << "Boundaries :" << endl;
3555  while(Util::nextc(is)) {
```

```
3556   IntVector index;
3557   string str;
3558   is >> str;
3559   cout << " \t" << str << endl;
3560   is >> index;
3561
3562   IntVector& gB = gBoundaries[str];
3563   gB.insert(gB.begin(),index.begin(),index.end());
3564
3565   /*internal mesh boundaries*/
3566   if(str.find("interMesh") != std::string::npos) {
3567    interBoundary b;
3568    sscanf(str.c_str(), "interMesh_%x_%x", &b.from,&b.to);
3569    b.f   = &gBoundaries[str];
3570    gInterMesh.push_back(b);
3571   }
3572  }
3573  /*start of buffer*/
3574  Int buffer_index = 0;
3575  forEach(gInterMesh,i) {
3576   interBoundary& b = gInterMesh[i];
3577   b.buffer_index = buffer_index;
3578   buffer_index += b.f->size();
3579  }
3580  is >> dec;
3581 }
3582 /*write mesh*/
3583 void Mesh::MeshObject::write(ostream& os) {
3584  os << hex;
3585  os.precision(12);
3586  os << v;
3587  os.precision(6);
3588  os << f;
3589  os << c;
3590  forEachIt(Boundaries,bdry,it)
3591   os << it->first << " " << it->second << endl;
3592  os << dec;
3593 }
3594 /*Is face in boundary*/
3595 bool Mesh::faceInBoundary(Int f) {
3596  forEachIt(Boundaries,gBoundaries,it) {
```

```
3597   IntVector& gB = it->second;
3598   forEach(gB,j) {
3599    if(gB[j] == f)
3600     return true;
3601   }
3602  }
3603  return false;
3604 }
3605 /*add boundary cells*/
3606 void Mesh::addBoundaryCells() {
3607  using namespace Constants;
3608  Int i,index;
3609 
3610  /*neighbor and owner cells of face*/
3611  gBCellsStart = gCells.size();
3612  gFO.assign(gFacets.size(),MAX_INT);
3613  gFN.assign(gFacets.size(),MAX_INT);
3614  for(i = 0;i < gBCellsStart;i++) {
3615   forEach(gCells[i],j) {
3616    index = gCells[i][j];
3617    if(gFO[index] == MAX_INT)
3618     gFO[index] = i;
3619    else
3620     gFN[index] = i;
3621   }
3622  }
3623  /*Flag boundary faces not in gBoundaries for auto deletion*/
3624  IntVector& gDelete = gBoundaries["delete"];
3625  forEach(gFN,i) {
3626   if(gFN[i] == MAX_INT) {
3627    if(!faceInBoundary(i))
3628     gDelete.push_back(i);
3629   }
3630  }
3631  /*add boundary cells*/
3632  forEachIt(Boundaries,gBoundaries,it) {
3633   IntVector& facets = it->second;
3634   forEach(facets,j) {
3635    i = facets[j];
3636    /*external patch*/
3637    if(gFN[i] == MAX_INT) {
```

```
3638     Cell c;
3639     c.push_back(i);
3640     gCells.push_back(c);
3641     gFN[i] = gCells.size() - 1;
3642    }
3643   }
3644  }
3645 }
3646 void Mesh::calcGeometry() {
3647  Int i;
3648
3649  /*allocate*/
3650  _fC.assign(gFacets.size(),Vector(0));
3651  _cC.assign(gCells.size(),Vector(0));
3652  _fN.assign(gFacets.size(),Vector(0));
3653  _cV.assign(gCells.size(),Scalar(0));
3654  _reversed.assign(gFacets.size(),false);
3655
3656  /* face centre*/
3657  forEach(gFacets,i) {
3658   Facet& f = gFacets[i];
3659   Vector C(0);
3660   forEach(f,j)
3661    C += gVertices[f[j]];
3662   _fC[i] = C / Scalar(f.size());
3663  }
3664
3665  /* cell centre */
3666  forEach(gCells,i) {
3667   Cell& c = gCells[i];
3668   Vector C(0);
3669   forEach(c,j)
3670    C += _fC[c[j]];
3671   _cC[i] = C / Scalar(c.size());
3672  }
3673  /* face normal */
3674  Vector v1,v2,v3,v;
3675  Scalar magN;
3676  forEach(gFacets,i) {
3677   Facet& f = gFacets[i];
3678   Vector N(0),C(0),Ni;
```

```
 Scalar Ntot = Scalar(0);
 v1 = _fC[i];
 forEach(f,j) {
  v2 = gVertices[f[j]];
  if(j + 1 == f.size())
   v3 = gVertices[f[0]];
  else
   v3 = gVertices[f[j + 1]];
  Ni = ((v2 - v1) ^ (v3 - v1));
  magN = mag(Ni);
  C += magN * ((v1 + v2 + v3) / 3);
  Ntot += magN;
  N += Ni;
 }
 _fC[i] = C / Ntot; /*corrected face centre*/
 v = _fC[i] - _cC[gFO[i]];
 if((v & N) < 0) {
  N = -N;
  _reversed[i] = true;
 }
 _fN[i] = N / Scalar(2);
}
/* cell volumes */
for(i = 0;i < gBCellsStart;i++) {
 Cell& c = gCells[i];
 Scalar V(0),Vi;
 Vector v = _cC[i],C(0);
 forEach(c,j) {
  v = _cC[i] - _fC[c[j]];
  Vi = mag(v & _fN[c[j]]);
  C += Vi * (2 * _fC[c[j]] + _cC[i]) / 3;
  V += Vi;
 }
 _cC[i] = C / V;      /*corrected cell centre */
 _cV[i] = V / Scalar(3);
}
/*boundary cell centre and volume*/
forEachS(gCells,i,gBCellsStart) {
 _cV[i] = _cV[gFO[gCells[i][0]]];
 _cC[i] = _fC[gCells[i][0]];
}
```

```
3720 }
3721 /*
3722 * Remove empty boundary
3723 */
3724 void Mesh::removeBoundary(IntVector& fs) {
3725  cout << "Removing faces: " << fs.size() << endl;
3726 
3727  Int count;
3728  IntVector Idf(gFacets.size(),0);
3729  IntVector Idc(gCells.size(),0);
3730 
3731  /*erase facet reference*/
3732  forEach(fs,i) {
3733   Int f = fs[i];
3734   Cell& co = gCells[gFO[f]];
3735   forEach(co,j) {
3736    if(co[j] == f) {
3737     co.erase(co.begin() + j);
3738     break;
3739    }
3740   }
3741   Cell& cn = gCells[gFN[f]];
3742   forEach(cn,j) {
3743    if(cn[j] == f) {
3744     cn.erase(cn.begin() + j);
3745     break;
3746    }
3747   }
3748  }
3749  /*updated facet id*/
3750  forEach(fs,i)
3751   Idf[fs[i]] = Constants::MAX_INT;
3752  count = 0;
3753  forEach(gFacets,i) {
3754   if(Idf[i] != Constants::MAX_INT)
3755    Idf[i] = count++;
3756   else
3757    gFacets[i].clear();
3758  }
3759  /*erase facets*/
3760  forEach(gFacets,i) {
```

```
 if(gFacets[i].size() == 0) {
  gFacets.erase(gFacets.begin() + i);
  gFO.erase(gFO.begin() + i);
  gFN.erase(gFN.begin() + i);
  _fC.erase(_fC.begin() + i);
  _fN.erase(_fN.begin() + i);
  --i;
 }
}
/*updated facet id*/
count = 0;
forEach(gCells,i) {
 if(gCells[i].size() != 0)
  Idc[i] = count++;
 else
  Idc[i] = Constants::MAX_INT;
}
/*erase cells*/
forEach(gCells,i) {
 if(gCells[i].size() == 0) {
  gCells.erase(gCells.begin() + i);
  _cC.erase(_cC.begin() + i);
  _cV.erase(_cV.begin() + i);
  --i;
 } else {
  forEach(gCells[i],j) {
   gCells[i][j] = Idf[gCells[i][j]];
  }
 }
}
/*facet owner and neighbor*/
forEach(gFacets,i) {
 gFO[i] = Idc[gFO[i]];
 gFN[i] = Idc[gFN[i]];
}
/*patches*/
forEachIt(Boundaries,gBoundaries,it) {
 IntVector& gB = it->second;
 forEach(gB,i)
  gB[i] = Idf[gB[i]];
}
```

```
3802
3803  cout << "Total faces: " << gFacets.size() << endl;
3804 }
3805 /*find nearest cell*/
3806 Int Mesh::findNearestCell(const Vector& v) {
3807  Scalar mindist,dist;
3808  Int bi = 0;
3809  mindist = mag(v - _cC[0]);
3810  for(Int i = 0;i < gBCellsStart;i++) {
3811   dist = mag(v - _cC[i]);
3812   if(dist < mindist) {
3813    mindist = dist;
3814    bi = i;
3815   }
3816  }
3817  return bi;
3818 }
3819 Int Mesh::findNearestFace(const Vector& v) {
3820  Scalar mindist,dist;
3821  Int bi = 0;
3822  mindist = mag(v - _fC[0]);
3823  forEach(gFacets,i) {
3824   dist = mag(v - _fC[i]);
3825   if(dist < mindist) {
3826    mindist = dist;
3827    bi = i;
3828   }
3829  }
3830  return bi;
3831 }
3832 void Mesh::getProbeCells(IntVector& probes) {
3833  forEach(probePoints,j) {
3834   Vector v = probePoints[j];
3835   Int index = findNearestCell(v);
3836   probes.push_back(index);
3837  }
3838 }
3839 void Mesh::getProbeFaces(IntVector& probes) {
3840  forEach(probePoints,j) {
3841   Vector v = probePoints[j];
3842   Int index = findNearestFace(v);
```

```
  probes.push_back(index);
 }
}
#include "mshMesh.h"
#include <sstream>

using namespace std;

void readMshMesh(std::istream& is,Mesh::MeshObject& mo) {
 char symbol,c;
 int id,ND,zone,findex,lindex,
  type,bctype,ftype,etype,
  node_start = 0,facet_start = 0;
 map<int,string> bnames;

 /*read id*/
 while((c = Util::nextc(is)) != 0) {
  int braces = 1;
  is >> symbol >> id;
  switch(id) {
  case 0x0:
   do{ is >> c; } while(c != ')');
   break;
  case 0x2:
   is >> ND >> symbol;
   break;
  case 0x10:
   is >> symbol >> zone;
   is >> findex >> lindex;
   is >> type >> ND;
   is >> symbol >> symbol;
   if(zone != 0) {
    Vertex v;
    for(int i = findex;i <= lindex;i++) {
     is >> v;
     mo.v.push_back(v);
    }
    is >> symbol >> symbol;
   } else {
    node_start = findex;
   }
```

```
3884   break;
3885  case 0x12:
3886   is >> symbol >> zone;
3887   is >> findex >> lindex;
3888   is >> type;
3889   if((c = Util::nextc(is)) == ')');
3890   else is >> etype;
3891   is >> symbol >> symbol;
3892
3893   while(symbol == '(') {
3894    do{ is >> c; } while(c != ')');
3895    is >> symbol;
3896   }
3897   if(zone == 0) {
3898    mo.c.resize(lindex);
3899   }
3900   break;
3901  case 0x13:
3902   is >> symbol >> zone;
3903   is >> findex >> lindex;
3904   is >> bctype;
3905   if((c = Util::nextc(is)) == ')') ftype = bctype;
3906   else is >> ftype;
3907   is >> symbol >> symbol;
3908
3909   if(zone != 0) {
3910    std::stringstream name;
3911    name << "zone" << zone;
3912    IntVector& gB = mo.bdry[name.str().c_str()];
3913
3914    Facet f;
3915    int n,c0,c1,k;
3916    for(int i = findex;i <= lindex;i++) {
3917     f.clear();
3918     is >> n;
3919     for(int j = 0;j < n;j++) {
3920      is >> k;
3921      f.push_back(k - node_start);
3922     }
3923     mo.f.push_back(f);
3924     gB.push_back(i - facet_start);
```

```
3925
3926      is >> c0 >> c1;
3927      if(c0 == 0) {
3928       mo.fo.push_back(Constants::MAX_INT);
3929      } else {
3930       mo.fo.push_back(c0 - 1);
3931      }
3932      if(c1 == 0) {
3933       mo.fn.push_back(Constants::MAX_INT);
3934      } else {
3935                      mo.fn.push_back(c1 - 1);
3936      }
3937     }
3938     is >> symbol >> symbol;
3939    } else {
3940     facet_start = findex;
3941    }
3942    break;
3943   case 0x39:
3944   case 0x45:
3945    is >> symbol >> dec >> zone >> hex;
3946    {
3947     string str;
3948     char buf[64];
3949     is >> str;
3950     int i = 0;
3951     do{ is >> c; } while(c != ')' && (buf[i++] = c));
3952     buf[i] = 0;
3953     bnames[zone] = buf;
3954    }
3955   default:
3956    while((c = Util::nextc(is))) {
3957     is >> c;
3958     if(c == '(') braces++;
3959     else if(c == ')') {
3960      braces--;
3961      if(!braces) break;
3962     }
3963    }
3964    break;
3965   }
```

```
3966  }
3967  /*rename*/
3968  for(map<int,string>::iterator it = bnames.begin();it != bnames.end();++it) {
3969   std::stringstream name;
3970   name << "zone" << dec << it->first;
3971   Boundaries::iterator it1 = mo.bdry.find(name.str().c_str());
3972   if(it1 != mo.bdry.end()) {
3973    mo.bdry[it->second] = it1->second;
3974    mo.bdry.erase(it1);
3975   }
3976  }
3977  /*add cells*/
3978  Int co,cn;
3979  forEach(mo.f,i) {
3980   co = mo.fo[i];
3981   cn = mo.fn[i];
3982   if(co != Constants::MAX_INT)
3983    mo.c[co].push_back(i);
3984   if(cn != Constants::MAX_INT)
3985    mo.c[cn].push_back(i);
3986  }
3987 }
3988 void writeMshMesh(std::ostream& os,Mesh::MeshObject& mo) {
3989  os << "(0 \"ASCII msh file\")" << endl << endl;
3990  os << "(0 \"Dimension:\")" << endl;
3991  os << "(2 3)" << endl << endl;
3992
3993  //vertices
3994  os << "(0 \"Vertices:\")" << endl;
3995  os << "(10 (0 1 " << mo.v.size() << " 0 3))" << endl << endl;
3996  os << "(10 (1 1 " << mo.v.size() << " 1 3)" << endl;
3997  os << "(" << endl;
3998  os.precision(10);
3999  forEach(mo.v,i)
4000   os << scientific << mo.v[i] << endl;
4001  os << "))" << endl << endl;
4002
4003  //facets
4004  os << "(0 \"Facets:\")" << endl;
4005  os << "(13 (0 1 " << mo.f.size() << " 0 0))" << endl << endl;
4006
```

```
4007 Int zone = 1;
4008 Int start = 1;
4009
4010 //internal
4011 Int nInternal = mo.f.size() - (mo.c.size() - mo.nc);
4012 os << "(0 \"Internal faces:\")" << endl;
4013 os << "(39 (" << dec << zone << hex << " interior " << "interior-1"
4014    << ")())" << endl;
4015 os << "(13 (" << zone << " " << start << " "
4016  << nInternal << " 2 0)" << endl;
4017 zone++;
4018 start += nInternal;
4019 os << "(" << endl;
4020 forEach(mo.f,f) {
4021  if(mo.fn[f] >= mo.nc)
4022   continue;
4023  Facet& mf = mo.f[f];
4024  os << mf.size() << " ";
4025  forEach(mf,j)
4026   os << mf[j] + 1 << " ";
4027  if(Mesh::_reversed[f])
4028   os << mo.fo[f] + 1 << " " << mo.fn[f] + 1 << endl;
4029  else
4030   os << mo.fn[f] + 1 << " " << mo.fo[f] + 1 << endl;
4031 }
4032 os << "))" << endl << endl;
4033
4034 //boundary
4035 forEachIt(Boundaries,mo.bdry,it) {
4036  const IntVector& fvec = it->second;
4037  os << "(0 \"" << it->first << "\")" << endl;
4038
4039  string bname = "pressure-inlet";
4040  Int bid = 4;
4041  if(it->first.find("WALL") != std::string::npos) {
4042   bname = "wall";
4043   bid = 3;
4044  }
4045  os << "(39 (" << dec << zone << hex << " " << bname << " " << it->first
4046       << ")())" << endl;
4047  os << "(13 (" << zone << " " << start << " "
```

```
4048      << start + fvec.size() - 1 << " " << bid << " 0)" << endl;
4049    zone++;
4050    start += fvec.size();
4051
4052    os << "(" << endl;
4053    forEach(fvec,i) {
4054     Int f = fvec[i];
4055     Facet& mf = mo.f[f];
4056     os << mf.size() << " ";
4057     forEach(mf,j)
4058      os << mf[j] + 1 << " ";
4059     if(Mesh::_reversed[f])
4060      os << mo.fo[f] + 1 << " 0" << endl;
4061     else
4062      os << "0 " << mo.fo[f] + 1 << endl;
4063    }
4064    os << "))" << endl << endl;
4065   }
4066
4067   //cells
4068   os << "(0 \"Cells:\")" << endl;
4069   os << "(12 (0 1 " << mo.nc << " 0 0))" << endl;
4070   os << "(12 (1 1 " << mo.nc << " 1 0)(" << endl;
4071   for(Int i = 0;i < mo.nc;i++)
4072    os << "4 ";
4073   os << endl << ")())" << endl;
4074  }#ifndef __MP_H
4075  #define __MP_H
4076
4077  #include "mpi.h"
4078  #include "tensor.h"
4079
4080  #if defined __DOUBLE
4081  # define MPI_SCALAR MPI_DOUBLE
4082  #else
4083  # define MPI_SCALAR MPI_FLOAT
4084  #endif
4085
4086  class MP {
4087  public:
4088   enum {
```

```
4089   FIELD, END
4090  };
4091  MP(int argc,char* argv[]);
4092  ~MP();
4093 public:
4094  typedef MPI_Request REQUEST;
4095
4096  static int n_hosts,host_id,name_len;
4097  static char host_name[512];
4098  static int _start_time;
4099  static void loop();
4100  static void barrier();
4101  static int iprobe(int&,int&);
4102  static void send(int,int);
4103  static void recieve(int,int);
4104  static void printH(const char* format,...);
4105  static void print(const char* format,...);
4106
4107  /*send and recieve messages*/
4108  template <class type>
4109  static void recieve(type* buffer,int size,int source,int message_id) {
4110   const int count = (size * sizeof(type) / sizeof(Scalar));
4111   MPI_Recv(buffer,count,MPI_SCALAR,source,message_id,MPI_COMM_WORLD,MPI_STATUS_IGNORE)
          ;
4112  }
4113  template <class type>
4114  static void send(type* buffer,int size,int source,int message_id) {
4115   const int count = (size * sizeof(type) / sizeof(Scalar));
4116   MPI_Send(buffer,count,MPI_SCALAR,source,message_id,MPI_COMM_WORLD);
4117  }
4118  template <class type>
4119  static void allsum(type* sendbuf,type* recvbuf,int size) {
4120   const int count = (size * sizeof(type) / sizeof(Scalar));
4121   MPI_Allreduce(sendbuf,recvbuf,count,MPI_SCALAR,MPI_SUM,MPI_COMM_WORLD);
4122  }
4123  template <class type>
4124  static void irecieve(type* buffer,int size,int source,int message_id,void* request) {
4125   const int count = (size * sizeof(type) / sizeof(Scalar));
4126   MPI_Irecv(buffer,count,MPI_SCALAR,source,message_id,MPI_COMM_WORLD,(MPI_Request*)
          request);
4127  }
```

```
4128  template <class type>
4129  static void isend(type* buffer,int size,int source,int message_id,void* request) {
4130   const int count = (size * sizeof(type) / sizeof(Scalar));
4131   MPI_Isend(buffer,count,MPI_SCALAR,source,message_id,MPI_COMM_WORLD,(MPI_Request*)
          request);
4132  }
4133  static void waitall(int count,void* request) {
4134   MPI_Waitall(count,(MPI_Request*)request,MPI_STATUS_IGNORE);
4135  }
4136 };
4137 #endif
4138 #ifndef __SYSTEM_H
4139 #define __SYSTEM_H
4140
4141 #include <string>
4142 #include <cstdarg>
4143 #ifdef _MSC_VER
4144 #    include <windows.h>
4145 #    include <process.h>
4146 #  include <sys/timeb.h>
4147 #else
4148 #    include <unistd.h>
4149 #    include <sys/stat.h>
4150 #  include <sys/time.h>
4151 #endif
4152
4153
4154 namespace System {
4155  /*get processor id*/
4156  inline int get_pid() {
4157 #ifdef _MSC_VER
4158   return _getpid();
4159 #else
4160   return getpid();
4161 #endif
4162  }
4163  /*system dependent directory operations*/
4164  inline int cd(std::string path) {
4165 #ifdef _MSC_VER
4166   return ::SetCurrentDirectory((LPCTSTR)path.c_str());
4167 #else
```

```
4168   return !::chdir(path.c_str());
4169 #endif
4170  }
4171  inline int mkdir(std::string path) {
4172 #ifdef _MSC_VER
4173   return ::CreateDirectory((LPCTSTR)path.c_str(),NULL);
4174 #else
4175   return !::mkdir(path.c_str(),S_IRWXU);
4176 #endif
4177  }
4178  inline int rmdir(std::string path) {
4179 #ifdef _MSC_VER
4180   return ::RemoveDirectory((LPCTSTR)path.c_str());
4181 #else
4182   return !::rmdir(path.c_str());
4183 #endif
4184  }
4185  /*time*/
4186  inline int get_time() {
4187 #ifdef _MSC_VER
4188   timeb tb;
4189   ftime(&tb);
4190   return int(tb.time * 1000 + tb.millitm);
4191 #else
4192   timeval tb;
4193   gettimeofday(&tb, NULL);
4194   return int(tb.tv_sec * 1000 + tb.tv_usec / 1000);
4195 #endif
4196  }
4197 }
4198
4199 #endif
4200 #include <cstdarg>
4201 #include "mp.h"
4202 #include "system.h"
4203
4204 /*statics*/
4205 int MP::n_hosts;
4206 int MP::host_id;
4207 int MP::name_len;
4208 char MP::host_name[512];
```

```
4209 int MP::_start_time = 0;
4210
4211 /*Initialize*/
4212 MP::MP(int argc,char* argv[]) {
4213  MPI_Init(&argc, &argv);
4214  MPI_Comm_size(MPI_COMM_WORLD, &n_hosts);
4215  MPI_Comm_rank(MPI_COMM_WORLD, &host_id);
4216  MPI_Get_processor_name(host_name, &name_len);
4217  _start_time = System::get_time();
4218  printf("Process [%d/%d] on %s : pid %d\n",
4219   host_id,n_hosts,host_name,System::get_pid());
4220  fflush(stdout);
4221 }
4222
4223 /*finalize*/
4224 MP::~MP() {
4225  MPI_Finalize();
4226 }
4227
4228 /*send*/
4229 void MP::send(int source,int message_id) {
4230  MPI_Send(MPI_BOTTOM,0,MPI_INT,source,message_id,MPI_COMM_WORLD);
4231 }
4232
4233 /*recieve*/
4234 void MP::recieve(int source,int message_id) {
4235  MPI_Recv(MPI_BOTTOM,0,MPI_INT,source,message_id,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
4236 }
4237
4238 /*barrier*/
4239 void MP::barrier() {
4240  MPI_Barrier(MPI_COMM_WORLD);
4241 }
4242
4243 /*probe for messages*/
4244 int MP::iprobe(int& source,int& message_id) {
4245  int flag;
4246  MPI_Status mpi_status;
4247  MPI_Iprobe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD,&flag,&mpi_status);
4248  if(flag) {
4249   message_id = mpi_status.MPI_TAG;
```

```
4250   source = mpi_status.MPI_SOURCE;
4251   return true;
4252  }
4253  return false;
4254 }
4255 /*print*/
4256 void MP::printH(const char* format,...) {
4257  printf("%d [%d] ",System::get_time() - _start_time,host_id);
4258  va_list ap;
4259  va_start(ap, format);
4260  vprintf(format, ap);
4261  va_end(ap);
4262  fflush(stdout);
4263 }
4264 void MP::print(const char* format,...) {
4265  va_list ap;
4266  va_start(ap, format);
4267  vprintf(format, ap);
4268  va_end(ap);
4269  fflush(stdout);
4270 }
4271 #ifndef __PREPARE_H
4272 #define __PREPARE_H
4273
4274 #include "field.h"
4275 #include "vtk.h"
4276
4277 namespace Prepare {
4278  int decomposeXYZ(Mesh::MeshObject&,Int*,Scalar*);
4279  void decomposeFields(std::vector<std::string>& fields,std::string,Int);
4280  int merge(Mesh::MeshObject&,Int*,std::vector<std::string>& fields,std::string,Int);
4281  int convertVTK(Mesh::MeshObject&,std::vector<std::string>& fields,Int);
4282  int probe(Mesh::MeshObject&,std::vector<std::string>& fields,Int);
4283 }
4284
4285 #endif
4286 #include "mesh.h"
4287 #include "prepare.h"
4288 #include "system.h"
4289
4290 using namespace std;
```

```
4291
4292 /*decompose application*/
4293 int main(int argc,char* argv[]) {
4294  /*message passing object*/
4295  MP mp(argc,argv);
4296  ifstream input(argv[1]);
4297
4298     /*read mesh & fields*/
4299  vector<string> fields;
4300  vector<Int> n;
4301  vector<Scalar> axis(4);
4302  axis[0] = 1;
4303  Util::ParamList params("general");
4304  params.enroll("mesh",&Mesh::gMeshName);
4305  params.enroll("fields",&fields);
4306  params.enroll("decompose",&n);
4307  params.enroll("axis",&axis);
4308  params.enroll("probe",&Mesh::probePoints);
4309  params.read(input);
4310
4311  /*Mesh*/
4312  if(mp.n_hosts > 1) {
4313   stringstream s;
4314   s << Mesh::gMeshName << mp.host_id;
4315   if(!System::cd(s.str()))
4316    return 1;
4317  }
4318  Mesh::readMesh();
4319
4320  /*cmd line*/
4321  int work = 0;
4322  Int start_index = 0;
4323  for(int i = 1;i < argc;i++) {
4324   if(!strcmp(argv[i],"-merge")) {
4325    work = 1;
4326   } else if(!strcmp(argv[i],"-vtk")) {
4327    work = 2;
4328   } else if(!strcmp(argv[i],"-probe")) {
4329    work = 3;
4330   } else if(!strcmp(argv[i],"-poly")) {
4331    Vtk::write_polyhedral = true;
```

```
4332   } else if(!strcmp(argv[i],"-start")) {
4333    i++;
4334    start_index = atoi(argv[i]);
4335   }
4336  }
4337
4338  Mesh::initGeomMeshFields(work != 0);
4339  cout << "fields " << fields << endl;
4340  atexit(Util::cleanup);
4341
4342  /*do work*/
4343  if(work == 1) {
4344   Prepare::merge(Mesh::gMesh,&n[0],fields,Mesh::gMeshName,start_index);
4345  } else if(work == 2) {
4346   Prepare::convertVTK(Mesh::gMesh,fields,start_index);
4347  } else if(work == 3) {
4348   Prepare::probe(Mesh::gMesh,fields,start_index);
4349  } else{
4350   Prepare::decomposeXYZ(Mesh::gMesh,&n[0],&axis[0]);
4351   Prepare::decomposeFields(fields,Mesh::gMeshName,start_index);
4352  }
4353  return 0;
4354 }
4355 #include <sstream>
4356 #include "prepare.h"
4357 #include "system.h"
4358
4359 using namespace std;
4360 using namespace Mesh;
4361
4362 /*duplicate fields*/
4363 template <class T>
4364 void duplicateFields(istream& is,ostream& of) {
4365  MeshField<T,CELL> f;
4366
4367  /*internal*/
4368  f.readInternal(is);
4369
4370  /*index file*/
4371  IntVector cLoc;
4372  ifstream index("index");
```

```
4373  index >> cLoc;
4374
4375  /*write it out*/
4376  of << "size "<< sizeof(T) / sizeof(Scalar) << endl;
4377  of << cLoc.size() << endl;
4378  of << "{" << endl;
4379  forEach(cLoc,j)
4380   of << f[cLoc[j]] << endl;
4381  of << "}" << endl;
4382
4383  /*boundaries*/
4384  char c;
4385  string bname,cname;
4386  while((c = Util::nextc(is)) && isalpha(c)) {
4387   BCondition<T> bc(" ");
4388   is >> bc;
4389   of << bc << endl;
4390  }
4391
4392  /*interMesh boundaries*/
4393  while((c = Util::nextc(index)) && isalpha(c)) {
4394   BCondition<T> bc(" ");
4395   index >> bc;
4396   of << bc << endl;
4397  }
4398 }
4399 /*decompose fields*/
4400 void Prepare::decomposeFields(vector<string>& fields,std::string mName,Int start_index
        ) {
4401  int size;
4402  std::string str;
4403
4404  for(Int ID = start_index;;ID++) {
4405   /*cd*/
4406   stringstream path;
4407   path << mName << ID;
4408   if(!System::cd(path.str()))
4409    break;
4410
4411   /*for each field*/
4412   forEach(fields,i) {
```

```
4413    /*read at time 0*/
4414    string str = "../" + fields[i] + "0";
4415    ifstream is(str.c_str());
4416    if(!is.fail()) {
4417     str = fields[i] + "0";
4418     ofstream of(str.c_str());
4419
4420     /*seekg to beg*/
4421     is >> str >> size;
4422     is.seekg(0,fstream::beg);
4423
4424     /*fields*/
4425     switch(size) {
4426     case 1 : duplicateFields<Scalar>(is,of); break;
4427     case 3 : duplicateFields<Vector>(is,of); break;
4428     case 6 : duplicateFields<STensor>(is,of); break;
4429     case 9 : duplicateFields<Tensor>(is,of); break;
4430     }
4431     /*end*/
4432    }
4433   }
4434   /*go back*/
4435   System::cd("..");
4436  }
4437 }
4438 /*decompose in x,y,z direction*/
4439 int Prepare::decomposeXYZ(Mesh::MeshObject& mo,Int* n,Scalar* nq) {
4440
4441  using Constants::MAX_INT;
4442  Int i,j,ID,count,total = n[0] * n[1] * n[2];
4443  Vector maxV(Scalar(-10e30)),minV(Scalar(10e30)),delta;
4444  Vector axis(nq[0],nq[1],nq[2]);
4445  Scalar theta = nq[3];
4446  Vector C;
4447
4448  /*decomposed mesh*/
4449  MeshObject* meshes = new MeshObject[total];
4450  IntVector* vLoc = new IntVector[total];
4451  IntVector* fLoc = new IntVector[total];
4452  IntVector* cLoc = new IntVector[total];
4453  for(i = 0;i < total;i++) {
```

```
4454  vLoc[i].assign(mo.v.size(),0);
4455  fLoc[i].assign(mo.f.size(),0);
4456 }
4457
4458 /*max and min points*/
4459 forEach(mo.v,i) {
4460  C = rotate(mo.v[i],axis,theta);
4461  for(j = 0;j < 3;j++) {
4462   if(C[j] > maxV[j]) maxV[j] = C[j];
4463   if(C[j] < minV[j]) minV[j] = C[j];
4464  }
4465 }
4466    delta = maxV - minV;
4467 for(j = 0;j < 3;j++)
4468  delta[j] /= Scalar(n[j]);
4469
4470 /*decompose cells*/
4471 MeshObject *pmesh;
4472 IntVector *pvLoc,*pfLoc,blockIndex;
4473
4474 blockIndex.assign(gBCellsStart,0);
4475
4476 for(i = 0;i < gBCellsStart;i++) {
4477  Cell& c = mo.c[i];
4478
4479  /* add cell */
4480  C = rotate(_cC[i],axis,theta);
4481  C = (C - minV) / delta;
4482  ID = Int(C[0]) * n[1] * n[2] +
4483    Int(C[1]) * n[2] +
4484    Int(C[2]);
4485  pmesh = &meshes[ID];
4486  pvLoc = &vLoc[ID];
4487  pfLoc = &fLoc[ID];
4488  pmesh->c.push_back(c);
4489  cLoc[ID].push_back(i);
4490  blockIndex[i] = ID;
4491
4492  /* mark vertices and facets */
4493  forEach(c,j) {
4494   Facet& f = mo.f[c[j]];
```

```
4495    (*pfLoc)[c[j]] = 1;
4496    forEach(f,k) {
4497     (*pvLoc)[f[k]] = 1;
4498    }
4499   }
4500  }
4501  /*add vertices & cells*/
4502  for(ID = 0;ID < total;ID++) {
4503   pmesh = &meshes[ID];
4504   pvLoc = &vLoc[ID];
4505   pfLoc = &fLoc[ID];
4506
4507   count = 0;
4508   forEach(mo.v,i) {
4509    if((*pvLoc)[i]) {
4510     pmesh->v.push_back(mo.v[i]);
4511     (*pvLoc)[i] = count++;
4512    } else
4513     (*pvLoc)[i] = Constants::MAX_INT;
4514   }
4515
4516   count = 0;
4517   forEach(mo.f,i) {
4518    if((*pfLoc)[i]) {
4519     pmesh->f.push_back(mo.f[i]);
4520     (*pfLoc)[i] = count++;
4521    } else
4522     (*pfLoc)[i] = Constants::MAX_INT;
4523   }
4524  }
4525  /*adjust IDs*/
4526  for(ID = 0;ID < total;ID++) {
4527   pmesh = &meshes[ID];
4528   pvLoc = &vLoc[ID];
4529   pfLoc = &fLoc[ID];
4530
4531   forEach(pmesh->f,i) {
4532    Facet& f = pmesh->f[i];
4533    forEach(f,j)
4534     f[j] = (*pvLoc)[f[j]];
4535   }
```

```
4536
4537  forEach(pmesh->c,i) {
4538   Cell& c = pmesh->c[i];
4539   forEach(c,j)
4540    c[j] = (*pfLoc)[c[j]];
4541  }
4542 }
4543 /*inter mesh faces*/
4544 IntVector* imesh = new IntVector[total * total];
4545 Int co,cn;
4546 forEach(mo.f,i) {
4547  if(gFN[i] < gBCellsStart) {
4548   co = blockIndex[gFO[i]];
4549   cn = blockIndex[gFN[i]];
4550   if(co != cn) {
4551    imesh[co * total + cn].push_back(fLoc[co][i]);
4552    imesh[cn * total + co].push_back(fLoc[cn][i]);
4553   }
4554  }
4555 }
4556
4557 /*write meshes to file */
4558 for(ID = 0;ID < total;ID++) {
4559  pmesh = &meshes[ID];
4560  pvLoc = &vLoc[ID];
4561  pfLoc = &fLoc[ID];
4562
4563  /*create directory and switch to it*/
4564  stringstream path;
4565  path << mo.name << ID;
4566
4567  System::mkdir(path.str());
4568  if(!System::cd(path.str()))
4569   return 1;
4570
4571  /*v,f & c*/
4572  ofstream of(mo.name.c_str());
4573  of << hex;
4574  of << pmesh->v << endl;
4575  of << pmesh->f << endl;
4576  of << pmesh->c << endl;
```

```
4577
4578  /*bcs*/
4579  forEachIt(Boundaries,mo.bdry,it) {
4580   IntVector b;
4581   Int f;
4582   forEach(it->second,j) {
4583    f = (*pfLoc)[it->second[j]];
4584    if(f != Constants::MAX_INT)
4585     b.push_back(f);
4586   }
4587   /*write to file*/
4588   if(b.size()) {
4589    of << it->first << " ";
4590    of << b << endl;
4591   }
4592  }
4593
4594  /*index file*/
4595  ofstream of2("index");
4596  of2 << cLoc[ID] << endl;
4597
4598  /*inter mesh boundaries*/
4599  for(j = 0;j < total;j++) {
4600   IntVector& f = imesh[ID * total + j];
4601   if(f.size()) {
4602    of << "interMesh_" << ID << "_" << j << " ";
4603    of << f << endl;
4604    of2 << "interMesh_" << ID << "_" << j << " "
4605     << "{\n\ttype GHOST\n}" << endl;
4606   }
4607  }
4608
4609  of << dec;
4610  /*go back*/
4611  if(!System::cd(".."))
4612   return 1;
4613 }
4614
4615 /*delete*/
4616 delete[] meshes;
4617 delete[] imesh;
```

```
4618  delete[] vLoc;
4619  delete[] fLoc;
4620  delete[] cLoc;
4621  return 0;
4622 }
4623 /*read fields*/
4624 template <class T>
4625 void readFields(istream& is,void* pFields,const IntVector& cLoc) {
4626  MeshField<T,CELL>& f = *((MeshField<T,CELL>*)pFields);
4627  Int size;
4628  char symbol;
4629  is >> size >> symbol;
4630  for(Int j = 0;j < size;j++) {
4631   is >> f[cLoc[j]];
4632  }
4633  is >> symbol;
4634 }
4635 /*create fields*/
4636 void createFields(vector<string>& fields,void**& pFields,Int start_index) {
4637  std::string str;
4638  Int size;
4639 
4640  /*for each field*/
4641  pFields = new void*[fields.size()];
4642  forEach(fields,i) {
4643   /*read at time 0*/
4644   stringstream path;
4645   path << fields[i] << start_index;
4646   str = path.str();
4647 
4648   ifstream is(str.c_str());
4649   if(!is.fail()) {
4650    /*fields*/
4651    is >> str >> size;
4652    switch(size) {
4653     case 1 : pFields[i] = new ScalarCellField(fields[i].c_str(),READWRITE); break;
4654     case 3 : pFields[i] = new VectorCellField(fields[i].c_str(),READWRITE); break;
4655     case 6 : pFields[i] = new STensorCellField(fields[i].c_str(),READWRITE); break;
4656     case 9 : pFields[i] = new TensorCellField(fields[i].c_str(),READWRITE); break;
4657    }
4658    /*end*/
```

```
4659   }
4660  }
4661 }
4662 /*open fields*/
4663 Int checkFields(vector<string>& fields,void**& pFields,Int step) {
4664  Int count = 0;
4665  forEach(fields,i) {
4666   stringstream fpath;
4667   fpath << fields[i] << step;
4668   ifstream is(fpath.str().c_str());
4669   if(is.fail())
4670    continue;
4671   count++;
4672   break;
4673  }
4674  if(count)
4675   Mesh::read_fields(step);
4676  return count;
4677 }
4678 /*Reverse decomposition*/
4679 int Prepare::merge(Mesh::MeshObject& mo,Int* n,
4680      vector<string>& fields,std::string mName,Int start_index) {
4681    /*create fields*/
4682  void** pFields;
4683  createFields(fields,pFields,start_index);
4684 
4685  /*indexes*/
4686  Int total = n[0] * n[1] * n[2];
4687  IntVector* cLoc = new IntVector[total];
4688  std::string str;
4689  Int size;
4690 
4691  for(Int ID = 0;ID < total;ID++) {
4692   stringstream path;
4693   path << mName << ID;
4694   str = path.str() + "/index";
4695   ifstream index(str.c_str());
4696   index >> cLoc[ID];
4697  }
4698 
4699  /*for each time step*/
```

```
4700  Int step = start_index;
4701  Mesh::read_fields(step);
4702  for(step = start_index + 1;;step++) {
4703   Int count = 0;
4704   for(Int ID = 0;ID < total;ID++) {
4705    stringstream path;
4706    path << mName << ID;
4707    forEach(fields,i) {
4708     stringstream fpath;
4709     fpath << fields[i] << step;
4710     str = path.str() + "/" + fpath.str();
4711     ifstream is(str.c_str());
4712     if(is.fail())
4713      continue;
4714     count++;
4715     /*read*/
4716     is >> str >> size;
4717     switch(size) {
4718     case 1 : readFields<Scalar>(is,pFields[i],cLoc[ID]); break;
4719     case 3 : readFields<Vector>(is,pFields[i],cLoc[ID]); break;
4720     case 6 : readFields<STensor>(is,pFields[i],cLoc[ID]); break;
4721     case 9 : readFields<Tensor>(is,pFields[i],cLoc[ID]); break;
4722     }
4723    }
4724   }
4725   if(count == 0) break;
4726   Mesh::write_fields(step);
4727  }
4728
4729  return 0;
4730 }
4731 /*Convert to VTK format*/
4732 int Prepare::convertVTK(Mesh::MeshObject& mo,vector<string>& fields,Int start_index) {
4733     /*create fields*/
4734  void** pFields;
4735  createFields(fields,pFields,start_index);
4736
4737  /*for each time step*/
4738  for(Int step = start_index;;step++) {
4739   if(!checkFields(fields,pFields,step))
4740    break;
```

```
4741
4742   /*write vtk*/
4743   Vtk::write_vtk(step);
4744  }
4745
4746  return 0;
4747 }
4748 /*Probe values at certain locations*/
4749 int Prepare::probe(Mesh::MeshObject& mo,vector<string>& fields,Int start_index) {
4750  /*probe points*/
4751  IntVector probes;
4752  getProbeFaces(probes);
4753  ofstream of("probes");
4754
4755     /*create fields*/
4756  void** pFields;
4757  createFields(fields,pFields,start_index);
4758
4759  /*for each time step*/
4760  for(Int step = start_index;;step++) {
4761   if(!checkFields(fields,pFields,step))
4762    break;
4763
4764   /*Interpolate*/
4765   forEachField(interpolateVertexAll());
4766
4767   /*write probes*/
4768 #define ADD(v,value,weight) {          \
4769   dist = magSq((v) - probeP);         \
4770   dist = weight / (dist + 1.0f);        \
4771   sum += (value) * dist;          \
4772   sumd += dist;             \
4773 }
4774 #define SUM(X) {              \
4775   Cell& c = gCells[X];           \
4776   forEach(c,m) {            \
4777    Facet& f = gFacets[c[m]];        \
4778    forEach(f,j) {           \
4779     ADD(gVertices[f[j]],(*it)[f[j]],1.0);    \
4780    }              \
4781   }               \
```

```
4782 }
4783 #define WRITE(T) {                    \
4784     std::list<MeshField<T,CELL>*>::iterator it1 =    \
4785   MeshField<T,CELL>::fields_.begin();      \
4786     for(MeshField<T,CELL>::vertexFieldsType::iterator it = \
4787      (MeshField<T,CELL>::vf_fields_)->begin(); it != \
4788   (MeshField<T,CELL>::vf_fields_)->end(); ++it,++it1) { \
4789   T sum(0.0);            \
4790   Scalar sumd(0.0);          \
4791   ADD(cC[c1],(*(*it1))[c1],2.0);       \
4792   ADD(cC[c2],(*(*it1))[c2],2.0);       \
4793   SUM(sc);                   \
4794      of << (sum/sumd) << " ";        \
4795  }                     \
4796 }
4797  forEach(probes,i) {
4798   Int fi = probes[i];
4799   Int c1 = gFO[fi];
4800   Int c2 = gFN[fi];
4801   Vector probeP = probePoints[i];
4802   Scalar dir = ((fC[fi] - probeP) & fN[fi]),dist;
4803   Int sc;
4804   if(dir >= 0) sc = c1;
4805   else sc = c2;
4806
4807   of << step << " " << i << " " << probePoints[i] << " ";
4808
4809   WRITE(Scalar);
4810   WRITE(Vector);
4811   WRITE(STensor);
4812   WRITE(Tensor);
4813
4814   of << endl;
4815  }
4816 #undef WRITE
4817 #undef SUM
4818 #undef ADD
4819  }
4820
4821  return 0;
4822 }#ifndef __UTIL_H
```

```
4823 #define __UTIL_H
4824
4825 #include "tensor.h"
4826 #include <map>
4827 #include <vector>
4828 #include <algorithm>
4829 #include <cstdarg>
4830
4831 /*vector IO*/
4832 template <class T>
4833 std::ostream& operator << (std::ostream& os, const std::vector<T>& p) {
4834  os << p.size() << std::endl;
4835  os << "{ " << std::endl;
4836  forEach(p,i)
4837   os << p[i] << std::endl;
4838  os << "}" << std::endl;
4839  return os;
4840 }
4841 template <class T>
4842 std::istream& operator >> (std::istream& is, std::vector<T>& p) {
4843  Int size;
4844  char symbol;
4845  is >> size >> symbol;
4846  p.resize(size);
4847  forEach(p,i)
4848   is >> p[i];
4849  is >> symbol;
4850  return is;
4851 }
4852
4853 std::ostream& operator << (std::ostream& os, const std::vector<Int>& p);
4854
4855 /*equal vectors*/
4856 template <class T>
4857 bool equal(std::vector<T>& v1,std::vector<T>& v2) {
4858  Int j;
4859  forEach(v1,i) {
4860   for(j = 0;j < v2.size();j++) {
4861    if(v1[i] == v2[j])
4862     break;
4863   }
```

```
4864   if(j == v2.size())
4865    return false;
4866  }
4867  return true;
4868 }
4869
4870 /*Utililty functions*/
4871 namespace Util {
4872  extern bool Terminated;
4873  Int hash_function(std::string s);
4874  int nextc(std::istream&);
4875  void cleanup();
4876
4877  /*string compare*/
4878  inline int compare(std::string& s1,std::string s2) {
4879   std::string t1 = s1,t2 = s2;
4880   std::transform(t1.begin(),t1.end(),t1.begin(),toupper);
4881   std::transform(t2.begin(),t2.end(),t2.begin(),toupper);
4882   return (t1 != t2);
4883  }
4884
4885  /*general string option list*/
4886  namespace A {
4887   struct Option {
4888    Int* val;
4889    std::vector<std::string> list;
4890    Option(void* v,Int N, ...) {
4891     val = (Int*)v;
4892     std::string str;
4893     list.assign(N,"");
4894     va_list ap;
4895     va_start(ap, N);
4896     for(Int i = 0;i < N;i++) {
4897      str = va_arg(ap,char*);
4898      list[i] = str;
4899     }
4900     va_end(ap);
4901    }
4902    Int getID(std::string str) {
4903     forEach(list,i) {
4904      if(!Util::compare(list[i],str))
```

```
4905      return i;
4906     }
4907     std::cout << "Unknown parameter : " << str << std::endl;
4908     return 0;
4909    }
4910    friend std::istream& operator >> (std::istream& is, Option& p) {
4911     std::string str;
4912     is >> str;
4913     *(p.val) = p.getID(str);
4914     return is;
4915    }
4916    friend std::ostream& operator << (std::ostream& os, const Option& p) {
4917     os << p.list[*(p.val)];
4918     return os;
4919    }
4920   };
4921  }
4922  using A::Option;
4923
4924  /*bool option*/
4925  struct BoolOption : public Option {
4926   BoolOption(void* v) :
4927   Option(v,2,"NO","YES")
4928   {
4929   }
4930  };
4931
4932  /*parameters*/
4933  template <typename T>
4934  class Parameters{
4935   std::map<std::string,T*> list;
4936  public:
4937   void enroll(std::string str,T* addr) {
4938    list[str] = addr;
4939   }
4940   bool read(std::string str,std::istream& is,bool out) {
4941    typename std::map<std::string,T*>::iterator it = list.find(str);
4942    if(it != list.end()) {
4943     is >> *(it->second);
4944     if(out) std::cout << *(it->second);
4945     return true;
```

```
4946    }
4947    return false;
4948   }
4949  };
4950  extern void read_params(std::istream&,std::string = "");
4951
4952  /*parameters list*/
4953  struct ParamList {
4954   std::string name;
4955   static std::map<std::string,ParamList*> list;
4956
4957   ParamList(std::string n) : name(n) {
4958    list[name] = this;
4959   }
4960   ~ParamList() {
4961    list.erase(name);
4962   }
4963
4964 #define addParam(T,N)         \
4965  Parameters<T> params_##N;      \
4966  void enroll(std::string str,T* addr) {   \
4967   params_##N.enroll(str,addr);    \
4968  }
4969   addParam(Int,Int);
4970   addParam(Scalar,Scalar);
4971   addParam(Vector,Vector);
4972   addParam(STensor,STensor);
4973   addParam(Tensor,Tensor);
4974   addParam(std::string,string);
4975   addParam(Option,Option);
4976   addParam(std::vector<Int>,vec_int);
4977   addParam(std::vector<std::string>,vec_string);
4978   addParam(std::vector<Scalar>,vec_scalar);
4979   addParam(std::vector<Vector>,vec_vector);
4980 #undef addParam
4981
4982   void read(std::istream& is,std::string str,bool out) {
4983 #define readp(N) params_##N.read(str,is,out)
4984    if(readp(Int));
4985    else if(readp(string));
4986    else if(readp(Option));
```

```
4987    else if(readp(Scalar));
4988    else if(readp(Vector));
4989    else if(readp(Tensor));
4990    else if(readp(STensor));
4991    else if(readp(vec_int));
4992    else if(readp(vec_scalar));
4993    else if(readp(vec_vector));
4994    else if(readp(vec_string));
4995    else if(out) {
4996     std::cout << "UNKNOWN";
4997    }
4998 #undef readp
4999   }
5000   void read(std::istream& is) {
5001    read_params(is,name);
5002   }
5003  };
5004  /*end*/
5005 }
5006
5007 #endif
5008 #include <string>
5009 #include "util.h"
5010
5011 using namespace std;
5012
5013 namespace Util {
5014  bool Terminated = false;
5015
5016  std::map<std::string,ParamList*> ParamList::list;
5017 }
5018
5019 Int Util::hash_function(std::string s) {
5020  Int h = 0;
5021  const char* p = s.c_str();
5022  while(*p) { h = 31 * h + *p++; }
5023  return h;
5024 }
5025 int Util::nextc(std::istream& is) {
5026  char c;
5027  is >> c;
```

```
5028  while(c == '#') {
5029   while((c = is.get()) && c != '\n');
5030   is >> c;
5031  }
5032  if(is.eof())
5033   return 0;
5034  is.putback(c);
5035  return c;
5036 }
5037 void Util::cleanup () {
5038  Terminated = true;
5039  printf("Exiting application\n");
5040 }
5041 void Util::read_params(istream& is,std::string block) {
5042  string str;
5043  char c;
5044  bool output = block.empty();
5045
5046 #define READ() {    \
5047  c = Util::nextc(is);  \
5048  if(!c) goto END;   \
5049  else if(c == '}') {   \
5050   is >> c;    \
5051   break;     \
5052  } else is >> str;   \
5053 }
5054
5055  while(true) {
5056   READ();
5057   is >> c;
5058
5059   map<string,ParamList*>::iterator it = ParamList::list.find(str);
5060   if((it == ParamList::list.end()) ||
5061    (!block.empty() && compare(str,block))) {
5062    int braces = 1;
5063    while((c = Util::nextc(is))) {
5064     is >> c;
5065     if(c == '{') braces++;
5066     else if(c == '}') {
5067      braces--;
5068      if(!braces) break;
```

```
5069     }
5070    }
5071    continue;
5072   }
5073
5074   if(output) cout << str << "\n{\n" << endl;
5075   ParamList* params = it->second;
5076   while(true) {
5077    READ();
5078    if(output) cout << "\t" << str << " = ";
5079    params->read(is,str,output);
5080    if(output) cout << endl;
5081   }
5082   if(output) cout << "}\n" << endl;
5083   if(!block.empty())
5084    break;
5085  }
5086 END:
5087  is.clear();
5088  is.seekg(0,ios::beg);
5089 }
5090
5091 std::ostream& operator << (std::ostream& os, const std::vector<Int>& p) {
5092  Int sz = p.size();
5093  if(sz >= 16) os << sz << endl << "{ ";
5094  else os << sz << "{ ";
5095  for(Int i = 0;i < sz;i++) {
5096   if(sz >= 16 && (i % 16) == 0)
5097    os << endl;
5098   os << p[i] << " ";
5099  }
5100  if(sz >= 16) os << endl << "}";
5101  else os << "}";
5102  return os;
5103 }
5104
5105
5106 #ifndef __VTK_H
5107 #define __VTK_H
5108
5109 #include "field.h"
```

```
5110
5111 namespace Vtk {
5112  void write_vtk(Int);
5113  extern bool write_polyhedral;
5114  extern bool write_cell_value;
5115 }
5116
5117 #endif
5118 #include "vtk.h"
5119
5120 using namespace std;
5121 using namespace Mesh;
5122
5123 bool Vtk::write_polyhedral = false;
5124 bool Vtk::write_cell_value = true;
5125
5126 static Int cell_count(Cell& c) {
5127  Facet* f;
5128  Int i,nFacets = c.size(),nVertices = 0,nTotal;
5129  for(i = 0;i < nFacets;i++) {
5130   f = &gFacets[c[i]];
5131   nVertices += f->size();
5132  }
5133  nTotal = nFacets + nVertices + 2;
5134  return nTotal;
5135 }
5136
5137 static void cell_vtk(std::ofstream& of, Cell& c) {
5138  Facet* f;
5139  Int i,j,nFacets = c.size(),nVertices = 0,nTotal;
5140  for(i = 0;i < nFacets;i++) {
5141   f = &gFacets[c[i]];
5142   nVertices += f->size();
5143  }
5144  nTotal = nFacets + nVertices + 2;
5145
5146  /*write*/
5147  of << nTotal - 1 << " " << nFacets << " ";
5148  for(i = 0;i < nFacets;i++) {
5149   f = &gFacets[c[i]];
5150   of << f->size() << " ";
```

```
5151   for(j = 0;j < f->size();j++) {
5152    of << (*f)[j] << " ";
5153   }
5154  }
5155  of << endl;
5156 }
5157 
5158 void Vtk::write_vtk(Int step) {
5159  Int total;
5160  stringstream path;
5161  path << gMeshName << step << ".vtk";
5162  ofstream of(path.str().c_str());
5163  if(write_polyhedral)
5164   of << "# vtk DataFile Version 2.0" << endl;
5165  else
5166   of << "# vtk DataFile Version 1.0" << endl;
5167  of << Mesh::gMeshName << endl;
5168  of << "ASCII" << endl;
5169  of << "DATASET UNSTRUCTURED_GRID" << endl;
5170     /*Geometry*/
5171  Int i;
5172  of << "POINTS " << gVertices.size() << " float" << endl;
5173  of.precision(12);
5174  forEach(gVertices,i)
5175   of << gVertices[i] << endl;
5176     of.precision(6);
5177  if(write_polyhedral) {
5178   /*polyhedral cells*/
5179   total = 0;
5180   for(i = 0;i < gBCellsStart;i++)
5181    total += cell_count(gCells[i]);
5182 
5183   of << "CELLS " << gBCellsStart << " " << total << endl;
5184   for(i = 0;i < gBCellsStart;i++)
5185    cell_vtk(of,gCells[i]);
5186 
5187   of << "CELL_TYPES " << gBCellsStart << endl;
5188   for(i = 0;i < gBCellsStart;i++)
5189    of << 42 << endl;
5190  } else {
5191   /*hexahedral cells*/
```

```
5192  of << "CELLS " << gBCellsStart << " " << gBCellsStart * 9 << endl;
5193  for(i = 0;i < gBCellsStart;i++) {
5194   Cell& c = gCells[i];
5195   Facet f1 = gFacets[c[0]];
5196   Facet f2 = gFacets[c[1]];
5197   of << f1.size() + f2.size() << " ";
5198   forEach(f1,j)
5199    of << f1[j] << " ";
5200   forEach(f2,j)
5201    of << f2[j] << " ";
5202   of << endl;
5203  }
5204  of << "CELL_TYPES " << gBCellsStart << endl;
5205  for(i = 0;i < gBCellsStart;i++) {
5206   of << "12" << endl;
5207  }
5208 }
5209 /*Fields*/
5210 total = ScalarCellField::count_writable() +
5211     VectorCellField::count_writable() +
5212   STensorCellField::count_writable() +
5213   TensorCellField::count_writable();
5214 if(write_cell_value) {
5215  of << "CELL_DATA " << gBCellsStart << endl;
5216  of << "FIELD attributes "<< total + 1 << endl;
5217  forEachField(writeVtkCellAll(of));
5218  of << "cellID 1 " << Mesh::gBCellsStart << " int" << endl;
5219  for(Int i = 0;i < Mesh::gBCellsStart;i++) of << i << endl;
5220 }
5221 of << "POINT_DATA " << gVertices.size() << endl;
5222 of << "FIELD attributes "<< total << endl;
5223 forEachField(writeVtkVertexAll(of));
5224 }
5225 #ifndef __SOLVE_H
5226 #define __SOLVE_H
5227
5228 #include "field.h"
5229
5230 void Solve(const MeshMatrix<Scalar>&);
5231 void Solve(const MeshMatrix<Vector>&);
5232 void Solve(const MeshMatrix<STensor>&);
```

```
5233 void Solve(const MeshMatrix<Tensor>&);
5234
5235 #endif
5236 #include "solve.h"
5237
5238 /* ************************************************************************
5239  *  Solve system of linear equations iteratively
5240  * ************************************************************************/
5241 template<class type>
5242 Scalar getResidual(const MeshField<type,CELL>& r,
5243       const MeshField<type,CELL>& cF,
5244       bool sync) {
5245  type res[2];
5246  res[0] = type(0);
5247  res[1] = type(0);
5248  for(Int i = 0;i < Mesh::gBCellsStart;i++) {
5249   res[0] += (r[i] * r[i]);
5250   res[1] += (cF[i] * cF[i]);
5251  }
5252  if(sync) {
5253   type global_res[2];
5254   MP::allsum(res,global_res,2);
5255   res[0] = global_res[0];
5256   res[1] = global_res[1];
5257  }
5258  return sqrt(mag(res[0]) / mag(res[1]));
5259 }
5260
5261 template<class type>
5262 void SolveT(const MeshMatrix<type>& M) {
5263
5264  using namespace Mesh;
5265  MeshField<type,CELL> r,p,AP;
5266  MeshField<type,CELL> r1(false),p1(false),AP1(false);
5267  MeshField<type,CELL>& cF = *M.cF;
5268  MeshField<type,CELL>& buffer = AP;
5269  ScalarCellField D = M.ap,iD = (1 / M.ap);
5270  Scalar res,ires;
5271  type alpha,beta,o_rr = type(0),oo_rr;
5272  Int j,iterations = 0;
5273  bool converged = false;
```

```
5274 register Int i;
5275
5276 /*****************************
5277  * Parallel controls
5278  *****************************/
5279 bool print = (MP::host_id == 0);
5280 int  end_count = 0;
5281 bool sync = (Controls::parallel_method == Controls::BLOCKED)
5282  && gInterMesh.size();
5283 std::vector<bool> sent_end(gInterMesh.size(),false);
5284
5285 /*****************************
5286  * Identify solver type
5287  *****************************/
5288 if(print) {
5289  if(M.flags & M.SYMMETRIC)
5290   MP::printH("SYMM-");
5291  else
5292   MP::printH("ASYM-");
5293  if(Controls::Solver == Controls::JACOBI)
5294   MP::print("JAC :");
5295  else if(Controls::Solver == Controls::SOR)
5296   MP::print("SOR :");
5297  else {
5298   switch(Controls::Preconditioner) {
5299   case Controls::NOP: MP::print("PCG :"); break;
5300   case Controls::DIAG: MP::print("DIAG-PCG :"); break;
5301   case Controls::SORP: MP::print("SOR-PCG :"); break;
5302   case Controls::DILU: MP::print("DILU-PCG :"); break;
5303   }
5304  }
5305 }
5306 /*****************************
5307  * Initialization
5308  *****************************/
5309 if(Controls::Solver == Controls::PCG) {
5310  if(!(M.flags & M.SYMMETRIC)) {
5311   /* Allocate BiCG vars*/
5312   r1.allocate();
5313   p1.allocate();
5314   AP1.allocate();
```

```
5315   } else {
5316    if(Controls::Preconditioner == Controls::SORP) {
5317     /*SOR and GS*/
5318     iD *= Controls::SOR_omega;
5319     D *= (2.0 / Controls::SOR_omega - 1.0);
5320    } else if(Controls::Preconditioner == Controls::DILU) {
5321     /*D-ILU(0)*/
5322     for(i = 0;i < gBCellsStart;i++) {
5323      Cell& c = gCells[i];
5324      forEach(c,j) {
5325       Int f = c[j];
5326       Int c1 = gFO[f];
5327       Int c2 = gFN[f];
5328       if(i == c1) {
5329        if(c2 > i) D[c2] -=
5330         (M.an[0][f] * M.an[1][f] * iD[c1]);
5331       } else {
5332        if(c1 > i) D[c1] -=
5333         (M.an[0][f] * M.an[1][f] * iD[c2]);
5334       }
5335      }
5336     }
5337     iD = (1 / D);
5338    }
5339    /*end*/
5340   }
5341  }
5342  /*****************************
5343   * Jacobi sweep
5344   *****************************/
5345 #define JacobiSweep() {        \
5346  AP = iD * getRHS(M);       \
5347  for(i = 0;i < gBCellsStart;i++)     \
5348   cF[i] = AP[i];        \
5349 }
5350  /*****************************
5351   *  Forward/backward GS sweeps
5352   *****************************/
5353 #define Sweep_(X,B,i) {        \
5354  Cell& c = gCells[i];       \
5355  type ncF = B[i];        \
```

```
5356  forEach(c,j) {          \
5357   Int f = c[j];         \
5358   if(i == gFO[f])         \
5359    ncF += X[gFN[f]] * M.an[1][f];   \
5360   else           \
5361    ncF += X[gFO[f]] * M.an[0][f];   \
5362  }             \
5363  ncF *= iD[i];          \
5364  X[i] = X[i] * (1 - Controls::SOR_omega) +  \
5365   ncF * (Controls::SOR_omega);    \
5366 }
5367 #define ForwardSweep(X,B) {       \
5368  for(i = 0;i < gBCellsStart;i++)     \
5369   Sweep_(X,B,i);         \
5370 }
5371 #define BackwardSweep(X,B) {      \
5372  for(int i = gBCellsStart - 1;i >= 0;i--)  \
5373   Sweep_(X,B,i);         \
5374 }
5375  /***********************************
5376   *  Forward/backward substitution
5377   ***********************************/
5378 #define Substitute_(X,B,i,forw,tr) {    \
5379  Cell& c = gCells[i];       \
5380  type ncF = B[i];         \
5381  forEach(c,j) {          \
5382   Int f = c[j];         \
5383   Int c1 = gFO[f];        \
5384      Int c2 = gFN[f];        \
5385   if(i == c1) {         \
5386    if((forw && (c2 < c1)) ||    \
5387      (!forw && (c1 < c2))) {    \
5388     ncF += X[c2] * M.an[1 - tr][f];  \
5389    }           \
5390   } else {          \
5391    if((forw && (c2 > c1)) ||    \
5392      (!forw && (c1 > c2)))     \
5393     ncF += X[c1] * M.an[0 + tr][f];  \
5394   }            \
5395  }             \
5396  ncF *= iD[i];          \
```

```
 X[i] = ncF;          \
}
#define ForwardSub(X,B,TR) {      \
 for(i = 0;i < gBCellsStart;i++)     \
  Substitute_(X,B,i,true,TR);     \
}
#define BackwardSub(X,B,TR) {      \
 for(int i = gBCellsStart - 1;i >= 0;i--)  \
  Substitute_(X,B,i,false,TR);    \
}
#define DiagSub(X,B) {        \
 for(i = 0;i < gBCellsStart;i++)     \
  X[i] = B[i] * iD[i];      \
}
 /***********************************
  *  Preconditioners
  ***********************************/
#define precondition_(R,Z,TR) {      \
 using namespace Controls;      \
 if(Preconditioner == Controls::NOP) {   \
  Z = R;          \
 } else if(Preconditioner == Controls::DIAG) { \
  DiagSub(Z,R);        \
 } else {          \
  if(Controls::Solver == Controls::PCG) {  \
   Z = type(0);       \
   ForwardSub(Z,R,TR);      \
   Z = Z * D;        \
   BackwardSub(Z,Z,TR);     \
  }           \
 }            \
}
#define precondition(R,Z) precondition_(R,Z,0)
#define preconditionT(R,Z) precondition_(R,Z,1)
 /***********************************
  *  SAXPY and DOT operations
  ***********************************/
#define Taxpy(Y,I,X,alpha_) {      \
 for(i = 0;i < gBCellsStart;i++)     \
  Y[i] = I[i] + X[i] * alpha_;    \
}
```

```
5438 #define Tdot(X,Y,sum) {        \
5439  sum = type(0);         \
5440  for(i = 0;i < gBCellsStart;i++)     \
5441   sum += X[i] * Y[i];       \
5442 }
5443  /***********************************
5444   *  Synchronized sum and exchange
5445   ***********************************/
5446 #define SUM_ALL(typ,var) if(sync) {   \
5447  typ t;           \
5448  MP::allsum(&var,&t,1);       \
5449  var = t;          \
5450 }
5451 #define EXCHANGE(var)  if(sync) {   \
5452  exchange_ghost(&var[0]);      \
5453 }
5454  /***********************************
5455   *  Residual
5456   ***********************************/
5457 #define CALC_RESID() {        \
5458  r = M.Su - M * cF;        \
5459  forEachS(r,k,gBCellsStart)      \
5460   r[k] = type(0);        \
5461  precondition(r,AP);        \
5462  forEachS(AP,k,gBCellsStart)      \
5463   AP[k] = type(0);       \
5464  res = getResidual(AP,cF,sync);     \
5465  if(Controls::Solver == Controls::PCG) {   \
5466   Tdot(r,AP,o_rr);       \
5467   SUM_ALL(type,o_rr);       \
5468   p = AP;          \
5469   if(!(M.flags & M.SYMMETRIC)) {    \
5470    r1 = r;         \
5471    p1 = p;         \
5472   }           \
5473  }            \
5474 }
5475  /************************
5476   *  Initialize residual
5477   ************************/
5478  CALC_RESID();
```

```
5479 ires = res;
5480 /*************************************************************
5481 * Initialize exchange of ghost cells just once.
5482 * Lower numbered processors send message to higher ones.
5483 *************************************************************/
5484 if(!sync) {
5485  end_count = gInterMesh.size();
5486  forEach(gInterMesh,i) {
5487   interBoundary& b = gInterMesh[i];
5488   if(b.from < b.to) {
5489    IntVector& f = *(b.f);
5490    forEach(f,j)
5491     buffer[j] = cF[gFO[f[j]]];
5492    MP::send(&buffer[0],f.size(),b.to,MP::FIELD);
5493   }
5494  }
5495 }
5496    /* ***************************
5497  * Iterative solution
5498  * ************************/
5499 while(iterations < Controls::max_iterations) {
5500  /*counter*/
5501  iterations++;
5502 
5503  /*select solver*/
5504  if(Controls::Solver == Controls::JACOBI) {
5505   /*Jacobi solver*/
5506   p = cF;
5507   JacobiSweep();
5508   for(i = 0;i < gBCellsStart;i++)
5509    AP[i] = cF[i] - p[i];
5510   /*end*/
5511  } else if(Controls::Solver == Controls::SOR) {
5512   /*Asynchronous SOR solver*/
5513   p = cF;
5514   ForwardSweep(cF,M.Su);
5515   for(i = 0;i < gBCellsStart;i++)
5516    AP[i] = cF[i] - p[i];
5517   /*end*/
5518  } else if(M.flags & M.SYMMETRIC) {
5519   /*conjugate gradient*/
```

```
5520    EXCHANGE(p);
5521    AP = M * p;
5522    Tdot(p,AP,oo_rr);
5523    SUM_ALL(type,oo_rr);
5524    alpha = sdiv(o_rr , oo_rr);
5525    Taxpy(cF,cF,p,alpha);
5526    Taxpy(r,r,AP,-alpha);
5527    precondition(r,AP);
5528    oo_rr = o_rr;
5529    Tdot(r,AP,o_rr);
5530    SUM_ALL(type,o_rr);
5531    beta = sdiv(o_rr , oo_rr);
5532    Taxpy(p,AP,p,beta);
5533    /*end*/
5534   } else {
5535    /* biconjugate gradient*/
5536    EXCHANGE(p);
5537    EXCHANGE(p1);
5538    AP = M * p;
5539    AP1 = M ^ p1;
5540    Tdot(p1,AP,oo_rr);
5541    SUM_ALL(type,oo_rr);
5542    alpha = sdiv(o_rr , oo_rr);
5543    Taxpy(cF,cF,p,alpha);
5544    Taxpy(r,r,AP,-alpha);
5545    Taxpy(r1,r1,AP1,-alpha);
5546    precondition(r,AP);
5547    preconditionT(r1,AP1);
5548    oo_rr = o_rr;
5549    Tdot(r1,AP,o_rr);
5550    SUM_ALL(type,o_rr);
5551    beta = sdiv(o_rr , oo_rr);
5552    Taxpy(p,AP,p,beta);
5553    Taxpy(p1,AP1,p1,beta);
5554    /*end*/
5555   }
5556   /* ******************************************
5557   * calculate norm of residual & check convergence
5558   * ******************************************/
5559   EXCHANGE(cF);
5560   res = getResidual(AP,cF,sync);
```

```
5561   if(res <= Controls::tolerance
5562    || iterations == Controls::max_iterations)
5563    converged = true;
5564 PROBE:
5565   /* ************************************************************
5566    * Update ghost cell values. Communication is NOT forced on
5567    * every iteration,rather a non-blocking probe is used to
5568    * process messages as they arrive.
5569    ************************************************************/
5570   if(!sync)
5571   {
5572    int source,message_id;
5573    /*probe*/
5574    while(MP::iprobe(source,message_id)) {
5575     /*find the boundary*/
5576     Int patchi;
5577     for(patchi = 0;patchi < gInterMesh.size();patchi++) {
5578      if(gInterMesh[patchi].to == source)
5579       break;
5580     }
5581     interBoundary& b = gInterMesh[patchi];
5582     /*parse message*/
5583     if(message_id == MP::FIELD) {
5584      IntVector& f = *(b.f);
5585      /*recieve*/
5586      MP::recieve(&buffer[0],f.size(),source,message_id);
5587      forEach(f,j)
5588       cF[gFN[f[j]]] = buffer[j];
5589      /*Re-calculate residual.*/
5590      CALC_RESID();
5591      if(res > Controls::tolerance
5592       && iterations < Controls::max_iterations)
5593       converged = false;
5594      /* For communication to continue, processor have to send back
5595       * something for every message recieved.*/
5596      if(converged) {
5597       /*send END marker*/
5598       if(!sent_end[patchi]) {
5599        MP::send(source,MP::END);
5600        sent_end[patchi] = true;
5601       }
```

```
5602      } else {
5603       /*send back our part*/
5604       forEach(f,j)
5605        buffer[j] = cF[gFO[f[j]]];
5606       MP::send(&buffer[0],f.size(),source,message_id);
5607      }
5608     } else if(message_id == MP::END) {
5609      /*END marker recieved*/
5610      MP::recieve(source,message_id);
5611      end_count--;
5612      if(!sent_end[patchi]) {
5613       MP::send(source,MP::END);
5614       sent_end[patchi] = true;
5615      }
5616     }
5617    }
5618   }
5619   /* ****************************************
5620   * Wait untill all partner processors send us
5621   * an END message i.e. until end_count = 0.
5622   * ****************************************/
5623   if(converged) {
5624    if(end_count > 0) goto PROBE;
5625    else break;
5626   }
5627   /********
5628    * end
5629    ********/
5630  }
5631
5632  /*solver info*/
5633  if(print)
5634   MP::print("Iterations %d Initial Residual "
5635   "%.5e Final Residual %.5e\n",iterations,ires,res);
5636
5637  /*barrier*/
5638  MP::barrier();
5639
5640  /*update boundary conditons*/
5641  updateExplicitBCs(cF);
5642 }
```

```
5643 /***************************
5644  * Explicit instantiations
5645  ***************************/
5646 void Solve(const MeshMatrix<Scalar>& A) {
5647  applyImplicitBCs(A);
5648  SolveT(A);
5649 }
5650 void Solve(const MeshMatrix<Vector>& A) {
5651  applyImplicitBCs(A);
5652  SolveT(A);
5653 }
5654 void Solve(const MeshMatrix<STensor>& A) {
5655  applyImplicitBCs(A);
5656  SolveT(A);
5657 }
5658 void Solve(const MeshMatrix<Tensor>& A) {
5659  applyImplicitBCs(A);
5660  SolveT(A);
5661 }
5662 /* *********************
5663  *       End
5664  * *******************/
5665 #include "field.h"
5666 #include "turbulence.h"
5667 #include "mp.h"
5668 #include "system.h"
5669 #include "solve.h"
5670
5671 using namespace std;
5672
5673 /*general properties*/
5674 namespace GENERAL {
5675  Scalar density = 1;
5676  Scalar viscosity = 1e-5;
5677  Scalar conductivity = 1e-4;
5678  Vector gravity = Vector(0,0,-9.81);
5679
5680  void enroll(Util::ParamList& params) {
5681   params.enroll("rho",&density);
5682   params.enroll("viscosity",&viscosity);
5683   params.enroll("conductivity",&conductivity);
```

```
5684   params.enroll("gravity",&gravity);
5685  }
5686 };
5687 
5688 /*solvers*/
5689 void piso(istream&);
5690 void diffusion(istream&);
5691 void potential(istream&);
5692 void transport(istream&);
5693 void walldist(istream&);
5694 
5695 /**
5696 \verbatim
5697  Main application entry point for different solvers.
5698  \endverbatim
5699  */
5700 int main(int argc,char* argv[]) {
5701 
5702  /*message passing object*/
5703  MP mp(argc,argv);
5704  ifstream input(argv[1]);
5705 
5706  /*main options*/
5707  Util::ParamList params("general");
5708  string sname;
5709  params.enroll("solver",&sname);
5710  params.enroll("mesh",&Mesh::gMeshName);
5711  Mesh::enroll(params);
5712  GENERAL::enroll(params);
5713  params.read(input);
5714 
5715  /*Mesh*/
5716  if(mp.n_hosts > 1) {
5717   stringstream s;
5718   s << Mesh::gMeshName << mp.host_id;
5719   if(!System::cd(s.str()))
5720    return 1;
5721  }
5722  Mesh::readMesh();
5723  Mesh::initGeomMeshFields();
5724  atexit(Util::cleanup);
```

```
5725
5726  /*call solver*/
5727  if(!Util::compare(sname,"piso")) {
5728   piso(input);
5729  } else if(!Util::compare(sname,"diffusion")) {
5730   diffusion(input);
5731  } else if(!Util::compare(sname,"transport")) {
5732   transport(input);
5733  } else if(!Util::compare(sname,"potential")) {
5734   potential(input);
5735  } else if(!Util::compare(sname,"walldist")) {
5736   walldist(input);
5737  }
5738
5739  return 0;
5740 }
5741 /**
5742  Iteration object that does common book keeping stuff
5743  for all solvers.
5744  */
5745 class Iteration {
5746 private:
5747  Int starti;
5748  Int endi;
5749  Int i;
5750  Int n_deferred;
5751  Int idf;
5752 public:
5753  Iteration() {
5754   Int step = Controls::start_step / Controls::write_interval;
5755   starti = Controls::write_interval * step + 1;
5756   endi = Controls::end_step;
5757   n_deferred = Controls::n_deferred;
5758   i = starti;
5759   idf = 0;
5760
5761   Mesh::read_fields(step);
5762   Mesh::getProbeCells(Mesh::probeCells);
5763   forEachField(initTimeSeries());
5764  }
5765  bool start() {
```

```
5766   return (i == starti);
5767  }
5768  bool end() {
5769   if(i > endi)
5770    return true;
5771   /*iteration number*/
5772   if(MP::host_id == 0) {
5773    if(Controls::state == Controls::STEADY)
5774     MP::printH("Step %d\n",i);
5775    else
5776     MP::printH("Time %f\n",i * Controls::dt);
5777   }
5778   return false;
5779  }
5780  void next() {
5781   idf++;
5782   if(idf <= n_deferred)
5783    return;
5784
5785   /*update time series*/
5786   forEachField(updateTimeSeries(i));
5787
5788   /*write result to file*/
5789   if((i % Controls::write_interval) == 0) {
5790    Int step = i / Controls::write_interval;
5791    Mesh::write_fields(step);
5792   }
5793
5794   /*increment*/
5795   i++;
5796  }
5797  ~Iteration() {
5798  }
5799  static Int get_start() {
5800   return Controls::start_step / Controls::write_interval;
5801  }
5802  static Int get_end() {
5803   return Controls::end_step / Controls::write_interval;
5804  }
5805 };
5806 /**
```

```
5807 \verbatim
5808 Navier stokes solver using PISO algorithm
5809 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
5810 References:
5811 Hrvoje Jasak, "Error analysis and estimation of FVM with
5812 applications to fluid flow".
5813 Description:
5814    The PISO algorithm is used to solve NS equations on collocated grids
5815 using Rhie-Chow interpolation to avoid wiggles in pressure field.
5816
5817 Prediction
5818 ~~~~~~~~~~
5819 Discretize and solve the momenum equation with current values of pressure.
5820 The velocities obtained will not satisfy continuity unless exact pressure
5821 happened to be specified.
5822
5823 Correction
5824 ~~~~~~~~~~
5825 Step 1)
5826   Determine velocity with all terms included except pressure gradient source
          contribution.
5827       ap * Up = H(U) - grad(p)
5828    Up = H(U) / ap - grad(p) / ap
5829      Droping grad(p) term:
5830         Ua = H(U) / ap
5831   One jacobi sweep is done to find Ua.
5832    Step 2)
5833      Solve poisson pressure equation to satisfy continuity with fluxes calculated
5834   from interpolated Ua.
5835       div(Up) = 0
5836    div(1/ap * grad(p)) = div(H(U)/ap)
5837    lap(p,1/ap) = div(Ua)
5838    Step 3)
5839   Correct the velocity with gradient of newly found pressure
5840       U -= grad(p)
5841    These steps are repeated two or more times for transient solutions.
5842 For steady state problems once is enough.
5843 \endverbatim
5844 */
5845 void piso(istream& input) {
5846  /*Solver specific parameters*/
```

```
5847 Scalar& rho = GENERAL::density;
5848 Scalar& viscosity = GENERAL::viscosity;
5849 Scalar velocity_UR = Scalar(0.8);
5850 Scalar pressure_UR = Scalar(0.5);
5851 Int n_PISO = 1;
5852 Int n_ORTHO = 0;
5853
5854 /*piso options*/
5855 Util::ParamList params("piso");
5856 params.enroll("velocity_UR",&velocity_UR);
5857 params.enroll("pressure_UR",&pressure_UR);
5858 params.enroll("n_PISO",&n_PISO);
5859 params.enroll("n_ORTHO",&n_ORTHO);
5860
5861 VectorCellField U("U",READWRITE);
5862 ScalarCellField p("p",READWRITE);
5863
5864 /*turbulence model*/
5865 ScalarFacetField F;
5866 bool Steady;
5867 Turbulence_Model::RegisterTable(params);
5868 params.read(input);
5869 Turbulence_Model* turb =
5870  Turbulence_Model::Select(U,F,rho,viscosity,Steady);
5871 turb->enroll();
5872
5873 /*read parameters*/
5874 Util::read_params(input);
5875
5876 /*wall distance*/
5877 if(turb->needWallDist())
5878  Mesh::calc_walldist(Iteration::get_start());
5879
5880 /*time*/
5881 Scalar time_factor = Controls::time_scheme_factor;
5882 Steady = (Controls::state == Controls::STEADY);
5883
5884 /*Calculate for each time step*/
5885 Iteration it;
5886 ScalarCellField po = p;
5887 VectorCellField gP = -gradV(p);
```

```
5888 F = flx(rho * U);
5889
5890 for(;!it.end();it.next()) {
5891  /*Form Navier-stokes equation*/
5892  VectorMeshMatrix M;
5893
5894  /*convection*/
5895  {
5896   ScalarFacetField mu = rho * viscosity;
5897   M = div(U,F,mu);
5898  }
5899
5900  /*viscous/turbulent stress*/
5901  turb->addTurbulentStress(M);
5902
5903  /*relax if steady state otherwise add time contribution*/
5904  if(Steady)
5905   M.Relax(velocity_UR);
5906  else {
5907   /*crank nicolson*/
5908   if(!equal(time_factor,1)) {
5909    VectorCellField po = M * U;
5910    M *= time_factor;
5911    M.Su -= (1 - time_factor) * po;
5912   }
5913   /*time derivative*/
5914   M += ddt(U,rho);
5915  }
5916
5917  /*solve momentum equation*/
5918  Solve(M == gP);
5919
5920  /*1/ap*/
5921  ScalarCellField api = (1 / M.ap);
5922  fillBCs(api,true);
5923  ScalarCellField rmu = rho * api * Mesh::cV;
5924
5925  /*PISO loop*/
5926  for(Int j = 0;j < n_PISO;j++) {
5927   /* Ua = H(U) / ap*/
5928   U = getRHS(M) * api;
```

```
5929    updateExplicitBCs(U,true);
5930
5931    /*solve pressure poisson equation to satisfy continuity*/
5932    {
5933     ScalarCellField rhs = div(rho * U);
5934     for(Int k = 0;k <= n_ORTHO;k++)
5935      Solve(lap(p,rmu) += rhs);
5936    }
5937
5938    /*explicit velocity correction : add pressure contribution*/
5939    gP = -gradV(p);
5940    U -= gP * api;
5941    updateExplicitBCs(U,true);
5942   }
5943
5944   /*update fluctuations*/
5945   updateExplicitBCs(U,true,true);
5946   F = flx(rho * U);
5947
5948   /*solve turbulence transport equations*/
5949   turb->solve();
5950
5951   /*explicitly under relax pressure*/
5952   if(Steady) {
5953    p.Relax(po,pressure_UR);
5954    gP = -gradV(p);
5955    po = p;
5956   }
5957  }
5958
5959  /*write calculated turbulence fields*/
5960  if(turb->writeStress) {
5961   ScalarCellField K("Ksgs",WRITE);
5962   STensorCellField R("Rsgs",WRITE);
5963   STensorCellField V("Vsgs",WRITE);
5964   K = turb->getK();
5965   R = turb->getReynoldsStress();
5966   V = turb->getViscousStress();
5967   Mesh::write_fields(Iteration::get_end());
5968  }
5969 }
```

```
5970 /**
5971  \verbatim
5972  Diffusion solver
5973  ~~~~~~~~~~~~~~~~
5974  Solver for pdes of parabolic heat equation type:
5975       d(rho*u)/dt = lap(T,rho*DT)
5976  \endverbatim
5977 */
5978 void diffusion(istream& input) {
5979  /*Solver specific parameters*/
5980  Scalar& rho = GENERAL::density;
5981  Scalar DT = Scalar(1);
5982  Scalar t_UR = Scalar(1);
5983 
5984  /*diffusion*/
5985  Util::ParamList params("diffusion");
5986  params.enroll("DT",&DT);
5987  params.enroll("t_UR",&t_UR);
5988 
5989  ScalarCellField T("T",READWRITE);
5990 
5991  /*read parameters*/
5992  Util::read_params(input);
5993 
5994  /*time*/
5995  Scalar time_factor = Controls::time_scheme_factor;
5996  bool Steady = (Controls::state == Controls::STEADY);
5997 
5998  /*Calculate for each time step*/
5999  ScalarFacetField mu = rho * DT;
6000 
6001  for(Iteration it;!it.end();it.next()) {
6002   ScalarMeshMatrix M;
6003 
6004   M = -lap(T,mu);
6005 
6006   if(Steady)
6007    M.Relax(t_UR);
6008   else {
6009    if(!equal(time_factor,1)) {
6010     ScalarCellField po = M * T;
```

```
6011     M *= time_factor;
6012     M.Su -= (1 - time_factor) * po;
6013    }
6014    M += ddt(T,rho);
6015   }
6016
6017   Solve(M);
6018  }
6019 }
6020 /**
6021   \verbatim
6022   Transport equation solver
6023   ~~~~~~~~~~~~~~~~~~~~~~~~~
6024   Given a flow field (U) and values of a scalar at the boundaries,
6025   the solver determines the distribution of the scalar.
6026      dT/dt + div(T,F,mu) = lap(T,mu)
6027   \endverbatim
6028  */
6029 void transport(istream& input) {
6030  /*Solver specific parameters*/
6031  Scalar& rho = GENERAL::density;
6032  Scalar DT = Scalar(4e-2);
6033  Scalar t_UR = Scalar(1);
6034
6035  /*transport*/
6036  Util::ParamList params("transport");
6037  params.enroll("DT",&DT);
6038  params.enroll("t_UR",&t_UR);
6039
6040  VectorCellField U("U",READWRITE);
6041  ScalarCellField T("T",READWRITE);
6042
6043  /*read parameters*/
6044  Util::read_params(input);
6045
6046  /*time*/
6047  Scalar time_factor = Controls::time_scheme_factor;
6048  bool Steady = (Controls::state == Controls::STEADY);
6049
6050  /*Calculate for each time step*/
6051  ScalarFacetField F,mu = rho * DT,gamma;
```

```
6052
6053  for(Iteration it;!it.end();it.next()) {
6054   ScalarMeshMatrix M;
6055
6056   F = flx(rho * U);
6057   M = div(T,F,mu)
6058    - lap(T,mu);
6059
6060   if(Steady)
6061    M.Relax(t_UR);
6062   else {
6063    if(!equal(time_factor,1)) {
6064     ScalarCellField po = M * T;
6065     M *= time_factor;
6066     M.Su -= (1 - time_factor) * po;
6067    }
6068    M += ddt(T,rho);
6069   }
6070   Solve(M);
6071  }
6072 }
6073 /**
6074   \verbatim
6075     Potential flow solver
6076  ~~~~~~~~~~~~~~~~~~~~~
6077  In potential flow the velocity field is irrotational (vorticity = curl(U) = 0).
6078  This assumption fails for boundary layers and wakes that exhibit strong vorticity,
6079  but it can still be used to initialize the flow field for further simulations.
6080
6081  For incompressible flow
6082        div(U) = 0
6083     Velocity is the gradient of velocity potential phi
6084        U = grad(phi)
6085      div(grad(phi)) = 0
6086      lap(phi) = 0
6087     phi is pressure for this solver. The initial flow field will inevitably not
          satisfy
6088  continuity due to imposed boundary conditons. Therefore we solve a pressure poisson
6089  equation and then correct the velocity with the gradient of p.
6090        lap(p) = div(U)
6091      U -= grad(p)
```

```
6092   \endverbatim
6093 */
6094 void potential(istream& input) {
6095  /*Solver specific parameters*/
6096  Int n_ORTHO = 0;
6097 
6098  /*potential*/
6099  Util::ParamList params("potential");
6100  params.enroll("n_ORTHO",&n_ORTHO);
6101 
6102  VectorCellField U("U",READWRITE);
6103  ScalarCellField p("p",READ);
6104 
6105  /*read parameters*/
6106  Util::read_params(input);
6107 
6108  /*set internal field to zero*/
6109  for(Int i = 0;i < Mesh::gBCellsStart;i++) {
6110   U[i] = Vector(0,0,0);
6111   p[i] = Scalar(0);
6112  }
6113  updateExplicitBCs(U,true);
6114  updateExplicitBCs(p,true);
6115 
6116  for(Iteration it;it.start();it.next()) {
6117   /*solve potential equation*/
6118   ScalarCellField divU = div(U);
6119   ScalarFacetField one = Scalar(1);
6120   for(Int k = 0;k <= n_ORTHO;k++)
6121    Solve(lap(p,one) == divU);
6122 
6123   /*correct velocity*/
6124   U -= grad(p);
6125   updateExplicitBCs(U,true);
6126  }
6127 }
6128 /**
6129  \verbatim
6130  Wall distance
6131  ~~~~~~~~~~~~~
6132  Reference:
```

```
6133    D.B.Spalding, Calculation of turbulent heat transfer in cluttered spaces
6134    Description:
6135    Poisson equation is solved to get approximate nearest wall distance.
6136          lap(phi,1) = -cV
6137    The boundary conditions are phi=0 at walls, and grad(phi) = 0 elsewhere.
6138  \endverbatim
6139 */
6140 void walldist(istream& input) {
6141  /*Solver specific parameters*/
6142  Int n_ORTHO = 0;
6143 
6144  /*walldist options*/
6145  Util::ParamList params("walldist");
6146  params.enroll("n_ORTHO",&n_ORTHO);
6147  Util::read_params(input);
6148 
6149  /*solve*/
6150  Mesh::calc_walldist(Iteration::get_start(),n_ORTHO);
6151 }
6152 void Mesh::calc_walldist(Int step,Int n_ORTHO) {
6153  ScalarCellField& phi = yWall;
6154    /*poisson equation*/
6155  {
6156   ScalarFacetField one = Scalar(1);
6157   for(Int k = 0;k <= n_ORTHO;k++)
6158    Solve(lap(phi,one) == -cV);
6159  }
6160  /*wall distance*/
6161  {
6162   VectorCellField g = grad(phi);
6163   yWall = sqrt((g & g) + 2 * phi) - mag(g);
6164  }
6165  /*write it*/
6166  yWall.write(step);
6167 }
6168 #include <cuda.h>
6169 #include "solve.h"
6170 
6171 /*number of threads in a block*/
6172 static const Int nThreads = 128;
6173 
```

```
6174 /*Matrix vector multiply*/
6175 template <class T>
6176 __global__
6177 void cudaMul(const Int* const rows,
6178     const Int* const cols,
6179     const Scalar* const an,
6180     const Int N,
6181     const T* const x,
6182     T* y
6183     ) {
6184  Int i = blockIdx.x * blockDim.x + threadIdx.x;
6185  if (i < N) {
6186   const Int start = rows[i];
6187   const Int end = rows[i + 1];
6188   T res = an[start] * x[cols[start]];
6189 
6190   for (Int j = start + 1; j < end; j++)
6191    res -= an[j] * x[cols[j]];
6192   y[i] = res;
6193  }
6194 }
6195 /*jacobi solver*/
6196 template<class T>
6197 __global__
6198 void cudaJacobi(const Int* const rows,
6199     const Int* const cols,
6200     const Scalar* const an,
6201     const T* const cF,
6202     T* const cF1,
6203     const T* const Su,
6204     T* r,
6205     const Int N,
6206     Scalar omega
6207     ) {
6208  Int i = blockIdx.x * blockDim.x + threadIdx.x;
6209  if (i < N) {
6210   const Int start = rows[i];
6211   const Int end = rows[i + 1];
6212   T res = Su[i], val = cF[i];
6213 
6214   for (Int j = start + 1; j < end; j++)
```

```
6215    res += an[j] * cF[cols[j]];
6216   res /= an[start];
6217
6218   r[i] = -val;
6219   val *= (1 - omega);
6220   val += res * (omega);
6221   r[i] += val;
6222   cF1[i] = val;
6223  }
6224 }
6225 /*Taxpy*/
6226 template<class T,class T1>
6227 __global__
6228 void cudaTaxpy(const Int N,
6229       const T1 alpha,
6230       const T* const x,
6231       const T* const y,
6232       T* const z
6233         ) {
6234  Int i = blockIdx.x * blockDim.x + threadIdx.x;
6235  if (i < N) {
6236   T temp;
6237   temp = x[i];
6238   temp *= alpha;
6239   temp += y[i];
6240   z[i] = temp;
6241  }
6242 }
6243 /*Txmy*/
6244 template<class T,class T1>
6245 __global__
6246 void cudaTxmy(const Int N,
6247      const T* const x,
6248      const T1* const y,
6249      T* const z
6250        ) {
6251  Int i = blockIdx.x * blockDim.x + threadIdx.x;
6252  if (i < N) {
6253   T temp;
6254   temp = x[i];
6255   temp *= y[i];
```

```
  z[i] = temp;
 }
}
/*Tdot*/
template <class T>
__global__
void Tdot(const T* const a,
    const T* const b,
    T* const c,
    const Int N
    ) {
    __shared__ T cache[nThreads];
    Int tid = threadIdx.x + blockIdx.x * blockDim.x;
    Int cacheIndex = threadIdx.x;

    T   temp = T(0),val;
    while (tid < N) {
  val = a[tid];
  val *= b[tid];
        temp += val;
        tid += blockDim.x * gridDim.x;
    }
    cache[cacheIndex] = temp;

    __syncthreads();

    Int i = blockDim.x / 2;
    while (i != 0) {
        if (cacheIndex < i)
            cache[cacheIndex] += cache[cacheIndex + i];
        __syncthreads();
        i /= 2;
    }

    if (cacheIndex == 0)
        c[blockIdx.x] = cache[0];
}
template<class T>
__host__
T cudaTdot(T* x,
     T* y,
```

```
6297      T* d_sum,
6298      T* sum,
6299      const Int nBlocks32,
6300      const Int N
6301      ) {
6302     Tdot <<< nBlocks32, nThreads >>> (x,y,d_sum,N);
6303  cudaMemcpy(sum,d_sum,nBlocks32 * sizeof(T),cudaMemcpyDeviceToHost);
6304  T c = T(0);
6305     for (Int i = 0; i < nBlocks32; i++)
6306         c += sum[i];
6307  return c;
6308 }
6309 /***********************************************
6310  * Template class to solve equations on GPU
6311  *  Solver must do many iterations to compensate
6312  *  for the latency caused by copying matrix
6313  *  from host to device.
6314  ***********************************************/
6315 template<class T>
6316 __host__
6317 void SolveT(const MeshMatrix<T>& M) {
6318  const Int N = Mesh::gBCellsStart;
6319  const Int Nall = M.ap.size();
6320  const Int nBlocks = (N + nThreads - 1) / nThreads;
6321  const Int nBlocks32 = ((nBlocks > 32) ? 32 : nBlocks);
6322
6323  //info
6324  if(M.flags & M.SYMMETRIC)
6325   MP::printH("Symmetric : ");
6326  else
6327   MP::printH("Asymmetric : ");
6328  if(Controls::Solver == Controls::SOR)
6329   MP::print("SOR :");
6330  else
6331   MP::print("PCG :");
6332
6333  /*******************************
6334   *  variables on host & device
6335   *******************************/
6336  Int*  d_rows;
6337  Int*  d_cols;
```

```
6338 Scalar* d_an;
6339 Scalar* d_anT;
6340 Scalar* d_pC;
6341 T*      d_cF;
6342 T*      d_Su;
6343 //PCG
6344 T*      d_r,*d_r1;
6345 T*      d_p,*d_p1,*d_AP,*d_AP1;
6346 T       alpha,beta,o_rr,oo_rr;
6347 T       local_res[2];
6348 //reduction
6349 T*      sum,*d_sum;
6350
6351 /*********************************
6352  * allocate memory on device
6353  *********************************/
6354 {
6355  CSRMatrix<T> A(M);
6356  cudaMalloc((void**) &d_rows,A.rows.size() * sizeof(Int));
6357  cudaMalloc((void**) &d_cols,A.cols.size() * sizeof(Int));
6358  cudaMalloc((void**) &d_an, A.an.size() * sizeof(Scalar));
6359  cudaMalloc((void**) &d_cF, Nall * sizeof(T));
6360  cudaMalloc((void**) &d_Su, Nall * sizeof(T));
6361
6362  cudaMemcpy(d_rows ,&A.rows[0] ,A.rows.size() * sizeof(Int), cudaMemcpyHostToDevice);
6363  cudaMemcpy(d_cols ,&A.cols[0] ,A.cols.size() * sizeof(Int), cudaMemcpyHostToDevice);
6364  cudaMemcpy(d_an ,&A.an[0] ,A.an.size() * sizeof(Scalar), cudaMemcpyHostToDevice);
6365  cudaMemcpy(d_cF ,&A.cF[0] ,Nall * sizeof(T),  cudaMemcpyHostToDevice);
6366  cudaMemcpy(d_Su ,&A.Su[0] ,Nall * sizeof(T),  cudaMemcpyHostToDevice);
6367
6368  cudaMalloc((void**) &d_r, Nall * sizeof(T));
6369  cudaMalloc((void**) &d_sum, nBlocks32 * sizeof(T));
6370  sum = (T*) malloc(nBlocks32 * sizeof(T));
6371
6372  if(Controls::Solver == Controls::SOR) {
6373   cudaMalloc((void**) &d_AP,Nall * sizeof(T));
6374   cudaMemcpy( d_AP,d_cF,Nall * sizeof(T),cudaMemcpyDeviceToDevice);
6375  } else if(Controls::Solver == Controls::PCG) {
6376   cudaMalloc((void**) &d_p, Nall * sizeof(T));
6377   cudaMalloc((void**) &d_AP, Nall * sizeof(T));
6378   {
```

```
6379    ScalarCellField pC = 1./M.ap;
6380    cudaMalloc((void**) &d_pC,N * sizeof(Scalar));
6381    cudaMemcpy(d_pC,&pC[0],N * sizeof(Scalar),cudaMemcpyHostToDevice);
6382   }
6383   if(!(M.flags & M.SYMMETRIC)) {
6384    cudaMalloc((void**) &d_r1, Nall * sizeof(T));
6385    cudaMalloc((void**) &d_p1, Nall * sizeof(T));
6386    cudaMalloc((void**) &d_AP1, Nall * sizeof(T));
6387    cudaMalloc((void**) &d_anT,A.anT.size() * sizeof(Scalar));
6388    cudaMemcpy(d_anT,&A.anT[0],A.anT.size() * sizeof(Scalar), cudaMemcpyHostToDevice);
6389   }
6390  }
6391 }
6392
6393 /*CG*/
6394 if(Controls::Solver == Controls::PCG) {
6395  cudaMemset(d_r,0,Nall * sizeof(T));
6396  cudaMemset(d_p,0,Nall * sizeof(T));
6397  cudaMul  <<< nBlocks, nThreads >>> (d_rows,d_cols,d_an,N,d_cF,d_AP);
6398  cudaTaxpy <<< nBlocks, nThreads >>> (N,Scalar(-1),d_AP,d_Su,d_r);
6399  cudaTxmy <<< nBlocks, nThreads >>> (N,d_r,d_pC,d_p);
6400  o_rr = cudaTdot(d_r,d_p,d_sum,sum,nBlocks32,N);
6401 }
6402 /*BiCG*/
6403 if(!(M.flags & M.SYMMETRIC) && (Controls::Solver == Controls::PCG)) {
6404  cudaMemcpy(d_r1,d_r,Nall * sizeof(T), cudaMemcpyDeviceToDevice);
6405  cudaMemcpy(d_p1,d_p,Nall * sizeof(T), cudaMemcpyDeviceToDevice);
6406 }
6407 //iterate until convergence
6408 Scalar res = 0;
6409 Int iterations = 0;
6410
6411 /* *************************
6412  * Iterative solvers
6413  * *************************/
6414 while(iterations < Controls::max_iterations) {
6415       /*counter*/
6416  iterations++;
6417
6418  /*select solver*/
6419  if(Controls::Solver == Controls::SOR) {
```

```
6420   iterations++;
6421   cudaJacobi <<< nBlocks, nThreads >>> (d_rows,d_cols,d_an,d_cF,d_AP,d_Su,d_r,N,
           Controls::SOR_omega);
6422   cudaJacobi <<< nBlocks, nThreads >>> (d_rows,d_cols,d_an,d_AP,d_cF,d_Su,d_r,N,
           Controls::SOR_omega);
6423  } else if(M.flags & M.SYMMETRIC) {
6424   /*conjugate gradient : from wiki*/
6425   cudaMul <<< nBlocks, nThreads >>> (d_rows,d_cols,d_an,N,d_p,d_AP);
6426   oo_rr = cudaTdot(d_p,d_AP,d_sum,sum,nBlocks32,N);
6427   alpha = sdiv(o_rr , oo_rr);
6428   cudaTaxpy <<< nBlocks, nThreads >>> (N,alpha,d_p,d_cF,d_cF);
6429   cudaTaxpy <<< nBlocks, nThreads >>> (N,-alpha,d_AP,d_r,d_r);
6430   oo_rr = o_rr;
6431   cudaTxmy <<< nBlocks, nThreads >>> (N,d_r,d_pC,d_AP);
6432   o_rr = cudaTdot(d_r,d_AP,d_sum,sum,nBlocks32,N);
6433   beta = sdiv(o_rr , oo_rr);
6434   cudaTaxpy <<< nBlocks, nThreads >>> (N,beta,d_p,d_AP,d_p);
6435   /*end*/
6436  } else {
6437   /* biconjugate gradient : from wiki */
6438   cudaMul <<< nBlocks, nThreads >>> (d_rows,d_cols,d_an,N,d_p,d_AP);
6439   cudaMul <<< nBlocks, nThreads >>> (d_rows,d_cols,d_anT,N,d_p1,d_AP1);
6440   oo_rr = cudaTdot(d_p1,d_AP,d_sum,sum,nBlocks32,N);
6441   alpha = sdiv(o_rr , oo_rr);
6442   cudaTaxpy <<< nBlocks, nThreads >>> (N,alpha,d_p,d_cF,d_cF);
6443   cudaTaxpy <<< nBlocks, nThreads >>> (N,-alpha,d_AP,d_r,d_r);
6444   cudaTaxpy <<< nBlocks, nThreads >>> (N,-alpha,d_AP1,d_r1,d_r1);
6445   oo_rr = o_rr;
6446   cudaTxmy <<< nBlocks, nThreads >>> (N,d_r,d_pC,d_AP);
6447   cudaTxmy <<< nBlocks, nThreads >>> (N,d_r1,d_pC,d_AP1);
6448   o_rr = cudaTdot(d_r1,d_AP,d_sum,sum,nBlocks32,N);
6449   beta = sdiv(o_rr , oo_rr);
6450   cudaTaxpy <<< nBlocks, nThreads >>> (N,beta,d_p,d_AP,d_p);
6451   cudaTaxpy <<< nBlocks, nThreads >>> (N,beta,d_p1,d_AP1,d_p1);
6452  }
6453
6454  /* ******************************************
6455  * calculate norm of residual & check convergence
6456  * ******************************************/
6457  local_res[0] = cudaTdot(d_r,d_r,d_sum,sum,nBlocks32,N);
6458  local_res[1] = cudaTdot(d_cF,d_cF,d_sum,sum,nBlocks32,N);
```

```
6459  res = sqrt(mag(local_res[0]) / mag(local_res[1]));
6460
6461  /*check convergence*/
6462  if(res <= Controls::tolerance)
6463   break;
6464 }
6465
6466 /*****************************
6467  *  Copy result back to cpu
6468  *****************************/
6469    //copy result
6470 cudaMemcpy(&((*M.cF)[0]), d_cF, N * sizeof(T), cudaMemcpyDeviceToHost);
6471
6472 //update boundary conditons
6473 updateExplicitBCs(*M.cF);
6474
6475 //info
6476 MP::print("Iterations %d Residue: %.5e\n",iterations,res);
6477 /*********************************
6478  * free device memory
6479  *********************************/
6480 {
6481  cudaFree(d_rows);
6482  cudaFree(d_cols);
6483  cudaFree(d_an);
6484  cudaFree(d_cF);
6485  cudaFree(d_Su);
6486
6487  cudaFree(d_r);
6488  cudaFree(d_sum);
6489  free(sum);
6490
6491  if(Controls::Solver == Controls::SOR) {
6492   cudaFree(d_AP);
6493  } else if(Controls::Solver == Controls::PCG) {
6494   cudaFree(d_p);
6495   cudaFree(d_AP);
6496   cudaFree(d_pC);
6497   if(!(M.flags & M.SYMMETRIC)) {
6498    cudaFree(d_r1);
6499    cudaFree(d_p1);
```

```
    cudaFree(d_AP1);
    cudaFree(d_anT);
   }
  }
 }
 /******************
  *    END
  ******************/
}

/***************************
 * Explicit instantiations
 ***************************
void Solve(const MeshMatrix<Scalar>& A) {
 applyImplicitBCs(A);
 SolveT(A);
}
void Solve(const MeshMatrix<Vector>& A) {
 applyImplicitBCs(A);
 SolveT(A);
}
void Solve(const MeshMatrix<STensor>& A) {
 applyImplicitBCs(A);
 SolveT(A);
}
void Solve(const MeshMatrix<Tensor>& A) {
 applyImplicitBCs(A);
 SolveT(A);
}
/* ********************
 *       End
 * ********************/
#ifndef __TURBULENCE_H
#define __TURBULENCE_H

#include "field.h"
#include "solve.h"
/**
  \verbatim
  Description of RANS turbulence models
  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
6541  Navier Stokes without source term:
6542     d(rho*u)/dt + div(rho*uu) = -grad(p) + div(mu*gu)
6543  RANS:
6544     d(rho*U)/dt + div(rho*UU) + div(rho*u'u') = -grad(P) + div(mu*gU)
6545     d(rho*U)/dt + div(rho*UU) = -grad(P) + div(mu*gU) - div(rho*u'u')
6546     d(rho*U)/dt + div(rho*UU) = -grad(P) + div(V + R)
6547  where Viscous (V) and Reynolds (R) stress tensors are
6548   V = mu*gU
6549     R = -rho*u'u'
6550  Boussinesq model for R:
6551     Traceless(R) = 2 * emu * Traceless(S)
6552   where S = (gU + gUt) / 2
6553     R - R_ii/3 = 2 * emu * (S - S_ii/3)
6554     R = 2 * emu * (S - S_ii/3) + R_ii/3
6555       = 2 * emu * ((gU + gUt)/2 - gU_ii/3) + R_ii/3
6556       = emu * gU + emu * (gUt - 2/3*gUt_ii) + R_ii/3
6557       = emu * gU + emu * dev(gUt,2) - 2/3*rho*k*I
6558  Viscous and Reynolds stress together:
6559     V + R = {mu * gU} + {emu * gU + emu * dev(gUt,2) - 2/3*rho*k*I}
6560           = (mu + emu) * gU + emu * dev(gUt,2) - 2/3*rho*k*I
6561           = ( eff_mu ) * gU + emu * dev(gUt,2) - 2/3*rho*k*I
6562  Volume integrated V+R i.e force:
6563     div(V + R) = div(eff_mu*gU) + div(emu * dev(gUt,2)) - div(2/3*rho*k*I)
6564                    Implicit        Explicit       Absored in pressure
6565                                                   p_m = p + 2/3*k*rho
6566  Final RANS equation after substituting div(V+R):
6567   d(rho*U)/dt + div(rho*UU) = -grad(P) + div(V + R)
6568     d(rho*U)/dt + div(rho*UU) = -grad(P_m) + div(eff_mu*gU) + div(emu * dev(gUt,2))
6569  Since the k term is absorbed into the pressure gradient, we only need models for
6570  turbulent diffusivity emu.
6571
6572     Base turbulence model:
6573        This default class has no turbulence model so it is a laminar solver.
6574        Only the viscous stress V is added to the NS equations. Turbulence models
6575        derived from this class add a model for Reynold's stress R usually by solving
6576        some turbulence transport equations.
6577     \endverbatim
6578 */
6579 struct Turbulence_Model {
6580
6581  VectorCellField& U;
```

```
6582 ScalarFacetField& F;
6583 Scalar& rho;
6584 Scalar& nu;
6585 bool& Steady;
6586
6587 Util::ParamList params;
6588 bool writeStress;
6589 /*constructor*/
6590 Turbulence_Model(VectorCellField& tU,ScalarFacetField& tF,Scalar& trho,Scalar& tnu,
         bool& tSteady) :
6591  U(tU),
6592  F(tF),
6593  rho(trho),
6594  nu(tnu),
6595  Steady(tSteady),
6596  writeStress(false),
6597  params("turbulence")
6598 {
6599 }
6600 /*overridable functions*/
6601 virtual void enroll() {
6602  using namespace Util;
6603  Option* op = new BoolOption(&writeStress);
6604  params.enroll("writeStress",op);
6605 };
6606 virtual void solve() {};
6607 virtual void addTurbulentStress(VectorMeshMatrix& M) {
6608  ScalarFacetField mu = rho * nu;
6609  M -= lap(U,mu);
6610 };
6611 /* V */
6612 STensorCellField getViscousStress() {
6613  STensorCellField V = 2 * rho * nu * sym(grad(U));
6614  return V;
6615 }
6616 /* R */
6617 virtual STensorCellField getReynoldsStress() {
6618  return STensor(0);
6619 }
6620 /* TKE */
6621 virtual ScalarCellField getK() {
```

```
6622   return Scalar(0);
6623  }
6624  /* Turbulence model selection */
6625  static Int turb_model;
6626  static bool bneedWallDist;
6627  static bool needWallDist() { return bneedWallDist;}
6628  static void RegisterTable(Util::ParamList& params);
6629  static Turbulence_Model* Select(VectorCellField& U,ScalarFacetField& F,
6630   Scalar& rho,Scalar& nu,bool& Steady);
6631 };
6632 /**
6633  * Eddy viscosity models based on Boussinesq's assumption
6634  * that the action of Reynolds and Viscous stress are similar.
6635  */
6636 struct EddyViscosity_Model : public Turbulence_Model {
6637  ScalarCellField eddy_mu;
6638  enum Model {
6639   SMAGORNSKY,BALDWIN,KATO
6640  };
6641  enum WallModel {
6642   NONE,STANDARD,LAUNDER
6643  };
6644  Model modelType;
6645  WallModel wallModel;
6646
6647  /*constructor*/
6648  EddyViscosity_Model(VectorCellField& tU,ScalarFacetField& tF,Scalar& trho,Scalar& tnu
         ,bool& tSteady) :
6649   Turbulence_Model(tU,tF,trho,tnu,tSteady),
6650   eddy_mu("emu",READWRITE),
6651   modelType(SMAGORNSKY),
6652   wallModel(STANDARD)
6653  {
6654  }
6655  /*Register options*/
6656  virtual void enroll() {
6657   using namespace Util;
6658   Option* op = new Option(&modelType,3,
6659    "SMAGORNSKY","BALDWIN","KATO");
6660   params.enroll("modelType",op);
6661   Turbulence_Model::enroll();
```

```
6662 }
6663 /*eddy_mu*/
6664 virtual void calcEddyViscosity(const TensorCellField& gradU) = 0;
6665
6666 /* V + R */
6667 virtual void addTurbulentStress(VectorMeshMatrix& M) {
6668  TensorCellField gradU = grad(U);
6669  calcEddyViscosity(gradU);
6670  setWallEddyMu();
6671  fillBCs(eddy_mu);
6672
6673  ScalarCellField eff_mu = eddy_mu + rho * nu;
6674  M -= lap(U,eff_mu);
6675  M -= div(eddy_mu * dev(trn(gradU),2));
6676 };
6677 /* R */
6678 virtual STensorCellField getReynoldsStress() {
6679  STensorCellField R = 2 * eddy_mu * dev(sym(grad(U))) -
6680       STensorCellField(Constants::I_ST) * (2 * rho * getK() / 3);
6681  return R;
6682 }
6683 /* S2 */
6684 ScalarCellField getS2(const TensorCellField& gradU) {
6685  ScalarCellField magS;
6686  if(modelType == SMAGORNSKY) {
6687   STensorCellField S = sym(gradU);
6688   magS = S & S;
6689  } else if(modelType == BALDWIN) {
6690   TensorCellField O = skw(gradU);
6691   magS = O & O;
6692  } else {
6693   STensorCellField S = sym(gradU);
6694   TensorCellField O = skw(gradU);
6695   magS = sqrt((S & S) * (O & O));
6696  }
6697  return (2 * magS);
6698 }
6699 /*Fix near wall cell values*/
6700 void FixNearWallValues(ScalarMeshMatrix& M) {
6701  using namespace Mesh;
6702  BasicBCondition* bbc;
```

```
6703   forEach(AllBConditions,d) {
6704    bbc = AllBConditions[d];
6705    if(bbc->fIndex == eddy_mu.fIndex && bbc->cIndex == Mesh::ROUGHWALL) {
6706     IntVector& wall_faces = *bbc->bdry;
6707     if(wall_faces.size()) {
6708      Int f,c1;
6709      forEach(wall_faces,i) {
6710       f = wall_faces[i];
6711       c1 = gFO[f];
6712       M.Fix(c1,(*M.cF)[c1]);
6713      }
6714     }
6715    }
6716   }
6717  }
6718  /* Wall functions */
6719  void setWallEddyMu() {
6720   using namespace Mesh;
6721   BasicBCondition* bbc;
6722   forEach(AllBConditions,d) {
6723    bbc = AllBConditions[d];
6724    if(bbc->fIndex == eddy_mu.fIndex && bbc->cIndex == Mesh::ROUGHWALL) {
6725     IntVector& wall_faces = *bbc->bdry;
6726     LawOfWall& low = bbc->low;
6727     if(wall_faces.size()) {
6728      forEach(wall_faces,i) {
6729       applyWallFunction(wall_faces[i],low);
6730      }
6731     }
6732    }
6733   }
6734  }
6735  /*overridable*/
6736  virtual void applyWallFunction(Int f,LawOfWall& low) = 0;
6737 };
6738 /**
6739  * Base two equation K-X turbulence model
6740  */
6741 struct KX_Model : public EddyViscosity_Model {
6742  /*model coefficients*/
6743  Scalar Cmu;
```

```
6744 Scalar SigmaK;
6745 Scalar SigmaX;
6746 Scalar C1x;
6747 Scalar C2x;
6748
6749 Scalar k_UR;
6750 Scalar x_UR;
6751
6752 /*turbulence fields*/
6753 ScalarCellField k;
6754 ScalarCellField x;
6755 ScalarCellField Pk;
6756
6757 /*constructor*/
6758 KX_Model(VectorCellField& tU,ScalarFacetField& tF,Scalar& trho,Scalar& tnu,bool&
         tSteady,const char* xname) :
6759  EddyViscosity_Model(tU,tF,trho,tnu,tSteady),
6760  k_UR(0.7),
6761  x_UR(0.7),
6762  k("k",READWRITE),
6763  x(xname,READWRITE)
6764 {
6765  wallModel = LAUNDER;
6766 }
6767 /*TKE*/
6768 virtual ScalarCellField getK() { return k; }
6769 /*Register options*/
6770 virtual void enroll() {
6771  using namespace Util;
6772  params.enroll("k_UR",&k_UR);
6773  params.enroll("x_UR",&x_UR);
6774  EddyViscosity_Model::enroll();
6775 }
6776 /* k-x model specific over-ridables*/
6777 virtual void calcEddyMu() = 0;
6778 virtual Scalar calcX(Scalar ustar,Scalar kappa,Scalar y) = 0;
6779 virtual Scalar getCmu(Int i) {
6780  return Cmu;
6781 }
6782 /* eddy viscosity*/
6783 virtual void calcEddyViscosity(const TensorCellField& gradU) {
```

```
6784   calcEddyMu();
6785   Pk = getS2(gradU) * eddy_mu;
6786  }
6787  /* wall function */
6788  virtual void applyWallFunction(Int f,LawOfWall& low) {
6789   using namespace Mesh;
6790   Int c1 = gFO[f];
6791   Int c2 = gFN[f];
6792
6793   /*calc ustar*/
6794   Scalar ustar;
6795   Scalar y = mag(unit(fN[f]) & (cC[c1] - cC[c2]));
6796   if(wallModel == STANDARD) {
6797    ustar = low.getUstar(nu,mag(U[c1]),y);
6798    k[c1] = pow(ustar,2) / sqrt(getCmu(c1));
6799   } else if(wallModel == LAUNDER) {
6800    ustar = pow(getCmu(c1),Scalar(0.25)) * sqrt(k[c1]);
6801   }
6802   x[c1] = calcX(ustar,low.kappa,y);
6803
6804   /* calculate eddy viscosity*/
6805   Scalar yp = (ustar * y) / nu;
6806   Scalar up = low.getUp(ustar,nu,yp);
6807   eddy_mu[c1] = (rho * nu) * (yp / up - 1);
6808
6809   /* turbulence generation and dissipation */
6810   if(wallModel == LAUNDER) {
6811    Scalar mag_dudy = mag((U[c2] - U[c1]) / y);
6812    Scalar mag_dudy_log = ustar / (low.kappa * y);
6813    Pk[c1] = (mag_dudy * mag_dudy_log) * eddy_mu[c1];
6814   }
6815  };
6816 };
6817
6818 #endif
6819 #ifndef __MIXING_LENGTH_H
6820 #define __MIXING_LENGTH_H
6821
6822 #include "turbulence.h"
6823
6824 struct MixingLength_Model : public EddyViscosity_Model {
```

```
6825  /*model coefficients*/
6826  Scalar mixingLength;
6827  Scalar C;
6828  Int wallDamping;
6829
6830  /*mixing length field*/
6831  ScalarCellField lm;
6832  Scalar kappa;
6833
6834  /*constructor*/
6835  MixingLength_Model(VectorCellField&,ScalarFacetField&,Scalar&,Scalar&,bool&);
6836
6837  /*others*/
6838  virtual void enroll();
6839  virtual void calcEddyViscosity(const TensorCellField& gradU);
6840  virtual void applyWallFunction(Int f,LawOfWall& low);
6841  virtual ScalarCellField getK();
6842  virtual void calcLengthScale() {
6843   lm = mixingLength;
6844  }
6845 };
6846
6847 #endif
6848 #include "mixing_length.h"
6849 /**
6850 \verbatim
6851 References:
6852     Book by Pope pg. 369
6853 Description:
6854  Velocity and time scales are modelled as:
6855   l* = lm
6856   u* = lm * |S|
6857   eddy_nu = u*l*
6858           = (lm^2) * |S|
6859     Generalization of the mixing length model for 3D flows:
6860     by Smagorinsky (1963).
6861     eddy_nu = (lm^2) * sqrt(2 * (S & S))
6862     by Baldwin & Lomax (1978)
6863     eddy_nu = (lm^2) * sqrt(2 * (O & O))
6864  The turbulent kinetic energy k can be approximated by equating turbulent
6865  viscosity eddy_nu with the one from Prandtl/Smagorinsky one equation models.
```

```
6866     u* = C * k^1/2
6867   eddy_nu = C * k^1/2 * lm
6868  Equating with the above eqn yields
6869     k = (lm / C)^2 * (2 * (S & S))
6870
6871  For high-Re flows, the mixing length close to the wall is set :
6872     lm = kappa * y_wall
6873  Thus for Smagornsky LES model
6874     lm = min(Cs * Delta, kappa * y_wall)
6875 \endverbatim
6876 */
6877 MixingLength_Model::MixingLength_Model(VectorCellField& tU,ScalarFacetField& tF,Scalar
        & trho,Scalar& tnu,bool& tSteady) :
6878  EddyViscosity_Model(tU,tF,trho,tnu,tSteady),
6879  mixingLength(0),
6880  C(0.55),
6881  kappa(0.41),
6882  wallDamping(1)
6883 {
6884 }
6885 void MixingLength_Model::enroll() {
6886  using namespace Util;
6887  params.enroll("mixing_length",&mixingLength);
6888  Option* op = new BoolOption(&wallDamping);
6889  params.enroll("wall_damping",op);
6890  params.enroll("kappa",&kappa);
6891  params.enroll("C",&C);
6892  EddyViscosity_Model::enroll();
6893 }
6894 ScalarCellField MixingLength_Model::getK() {
6895  return pow(lm / C,2.0) * getS2(grad(U));
6896 }
6897 void MixingLength_Model::calcEddyViscosity(const TensorCellField& gradU) {
6898  calcLengthScale();
6899  if(wallDamping)
6900   lm = min(kappa * Mesh::yWall,lm);
6901  eddy_mu = rho * pow(lm,Scalar(2)) * sqrt(getS2(gradU));
6902 }
6903 void MixingLength_Model::applyWallFunction(Int f,LawOfWall& low) {
6904  using namespace Mesh;
6905  Int c1 = gFO[f];
```

```
 Int c2 = gFN[f];

 /*calc ustar*/
 Scalar ustar = 0.0;
 Scalar y = mag(unit(fN[f]) & (cC[c1] - cC[c2]));
 if(wallModel == STANDARD)
  ustar = low.getUstar(nu,mag(U[c1]),y);

 /* calculate eddy viscosity*/
 Scalar yp = (ustar * y) / nu;
 Scalar up = low.getUp(ustar,nu,yp);
 eddy_mu[c1] = (rho * nu) * (yp / up - 1);
}


#ifndef __KE_H
#define __KE_H

#include "turbulence.h"

struct KE_Model : public KX_Model {
 /*constructor*/
 KE_Model(VectorCellField&,ScalarFacetField&,Scalar&,Scalar&,bool&);

 /*others*/
 virtual void enroll();
 virtual void solve();
 virtual void calcEddyMu() {
  eddy_mu = (rho * Cmu * k * k) / x;
 };
 virtual Scalar calcX(Scalar ustar,Scalar kappa,Scalar y) {
  return pow(ustar,Scalar(3)) / (kappa * y);
 }
};

#endif
#include "ke.h"
/*
References:
   http://www.cfd-online.com/Wiki/Standard_k-epsilon_model
*/
```

```
6947 KE_Model::KE_Model(VectorCellField& tU,ScalarFacetField& tF,Scalar& trho,Scalar& tnu,
         bool& tSteady) :
6948  KX_Model(tU,tF,trho,tnu,tSteady,"e")
6949 {
6950  Cmu = 0.09;
6951  SigmaK = 1;
6952  SigmaX = 1.314;
6953  C1x = 1.44;
6954  C2x = 1.92;
6955 }
6956 void KE_Model::enroll() {
6957  using namespace Util;
6958  KX_Model::enroll();
6959  params.enroll("Cmu",&Cmu);
6960  params.enroll("SigmaK",&SigmaK);
6961  params.enroll("SigmaE",&SigmaX);
6962  params.enroll("C1e",&C1x);
6963  params.enroll("C2e",&C2x);
6964 }
6965 void KE_Model::solve() {
6966  ScalarMeshMatrix M;
6967  ScalarFacetField mu;
6968
6969  /*turbulent dissipation*/
6970  mu = cds(eddy_mu) / SigmaX + rho * nu;
6971  M = div(x,F,mu)
6972   - lap(x,mu);
6973  M -= src(x,
6974   (C1x * Pk * x / k),                  //Su
6975   -(C2x * rho * x / k)                 //Sp
6976   );
6977  if(Steady)
6978   M.Relax(x_UR);
6979  else
6980   M += ddt(x,rho);
6981  FixNearWallValues(M);
6982  Solve(M);
6983  x = max(x,Constants::MachineEpsilon);
6984
6985  /*turbulent kinetic energy*/
6986  mu = cds(eddy_mu) / SigmaK + rho * nu;
```

```
 M = div(k,F,mu)
  - lap(k,mu);
 M -= src(k,
  Pk,                                   //Su
  -(rho * x / k)                        //Sp
  );
 if(Steady)
  M.Relax(k_UR);
 else
  M += ddt(k,rho);
 if(wallModel == STANDARD)
  FixNearWallValues(M);
 Solve(M);
 k = max(k,Constants::MachineEpsilon);
}
#ifndef __KW_H
#define __KW_H

#include "turbulence.h"

struct KW_Model : public KX_Model {
 /*constructor*/
 KW_Model(VectorCellField&,ScalarFacetField&,Scalar&,Scalar&,bool&);

 /*others*/
 virtual void enroll();
 virtual void solve();
 virtual void calcEddyMu() {
  eddy_mu = (rho * k) / x;
 };
 virtual Scalar calcX(Scalar ustar,Scalar kappa,Scalar y) {
  return ustar / (kappa * y * sqrt(Cmu));
 }
};

#endif
#include "kw.h"
/*
References:
    http://www.cfd-online.com/Wiki/Wilcox%27s_k-omega_model
*/
```

```
7028 KW_Model::KW_Model(VectorCellField& tU,ScalarFacetField& tF,Scalar& trho,Scalar& tnu,
         bool& tSteady) :
7029  KX_Model(tU,tF,trho,tnu,tSteady,"w")
7030 {
7031  Cmu = 0.09;
7032  SigmaK = 2;
7033  SigmaX = 2;
7034  C1x = 5./9;
7035  C2x = 3./40;
7036 }
7037 void KW_Model::enroll() {
7038  using namespace Util;
7039  KX_Model::enroll();
7040  params.enroll("Cmu",&Cmu);
7041  params.enroll("SigmaK",&SigmaK);
7042  params.enroll("SigmaW",&SigmaX);
7043  params.enroll("C1w",&C1x);
7044  params.enroll("C2w",&C2x);
7045 }
7046 void KW_Model::solve() {
7047  ScalarMeshMatrix M;
7048  ScalarFacetField mu;
7049
7050  /*turbulent dissipation*/
7051  mu = cds(eddy_mu) / SigmaX + rho * nu;;
7052  M = div(x,F,mu)
7053   - lap(x,mu);
7054  M -= src(x,
7055   (C1x * Pk * x / k),                  //Su
7056   -(C2x * x * rho)                     //Sp
7057   );
7058  if(Steady)
7059   M.Relax(x_UR);
7060  else
7061   M += ddt(x,rho);
7062  FixNearWallValues(M);
7063  Solve(M);
7064  x = max(x,Constants::MachineEpsilon);
7065
7066  /*turbulent kinetic energy*/
7067  mu = cds(eddy_mu) / SigmaK + rho * nu;;
```

```
 M = div(k,F,mu)
  - lap(k,mu);
 M -= src(k,
  Pk,                                   //Su
  -(Cmu * x * rho)                      //Sp
  );
 if(Steady)
  M.Relax(k_UR);
 else
  M += ddt(k,rho);
 if(wallModel == STANDARD)
  FixNearWallValues(M);
 Solve(M);
 k = max(k,Constants::MachineEpsilon);
}
#ifndef __LES_H
#define __LES_H

#include "mixing_length.h"

struct LES_Model : public MixingLength_Model {
 /*model coefficients*/
 Scalar Cs;

 /*constructor*/
 LES_Model(VectorCellField&,ScalarFacetField&,Scalar&,Scalar&,bool&);

 /*others*/
 virtual void enroll();
 virtual void calcLengthScale();
};

#endif
#include "les.h"
/*
References:
   http://www.cfd-online.com/Wiki/Smagorinsky-Lilly_model
*/
LES_Model::LES_Model(VectorCellField& tU,ScalarFacetField& tF,Scalar& trho,Scalar& tnu
    ,bool& tSteady) :
 MixingLength_Model(tU,tF,trho,tnu,tSteady),
```

```
 Cs(0.11)
{
}
void LES_Model::enroll() {
 params.enroll("Cs",&Cs);
 MixingLength_Model::enroll();
}
void LES_Model::calcLengthScale() {
 ScalarCellField delta = pow(Mesh::cV,Scalar(1./3));
 lm = Cs * delta;
}


#ifndef __REALIZABLEKE_H
#define __REALIZABLEKE_H

#include "turbulence.h"

struct REALIZABLE_KE_Model : public KX_Model {
 /*model coefficients*/
 ScalarCellField CmuF;
 ScalarCellField C1;
 ScalarCellField magS;
 Scalar A0;

 /*constructor*/
 REALIZABLE_KE_Model(VectorCellField&,ScalarFacetField&,Scalar&,Scalar&,bool&);

 /*others*/
 virtual void enroll();
 virtual void solve();
 virtual void calcEddyMu() {
  eddy_mu = (rho * CmuF * k * k) / x;
 };
 virtual Scalar calcX(Scalar ustar,Scalar kappa,Scalar y) {
  return pow(ustar,Scalar(3)) / (kappa * y);
 }
 virtual Scalar getCmu(Int i) {
  return CmuF[i];
 }
 virtual void calcEddyViscosity(const TensorCellField& gradU);
```

```
};

#endif
#include "realizableke.h"

/*
References:
   http://www.cfd-online.com/Wiki/Realisable_k-epsilon_model
 http://www.laturbolenza.com/?p=92
*/
REALIZABLE_KE_Model::REALIZABLE_KE_Model(VectorCellField& tU,ScalarFacetField& tF,
    Scalar& trho,Scalar& tnu,bool& tSteady) :
 KX_Model(tU,tF,trho,tnu,tSteady,"e"),
 CmuF(0.09),
 A0(4.04)
{
 SigmaK = 1.0;
 SigmaX = 1.2;
 C2x = 1.9;
}
void REALIZABLE_KE_Model::enroll() {
 using namespace Util;
 KX_Model::enroll();
 params.enroll("SigmaK",&SigmaK);
 params.enroll("SigmaE",&SigmaX);
 params.enroll("C2e",&C2x);
}
void REALIZABLE_KE_Model::calcEddyViscosity(const TensorCellField& gradU) {
 /*calculate CmuF*/
 STensorCellField S = sym(gradU);
 {
  TensorCellField O = skw(gradU);
  ScalarCellField Ustar = sqrt((S & S) + (O & O));
  ScalarCellField Sbar = sqrt(S & S);
  ScalarCellField W = ((mul(S,S) & S) / pow(Sbar,3.0)) * sqrt(6.0);
  W = min(max(W,-1.0),1.0);
  ScalarCellField As = sqrt(6.0) * cos(acos(W) / 3.0);
  CmuF = 1.0 / (A0 + As * Ustar * k / x);
  CmuF = min(CmuF,0.09);
 }
 /*calculate C1*/
```

```
7189  magS = sqrt((S & S) * 2.0);
7190  {
7191   ScalarCellField eta = magS * (k / x);
7192   C1 = max(eta/(eta + 5.0),0.43);
7193  }
7194  /*calculate viscosity*/
7195  KX_Model::calcEddyViscosity(gradU);
7196 }
7197 void REALIZABLE_KE_Model::solve() {
7198  ScalarMeshMatrix M;
7199  ScalarFacetField mu;
7200
7201  /*turbulent dissipation*/
7202  mu = cds(eddy_mu) / SigmaX + rho * nu;
7203  M = div(x,F,mu)
7204   - lap(x,mu);
7205  M -= src(x,
7206   (C1 * rho * magS * x),               //Su
7207   -(C2x * rho * x / (k + sqrt(nu * x))) //Sp
7208   );
7209  if(Steady)
7210   M.Relax(x_UR);
7211  else
7212   M += ddt(x,rho);
7213  FixNearWallValues(M);
7214  Solve(M);
7215  x = max(x,Constants::MachineEpsilon);
7216
7217  /*turbulent kinetic energy*/
7218  mu = cds(eddy_mu) / SigmaK + rho * nu;
7219  M = div(k,F,mu)
7220   - lap(k,mu);
7221  M -= src(k,
7222   Pk,                                  //Su
7223   -(rho * x / k)                       //Sp
7224   );
7225  if(Steady)
7226   M.Relax(k_UR);
7227  else
7228   M += ddt(k,rho);
7229  if(wallModel == STANDARD)
```

```
 FixNearWallValues(M);
 Solve(M);
 k = max(k,Constants::MachineEpsilon);
}
#ifndef __RNG_KE_H
#define __RNG_KE_H

#include "ke.h"

struct RNG_KE_Model : public KE_Model {
 /*model coefficients*/
 Scalar eta0;
 Scalar beta;
 /*calculate C2eStar*/
 ScalarCellField C2eStar;

 /*constructor*/
 RNG_KE_Model(VectorCellField&,ScalarFacetField&,Scalar&,Scalar&,bool&);

 virtual void enroll();
 virtual void solve();
 virtual void calcEddyViscosity(const TensorCellField& gradU);
};

#endif
#include "rngke.h"
/*
References:
   http://www.cfd-online.com/Wiki/RNG_k-epsilon_model
*/
RNG_KE_Model::RNG_KE_Model(VectorCellField& tU,ScalarFacetField& tF,Scalar& trho,
    Scalar& tnu,bool& tSteady) :
 KE_Model(tU,tF,trho,tnu,tSteady),
 eta0(4.38),
 beta(0.012)
{
 Cmu = 0.0845;
 SigmaK = 0.7194;
 SigmaX = 0.7194;
 C1x = 1.42;
 C2x = 1.68;
```

```
7270 }
7271 void RNG_KE_Model::enroll() {
7272  using namespace Util;
7273  KE_Model::enroll();
7274  params.enroll("eta0",&eta0);
7275  params.enroll("beta",&beta);
7276 }
7277 void RNG_KE_Model::calcEddyViscosity(const TensorCellField& gradU) {
7278  /*calculate C2eStar*/
7279  {
7280   ScalarCellField eta = sqrt(getS2(gradU)) * (k / x);
7281   C2eStar = C2x + Cmu * pow(eta,3.0) * (1 - eta / eta0) /
7282    (1 + beta * pow(eta,3.0));
7283   C2eStar = max(C2eStar,0.0);
7284  }
7285  /*calculate viscosity*/
7286  KE_Model::calcEddyViscosity(gradU);
7287 }
7288 void RNG_KE_Model::solve() {
7289  ScalarMeshMatrix M;
7290  ScalarFacetField mu;
7291 
7292  /*turbulent dissipation*/
7293  mu = cds(eddy_mu) / SigmaX + rho * nu;;
7294  M = div(x,F,mu)
7295   - lap(x,mu);
7296  M -= src(x,
7297   (C1x * Pk * x / k),                  //Su
7298   -(C2eStar * rho * x / k)             //Sp
7299   );
7300  if(Steady)
7301   M.Relax(x_UR);
7302  else
7303   M += ddt(x,rho);
7304  FixNearWallValues(M);
7305  Solve(M);
7306  x = max(x,Constants::MachineEpsilon);
7307 
7308  /*turbulent kinetic energy*/
7309  mu = cds(eddy_mu) / SigmaK + rho * nu;
7310  M = div(k,F,mu)
```

```
7311   - lap(k,mu);
7312  M -= src(k,
7313   Pk,                                  //Su
7314   -(rho * x / k)                       //Sp
7315   );
7316  if(Steady)
7317   M.Relax(k_UR);
7318  else
7319   M += ddt(k,rho);
7320  if(wallModel == STANDARD)
7321   FixNearWallValues(M);
7322  Solve(M);
7323  k = max(k,Constants::MachineEpsilon);
7324 }
```

Curriculum Vitae

Name:	Daniel Abdi
Post-Secondary Education and Degrees:	Addis Ababa University, Addis Ababa, Ethiopia 1998 - 2003 B.Sc
	Indian Institute of Technology, India, Roorkee 2004 - 2006 M.Tech.
	Florida International University, Miami , FL 2009 - 2012
	University of Western Ontario, London, ON 2012 - 2013 Ph.D.
Honours and Awards:	Chi Epsilon 2010-2012
Related Work Experience:	Research Assistant at Florida International University 2009 - 2012
	Reasearch Assistant at The University of Western Ontario 2012 - 2013

Publications:

- D. Abdi and G. Bitsuamlak. (2013), *Asynchronous parallelization of CFD solver*, Computers and Fluids (**submitted**)
- D. Abdi and G. Bitsuamlak. (2013), *Numerical evaluation of the effect of multiple roughness changes*, Wind and Structures (**submitted**)
- D. Abdi and G. Bitsuamlak. (2013), *Effect of turbulence models on wind simulations in complex terrain*, Journal of Wind and Industrial Aero (**submitted**)

- D. Abdi and G. Bitsuamlak. (2013), *Development of computational tools for large scale wind simulations*, ATC-SEI Advances in Hurricane Engineering Conference
- D. Abdi and G. Bitsuamlak. (2012), *Assessing the effect of boundary conditions on simulating atmospheric boundary layer*, 2012 Joint Conference EMI/PMC.
- D. Abdi and G. Bitsuamlak. (2010), *Estimation of surface roughness using CFD*, The Fifth International Symposium on Computational Wind Engineering (CWE2010).
- D. Abdi, S. Levin, and G. Bitsuamlak (2009), *Application of an artificial neural network model for boundary layer wind tunnel profile development*, 11th Americas conference on wind engineering.
- D.Abdi and G.Bitsuamlak. (2013), *Numerical evaluation of roughness effects using urban models of different complexity*, 2013, CWE2014 (**abstract submitted**)