1-13-2023 2:00 PM

A Computational Framework for Aerodynamic and Aeroelastic Modeling of Wind Loads on Tall Buildings

Abiy Fantaye Melaku, *The University of Western Ontario*

Supervisor: Bitsuamlak, Girma T., *The University of Western Ontario*

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

Abstract

Driven by the burgeoning growth of computing power over the last few decades, the capability of computational fluid dynamics (CFD) to simulate turbulent flows of practical interest has progressed rapidly. In the past, a notable research effort has been dedicated to applying CFD for modeling wind loads on structures, particularly for tall buildings. However, the current state of CFD for wind load evaluation of tall buildings using Large-Eddy Simulation (LES) has several critical challenges, including the treatment of atmospheric boundary layer (ABL) flow conditions, turbulence modeling of separated flows around buildings, and simulation of wind-structure interaction for dynamically sensitive buildings. For CFD to be a practically useful wind engineering tool, these challenges must be addressed adequately, meeting the rigors of the current wind engineering practice. This thesis presents the development of a CFD-based framework for accurate aerodynamic and aeroelastic modeling of tall buildings with the objective of overcoming these key limitations. The capabilities of the framework are demonstrated using a series of case studies.

The CFD-based framework is developed in three major phases. In the first phase, computationally efficient methods were developed for modeling the characteristics of the approaching ABL turbulence. A novel synthetic inflow turbulence generation method is proposed that satisfies two-point flow statistics coupled with an implicit ground roughness modeling technique to represent the local terrain effect. In the next phase of the framework, aerodynamic wind loads on tall buildings having different surrounding configurations are simulated and validated against wind tunnel results. Initially, the cladding and overall loads, as well as responses of an isolated standard tall building, are investigated. Then, the framework is applied to a more realistic case involving a complex-shaped tall building located in a city center. In the final phase of research, the capability of the framework is extended by implementing a high-fidelity fluid-structure interaction (FSI) procedure to model the aeroelastic response of tall buildings. The implemented FSI algorithm uses a partitioned approach that couples a transient fluid solver with a multi-degree-of-freedom model of the building. Then the FSI procedure is applied to simulate the aeroelastic response of a tall flexible building. Overall, comparing the results from each phase of the study with wind tunnel measurements showed an encouraging level of agreement. It is expected that the framework presented in this thesis is of practical importance to the wind-resistant design of tall buildings.

Keywords: Atmospheric boundary layer (ABL), wind loads, computational fluid dynamics (CFD), inflow turbulence generation, large-eddy simulation (LES), spectral representation method, tall building, wind-induced response, fluid-structure interaction (FSI), aeroelastic modeling, structural dynamics, computational efficiency, validation, open-source code, software implementation

Summary for Lay Audience

Over the last few decades, the power of computers has shown rapid growth, making it possible to simulate complex wind flows virtually using numerical models. This development attracted interest in wind-induced loads on structures, particularly tall buildings, that are more susceptible to wind effects. However, computational modeling of wind load on tall buildings accurately has several critical challenges. To mention some, replicating the natural wind around the building site, capturing its complex behavior around buildings, and simulating its interaction with the motion of the building. To make computational methods practical design tools, these challenges must be resolved by adhering to the rigor of established experimental testing practices. This thesis presents the development of a high-fidelity computational framework for accurately modeling wind loads on tall buildings. The capabilities of the framework are demonstrated using a series of case studies.

The computational framework is developed in three major phases. In the first phase, a new technique was developed to mimic the natural wind approaching a building with minimal computational cost. In the next step of the framework, wind effects on tall buildings situated in different surrounding configurations are simulated and compared with experimental measurements. At this stage of development, the structure is modeled by neglecting its swaying motion. The primary wind effects simulated on the building are the pressure exerted on its façades and the cumulative force on the entire building structure. The final research phase expands the framework to model the two-way interaction between the building and the wind. The wind blows on the structure causing it to sway, and the tower disturbs the air around it while swinging. Such kind of simulation was achieved by integrating two computational entities: the wind flow model and the structural model of the building. Then, this technique is applied to simulate the swaying motion of a tall building under strong wind conditions. Finally, comparing the results from each phase of the study with experimental measurements showed an encouraging level of agreement. The framework presented in this thesis is expected to have practical relevance for designing tall buildings that can withstand wind effects.

Acknowledgments

I want to express my appreciation and sincere gratitude to my supervisor Prof. Girma Bitsuamlak for his continued support and guidance. A good supervisor shows the path, helps students when they face challenges and gives encouragement and critiques while pushing them towards their ultimate potential. Prof. Girma Bitsuamlak balanced all these roles in the most friendly way. I would also like to thank Prof. Gregory Kopp, Prof. Catherine Gorle, Dr. John Kilpatrick, and Dr. Jubayer Chowdhury for their constructive comment on this dissertation.

My appreciations go to my colleagues and all members of Prof. Girma Bitsuamlak's research group. I am especially indebted to Tsinuel Geleta, Anant Gairola and Kimberley Adamek for their unreserved support and proofreading of this thesis. The discussions with Dr. Tibebu Birhane were also valuable in this study. I would also like to acknowledge Christopher Howlett, Thomas Boekels, and Dr. Eric Lalonde for their assistance related to some of the experimental data used in this study. My appreciation also goes to Dr. Chieh-Hsun Wu for helping me conduct wind field measurements for validation. In addition, I would like to thank members of the Boundary Layer Wind Tunnel Laboratory at Western University, who assisted me in the wind tunnel tests. The inspiring conversations with Dr. Tadilo Bogale, Dr. Anwar Awol, and Dr. Meseret Kahsay are also unforgettable.

I would also like to thank the SharcNet and Digital Research Alliance of Canada for providing a high-performance computing resource used in this study with excellent technical support. I am very grateful for the financial support from the Government of Ontario-Trillium Scholarship.

Most importantly, I give my special heartfelt thanks to my parents, Fantaye Melaku and Almaz Nigatu, my sister Etagegnew Fantaye and the rest of my family members for their continued support and encouragement during this study.

Co-Authorship Statement

This thesis has been prepared in accordance with the regulations for an Integrated Article thesis format stipulated by the School of Graduate and Postdoctoral Studies at the University of Western Ontario and has been co-authored as:

From **Chapter 2**, "A divergence-free inflow turbulence generator using spectral representation method for large-eddy simulation of ABL flows" is published in *Journal of Wind Engineering and Industrial Aerodynamics* under the co-authorship of Abiy F. Melaku and Girma T. Bitsuamlak.

From **Chapter 3**, "Computationally efficient simulation of multivariate wind velocity field using a low-rank representation of the cross-power spectral density matrix" is submitted to *Journal of Engineering Mechanics* under the co-authorship of Abiy F. Melaku and Girma T. Bitsuamlak.

From **Chapter 4**, "LES for predicting wind loads and responses: prospect for wind-resistant tall building design" is submitted to *Journal of Wind Engineering and Industrial Aerodynamics* under the co-authorship of Abiy F. Melaku and Girma T. Bitsuamlak.

From **Chapter 5**, "Wall-modeled large-eddy simulation for evaluating wind loads on a tall building located in a city center: comparison with experimental data" is prepared for submission to *Engineering Structures* under the co-authorship of Abiy F. Melaku, Jeroen Janssen and Girma T. Bitsuamlak.

From **Chapter 6**, "windFSI: An open-source fluid-structure interaction framework for aeroelastic modeling of flexible structures" is prepared for submission to *Advances in Engineering Software* under the co-authorship of Abiy F. Melaku and Girma T. Bitsuamlak.

For the work presented in Chapter 5, the experimental Boundary Layer Wind Tunnel Laboratory data is provided by Thornton Tomasetti.

Contents

List of Figures

List of Tables

List of Appendices

List of Abbreviations

ABL	Atmospheric Boundary Layer
ASCE	American Society of Civil Engineers
BLWT	Boundary Layer Wind Tunnel
BLWTL	Boundary Layer Wind Tunnel Laboratory
CAARC	Commonwealth Advisory Aeronautical Research Council
CFD	Computational Fluid Dynamics
CPU	Central Processing Unit
CPSD	Cross-Power Spectral Density
CWE	Computational Wind engineering
CSS	Conventional Serial Staggered
CDRFG	Consistent Discrete Random Flow Generation
DES	Detached Eddy Simulation
DFSR	Divergence-free Spectral Representation Method
DOF	Degree of Freedom
DNS	Direct Numerical Simulation
ESDU	Engineering Science Data Unit
FEM	Finite Element Method
FFT	Fast Fourier Transform
FPI	Fixed-point Iteration
FVM	Finite Volume Method
HFPI	High-Frequency Pressure Integration
HPC	High-Performance Computing
LES	Large Eddy Simulation
MAPE	Mean Absolute Percentage Error
MDOF	Multi Degree Of Freedom
MPI	Message Passing Interface

OOP	Object-Oriented Programming
OpenFOAM	Open-Source Field Operation and Manipulation
OpenMP	Open Multi-Processing
PISO	Pressure Implicit with Splitting Operators
PIV	Particle Image Velocimetry
POD	Proper Orthogonal Decomposition
PSD	Power Spectral Density
RANS	Reynolds Averaged Navier-Stokes
Re	Reynolds number
RMS	Root Mean Square
SGS	Sub-grid Scale Stress
SRM	Spectral Representation Method
VBLWT	Virtual Boundary Layer Wind Tunnel

Chapter 1

Introduction

1.1 Background and motivation

In 2050, it is projected that more than two-thirds of the world population will be dwelling in urban areas (UNDESA, 2018). To accommodate this ever-growing urban population sustainably, designing cost-effective tall buildings satisfying structural strength and serviceability requirements is crucial. Partly driven by the ongoing rapid urbanization and complemented by the advancements in the use of lightweight construction materials, innovative structural systems, and design methods, the race toward new heights is making the current generations of tall buildings increasingly flexible. As a result, they are becoming highly vulnerable to wind-induced dynamic actions. The design of tall buildings for wind poses several wind engineering challenges, starting from accurately estimating the wind loads acting on the main structural system up to limiting the motions of the building to an acceptable level for the comfort of occupants. In the past, several research works have been dedicated to the development of different analytical and experimental methods for predicting dynamic loads on tall buildings. Wind-induced vibration of tall buildings is a highly complex problem that is not analytically tractable. This is primarily due to the difficulty of analytically treating complex turbulent flows around bluff bodies (Boggs, 1991). The gust response method originally proposed by Davenport (1961a, 1967) has been widely used for predicting the wind-induced response of structures. This method is capable of incorporating the dynamic properties of the structure and has been adopted by most of the present-day design codes. However, its application is primarily limited to along-wind response and fails to account for the across-wind and motion-induced (aeroelastic) effects. Given the limitation of the gust response method, wind tunnel testing is the only known way to directly determine the wind loads and the wind-induced response of tall buildings accurately (Irwin, 2009).

In recent years, however, due to the burgeoning growth of High-Performance Comput-

ing (HPC) and improved numerical methods, Computational Wind Engineering (CWE) tools, particularly Computational Fluid Dynamics (CFD), have shown remarkable potential for simulating wind loads on buildings. It is widely recognized that CFD offers several advantages over wind tunnel testing, including the ability to simulate full-scale conditions without any physical constraints. Due to the high versatility and shorter model development cycle, CFD can also virtually (if computing cost is not an issue) deliver results faster than experimental studies. Moreover, CFD provides a wealth of high-resolution relevant flow field data that helps understand important flow phenomena governing wind loads. Despite the aforementioned promises, the practical use of CFD for computational wind load evaluation still remains challenging. For example, for applications in environmental problems such as indoor flows, pollutant dispersion, and pedestrian-level winds, CFD is routinely utilized in practical design. For structural load prediction, however, the fact that peak quantities are much more important than mean flow values makes computational wind load evaluation a less forgiving task. Furthermore, when the structure is sufficiently sensitive to dynamic effects, wind load estimation using CFD normally requires the frequency content of the overall load to be accurately captured in addition to its magnitude. These and other contributing factors resulted in a lack of confidence in employing CFD for wind load prediction (Holmes, 2007; Irwin et al., 2013), which led to its slow adoption in the industry. Thus, it is imperative that significant research effort is still needed to develop and validate CFD-based wind load evaluation procedures tailored to the essential requirements of wind engineering practice.

The key challenges of wind load evaluation on tall buildings using CFD stem from three origins. First and foremost, considering that wind effects on structures are sensitive to the atmospheric boundary layer (ABL) turbulence, the CFD models must accurately reproduce the upstream ABL flow characteristics. Second, wind flow around tall buildings is a high Reynolds number flow characterized by peculiar aerodynamic features like impingement, separation, reattachment, recirculation, vortex shedding, etc. Thus, accurate wind load evaluation necessitates adequate resolution of these aerodynamic features using turbulence models of the desired fidelity. For transient wind load evaluation, this normally requires the use of high-fidelity turbulence models capable of handling complex unsteady turbulent flows such as large-eddy simulation (LES) (Dagnew and Bitsuamlak, 2013; Murakami, 1990; Tamura et al., 2008). This challenge is further exacerbated for tall buildings located in built-up areas where the effect of the surrounding buildings must also be modeled. The third challenge manifests itself in tall buildings that are dynamically sensitive and experience noticeable motion-induced (aeroelastic) effects. In such circumstances, the aerodynamic loads and the mechanical properties of the building are coupled, thus requiring a fluid-structure interaction (FSI) phenomenon to be aptly simulated numerically.

In addition to addressing these challenges, carefully instrumented wind tunnel testing is critically needed to validate the CFD models at each stage of development. The validation task needs to be carried out at different levels, including the approaching flow characteristics, surface pressure fluctuations, integrated forces, and the wind-induced responses of the structure. Also, to establish the necessary confidence, the desired quantities from the experiment and CFD must be compared thoroughly using one-point and two-point statistics of the measurements (Melaku and Bitsuamlak, 2021). The comparison, specially for surface pressure fluctuations, needs to be performed not only with lower-order statistics such as mean and root-mean-square as it is usually done in most computational wind evaluation studies but also using higher-order statistics like skewness and kurtosis (Dagnew and Bitsuamlak, 2013; Sagaut and Deck, 2009).

In this thesis, different aspects of the key challenges of CFD-based wind load evaluation mentioned above are addresses stage by stage with the objective of developing a high-fidelity computational framework for aerodynamic and aeroelastic modeling of wind loads on tall buildings. The thesis presents a collection of five stand-alone papers that collaborate toward this common objective. Figure 1.1 illustrates the main components of the proposed computational framework. The figure is inspired by the famous Alan Davenport's *Wind Loading Chain*. Moving from left to right, the first step in CFD-based wind load evaluation is specifying appropriate inflow boundary conditions characterizing the upcoming ABL turbulence. The next important step involves modeling the effect of ground roughness representative of the local terrain. After reproducing the approaching ABL flow, the aerodynamic wind loads are simulated using LES. Finally, the responses of the structure, including aeroelastic effects resulting from wind-structure interaction, are modeled by coupling a transient CFD solver with the FEM model of the structure.

1.2 Research objectives

With a clear need to improve the accuracy of numerical wind load evaluation on tall buildings, the main objective of this thesis is to develop a high-fidelity computational framework for aerodynamic and aeroelastic simulation of wind effects on tall buildings. The specific objectives of the research are:

1. Developing inflow turbulence generation and ground roughness modeling methods for large-eddy simulation of the ABL flows. The inflow generation method will be able to take wind profiles, velocity spectra, and two-point statistics of ABL turbulence as input to generate a realistic ABL turbulence for wind load evolution. Whereas the roughness

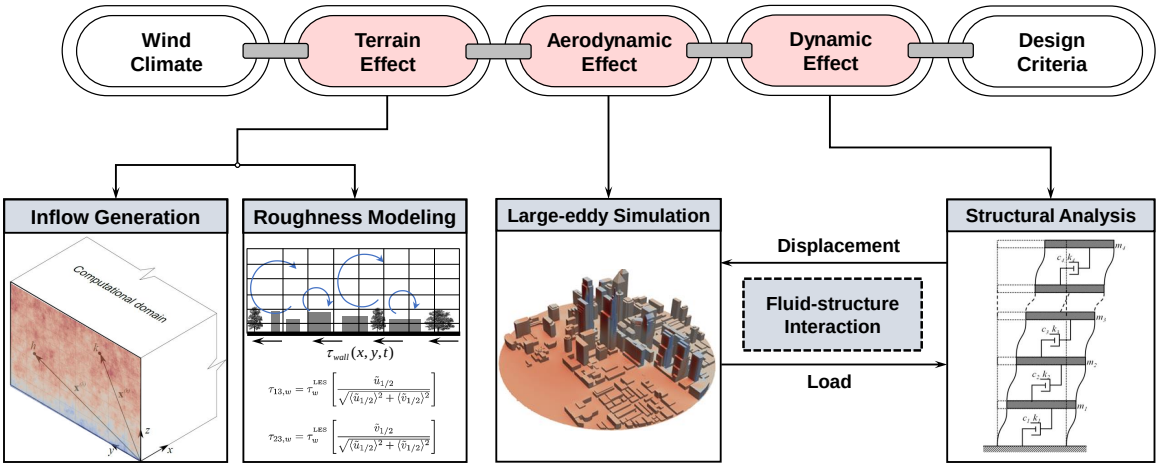


Figure 1.1: Important components of the computational framework demonstrated using Alan Davenport's *Wind Loading Chain*

modeling technique takes aerodynamic roughness length as an input parameter to model the effect of the local terrain.

2. Improving the computational speed of the inflow generation method for large-scale applications. Considering that high-fidelity LES of ABL flow requires generating inflow turbulence over a large number of grid points, the developed method needs to be computationally efficient.

3. Investigate the wind loads and responses of a tall building employing LES with the proposed inflow turbulence generation method. This must be first demonstrated using a generic tall building with an isolated configuration. The results from LES will be validated extensively against wind tunnel measurements.

4. Assessing the accuracy of LES for predicting wind loads on a tall building located in a city center with a realistic urban setup.

5. Developing a high-fidelity fluid-structure interaction framework for aeroelastic modeling of tall buildings. Specifically, the framework will implement a partitioned approach coupling OpenFOAM and modal structural solver. The accuracy and deficiency of different coupling algorithms for aeroelastic applications will be investigated with the help of numerical examples.

1.3 Outline of the thesis

This thesis is prepared based on the "Integrated-Article" format. The thesis contains a compilation of papers under review or published in peer-reviewed journals. Each paper addresses one of the five research objectives identified in the previous section. The research is pursued in three prominent themes, which are: (1) modeling of the approaching ABL turbulence, (2) LES-based wind load and response evaluation on tall buildings, and (3) fluid-structure interaction for aeroelastic modeling. Chapters 2 and 3 address the challenges related to modeling the approaching ABL turbulence. Chapters 4 and 5 present LES-based wind load valuation for tall buildings with isolated as well as complex urban surroundings. Chapters 6 propose a fluid-structure framework for computational aeroelastic modeling. Finally, Chapter 7 summarizes the main findings from the current research and provides recommendations for future research directions.

1.3.1 Modeling of the approaching ABL turbulence

Chapter 2: Synthetic inflow turbulence generator for large-eddy simulation of ABL flows using spectral representation method

In this chapter, a new synthetic inflow turbulence generation technique with explicitly defined two-point flow statistics is developed based on the spectral representation method. The efficacy of the method in representing one-point and two-point statistics is demonstrated by comparing the generated turbulence with wind field measurements taken in a boundary layer wind tunnel. Furthermore, this chapter implements an implicit ground roughness modeling technique for ABL flow. The proposed method is then applied to the LES of ABL flows for three exposure conditions, and the incident wind profiles are examined. Considering that the generation of realistic inflow turbulence is the first necessary step to conduct a successful LES, the methods developed in this chapter offer a unique advantage for wind load evaluation studies. Finally, the developed procedure is implemented into the OpenFOAM framework and disseminated open-source for the wider CWE research community.

Chapter 3: Computationally efficient inflow turbulence generation using a low-rank matrix decomposition

This chapter proposes a computationally efficient wind turbulence generation method that operates on a low-rank representation of the cross-power spectral density (CPSD) matrix. Inflow turbulence generation using the spectral representation method normally requires decomposing the CPSD matrix at multiple frequencies. Considering that the decomposition of the full

CPSD matrix is prohibitively expensive, this chapter presents the application of the Nyström technique to estimate the eigen-decomposition of the CPSD matrix from a small subset of systematically sampled informative points. The accuracy and computational efficiency of the proposed method relative to the conventional eigen-decomposition are investigated. Also, the chapter studies factors affecting the accuracy of the proposed method, such as the percentage of points sampled and the sampling technique used.

1.3.2 LES-based wind load and response evaluation on tall buildings

Chapter 4: LES for predicting wind loads and responses of a standard tall building: prospect for wind-resistant tall building design

This chapter investigates the capability of LES for predicting transient wind loads on the CAARC (Commonwealth Advisory Aeronautical Research Council) standard tall building. We present the application of the inflow turbulence generation and implicit ground-roughness modeling techniques developed in Chapter 2. To validate the LES results, we conducted experimental measurements using High-Frequency Pressure Integration (HFPI) model in a boundary layer wind tunnel. Also, the wind-induced responses of the building were investigated using the dynamic properties of a 60-story reinforced concrete building with a moment-resisting frame system. The validation of the numerical models is carried out stage by stage. First, the characteristics of the approaching flow are examined. Then, the statistics of the cladding loads and the base aerodynamic loads were compared. Finally, the performance of the LES for predicting wind-induced response is evaluated.

Chapter 5: LES-based wind load evaluation on a tall building located in a city center: comparison with experimental data

In this chapter, wind load on a tall building located in a realistic urban environment is studied using LES. The chapter demonstrates the application of the proposed framework to a realistic scenario involving a typical wind tunnel study in a tall building design project. The challenges of CFD modeling for buildings in complex urban setups, including specification of boundary conditions, mesh generation, and turbulence modeling, are highlighted and addressed. The study first validates the incident wind profiles from LES with those reported in the experimental data. Finally, the spectra as well as statistics of base shear and moment coefficients predicted from the LES are compared against experimental measurements for various configurations.

1.3.3 Fluid-structure interaction for aeroelastic modeling

Chapter 6: Fluid-structure interaction framework for computational aeroelastic modeling of tall buildings

This chapter provides the formulation and implementation of a high-fidelity Fluid-Structure Interaction (FSI) framework for computational aeroelastic modeling of flexible structures. The FSI framework was developed by coupling OpenFOAM's transient solver with an in-house structural solver. The FSI framework employs a partitioned procedure where fluid and structure subsystems are solved separately, and the coupling is achieved by exchanging data at each time step. In this study, the structural solver is directly integrated into the CFD solver architecture to reduce the communication overhead between the fluid and structural solvers. Two coupling algorithms representing "weak" and "strong" methods were considered. The framework is implemented using C++ programming language, applying an object-oriented programming paradigm. The software architecture of the framework is designed to be versatile so that it can easily be extended to simulate various wind-structure problems involving structures with complex mode shapes and non-linear material properties. Finally, the main capabilities of the proposed FSI framework are demonstrated using two numerical examples, including the vortex-induced crosswind oscillation of a circular cylinder and the wind-induced vibration of a tall building.

Chapter 2

Synthetic inflow turbulence generator for large-eddy simulation of ABL flows using spectral representation method

2.1 Introduction

Over the last few decades, the exponential growth of computational power has made large-eddy simulation (LES) a more accessible tool for studying turbulent flows of practical interest. Due to this advancement, LES is now being used more often to study various wind engineering problems. To name a few, LES has been used to evaluate wind load on buildings and components(Aboshosha et al., 2015c; Dagnew and Bitsuamlak, 2013, 2014; Daniels et al., 2013; Elshaer et al., 2016; Li et al., 2015; Melaku et al., 2022; Nozawa and Tamura, 2002; Tamura et al., 2008; Tamura and Ono, 2003; Tanaka et al., 2013; Yan and Li, 2015), pedestrian level winds (Adamek et al., 2017; Razak et al., 2013; Tominaga et al., 2008b; Yuan et al., 2016), pollutant dispersion (Gousseau et al., 2011; Tomas et al., 2015; Tominaga and Stathopoulos, 2011; Xie and Castro, 2009) and more recently to simulate non-synoptic wind fields such as downbursts (Aboshosha et al., 2015b; Vermeire et al., 2011) and tornados (Gairola and Bitsuamlak, 2019; Lewellen et al., 1997). One of the main challenges of conducting a successful LES study is generating the inflow boundary condition that characterizes the incoming turbulence. Especially, LES of the atmospheric boundary layer (ABL) flows requires specifying inlet turbulence with the correct mean velocity, turbulence intensity, and integral length scale profiles in line with experimental or field measurements. It is also required that the generated turbulence captures the spatiotemporal correlation of the flow describing the particular terrain being simulated (Aboshosha et al., 2015c; Dagnew and Bitsuamlak, 2014; Huang et al., 2010;

Melaku et al., 2017; Yan and Li, 2015). In addition to having the required wind profiles and spatiotemporal correlation, the generated turbulence is generally required to be divergence-free for incompressible flow simulation. Violating the divergence-free condition can introduce non-physical pressure fluctuations in the simulation (Kim et al., 2013; Patruno and de Miranda, 2020; Poletto et al., 2013). The present study aims at developing a synthetic inflow generation method based on the spectral representation method that meets these requirements for LES of ABL flows.

Different inflow generation methods have been developed in the past, and these methods can generally be grouped into two categories: precursor methods and synthetic methods. In the first group, velocity data extracted from a precursor simulation is used as an inflow in the successor(main) simulation (Lund et al., 1998). Whereas in the second group, the inflow turbulence is generated artificially by employing statistical methods. For a comprehensive review of different inflow generation methods, the reader is advised to refer to Tabor and Baba-Ahmadi (2010), Wu (2017), and Dhamankar et al. (2018).

From a computational wind engineering (CWE) standpoint, the most direct and accurate precursor simulation would be replicating the entire wind tunnel geometry in CFD and developing a turbulent boundary layer over a long fetch distance naturally (Jørgensen et al., 2012; Tanaka et al., 2013; Thordal et al., 2019). However, the extra computational load and lack of flexibility make this method uneconomical for practical use. Alternatively, the cost of running such an expensive simulation can be reduced using recycling (Lund et al., 1998; Nozawa and Tamura, 2002) techniques. Nonetheless, these alternatives can introduce spurious periodicity on the generated turbulence and remain costly compared to synthetic methods.

Compared to precursor methods, synthetic methods are more flexible (give tight control over flow statistics) and take less computational time. In the literature, there are at least three categories of synthetic inflow generation methods. In the first category of methods, the inlet turbulence is generated by superposing sinusoidal waves of different amplitudes, frequencies, and phase shifts. These methods are commonly called Fourier techniques and use sine and cosine functions correlated over space and time. Examples of turbulence generators in this group include techniques proposed by Kraichnan (1970), Lee et al. (1992), Kondo et al. (1997), Smirnov et al. (2001), Huang et al. (2010), Castro and Paz (2013) and Aboshosha et al. (2015c). Synthetic inflow generation methods that are widely used in CWE studies, particularly for wind load evaluation, belong to this group (Dagnew and Bitsuamlak, 2014; Elshaer et al., 2016; Lu et al., 2012; Melaku et al., 2022; Ricci et al., 2017; Yu et al., 2018). The second group includes methods that use digital filtering techniques (Di Mare et al., 2006; Kim et al., 2013; Klein et al., 2003; Lamberti et al., 2018; Xie and Castro, 2008). In these methods, a random set of numbers is initially generated, then digital filters are used to impose spatial and temporal

correlations. However, for boundary layer flows, imposing an integral length scale that varies over the inlet plane can be challenging as outlined in Dhamankar et al. (2018). In the third family, the fluctuating velocity field is generated based on the classical view of turbulence as a superposition of coherent structures with a given shape, length, and time scale (Jarrin et al., 2006; Pamiès et al., 2009). Methods under this category include artificial vortex methods (Mathey et al., 2006), synthetic-eddy methods (Jarrin et al., 2006; Kim and Haeri, 2015; Pamiès et al., 2009; Poletto et al., 2013) and turbulent spot methods (Kornev and Hassel, 2007; Kornev et al., 2008; Kröger and Kornev, 2018). Although methods in this category are very promising, specifying arbitrary target velocity spectra explicitly/directly can be challenging (Yan and Li, 2015).

Recently, Huang et al. (2010) and Aboshosha et al. (2015c) noted that the inflow turbulence that satisfies two-point flow statistics defined by the frequency-dependent coherency function is crucial for wind engineering applications. For instance, Aboshosha et al. (2015c) demonstrated the importance of inflow turbulence with a proper coherency function for the accurate estimation of wind loads on tall buildings. Nevertheless, the method developed by Aboshosha et al. (2015c) imposes the coherency function using an empirically tuned parameter, which often needs problem-specific calibration and does not always guarantee a fully-correlated velocity field in the three principal directions. To this end, from the available synthetic inflow generation schemes, the method developed by Kondo et al. (1997) is capable of generating a point-to-point correlated velocity field with explicitly defined target two-point statistics. However, the procedure used in Kondo et al. (1997) takes considerable computational time, and the generated turbulence does not satisfy the continuity equation, requiring a separate divergence-free operation. Nevertheless, the computational efficiency and accuracy of the procedure used in Kondo et al. (1997) can be significantly improved using recent developments in spectral representation methods.

The spectral representation method is one of the most commonly used and robust procedures for synthesizing random fields (Spanos and Zeldin, 1998). It was first proposed by Shinozuka (Shinozuka, 1971; Shinozuka and Jan, 1972) for the simulation of multidimensional, multivariate, Gaussian stationary processes with a prescribed target cross-power spectral density (CPSD) matrix. Several researchers have applied this method for simulation of the turbulent wind fields, earthquake ground motions, ocean waves, etc., (Deodatis, 1996a,b; Di Paola, 1998; Mann, 1998; Morooka et al., 1997; Shinozuka and Deodatis, 1988; Solari and Carassale, 2000; Tucker et al., 1984). The atmospheric turbulent velocity field generated by the spectral representation method has been shown to satisfy the targeted statistical description of ABL flow (Carassale and Solari, 2006; Di Paola, 1998). Owing to this advantage, Hémon and Santi (2007) highlighted that the method could be an ideal approach for synthetically generating

inflow boundary condition for LES of ABL flows.

In this paper, we propose a computationally efficient synthetic inflow turbulence generation method based on the spectral representation method of Deodatis (1996b) with a posteriori divergence-free operation developed by Kim et al. (2013). The proposed procedure is named the Divergence-free Spectral Representation (DFSR) method. The computational efficiency is achieved by using interpolation-enhanced schemes for the CPSD matrix decomposition, followed by the application of the Fast Fourier Transform (FFT) technique for the simulation of the velocity-time series that reduces the computational cost significantly.

The paper is organized as follows. Section 2.2 presents the detailed procedure used to simulate the velocity field using the spectral representation method. The wind tunnel measurements conducted for validating the proposed method are described in Section 2.3. In Section 2.4, the generated velocity field is validated against the experimental data at selected points, and the performance is compared with the "Consistent Discrete Random Flow Generation" (CDRFG) method of Aboshosha et al. (2015c). Finally, in Section 2.5, the proposed method is applied to LES of a neutrally stratified ABL flow for three exposure conditions, and the results from LES are compared with the experimentally measured profiles.

2.2 Numerical procedure

In this section, the procedure for generating the inlet turbulence and the subsequent steps used to make it divergence-free are described. The inlet turbulence is simulated as a multivariate stochastic velocity field employing the method used in Deodatis (1996b). To make the generated turbulence divergence-free, a posteriori procedure developed by Kim et al. (2013) is adopted. The key steps in the proposed inflow generation method are summarized using a flowchart shown in Figure 2.1.

2.2.1 Proposed inflow generation method

Let x, y, z be a Cartesian coordinate system; x, y, and z representing the longitudinal, lateral, and vertical directions, respectively as shown in Figure 2.2. A statistically stationary velocity vector field $\boldsymbol{u}(\boldsymbol{x}, t)$ on a $yz-$plane that varies with position $\boldsymbol{x}(y, z)$ and time t can be represented as

$$\boldsymbol{u}(y, z; t) = \bar{\boldsymbol{u}}(z) + \boldsymbol{u}'(y, z; t), \tag{2.1}$$

where $\bar{\boldsymbol{u}}(z)$ and $\boldsymbol{u}'(y, z; t)$ represent the mean and the fluctuating parts, respectively. The mean part $\bar{\boldsymbol{u}}(z)$ can be determined from the logarithmic law profile (Tennekes, 1973)

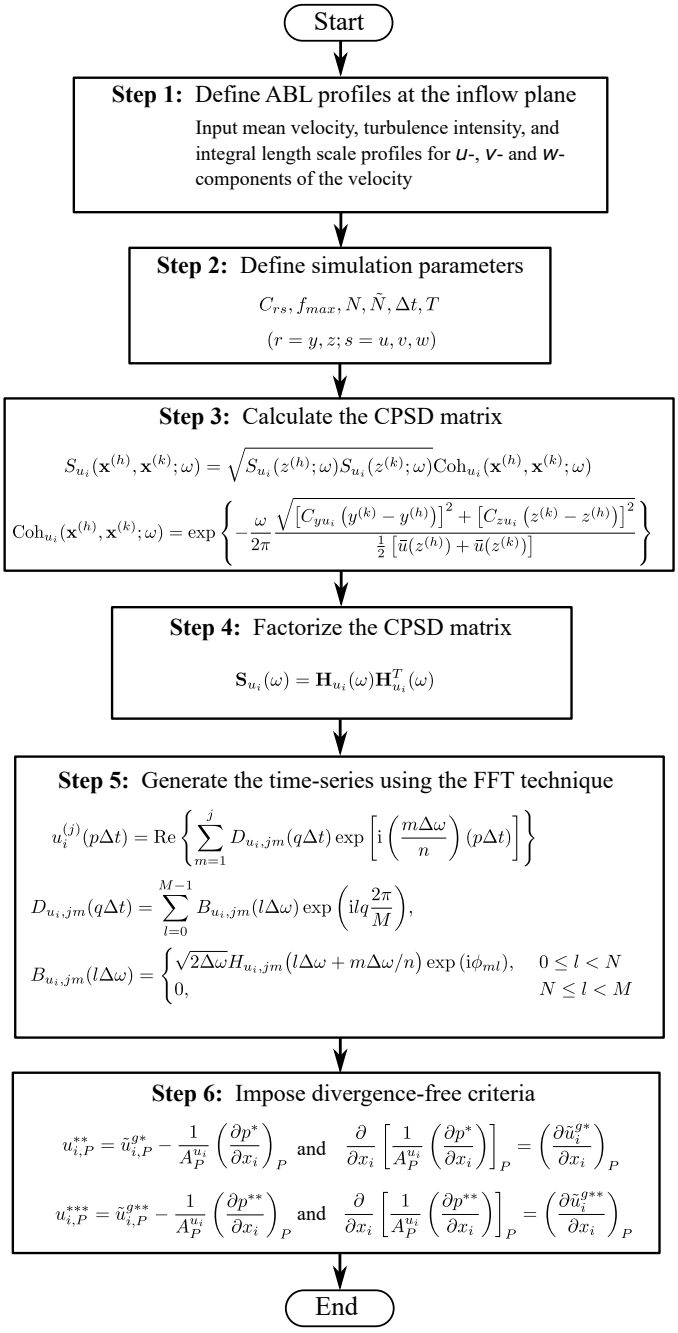


Figure 2.1: Flow chart summarizing the simulation procedure for the Divergence-free Spectral Representation (DFSR) method

$$\bar{u}(z) = \frac{1}{\kappa} u_* \ln\left(\frac{z}{z_0}\right), \tag{2.2}$$

where κ is the von Karman constant, u_* is the shear friction velocity and z_0 is the aerodynamic roughness height. It is accepted that the log-law model works only in the lower ABL i.e., $z < 200$ m (Cook, 1997). For higher elevations, a more accurate model by Deaves and Harris (1978) needs to be used.

The fluctuating components of the velocity are represented statistically by their corresponding cross-power spectral density functions. Considering a discrete spatial domain containing n points, for any pair of points h and k in the domain with position vectors $\boldsymbol{x}^{(h)}$ and $\boldsymbol{x}^{(k)}$ shown in Figure 2.2, the CPSD function $S_{u_i}(\boldsymbol{x}^{(h)}, \boldsymbol{x}^{(k)}; \omega)$ of the velocity component $u_i (i = 1, 2, 3)$ is given by

$$S_{u_i}(\boldsymbol{x}^{(h)}, \boldsymbol{x}^{(k)}; \omega) = \sqrt{S_{u_i}(z^{(h)}; \omega) S_{u_i}(z^{(k)}; \omega)}\,\mathrm{Coh}_{u_i}(\boldsymbol{x}^{(h)}, \boldsymbol{x}^{(k)}; \omega) \qquad (h, k = 1, 2, ..n), \tag{2.3}$$

where $\mathrm{Coh}_{u_i}(\boldsymbol{x}^{(h)}, \boldsymbol{x}^{(k)}; \omega)$ is the coherency function for the velocity component u_i and ω is the angular frequency. The coherency function represents the correlation of the velocity field for various frequencies and is defined by Davenport (1961b) as

$$\mathrm{Coh}_{u_i}(\boldsymbol{x}^{(h)}, \boldsymbol{x}^{(k)}; \omega) = \exp\left\{-\frac{\omega}{2\pi}\frac{\sqrt{\left[C_{yu_i}\left(y^{(k)} - y^{(h)}\right)\right]^2 + \left[C_{zu_i}\left(z^{(k)} - z^{(h)}\right)\right]^2}}{\frac{1}{2}\left[\bar{u}(z^{(h)}) + \bar{u}(z^{(k)})\right]}\right\}. \tag{2.4}$$

In Eq.(2.4), C_{yu_i} and C_{zu_i} refer to the coherency decay coefficients in y- and z-direction for the velocity component u_i, respectively. Solari and Piccardo (2001) provided average values and coefficients of variation for C_{yu_i} and C_{zu_i} by combining data from several studies.

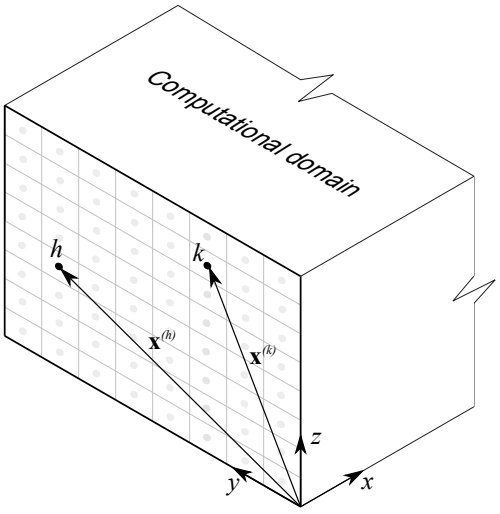


Figure 2.2: Inflow plane coordinate system definition

In wind engineering applications, it is often assumed that the CPSD function between different components of the velocity is small (Solari and Tubino, 2002). In the current study, for simplicity and to reduce the computational cost, the time series for each component of the velocity is simulated independently (i.e., without using the combined CPSD matrix of u and w components), which can result in zero Reynolds shear stresses. However, this limitation can be circumvented by applying the transformation procedure used in Lund et al. (1998) with the desired Reynolds shear stress values. For the auto-spectrum of each velocity component, the well-known von Karman model (Simiu and Scanlan, 1996) is adopted in the following form

$$\begin{aligned} \frac{S_{u_i}(z;\omega)}{\sigma^2_{u_i}(z)} &= \frac{4\left[(L_{u_i}(z)/\bar{u}(z)\right]}{\left[1+70.8(2\pi\omega L_{u_i}(z)/\bar{u}(z))^2)\right]^{5/6}} && (i=1), \\ \frac{S_{u_i}(z;\omega)}{\sigma^2_{u_i}(z)} &= \frac{4\left[L_{u_i}(z)/\bar{u}(z)\right]\left[1+188.4(4\pi\omega L_{u_i}(z)/\bar{u}(z))^2\right]}{\left[1+70.8(4\pi\omega L_{u_i}(z)/\bar{u}(z))^2)\right]^{11/6}} && (i=2,3), \end{aligned} \tag{2.5}$$

where σ_{u_i} and L_{u_i} refer to the standard deviation and the integral length scale of turbulence for the velocity component u_i, respectively.

Thus, the one-sided target cross-spectral density matrix $\mathbf{S}_{u_i}(\omega)$ for the velocity component u_i can be represented as

$$\mathbf{S}_{u_i}(\omega) = \begin{bmatrix} S_{u_i,11}(\omega) & S_{u_i,12}(\omega) & \dots & S_{u_i,1n}(\omega) \\ S_{u_i,21}(\omega) & S_{u_i,22}(\omega) & \dots & S_{u_i,2n}(\omega) \\ \vdots & \vdots & \ddots & \vdots \\ S_{u_i,n1}(\omega) & S_{u_i,n2}(\omega) & \dots & S_{u_i,nn}(\omega) \end{bmatrix}. \tag{2.6}$$

Assuming the imaginary part of $\mathbf{S}_{u_i}(\omega)$ matrix to be small (Simiu and Scanlan, 1996), the velocity component in the $i-$direction at location $\boldsymbol{x}^{(j)}$ can be simulated by using the formula in Deodatis (1996b) as

$$u_i^{(j)}(t) = \sqrt{2\Delta\omega}\sum_{m=1}^{j}\sum_{l=1}^{N}\left|H_{u_i,jm}(\omega_{ml})\right|\cos(\omega_{ml}t+\phi_{ml}), \tag{2.7}$$

where N is the number of frequency intervals; $\Delta\omega = \omega_{up}/N$; ω_{up} is the upper cut-off frequency above which the cross-spectral matrix $\mathbf{S}_{u_i}(\omega)$ can be assumed to be zero; ω_{ml} is a double indexed frequency given as

$$\omega_{ml} = (l-1)\Delta\omega + \frac{m}{n}\Delta\omega \qquad l = 1,2,3,\dots,N \tag{2.8}$$

and ϕ_{ml} is a random phase angle uniformly distributed in the interval $[0, 2\pi]$. Sampling ϕ_{ml}

from a uniform distribution results in Gaussian velocity components, and a similar assumption is often used in spectral methods since the phase information within the flow is unknown a priori (Kondo et al., 1997; Lee et al., 1992).

The term $H_{u_i,jm}(\omega_{ml})$ in Eq.(2.7) is the element of the lower triangular matrix $\mathbf{H}_{u_i}(\omega)$ which is calculated from the Cholesky factorization of the CPSD matrix $\mathbf{S}_{u_i}(\omega)$ in the following form:

$$\mathbf{S}_{u_i}(\omega) = \mathbf{H}_{u_i}(\omega)\mathbf{H}_{u_i}^{\mathsf{T}}(\omega), \tag{2.9}$$

where $\mathbf{H}_{u_i}^{\mathsf{T}}$ is the transpose of the lower triangular matrix $\mathbf{H}_{u_i}(\omega)$.

For a large number of points on the inlet plane, the Cholesky factorization of the CPSD matrix given in Eq.(2.9) takes considerable computational time. Elements of the CPSD, as well as the decomposed matrix $\mathbf{H}_{u_i}(\omega)$, are a continuous function of frequency. Thus, any element in the $\mathbf{H}_{u_i}(\omega)$ matrix can be interpolated between adjacent frequencies. As demonstrated by Carassale and Solari (2006), Ding et al. (2006) and Tao et al. (2017), factorizing the CPSD matrix at selected frequencies and interpolating for the rest can significantly reduce the computational and memory demand. Eqs. (2.7) and (2.8) require the CPSD matrix to be decomposed for nN frequencies. However, in the current study, only $\tilde{N} \ll nN$ number of frequencies are used for factorizing the CPSD matrix. To interpolate the intermediate frequencies, a spline interpolation is adopted based on the recommendation provided in Tao et al. (2017). The distribution of sample interpolation frequencies is shown in Figure 2.3. The frequencies used for the interpolation are calculated as

$$\tilde{\omega}_\alpha = \left(\frac{\omega_{up}}{nN}\right)(nN)^{(\alpha-1)/(\tilde{N}-1)}, \qquad (\alpha = 1, 2, 3, \ldots, \tilde{N}). \tag{2.10}$$

2.2.1.1 Time series generation using Fast Fourier Transform (FFT)

The computational burden of simulating the flow field using Eq.(2.7) can also be drastically reduced by utilizing the Fast Fourier Transform (FFT) technique as demonstrated by Yang (1972, 1973) and Deodatis (1996b). Based on the procedure presented in Deodatis (1996b), for double-indexed frequency, in order to apply FFT technique to Eq.(2.7), first, it needs to be rewritten in the following form:

$$\begin{gathered} u_i^{(j)}(p\Delta t) = \mathrm{Re}\left\{\sum_{m=1}^{j} D_{u_i,jm}(q\Delta t)\exp\left[\mathrm{i}\left(\frac{m\Delta\omega}{n}\right)(p\Delta t)\right]\right\}, \\ p = 0, 1, 2, \ldots, M\times n - 1, \qquad j = 1, 2, 3, \ldots, n, \end{gathered} \tag{2.11}$$

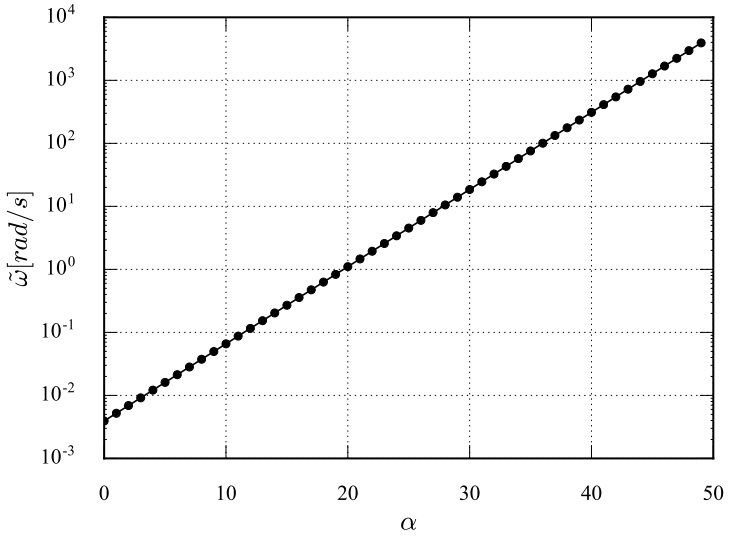


Figure 2.3: Sample frequency sequences calculated using Eq.(2.10) to interpolate the CPSD matrix for intermediate frequencies.

where i = $\sqrt{-1}$ is an imaginary unit; $M \geq 2N$, $q = 0, 1, 2, ..., M-1$ is the remainder of p/M and $D_{u_i,jm}(q\Delta t)$ is given by

$$D_{u_i,jm}(q\Delta t) = \sum_{l=0}^{M-1} B_{u_i,jm}(l\Delta\omega) \exp\left(\mathrm{i}lq\frac{2\pi}{M}\right), \tag{2.12}$$

where $B_{u_i,jm}(l\Delta\omega)$ can be computed as

$$B_{u_i,jm}(l\Delta\omega) = \begin{cases} \sqrt{2\Delta\omega} H_{u_i,jm}(l\Delta\omega + m\Delta\omega/n) \exp\left(\mathrm{i}\phi_{ml}\right), & 0 \leq l < N \\ 0, & N \leq l < M \end{cases}. \tag{2.13}$$

From Eq.(2.12), it can be seen that $D_{u_i,jm}(q\Delta t)$ is the inverse Fourier transform of $B_{u_i,jm}(l\Delta\omega)$ and can be computed using the FFT algorithm efficiently in $O(N \log N)$ complexity, instead of using the original cosine series representation that has $O(N^2)$ time complexity. To benefit from the computational efficiency of the FFT algorithm, Δt in Eq.(2.11) should satisfy the following condition:

$$\Delta t = \frac{2\pi}{2\omega_{up}}, \tag{2.14}$$

where ω_{up} is defined earlier as the upper cut-off frequency.

To further reduce the computational cost of decomposing the CPSD matrix, in Chapter 3, a new factorization technique that uses a low-rank representation of the CPSD matrix is proposed.

2.2.2 Divergence-free modification

The wind field generated using the spectral representation method described above is not divergence-free, and its direct application as an inflow boundary condition might introduce fictitious pressure fluctuations (Gresho and Sani, 1987; Kim et al., 2013; Poletto et al., 2013). The method proposed by Kim et al. (2013) addresses this issue by inserting the generated flow field on a vertical 2D plane close to the inlet. The inserted flow field acts as an intermediate value in the velocity-pressure coupling procedure. The flow is then corrected to satisfy the divergence-free condition during the pressure-correction step. Applying the method to the LES of a channel flow, Kim et al. (2013) showed that the correction has minimal effect on the flow statistics while reducing unnecessary pressure fluctuations. In the current study, the same procedure is adopted. Very recently, Patruno and de Miranda (2020) addressed the insurgence of pressure fluctuations due to incompatible inflow boundary conditions by using a Variationally Based Inflow Correction (VBIC) method that corrects the synthetically generated inflows to avoid pressure fluctuations. Although the current study uses the method developed by Kim et al. (2013), it should be noted that a similar method, like the VBIC, is also equally applicable to correct the generated turbulence to satisfy the divergence-free condition.

The divergence-free procedure developed by Kim et al. (2013) is a modification of the existing implementation of the Pressure-Implicit with Splitting of Operators (PISO) solver in OpenFOAM for grid points on the 2D inflow application plane. Kim et al. (2013) demonstrated that such a modification has a negligible effect on the solution accuracy by employing analytical and numerical error estimation techniques. Figure 2.4 shows a schematic representation of the computational domain and the 2D inflow application plane.

2.2.3 Treatment of ground roughness boundary condition

For the LES study of ABL flows, in addition to inlet boundary conditions, a special treatment to the ground boundary condition is required. In wall-bounded flow like ABL flows, as we get close to the ground surface, the flow is dominated by vortices with characteristic lengths much smaller than those at the free stream flow (Piomelli and Balaras, 2002). Resolving these small-size eddies requires an enormous number of computational grids near the wall and becomes computationally expensive. For flows with high Reynolds number, such as ABL flows, wall-modeled LES becomes the only alternative, considering the available computational power. Among the available methods, the wall stress boundary condition has been widely applied to LES modeling of ABL flows (Bou-Zeid et al., 2005; Grötzbach, 1987; Schumann, 1975; Thomas and Williams, 1999). The wall shear stress model, originally introduced by Schumann

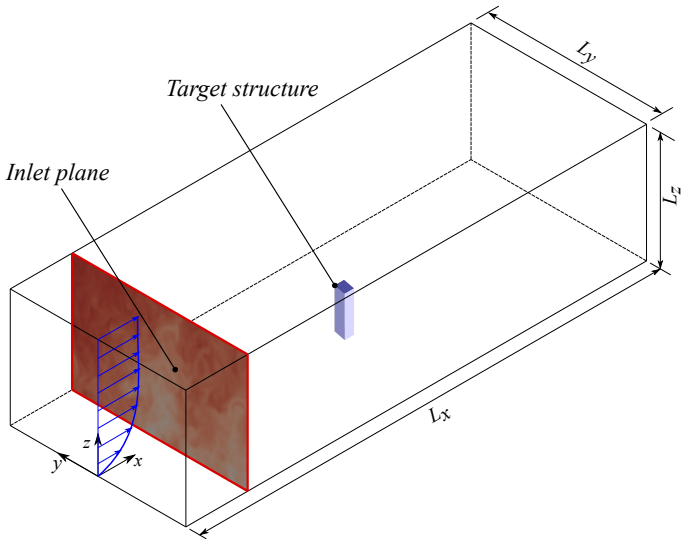


Figure 2.4: A schematic illustration of the computational domain with the inflow application plane

(1975), relates the wall shear stress to the velocity at the wall-adjacent cell center as

$$\tau_w^{\text{LES}} = -\left[\frac{\kappa}{\log[(\Delta z/2)/z_0]}\right]^2 \left(\langle\tilde{u}_{1/2}\rangle^2 + \langle\tilde{v}_{1/2}\rangle^2\right) \tag{2.15}$$

$$\tau_{13,w} = \tau_w^{\text{LES}} \left[\frac{\tilde{u}_{1/2}}{\sqrt{\langle\tilde{u}_{1/2}\rangle^2 + \langle\tilde{v}_{1/2}\rangle^2}}\right] \tag{2.16}$$

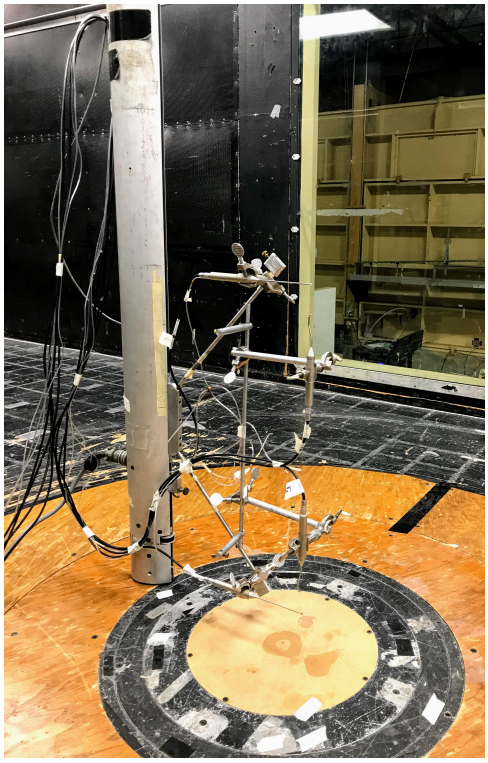
$$\tau_{23,w} = \tau_w^{\text{LES}} \left[\frac{\tilde{v}_{1/2}}{\sqrt{\langle\tilde{u}_{1/2}\rangle^2 + \langle\tilde{v}_{1/2}\rangle^2}}\right] \tag{2.17}$$

, where κ is the von Karman constant; Δz is the height of the wall adjacent cell; $\tilde{u}_{1/2}$ and $\tilde{v}_{1/2}$ are the filtered stream-wise and span-wise velocities at the first cell center away from the surface, respectively. The symbol $\langle\cdot\rangle$ denotes the plane-averaged velocity at the first grid points. Finally, the wall stress calculated from Eqs.(2.16, 2.17) is applied to the ground surface, accounting for the effect of the roughness. This method was implemented to work seamlessly with OpenFOAM's transient solver. To use this boundary condition for the LES, one needs to specify only z_0 representative of the exposure condition being simulated.

2.3 Experimental measurements

For validating the proposed method, experimental measurements of wind profiles were performed in the Boundary Layer Wind Tunnel Laboratory (BLWTL) at Western University. A

turbulent boundary layer flow is simulated in a high-speed closed-circuit wind tunnel section with a 39 m length, 3.4 m width, and 2.5 m height. To generate a fully developed ABL flow turbulence, the BLWTL facility uses three spires, a barrier, and roughness elements as shown in Figure 2.5b and 2.5c. The turntable is located 33 m from the inlet of the tunnel. By changing the heights of the roughness elements, three different terrain conditions that correspond to open, suburban, and urban exposure were simulated.



(a) Cobra probes setup

(b) Spires and barrier

(c) Roughness elements

Figure 2.5: Measurement configuration and wind tunnel setup

The experiment was conducted at a geometric scale of 1:400 in an empty tunnel configuration. The ABL profiles are measured at the center of the turntable. The u-, v- and w-components of the velocity were recorded using Cobra Probes mounted on a movable anchor as shown in Figure 2.5a. To create the profiles, measurements were taken at 28 points aligned vertically. Table 2.1 shows details of the wind tunnel configuration used. The simulation of the terrains is based on the wind characteristics provided in ESDU (2001a,b); ESDU-85020 (2001) for mean velocity and turbulence intensity profiles. Figures 2.6a-c show the comparison of the measured stream-wise mean velocity and turbulence intensity profiles with those found from ESDU (2001a,b); ESDU-85020 (2001). The mean velocity profiles are normalized by the mean wind speed at a reference height z_{ref} = 0.45 m, which corresponds to a full-scale height of 180 m. In Figures 2.7a-c, the stream-wise velocity spectrum at the reference height is shown

together with the well-known von Karman spectrum for each exposure condition.

Table 2.1: Terrains simulated and wind tunnel setup used

Terrain	Roughness block height [m]	$z_{0,FS}$ [m]	u_* [m/s]
Open	0.0254	0.03	0.629
Suburban	0.0635	0.30	0.785
Urban	0.0889	0.70	0.878

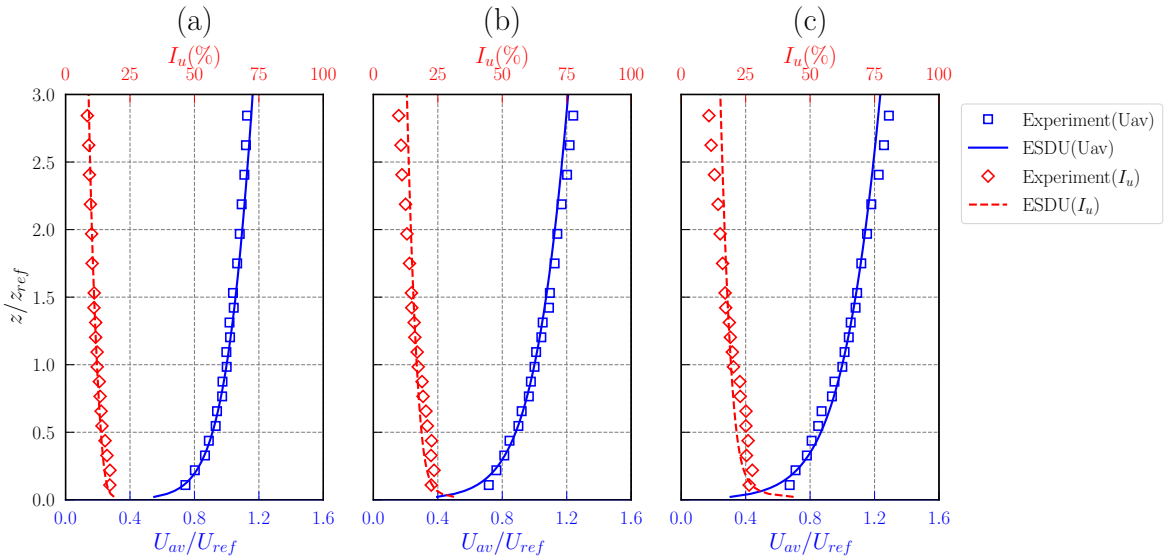


Figure 2.6: Comparison of measured mean velocity and longitudinal turbulence intensity profiles with ESDU (2001a,b); ESDU-85020 (2001) standard: (a) open; (b) suburban and (c) urban

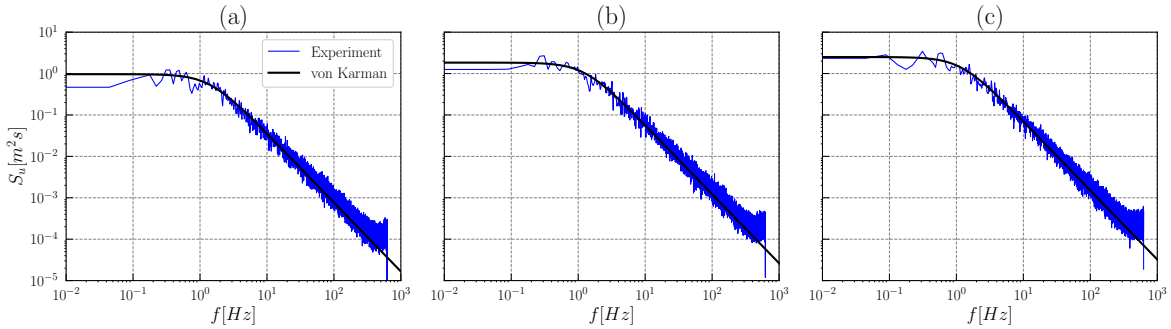


Figure 2.7: Comparison of the longitudinal velocity spectra 0.45 m above the ground with the von Karman spectrum: (a) open; (b) suburban and (c) urban

2.4 Validation of the generated velocity field

The first step of validating the proposed method involves a statistical comparison of the flow generated using the procedure outlined in Section 2.2.1 against the targeted experimental measurements. Turbulent velocity fluctuations in the three orthogonal directions are generated at four points given in Table 2.2. At each point, the target wind profiles, including mean velocity (U_{av}), turbulence intensities (I_u, I_v, I_w) and integral length scales of turbulence (^xL_u,^xL_v,^xL_w) are taken from the experimental data for open exposure condition. In Table 2.2, the first two points (Point 1 and 2) are separated by 0.05 m vertical distance, while the last two points (Point 3 and 4) have a 0.15 m lateral separation. The suitability of the current method to generate ABL-like turbulence is shown by comparing the generated velocity fluctuations with the experimental measurement statistically. The current method is also compared with the CDRFG method of Aboshosha et al. (2015c) since both methods model the two-point statistics of the flow using the frequency-dependent coherency function. For a seamless comparison with the experimental measurements, the simulation is performed at 1:400 scale for 60 s duration with $\Delta t = 8\times10^{-4}$ s. The other essential input parameter used in the simulation are set the same as the experimental measurements and are given in Table 2.2. A detailed description of the experimental setting can be found in Section 2.3.

Table 2.2: Characteristics of the ABL profile and parameters used in the simulation

Point	$y(m)$	$z(m)$	$U_{av}(m/s)$	$I_u(\%)$	$I_v(\%)$	$I_w(\%)$	$^xL_u(m)$	$^xL_v(m)$	$^xL_w(m)$
1	0.00	0.25	12.72	14.13	10.57	8.07	1.11	0.32	0.25
2	0.00	0.30	12.84	13.79	10.02	8.11	1.11	0.33	0.27
3	0.00	0.35	13.17	13.04	9.60	7.45	1.17	0.34	0.19
4	0.15	0.35	13.17	13.04	9.60	7.45	1.17	0.34	0.19
Simulation parameters									
$C_{yu} = 10.0, C_{zu} = 9.0, C_{yv} = 3.5, C_{zv} = 4.5, C_{yw} = 7.5$ and $C_{zw} = 3.0$ $N = 2^{14}, \tilde{N} = 50, f_{max} = 625$ Hz, $\Delta t = 8 \times 10^{-4}$ s, $T = 60$ s									

2.4.1 Evaluation of one-point statistics

Figure 2.8 shows a sample plot of the spectral content of the generated velocity fluctuations from the DFSR and CDRFG methods at Point 2 for each velocity component. The figure also shows the comparison with experimental measurement and the targeted von Karman spectrum. For the comparison shown in Figure 2.8, the target spectra for both the DFSR and CDRFG methods were discretized using the same frequency step. As depicted in Figure 2.8, the re-

sulting spectra from the DFSR and CDRFG methods generally follow the target spectra well. However, it should be noted that the accuracy of each method depends on the frequency step used in the simulation. Table 2.3 shows the relative errors in the standard deviation of the time series generated using each method at different frequency steps. The relative errors reported in Table 2.3 are the average of the 4 points used in the simulation. As shown in Table 2.3, the DFSR method generally gives a smaller relative error than the CDRFG method for the same frequency step.

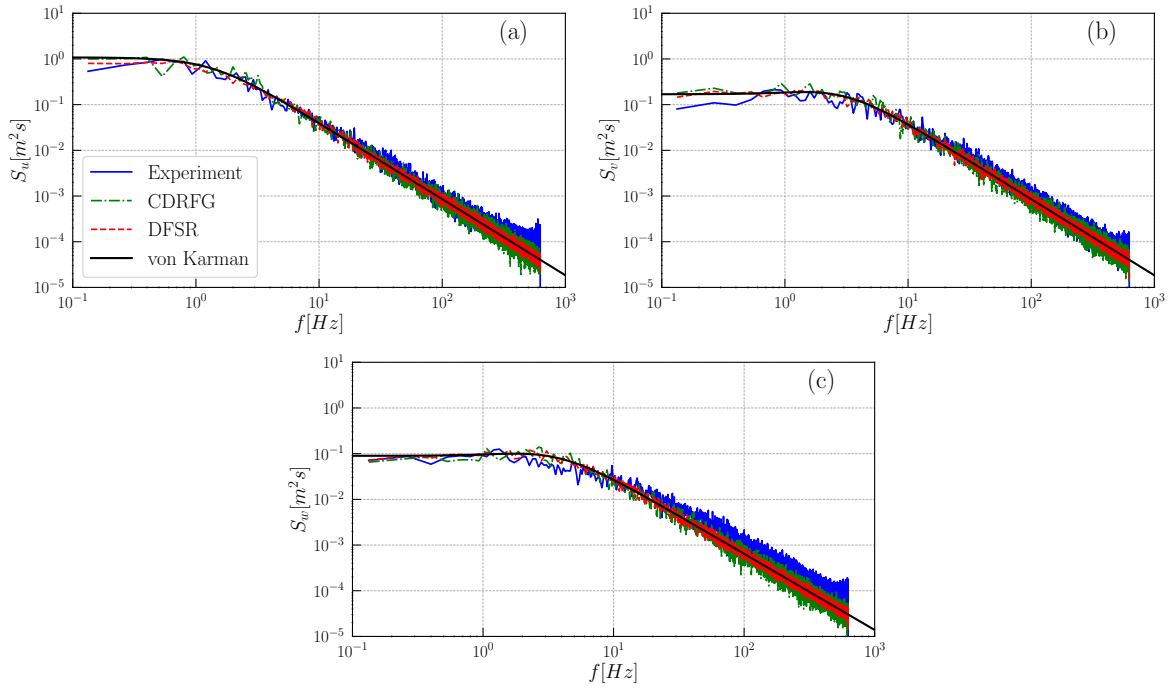


Figure 2.8: Comparison of velocity spectra at Point 2: (a) S_u; (b) S_v and (c) S_w

Table 2.3: Relative error in standard-deviation for different frequency step

	Error(%)					
	σ_u		σ_v		σ_w	
$\Delta f(Hz)$	CDRFG	DFSR	CDRFG	DFSR	CDRFG	DFSR
4.88	26.23	14.61	13.12	6.94	12.11	4.54
2.44	16.93	11.13	7.24	2.24	6.89	2.24
1.22	9.23	5.53	4.09	1.83	3.02	1.76

2.4.2 Evaluation of two-point statistics

In the previous section, the one-point statistics of the ABL turbulence generated using the DFSR method was studied in comparison with the experimental data. Here we evaluate the two-point statistics of the generated velocity field by examining the coherency function and the spatial correlation.

Figure 2.9 shows the coherency function of the generated turbulence in the vertical and lateral directions. In the same figure, the coherency functions calculated from the experimental data and the target function given in Eq.(2.4) are shown. To calculate the target theoretical coherency function using Eq.(2.4), the required coherency decay coefficients, C_{yu_i} and C_{zu_i} are determined fitting the experimental data. The respective values of C_{yu_i} and C_{zu_i} are given in Table 2.2. The coherency function $\mathrm{Coh}_{h,k}$ from a sample velocity time-series for any two points labeled h and k can be computed as

$$[\mathrm{Coh}_{h,k}(\omega)]^2 = \frac{[S_{h,k}(\omega)]^2}{S_{h,h}(\omega)S_{k,k}(\omega)}, \tag{2.18}$$

where $S_{h,k}(\omega)$ is the cross-power spectral density function between the two points and $S_{h,h}(\omega)$, $S_{k,k}(\omega)$ represent the auto-power spectral density function at point h and k, respectively.

In Figures 2.9a-c, the coherency function for each component of the velocity in the vertical direction is determined using Points 1 and 2. Similarly, Figures 2.9e-f show the coherency functions in the lateral direction calculated using Point 3 and 4. As shown in Figure 2.9, the coherency functions from the DFSR method are generally in excellent agreement with the target function and the experimental measurement for a wide range of frequencies when compared to the CDRFG method. The accuracy of the DFSR method is attributed to the fact that the coherency between the two points is explicitly modeled in the simulation procedure by defining the cross-spectral density matrix given in Eq.(2.6). Furthermore, the spectral representation method gives robust control over all velocity components in the three principal directions and does not require problem-specific tuning, as is typically done in the existing synthetic inflow generation methods such as CDRFG (Aboshosha et al., 2015c), DSRFG (Huang et al., 2010) and MDSRFG (Castro and Paz, 2013).

The spatial correlation of the simulated velocity field from the current method is also investigated by generating a velocity-time series over a horizontal line that contains 30 simulation points. The line is located 0.45 m above the ground and spans 1.5 m in the y-direction. The lateral spatial correlation of the generated flow field is shown in Figure 2.10 for both the DFSR and CDRFG methods in comparison with the theoretical curve. The targeted theoretical curve is computed by integrating Eq.(2.4) over the frequency ranged, $[0, \omega_{up}]$ of the simulation (Hémon and Santi, 2007; Huang et al., 2010). Using a similar expression provided in Hémon

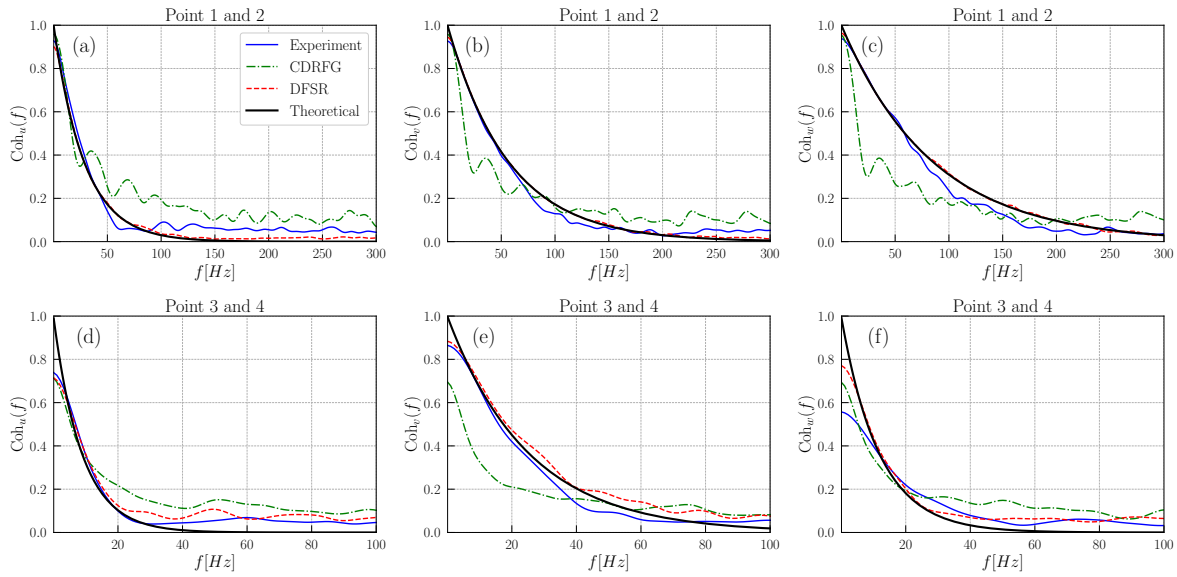


Figure 2.9: Comparison of the coherency function between points separated vertically (Point 1 and 2) and laterally (Point 3 and 4) for u, v and w components. The theoretical curves shown in the plots are calculated using Eq.(2.4).

and Santi (2007), the theoretical/target spatial correlation for the velocity components u_i is determined as

$$S\,c_{h,k}^{u_i} = \sum_{l} \sqrt{S_{u_i}(\boldsymbol{x}^{(h)}, \omega_l) S_{u_i}(\boldsymbol{x}^{(k)}, \omega_l)}\,\mathrm{Coh}_{u_i}(\boldsymbol{x}^{(h)}, \boldsymbol{x}^{(k)}, \omega_l), \tag{2.19}$$

where h and k are two points in space, between which the spatial correlation is sought and l denotes index of the frequencies.

For the u-component, the spacial correlations from both DFSR and CDRFG methods have less than a 5% deviation from the theoretical curve. For the v and w components, however, the DFSR method resulted in an average error of 4.6% and 7.7%, respectively, while the CDRFG method gave an error up to 60%.

Figure 2.11 shows snapshots of the instantaneous velocity field generated using the CDRFG and DFSR methods on a vertical plane. The velocity contours in Figure 2.11 show both small and large-scale turbulence fluctuations, which can translate to eddies of various sizes when applied to LES. In Figure 2.11, visually comparing the snapshots of the generated turbulence, it can be seen that large-scale fluctuations are more apparent in the velocity contours of the DFSR method compared to that of the CDRFG method for v- and w- components of the velocity. This is further related to the accuracy of each method to model the spatial correlation (see also Figure 2.10). It is worth mentioning that inflow turbulence with appropriate spatial correlation results

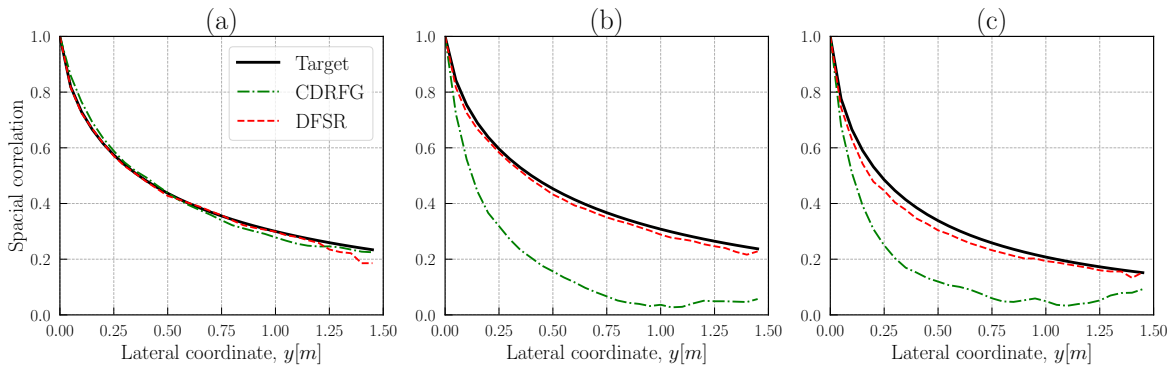


Figure 2.10: Comparison of spatial correlation of the generated velocity field using the CDRFG and DFSR methods: (a) u-, (b) v- and (c) w-components

in a realistic flow field downstream of the inlet and minimizes the decay of turbulence in the computational domain.

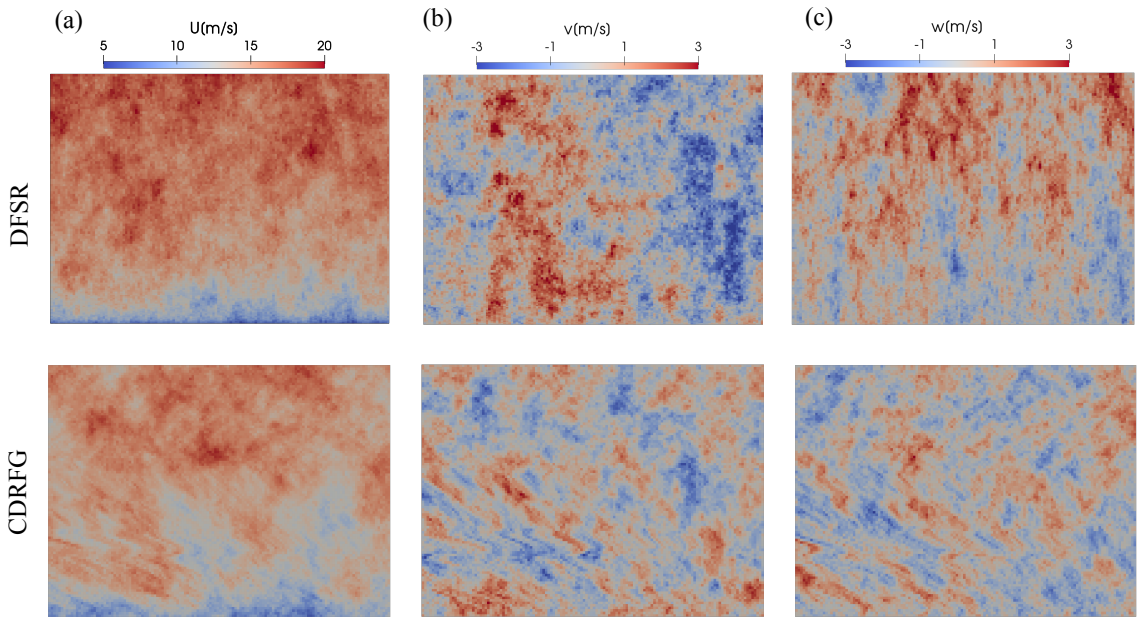


Figure 2.11: Instantaneous velocity field distribution generated using the DFSR (first row) and CDRFG (second row) methods for open exposure condition. The columns (a), (b) and (c) are the u-, v- and w- components, respectively. The dimension of the plane is 3.4 m $\times$ 2.5 m.

When generating inflow on a plane with many points (like in Figure 2.11), performing Cholesky factorization of the CPSD matrix and simulation of the velocity time series at each point can be time-consuming. A brief discussion regarding the computational cost of the current method and details of the implemented code is found in Appendix A.

2.5 Application of DFSR for LES of neutrally stratified ABL flow

The previous section demonstrated that the proposed method can generate a turbulent wind field that satisfies the prescribed one- and two-point statistics of ABL turbulence. In this section, the DFSR method is applied to simulate a neutrally stratified ABL flow in LES. The LES study is conducted for three exposure conditions: open, suburban, and urban targeting the experimental measurements described in Section 2.3. The performance of the proposed approach is compared with the CDRFG method and a precursor simulation that replicates the entire wind tunnel geometry. All the numerical simulations are carried out in OpenFOAM 5.0 using the same geometric scale as the experimental measurements.

2.5.1 Computational domain and grid generation

Two types of computational domains are used in the current study (see Figure 2.12). The first computational domain is shown in Figure 2.12a and is used to test the proposed inflow generation method. The cross-sectional dimension of this domain is taken to be the same as that of the boundary layer wind-tunnel near the turntable (see Figure 2.12). Hence, a 3.4 m wide and 2.5 m high domain is used as shown in Figure 2.12a. However, in the stream-wise direction, to cut computational cost, the length of the domain is set to 10 m. The second computational domain replicates the whole boundary layer wind tunnel geometry, as shown in Figure 2.12b. The roughness elements shown in Figure 2.12b have a $length \times width \times height$ of 0.10 m $\times$ 0.05 m $\times$ 0.025 m and are spaced 0.40 m apart in each direction. Three spires are used in the model with a 1.22 m height and 0.10 m bottom width. A barrier that has 0.38 m height and 0.1 m width was modeled (see Figure 2.12b). In total, seven LES cases were performed, and the details of these simulations are summarized in Table 2.4. The first six simulations use the domain shown in Figure 2.12a, while the last case, labeled VWT (Virtual Wind Tunnel), uses the domain shown in Figure 2.12b.

In the upper part of the domain shown in Figure 2.12a, cubical cells are used, while the lower 20% part uses rectangularly stretched cells that have a maximum aspect ratio of 4 near the ground surface. The total cell count for these cases is approximately 6.5 million. Case names with prefixes DF and CD in Table 2.4 are flat tertian simulations that use the inflow generated from the DFSR and CDRFG methods, respectively. Considering the cost of the simulation, the VWT case is run only for an open exposure condition. The grid for the VWT case is generated using different refinement levels. The spires, barrier, roughness elements, and tunnel floor have additional local refinements, while in the rest of the tunnel, a uniform grid is

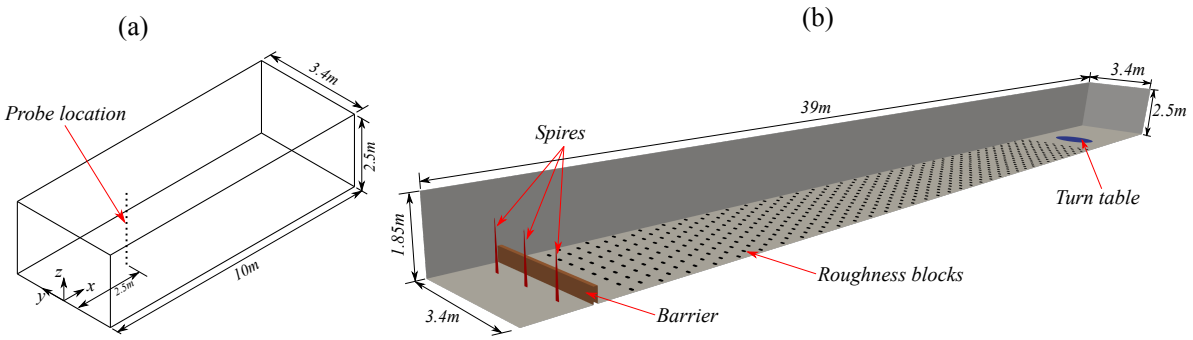


Figure 2.12: Geometry of the computational domain used: (a) domain used for testing the inflow and (b) geometry of the boundary layer wind tunnel for VWT case.

Table 2.4: Summary of simulation set-up for different case studies

Case name	Exposure	$z_0[m]$	$U_{ref}[m/s]$	Inflow	N_x, N_y, N_z	Grids
DF1	Open	0.03	13.66	DFSR	400,136,120	6.5×10^6
DF2	Suburb	0.30	12.55	DFSR	400,136,120	6.5×10^6
DF3	Urban	0.70	12.19	DFSR	400,136,120	6.5×10^6
CD1	Open	0.03	13.66	CDRFG	400,136,120	6.5×10^6
CD2	Suburb	0.30	12.55	CDRFG	400,136,120	6.5×10^6
CD3	Urban	0.70	12.19	CDRFG	400,136,120	6.5×10^6
VWT	Open	0.03	13.22	Uniform		15×10^6

employed. For cases that use the DFSR and CDRFG methods, 2.5 m (one domain height) away from the inlet, a vertical line probe is placed to measure the wind profiles (see Figure 2.12a). This measurement location is chosen considering CWE applications, where the location of interest is usually one domain height away from the inlet if one uses guidelines such as the COST (Franke, 2006) recommendation. For the VWT case, the wind profiles are measured 33 m away from the inlet at the center of the turntable, the same way as the experimental measurements described in Section 2.3.

2.5.2 Boundary conditions

For the cases that use the CDRFG (Aboshosha et al., 2015c) technique, the generated synthetic inflow is applied at the inlet. Whereas, for DFSR cases, at the inlet, a mean logarithmic velocity profile is imposed as shown in Figure 2.4, and the generated turbulence is inserted on a vertical plane close to the inlet (0.625 m) as described in Section 2.2.2. For the side and top walls,

no-slip boundary condition is applied. At the outlet, a Neumann condition for the velocity field is specified with a zero-pressure outlet. Furthermore, for cases that use the domain in Figure 2.12a, a wall model is necessary to account for the effect of the ground roughness. Hence, a wall shear stress boundary condition described in Section 2.2.3 is adopted. To use this method, the center of the wall adjacent cell must lie in the logarithmic layer (Grötzbach, 1987; Piomelli and Balaras, 2002). In the current study, it may be noted that for simulations that use the wall shear stress model, all the wall-adjacent cell centers extended into the logarithmic region.

For the VWT case, a uniform velocity of 15 m/s is imposed at the inlet corresponding to the tunnel fan speed. At the outlet, the same boundary condition used in the CDRFG and DFSR cases is adopted. On the other surfaces, a no-slip wall boundary condition is imposed, including the spires, barrier, roughness elements, and bottom, top and side faces of the tunnel. In the VWT case, since the roughness elements are explicitly modeled, no wall treatment is needed.

2.5.3 Numerical method

For the subgrid-scale (SGS) modeling, the standard Smagorinsky model (Smagorinsky, 1963) is used with its model constant C_S set to OpenFOAM's default value of 0.158. The filter width for the LES, Δ is defined as the cube-root of the cell volume. It is worth mentioning that the standard Smagorinsky SGS model suffers from excessive dissipation in laminar or high-shear regions (e.g., close to walls) (de Villiers, 2006). Thus, for wall areas the C_S constant is adjusted using the Van Driest (Driest, 1956) damping function given as

$$D = 1 - \exp\{-z^+/A^+\} \tag{2.20}$$

where z^+ is the wall normal distance in the wall units and $A^+ = 26$ is the Van Driest constant.

For the spatial discretization, a second-order accurate scheme is employed with linear interpolation. The time discretization is performed using a second-order accurate backward scheme. For the simulation of the coupled pressure-velocity equation, a transient solver based on the PISO (Pressure-Implicit with Splitting of Operation) algorithm is adopted. In most of the simulated cases, the Courant-Friedrichs-Lewy (CFL) number is set to $C_o < 0.7$. Simulation cases given in Table 2.4 are run over 36 s duration with a time step of 0.001 s. For the property of the fluid, the properties of air with a kinematic viscosity of $1.5 \times 10^{-5} m^2/s$ and density of 1.25 kg/m^3 are used.

2.5.4 Results and comparative discussion

After running the LES cases given in Table 2.4, the results from each case are discussed comparatively, taking the experimental data as a reference.

Figure 2.13 shows the downstream evolution of the coherent structures for open exposure condition for the CDRFG, DFSR, and VWT cases. As shown in Figure 2.13a, for the CDRFG case, the flow structures near the inlet seem relatively stretched in the longitudinal direction compared to those seen downstream. Whereas, in the DFSR case, flow structures remain qualitatively the same downstream. For the DFSR method, close to the inlet, where the inflow turbulence is applied (see Figure 2.13b), it is expected that the divergence-free operation alters the statistical qualities of the inflow. However, comparing the flow statistics before and after the divergence-free operation, it was evident that the changes are generally small, usually below 3%.

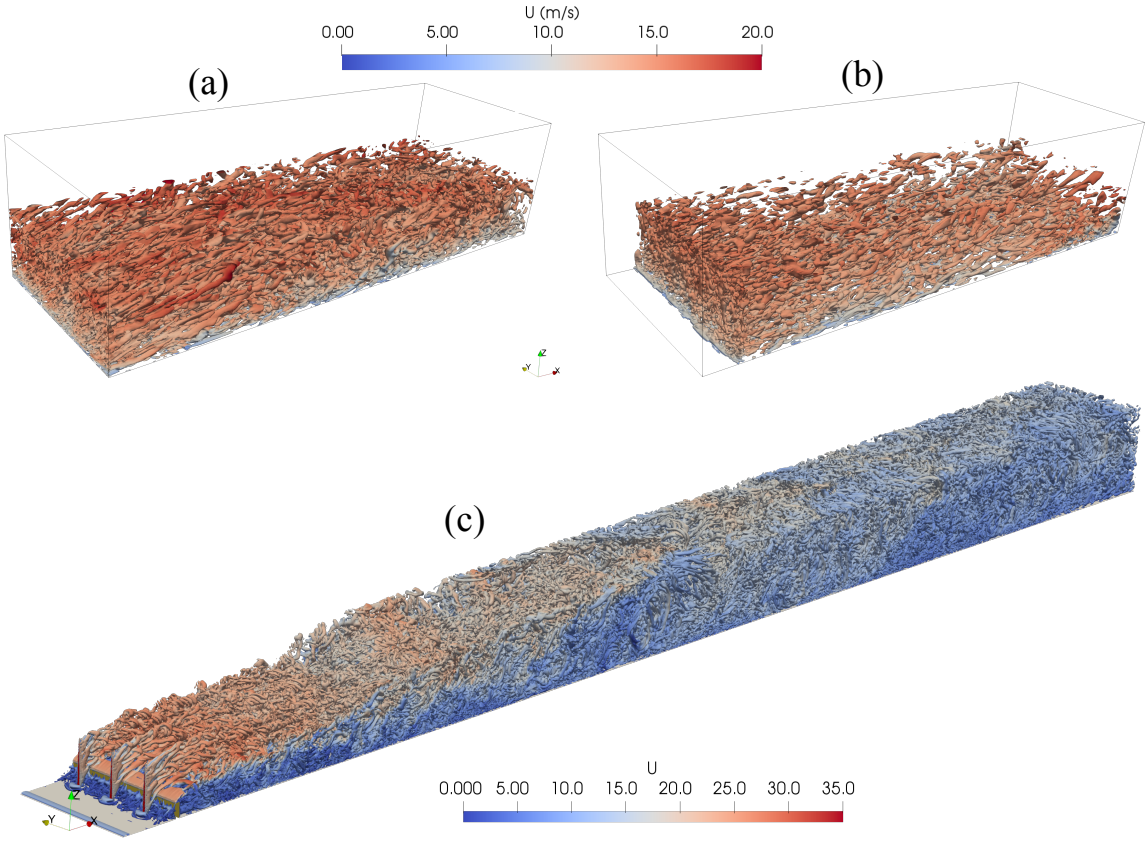


Figure 2.13: Iso-surface of $Q = 100$ colored by the magnitude of the instantaneous velocity field at $10s$ for open exposure condition: (a) DFSR; (b) CDRFG; (c) VWT

2.5.4.1 Comparison of ABL wind profiles

The performance of the DFSR method is evaluated by comparing the wind profiles from the CFD simulation with the experimentally measured data. The wind profiles are recorded at locations shown in Figure 2.12. Table 2.5 compares the absolute error in wind profiles averaged over the height for all simulated cases.

Figure 2.14 shows the comparison of the mean velocity profiles from the DFSR, CDRFG, and VWT cases with those measured from the experiment. Compared to the experimental data, the average difference over the height is roughly below 2.0% for all cases studied(see Table 2.5). However, the difference is more pronounced as the terrain becomes rougher.

Table 2.5: Comparison of relative error(%) in wind profiles averaged over the height

Terrain	Method	U_{av}	I_u	I_v	I_w	${}^xL_u$	${}^xL_v$	${}^xL_w$
Open	CDRFG	0.8	3.3	21.2	37.1	88.4	54.6	57.8
	DFSR	0.8	4.3	12.6	25.6	46.1	12.1	22.9
	VWT	0.7	5.4	8.8	5.4	11.5	17.9	18.7
Suburban	CDRFG	1.2	3.7	21.7	36.7	64.5	78.7	54.8
	DFSR	1.6	5.3	13.9	28.1	58.0	22.8	29.1
Urban	CDRFG	1.8	4.7	23.6	36.1	70.7	75.7	55.0
	DFSR	2.1	5.3	8.5	27.5	44.1	35.5	19.7

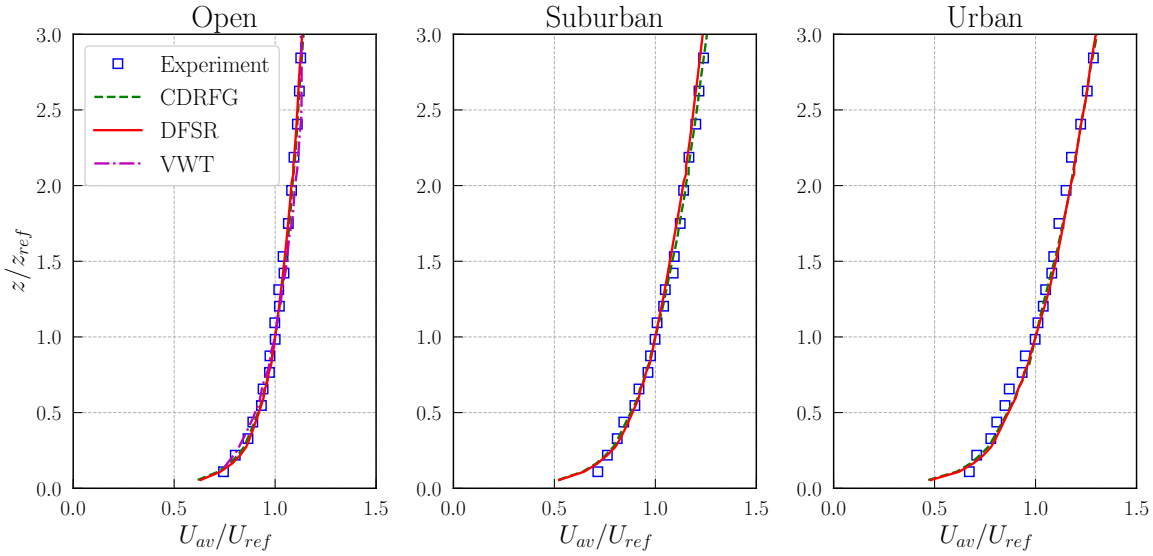


Figure 2.14: Comparison of mean velocity profiles from the DFSR, CDRFG, VWT cases for open, suburban and urban exposure conditions.

Comparison of the turbulence intensity profiles with the experimental data is shown in Figure 2.15. The deviation from the target experimental profiles for open, suburban, and urban

exposure condition are given in Table 2.5 for each component of the velocity. The stream-wise turbulence intensity profiles for both the CDRFG and DFSR cases are in excellent agreement with the experimental data. However, in lateral and vertical directions, the DFSR method showed better prediction than the CDRFG method. This improvement, as demonstrated in Section 2.4.2, is attributed to the capability of the DFSR method to generate inflow turbulence with realistic spatial correlation. Thus, the inflow is sustained up to the target location showing relatively small decay. Since the VWT case replicates the physical wind tunnel simulation, the turbulence intensity profiles are generally in excellent agreement with the experimental data compared to the CDRFG and DFSR cases.

The integral length scale profiles are computed from the velocity time-series at each point using Taylor's frozen turbulence hypothesis, and the results are depicted in Figure 2.16. The deviations of the integral length scale profiles from the experimental values are also reported in Table 2.5. As shown in Figure 2.16, for ${}^{x}L_u$ profile, the results from both the CDRFG and DFSR methods are generally higher than the experimental values. However, for ${}^{x}L_v$ and ${}^{x}L_w$ profiles, the DFSR cases seem to produce comparable profiles with the experimental measurements. As expected, the integral length scale profiles from the VWT simulation matched well with the experimental data.

2.5.4.2 Comparison of the velocity spectra

Figure 2.17 shows the spectra of the velocity fluctuations measured at the target location 0.45 m above the ground surface. The figure shows the spectra for the u, v, and w components comparing the LES cases with the experimental data for the three exposure conditions. The figure also shows the von Karman spectrum, which was used as a target spectrum to generate the inflow turbulence. In the inertial subrange, the widely known $-5/3$ slope is observed in all cases. The spectral plots also show a rapid drop at a frequency of $f \approx 40Hz$ due to the grid resolution limit. This issue has been noticed in LES and is due to the numerical dissipation that occurs for eddies smaller than the grid-scale (Thomas and Williams, 1999). As seen in Figure 2.17, for the u component of the velocity, the spectra from both the CDRFG and DFSR cases agree well with the experiment and generally follow the von Karman spectrum. However, for the v and w components, the spectra obtained from the DFSR cases seem to follow the experimental measurements more closely when compared to the CDRFG cases. For the VWT simulation, the spectra of all the velocity components are in agreement with the experimental measurement for a wide range of frequencies. These observations are generally consistent in all the three exposure conditions as shown in Figure 2.17.

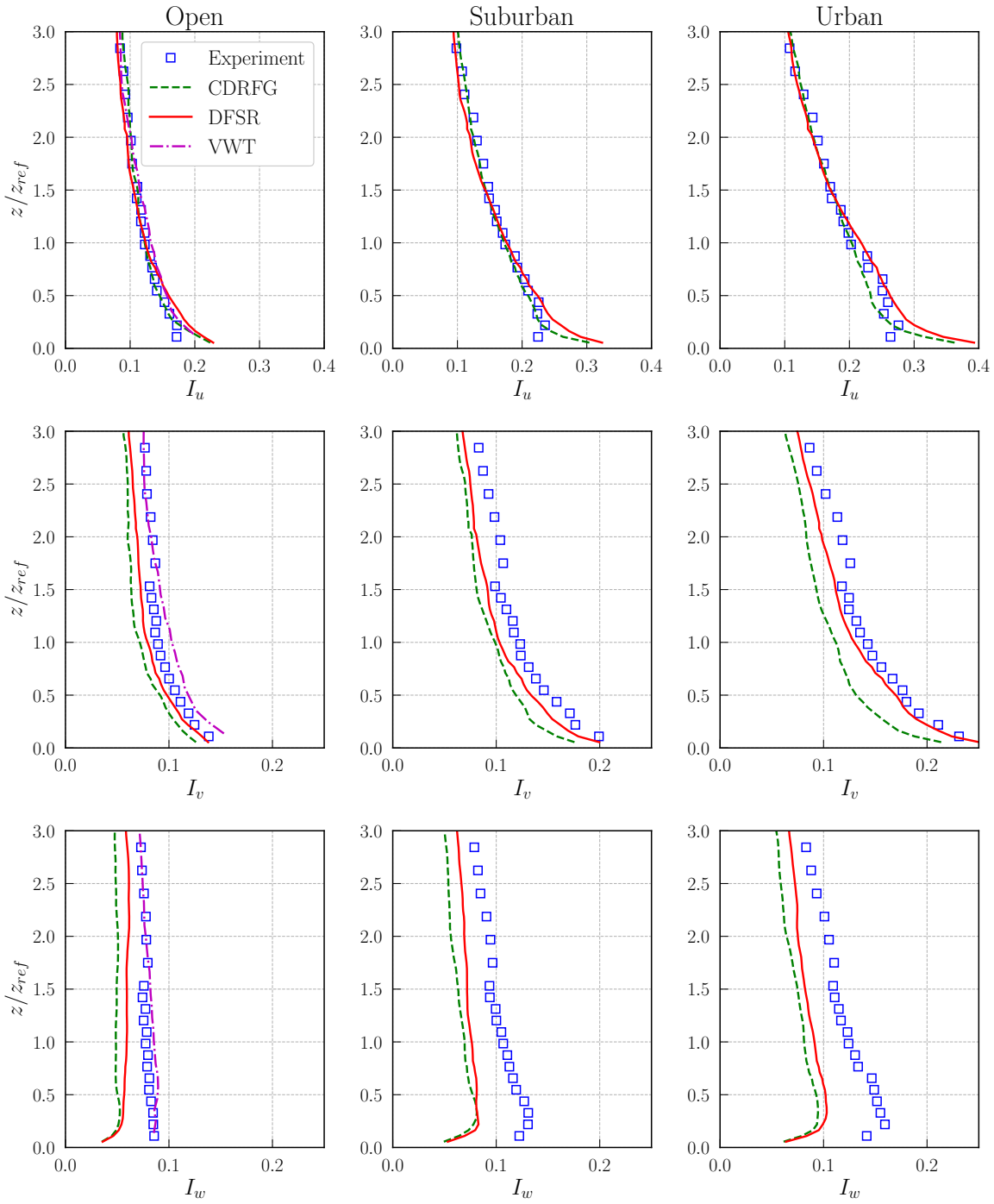


Figure 2.15: Comparison of turbulence intensity profiles from the DFSR, CDRFG and VWT cases for open, suburban and urban exposure conditions.

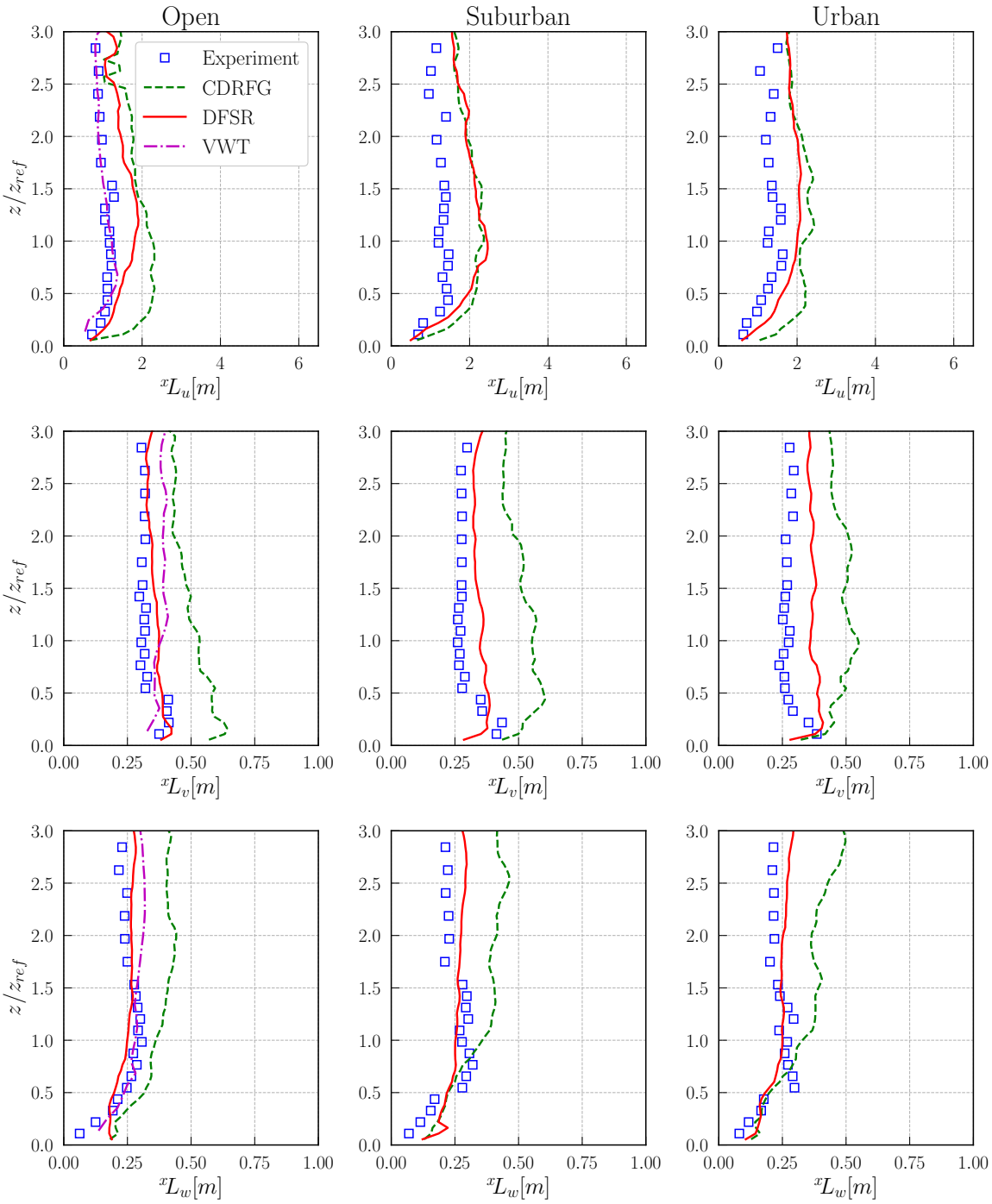


Figure 2.16: Comparison of profiles of integral length scale of turbulence for DFSR, CDRFG and VWT cases for open, suburban and urban exposure conditions.

2.5.4.3 Wind pressure flactuations

The effect of the divergence-free modification on the pressure fluctuation can be seen if the turbulence generated using the spectral representation method is directly applied at the inlet,

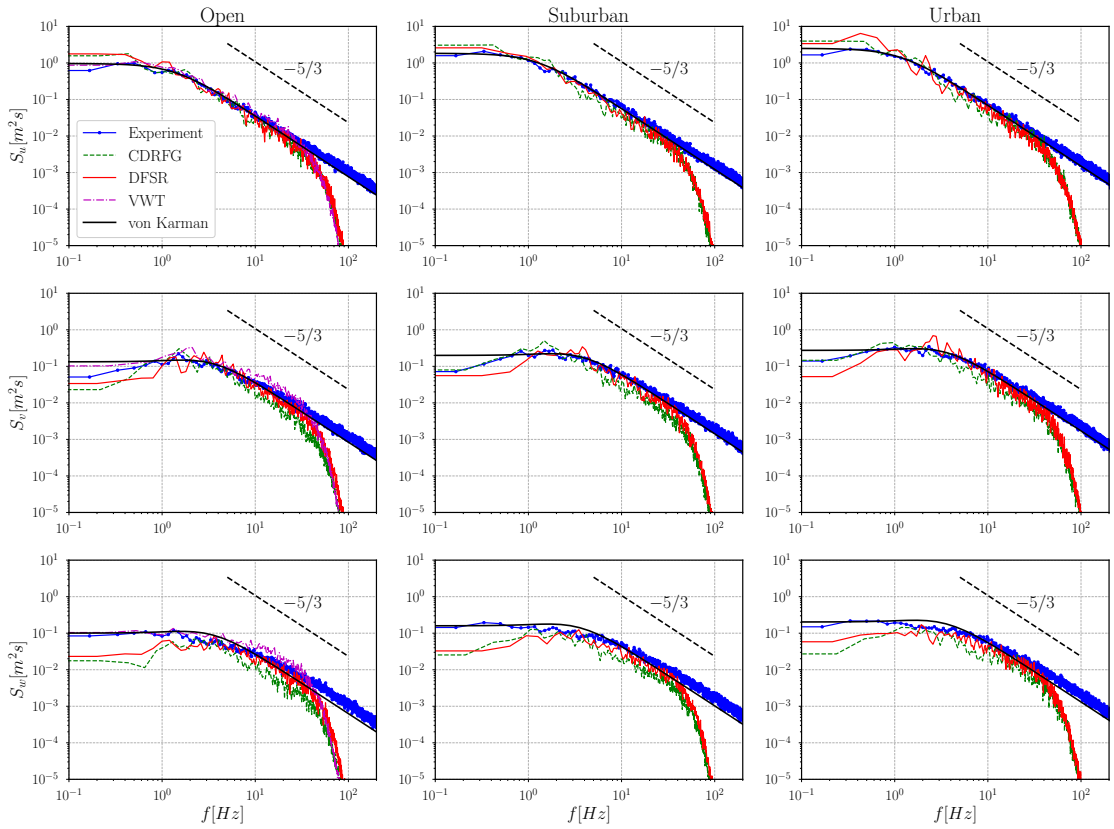


Figure 2.17: Comparison of velocity spectra at a point located 0.45 m above the ground

instead of using the procedure described in Section 2.2.2. Thus, two scenarios were compared: one that uses the inflow directly at the inlet and the other that applies the DFSR method. Figure 2.18 shows the standard deviation of the wall surface pressure fluctuation monitored on the ground surface for these two scenarios. The pressure fluctuation shown in Figure 2.18 is kinematic pressure (pressure normalized by the air density) measured for open exposure condition. For both cases, close to the inlet, non-physical peak pressure fluctuations are observed; however, these fluctuations decay as we move further from the inlet. A similar observation was also reported in other studies, especially when using synthetic inflow generation methods (Kim et al., 2013; Patruno and de Miranda, 2020; Patruno and Ricci, 2018). From Figure 2.18, it is clearly seen that the divergence-free modification used in the DFSR method reduced the pressure fluctuations significantly as compared to the case that applies the inflow directly at the inlet. It should be noted that the turbulence applied at the inlet should satisfy the continuity equation so that such artificial fluctuations are suppressed. For example, in CWE applications, this has important implications because these artificial fluctuations can contaminate the pressure measurements on bluff bodies downstream.

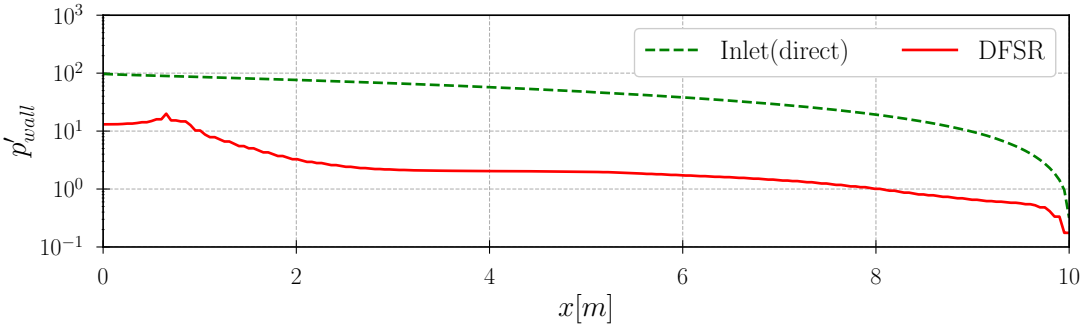


Figure 2.18: Comparison of pressure fluctuations on the ground surface

2.6 Conclusions

A divergence-free synthetic inflow turbulence generation method is developed for LES and applied to ABL flow simulation. The method uses the spectral representation method to generate inflow turbulence with explicitly specified two-point statistics followed by a posterior divergence-free modification. The method is named the "Divergence-free Spectral Representation" (DFSR) method. The velocity spectra, coherency function, and spatial correlation obtained from the proposed method are shown to be in excellent agreement with the experimental data.

Finally, the DFSR method is applied successfully to LES of neutrally stratified ABL flows for open, suburban, and urban exposure conditions. The downstream wind profiles from the DFSR method were compared to those found using the "Consistent Discrete Random Flow Generation" (CDRFG) method (Aboshosha et al., 2015c) and a precursor Virtual Wind Tunnel (VWT) simulation. The mean velocity, turbulence intensity, and integral length scale profiles, as well as the velocity spectra from the DFSR method, are generally in satisfactory agreement with the experimental data. It has been shown that the capability of the DFSR method to generate inflow turbulence with proper spatial correlation has resulted in a reduction of turbulence decay downstream as compared to the CDRFG method. Despite the high computational cost, the results obtained from the VWT simulation showed excellent agreement with the experimental data compared to both DFSR and CDRFG methods. It was also demonstrated that the procedure used to impose the divergence-free condition in the DFSR method was able to reduce the artificial pressure fluctuations in the computational domain. Based on the results presented, the proposed method is expected to be well-suited to CWE applications, particularly for LES-based wind load evaluation studies.

Chapter 3

Computationally efficient inflow turbulence generation using a low-rank matrix decomposition

3.1 Introduction

The spectral representation method (SRM) is one of the most popular techniques for synthesizing a random field given its statistical description. Due to its accuracy and straightforward formulation, the method is broadly applied in the statistical simulation of random fields, such as wind velocity fields (Deodatis, 1996b; Di Paola, 1998; Li and Kareem, 1993; Solari and Carassale, 2000), earthquake ground motions (Deodatis, 1996a; Li and Kareem, 1991; Shinozuka and Deodatis, 1988), and ocean waves (Li and Kareem, 1993; Tucker et al., 1984). For wind-resistant design, the method is commonly used to generate sample wind velocity functions for the buffeting analysis of flexible structures such as long-span bridges (Cao et al., 2000; Carassale and Solari, 2006; Yang et al., 1997) and tall buildings (Chen, 2008; Wu et al., 2007; Zhang et al., 2008). More recently, SRM has attracted renewed interest in computational wind engineering studies to generate inflow turbulence for computational fluid dynamics (CFD) simulations (Kondo et al., 1997; Melaku and Bitsuamlak, 2021; Wang and Chen, 2020). These examples typically involve the simulation of the multivariate, multi-dimensional, and non-homogeneous wind field in a spatial domain containing a large number of points. One of the key challenges in the use of SRM for such large-scale applications stems from its immense computational cost both in storage and execution time (Kareem, 2008).

For prescribed target cross-power spectral density (CPSD) functions, generating a multivariate wind field using SRM involves two main steps. First, the target CPSD matrix is de-

composed at a finite sequence of frequencies using either Cholesky or eigen-decomposition (Di Paola, 1998; Li and Kareem, 1995; Shinozuka and Jan, 1972; Shinozuka et al., 1990). Then, using the decomposed CPSD matrix, the velocity time series is simulated by superposing harmonic waves with randomly generated phase information (Shinozuka, 1971; Shinozuka and Jan, 1972). It has been demonstrated that the latter process can be efficiently computed using the fast Fourier transform (FFT) technique (Deodatis, 1996b; Yang, 1972). However, the decomposition of the CPSD matrix is the most computationally demanding part of the procedure. In terms of computation speed, compared to eigen-decomposition, the Cholesky decomposition is generally less expensive. However, eigen-decomposition is often attractive because it offers a physically meaningful interpretation of the simulated process with each eigenmode representing the basis of spatial distribution (Chen and Kareem, 2005; Di Paola, 1998). Moreover, for large-scale applications, by truncating the higher mode contributions, significant computational savings can be achieved when generating the time series. Nevertheless, the fundamental challenge for both Cholesky and eigen-decomposition is that, for n number of simulation points, the computational cost of factorizing the CPSD matrix inherently scales cubically with n (i.e., $O(n^3)$), which becomes prohibitively expensive for large-scale simulations.

To alleviate the computational burden of factorizing large CPSD matrices in SRM, a host of cost-effective alternatives have been devised in the past. Notably, Yang et al. (1997) and Cao et al. (2000) proposed a computationally efficient procedure by explicitly driving a closed-form expansion of the Cholesky decomposition. However, the method is limited to the simulation of homogeneous wind fields for points equally spaced along a line. Another group of methods employs interpolation-enhanced schemes where the decomposition of the CPSD matrix is done only at limited frequencies, and values for intermediate frequencies are interpolated (Ding et al., 2006; Gao et al., 2012; Huang et al., 2013; Li et al., 2011; Melaku and Bitsuamlak, 2021; Tao et al., 2018). Although these approaches clearly reduce the overall computational and storage cost (Tao et al., 2018), practically, they only operate well with Cholesky decomposition. In principle, it is possible to apply interpolation for eigen-decomposition as well (Carassale and Solari, 2006); nevertheless, the fact that eigenvectors are not a well-behaved function of frequency makes the interpolation task particularly challenging, unless through a stepwise approach (Tao et al., 2018). On the other hand, more recently, revisiting the stochastic wave-based model proposed by Shinozuka and Deodatis (1991b), alternative formulations via a wavenumber–frequency spectrum were proposed, ultimately succeeding the decomposition of the CPSD matrix (Benowitz and Deodatis, 2015; Chen et al., 2018; Peng et al., 2016; Song et al., 2018). While the wave-based model holds a promising research direction, its extension to a non-homogeneous wind field with arbitrary coherency and power spectral density (PSD)

functions is not trivial.

In this study, we propose a computationally efficient procedure for simulating a multivariate non-homogeneous wind velocity field using a low-rank representation of the CPSD matrix. Low-rank matrix decomposition is a well-established technique to reduce dimensionality in many statistical and machine learning-related applications (Belabbas and Wolfe, 2009; Kumar et al., 2009; Williams and Seeger, 2001). The proposed technique is specifically developed for SRM that uses eigen-decomposition or better known as the proper orthogonal decomposition (POD) method (Di Paola and Gullo, 2001). Fundamental to the POD-based approach is that only a limited number of modes contain most of the variance in the fluctuation (Di Paola and Gullo, 2001). Therefore, the main idea behind the proposed approach is to compute these most pertinent eigenvalues and eigenvectors of the CPSD matrix without decomposing the entire matrix. To achieve this objective, we apply Nyström method on a subset of columns systematically sampled from the CPSD matrix. The Nyström method is a classical technique originally developed for the numerical solution of eigenfunction problems (Baker, 1977). Later on, the method proved to be effective to speed up applications that involve the eigen-decomposition of large symmetric positive semi-definite matrices (Williams and Seeger, 2001). This makes Nyström method an attractive technique to improve the computational efficiency of multivariate wind field simulation in the framework of POD-based SRM.

The original formulation for the POD-based SRM is first briefly revisited in the current study. Next, the application of the Nyström method to the eigen-decomposition of the CPSD matrix is discussed with important error metrics and bound to assess the accuracy of the proposed method. Considering that the precision of the Nyström method greatly depends on the way informative columns are selected from CPSD, the application of different sampling techniques is presented. Finally, the trade-off between numerical accuracy and computational efficiency for the proposed method relative to the conventional POD-based approach is investigated using two numerical examples.

3.2 Wind field simulation using POD-based SRM

Let us assume the correlation between the three components of wind velocity is weak, and each component can be simulated independently. Thus, the wind velocity field in space can be represented as a one-dimensional stochastically stationary process that varies with position and time. Suppose that $\mathbf{V}^0(t) = [V_1(t), V_2(t), V_3(t), \ldots, V_n(t)]$ represents nV-1D zero-mean stationary velocity process on n discrete points in space. The target cross-correlation matrix

$\mathbf{R}^0(\tau)$ for a time lag $\tau[-\infty, +\infty]$ is defined by

$$\mathbf{R}^0(\tau) = \begin{bmatrix} R^0_{11}(\tau) & R^0_{12}(\tau) & \dots & R^0_{1n}(\tau) \\ R^0_{21}(\tau) & R^0_{22}(\tau) & \dots & R^0_{2n}(\tau) \\ \vdots & \vdots & \ddots & \vdots \\ R^0_{n1}(\tau) & R^0_{n2}(\tau) & \dots & R^0_{nn}(\tau) \end{bmatrix}, \tag{3.1}$$

and the corresponding two-sided target CPSD matrix $\mathbf{S}^0(\omega)$ is expressed as

$$\mathbf{S}^0(\omega) = \begin{bmatrix} S^0_{11}(\omega) & S^0_{12}(\omega) & \dots & S^0_{1n}(\omega) \\ S^0_{21}(\omega) & S^0_{22}(\omega) & \dots & S^0_{2n}(\omega) \\ \vdots & \vdots & \ddots & \vdots \\ S^0_{n1}(\omega) & S^0_{n2}(\omega) & \dots & S^0_{nn}(\omega) \end{bmatrix}, \tag{3.2}$$

where $\omega[-\infty, +\infty]$ is the angular frequency. The diagonal elements $R^0_{jj}(\tau)$ and $S^0_{jj}(\omega)(j = 1, 2, \dots, n)$ represent the auto-correlation and auto-power spectral density functions for the velocity process $V_j(t)$, respectively. For any two velocity processes $V_j(t)$ and $V_k(t)(j, k = 1, 2, 3, \dots, n; j \neq k)$, $R^0_{jk}(\tau)$ represents their cross-correlation function; while $S^0_{jk}(\omega)$ is their CPSD function. The superscript 0 in Eq. (3.1) and (3.2) denotes the target functions. It is worth noting that, for a stationary process, both the auto-/cross- correlation and spectral density functions are even functions.

Each entry of the cross-correlation and CPSD matrices given in Eq. (3.1) and (3.2) are linked by Wiener-Khintchine's Fourier transform pair as,

$$S^0_{jk}(\omega) = \frac{1}{2\pi} \int_{-\infty}^{\infty} R^0_{jk}(\tau) e^{-\mathrm{i}\omega\tau} \mathrm{d}\tau \quad j, k = 1, 2, 3, \dots, n, \tag{3.3a}$$

$$R^0_{jk}(\tau) = \int_{-\infty}^{\infty} S^0_{jk}(\omega) e^{\mathrm{i}\omega\tau} \mathrm{d}\omega \quad j, k = 1, 2, 3, \dots, n, \tag{3.3b}$$

where $\mathrm{i} = \sqrt{-1}$ represents an imaginary unit. From a practical standpoint, especially when modeling wind velocity processes, both $R^0_{jk}(\tau)$ and $S^0_{jk}(\omega)$ can be regarded as real functions, and the imaginary part is usually negligible (Simiu and Scanlan, 1996). Thus, in the current study, it is assumed that the matrices given in Eqs. (3.1) and (3.2) are real. One of the principal properties of the CPSD matrix is that it is symmetric (or Hermitian if it is complex-valued) and is a positive semi-definite matrix.

Now suppose that the CPSD matrix $\mathbf{S}^0(\omega)$ can be factorized in the following form:

$$\mathbf{S}^0(\omega) = \mathbf{D}(\omega)\mathbf{D}^{\mathsf{T}}(\omega), \tag{3.4}$$

where $\mathbf{D}(\omega)$ is $n \times n$ square matrix, and the superscript T denotes a matrix transpose operator. The expression in Eq. (3.5) does not particularly lead to a unique factorization of the matrix $\mathbf{S}^0(\omega)$ (Di Paola, 1998; Li and Kareem, 1995; Shinozuka et al., 1990). In fact, there are various ways of factorizing the CPSD matrix into such form. For stochastic simulation of wind velocity field, the two most commonly used approaches are the Cholesky and eigenvalue decomposition. If one uses Cholesky decomposition, $\mathbf{D}(\omega) = \mathbf{H}(\omega)$, where $\mathbf{H}(\omega)$ is a lower-triangular matrix. Alternately, if eigenvalue decomposition is employed, the matrix $\mathbf{D}(\omega)$ is computed from the eigen-decomposition of $\mathbf{S}^0(\omega)$ as

$$\mathbf{D}(\omega) = \mathbf{\Psi}(\omega)\sqrt{\mathbf{\Lambda}(\omega)}, \tag{3.5}$$

where $\mathbf{\Psi}(\omega)$ and $\mathbf{\Lambda}(\omega)$ contain the eigenvalues and eigenvectors of the CPSD matrix expressed in the following form:

$$\mathbf{S}^0(\omega) = \mathbf{\Psi}(\omega)\mathbf{\Lambda}(\omega)\mathbf{\Psi}^T(\omega), \tag{3.6a}$$

$$\mathbf{\Psi}(\omega) = [\psi_1(\omega), \psi_2(\omega), \ldots, \psi_n(\omega)], \tag{3.6b}$$

$$\mathbf{\Lambda}(\omega) = \mathbf{diag}[\lambda_1(\omega), \lambda_2(\omega), \ldots, \lambda_n(\omega)]. \tag{3.6c}$$

Here in Eq. (3.6), $\psi_k(\omega)$ and $\lambda_k(\omega)$ ($k = 1, 2, \ldots, n$) are the k-th eigen-vector, and eigenvalue of $\mathbf{S}^0(\omega)$ for frequency ω, respectively. The matrix $\mathbf{\Psi}(\omega)$ is an orthogonal matrix satisfying the condition $\mathbf{\Psi}(\omega)\mathbf{\Psi}^T(\omega) = \mathbf{I}$, $\mathbf{I}$ being an $(n \times n)$ identity matrix. Whereas, the diagonal matrix $\mathbf{\Lambda}(\omega) = \mathbf{diag}[\lambda_1(\omega), \lambda_2(\omega), \ldots, \lambda_n(\omega)]$ holds the eigenvalues, and each diagonal entry $\lambda_k(\omega)$ represents the k-th eigenvalue corresponding to the eigen-vector $\psi_k(\omega)$. In the subsequent discussions, the eigenvalue are assumed to be sorted in a descending order $(\lambda_1(\omega) > \lambda_2(\omega) \cdots > \lambda_n(\omega))$. Since the CPSD matrix is positive definite, all the eigenvalues are real and non-negative. The matrix $\sqrt{\mathbf{\Lambda}(\omega)}$ in Eq. (3.5) is a diagonal matrix formed by taking the square roots of the corresponding diagonal entries in $\mathbf{\Lambda}(\omega)$.

One of the main reasons that the eigenvalue decomposition is more attractive than the Cholesky decomposition is that the eigenvalues and eigenvectors are physically more meaningful. They give insight into the spatial modal structure of the generated wind field (Di Paola, 1998). Furthermore, since the CPSD matrix exhibit fast decay in the magnitude of the eigenvalues, only the first few leading modes are sufficient to capture a larger fraction of the total energy of the fluctuation. These properties lend to the application of POD-based SRM, retaining only the most pertinent modes. The approximation to the CPSD matrix employing the

leading r modes can thus be constructed as

$$\mathbf{S}^0(\omega) \simeq \mathbf{S}_r(\omega) = \sum_{k=1}^{r} \lambda_k(\omega)\boldsymbol{\psi}_k(\omega)\boldsymbol{\psi}_k^T(\omega) = \sum_{k=1}^{r} \mathbf{d}_k(\omega)\mathbf{d}_k^T(\omega), \tag{3.7}$$

where $\mathbf{d}_k(\omega)$ refers to the k-th column of the matrix $\mathbf{D}(\omega)$.

The expression on the right-hand side of Eq. (3.7) is essentially an optimal rank-r approximation $\mathbf{S}_r(\omega)$ to the target CPSD matrix $\mathbf{S}^0(\omega)$. An elaborate procedure on how to efficiently compute this low-rank approximation using the Nyström method is presented in Section 3.3.

Once the target CPSD functions are defined, the stationary velocity process $\mathbf{V}(t)$ can be generated using Fourier–Stieltjes transform (Lumley and Panofsky, 1964; Priestley, 1981; Vanmarcke, 2010). If we assume $\mathbf{V}(t)$ is a stochastically continuous process, $V_j(t)(j = 1, 2, \ldots, n)$ can be represented as the real part of a more general Fourier–Stieltjes integral of the form:

$$V_j(t) = \mathscr{R}\left\{\int_{-\infty}^{\infty} e^{\mathrm{i}\omega t}\mathrm{d}Z_j(\omega)\right\} \quad j = 1, 2, \ldots, n, \tag{3.8}$$

where $\mathscr{R}\{\cdot\}$ stands for the real part, and $Z_j(\omega)$ is a complex-valued stochastic process as a function of frequency ω with orthogonal Fourier increments $\mathrm{d}Z_j(\omega) = \{Z_j(\omega+\mathrm{d}\omega) - Z_j(\omega)\}$. The random increment $\mathrm{d}Z_j(\omega)$ is a zero-mean complex process satisfying the following conditions (Lumley and Panofsky, 1964):

$$\mathbb{E}[\mathrm{d}Z_j(\omega)] = 0, \tag{3.9a}$$

$$\mathbb{E}[\mathrm{d}Z_j(\omega)\mathrm{d}Z_k^*(\omega')] = \begin{cases} 0, & \text{if} \quad \omega \neq \omega' \\ S_{jk}^0(\omega)\mathrm{d}\omega, & \text{if} \quad \omega = \omega', \quad j,k = 1, 2, \ldots, n, \end{cases} \tag{3.9b}$$

where $\mathbb{E}[\cdot]$ denotes mathematical expectation, and the * indicates a complex conjugate operator. The requirement given in Eq. (3.9b) imposes the orthogonality condition on $\mathrm{d}Z_j(\omega)$, i.e., the increments at non-overlapping frequencies are not correlated. Fundamentally, this requirement links the statistical properties of $Z_j(\omega)$ to the target CPSD functions of the process $V_j(t)$.

For numerical simulation, the stochastic integral given in Eq. (3.8) needs to be cast in the form of the Riemann sum. For a practical application, let us now assume that the CPSD functions have negligible power for frequencies higher than the cut-off frequency $\omega_u = \Delta\omega N$.

Then, Eq. (3.8) can be re-written in discretized form as follows:

$$\begin{aligned} V_j(t) &= \mathscr{R}\left\{\int_{-\omega_u}^{\omega_u} e^{\mathrm{i}\omega t}\mathrm{d}Z_j(\omega)\right\} \\ &\simeq \mathscr{R}\left\{\sum_{l=-N}^{N} e^{\mathrm{i}\omega_l t}\Delta Z_j(\omega_l)\right\} \end{aligned} \tag{3.10}$$

where N is the number of frequency intervals, $\Delta\omega = \omega_u/N$ is the frequency step, and $\Delta Z_j(\omega_l)$ is a discrete approximation of the random Fourier increment. There are different ways of constructing the realizations of $\Delta Z_j(\omega_l)$, while satisfying the conditions in Eq. (3.9) (Deodatis, 1996b; Ding et al., 2011; Lumley and Panofsky, 1964). Considering the relationship $S^0_{jk}(\omega) = \sum_{m=1}^{n} D_{jm}(\omega)D_{mk}(\omega)$ from Eq. (3.4) and the requirement in Eq. (3.9b), $\Delta Z_j(\omega_l)$ can simply be constructed using

$$\Delta Z_j(\omega_l) = \sum_{m=1}^{n} D_{jm}(\omega_l)\sqrt{\Delta\omega}e^{\mathrm{i}\phi_{ml}} \quad j = 1, 2, \ldots, n, \tag{3.11}$$

in which $D_{jm}(\omega_l)$ are the elements of the decomposed matrix $\mathbf{D}(\omega)$, ϕ_{ml} represents random phase angles uniformly distributed in the interval $[0, 2\pi]$. Since ϕ_{ml} are drawn from a uniform distribution, by virtue of the central limit theorem, $V_j(t)$ is asymptotically Gaussian when $N \to \infty$(Shinozuka and Deodatis, 1991a). The discrete frequencies ω_l are located at the center of each frequency step, and calculated by

$$\omega_l = (l-1)\Delta\omega + \frac{1}{2}\Delta\omega \qquad l = 1, 2, \ldots, N. \tag{3.12}$$

After substituting Eq. (3.11) into Eq. (3.10), the final simulation formula can be derived as

$$\begin{aligned} V_j(t) &\simeq \mathscr{R}\left\{\sum_{m=1}^{n}\sum_{l=-N}^{N} D_{jm}(\omega_l)\sqrt{\Delta\omega}e^{\mathrm{i}(\omega_l t+\phi_{ml})}\right\} \\ &= 2\sum_{m=1}^{n}\sum_{l=1}^{N} D_{jm}(\omega_l)\sqrt{\Delta\omega}\cos(\omega_l t + \phi_{ml}) \end{aligned} \tag{3.13}$$

The velocity process $\mathbf{V}(t)$ resulting from Eq. (3.13) has a period associated with the smallest frequency $\omega_0 = \Delta\omega/2$ according to Eq. (3.12). Therefore, the generated time series is periodic with:

$$T_0 = \frac{4\pi}{\Delta\omega} = \frac{4\pi N}{\omega_u}. \tag{3.14}$$

Another important remark about Eq. (3.13) is that the generated stochastic wind process $V_j(t)$ is not ergodic. In principle, it is possible to impose ergodic property through double indexing of the sample frequencies in Eq. (3.12) as demonstrated by Deodatis (1996b). However, for large-

scale applications with many simulation points, the period T_0 required to achieve ergodicity in correlation would be very long for common practical applications. Despite this shortcoming, the choice of discrete frequencies based on Eq. (3.12) renders a rate of convergence for correlations that is proportional to $1/N^2$ (Shinozuka and Deodatis, 1991a). With regard to the time step, in order to avoid the "aliasing" effect that results from under-sampling of a continuous fluctuating process, Δt should satisfy the following condition:

$$\Delta t \leq \frac{2\pi}{2\omega_u}. \tag{3.15}$$

The computational cost of simulating the velocity field using the expression in Eq. (3.13) can be expedited using the Fast Fourier Transform (FFT) technique (Deodatis, 1996b; Yang, 1972, 1973). To apply the FFT algorithm to Eq. (3.13), for a single-indexed frequency, the simulation formula needs to be reformulated in the following form:

$$\begin{gathered} V_j(p\Delta t) = \mathscr{R}\left\{\sum_{m=1}^{n} A_{jm}(q\Delta t)\exp\left[\mathrm{i}\left(\frac{\Delta\omega}{2}\right)(p\Delta t)\right]\right\}, \\ p = 0, 1, 2, \ldots, N_T, \qquad j = 1, 2, 3, \ldots, n, \end{gathered} \tag{3.16}$$

where $q = 0, 1, 2, \ldots, M$ is the remainder of p/M; $M = 2N$; N_T is the number of time steps and $A_{jm}(p\Delta t)$ is given by

$$A_{jm}(q\Delta t) = \sum_{l=0}^{M-1} B_{jm}(l\Delta\omega)\exp\left(\mathrm{i}lq\frac{2\pi}{M}\right), \tag{3.17}$$

where $\mathrm{i} = \sqrt{-1}$ is a complex unit, and $B_{jm}(l\Delta\omega)$ is computed as

$$B_{jm}(l\Delta\omega) = \begin{cases} 2\sqrt{\Delta\omega}D_{jm}(l\Delta\omega + \Delta\omega/2)\exp(\mathrm{i}\phi_{ml}), & 0 \leq l < N \\ 0, & N \leq l < M, \end{cases} \tag{3.18}$$

in which $D_{jm}(l\Delta\omega + \Delta\omega/2)$ represents elements of the matrix $\mathbf{D}(\omega)$ determined from Eq. (3.4). From Eq. (3.17) it can be seen that $A_{jm}(q\Delta t)$ is the inverse Fourier transform of $B_{jk}(l\Delta\omega)$, and can be computed efficiently by any standard FFT routine. The simulation formulation in Eq. (3.13) takes a computational time that scales with $O(N^2)$. Whereas a typical FFT implementation scales with $O(N\log N)$ law, which drastically reduces the overall computing cost.

3.3 Proposed method

This section presents the application of the Nyström method to the spectral representation method. The Nyström method originated as a technique for obtaining the approximate numerical solution of integral equations (Baker, 1977). Thus, in the first part of the section, we demonstrate the use of classical Nyström approximation for the numerical treatment of continuous random process over a one-dimensional space as an eigenvalue problem. Then, the method is extended to the simulation of a multivariate velocity process with a prescribed target CPSD matrix. Finally, different landmark selection techniques and important error metrics to assess the accuracy of the Nyström method are briefly discussed.

3.3.1 The Nyström method for simulating random process over a linear domain

Consider a zero-mean random process $u(x,t)$ representing a continuous field along a linear domain $\mathcal{L}$ with span length ℓ, where x is the abscissa. Let $\lambda_k(\omega)$ and $\psi_k(x;\omega)(k = 1,2,\dots)$ be the eigenvalues and the corresponding eigenfunctions of the process which are non-null solution of the linear integral equation of the form (Carassale and Solari, 2002; Tubino and Solari, 2005):

$$\int_{\mathcal{L}} S_u(x,x';\omega)\psi_k(x';\omega)\mathrm{d}x' = \lambda_k(\omega)\psi_k(x;\omega), \tag{3.19}$$

where $S_u(x,x',\omega)$ is the CPSD function of the process often referred as the kernel function (Baker, 1977). For positive semi-definite $S_u(x,x',\omega)$, the eigenvalues are real and non-negative, while the eigenfunctions are real-valued orthogonal basis functions that satisfy the condition (Carassale and Solari, 2006):

$$\int_{\mathcal{L}} \psi_j(x;\omega)\psi_k(x;\omega)\mathrm{d}x = \delta_{jk}. \tag{3.20}$$

For a set of evenly-spaced sample points $X = \{x_1, x_2, \dots x_n\}$, the basic idea of Nyström method is to approximate the integral in Eq. (3.19) using a simple quadrature rule as

$$\frac{\ell}{n}\sum_{j=1}^{n} S_u(x,x_j;\omega)\psi_k(x_j;\omega) \simeq \lambda_k(\omega)\psi_k(x;\omega), \tag{3.21}$$

Here, if we choose x from $\{x_1, x_2, \dots x_n\}$, the expression in Eq. (3.19) leads to matrix eigenvalue problem

$$\mathbf{S}_u(\omega)\hat{\mathbf{\Psi}}(\omega) = \hat{\mathbf{\Psi}}(\omega)\hat{\mathbf{\Lambda}}(\omega), \tag{3.22}$$

where $\mathbf{S}_u(\omega)$ is the CPSD matrix of the discrete system with matrices $\hat{\mathbf{\Psi}}(\omega)$ and $\hat{\mathbf{\Lambda}}(\omega)$ containing its eigenvectors and eigenvalues. Finally, the *Nyström extension* of eigenvalues and eigenfunctions of the continuous field are derived by matching Eqs. (3.22) and (3.21) to give (Williams and Seeger, 2001):

$$\lambda_k(\omega) \approx \left(\frac{\ell}{n}\right) \hat{\lambda}_k(\omega), \tag{3.23}$$

$$\psi_k(x;\omega) \approx \sqrt{\frac{n}{\ell}} \frac{1}{\hat{\lambda}_k(\omega)} \sum_{j=1}^{n} S_u(x, x_j;\omega) \hat{\psi}_k(x_j;\omega) \tag{3.24}$$

It is worth noting that for bi-dimensional continuous random process $u(x,t)$, closed-form solution of eigenvalues and eigenfunctions is possible with homogeneous turbulence assumption, as clearly demonstrated by Carassale and Solari (2006). The advantage of the Nyström extension is that it allows us to recover approximate eigenvalues and eigenfunctions for any non-homogeneous multi-dimensional process given a set of sample points arbitrarily distributed in space. Most importantly, the expressions in Eqs. (3.23) and (3.24) motivates analogous extensions of the method to any positive semi-definite CPSD matrix as demonstrated in the following section.

3.3.2 Approximate Eigen decomposition of the CPSD matrix using Nyström Method

For the CPSD matrix defined over a set of discrete points in space, the Nyström method works on the low-rank representation of a given matrix by sampling a representative column subset from the target matrix (Williams and Seeger, 2001). Recalling that the target CPSD matrix $\mathbf{S}^0(\omega)$ is a symmetric positive semi-definite $n \times n$ matrix, it can be represented using Nyström method by selecting $c \ll n$ informative columns from $\mathbf{S}^0(\omega)$ that correspond to c landmark points in space. Once the sample columns are selected, an $n \times c$ subset matrix $\mathbf{C}(\omega)$ is formed. Without losing generality, the rows and columns of $\mathbf{S}^0(\omega)$ can be reordered so that $\mathbf{S}^0(\omega)$ and $\mathbf{C}(\omega)$ can be re-written in a block matrix form as (Kumar et al., 2009; Williams and Seeger, 2001)

$$\mathbf{S}^0(\omega) = \begin{bmatrix} \mathbf{W}(\omega) & \mathbf{S}_{21}^{\mathsf{T}}(\omega) \\ \mathbf{S}_{21}(\omega) & \mathbf{S}_{22}(\omega) \end{bmatrix} \quad \text{and} \quad \mathbf{C}(\omega) = \begin{bmatrix} \mathbf{W}(\omega) \\ \mathbf{S}_{21}(\omega) \end{bmatrix}, \tag{3.25}$$

where $\mathbf{W}(\omega) \in \mathbb{R}^{c \times c}$ represents the intersection of the selected c columns with the corresponding c rows of $\mathbf{S}^0(\omega)$. Alternatively, the matrix $\mathbf{W}(\omega)$ can be seen as the CPSD matrix of the selected c landmark points. The matrices $\mathbf{S}_{21}(\omega) \in \mathbb{R}^{(n-c)\times c}$ and $\mathbf{S}_{22}(\omega) \in \mathbb{R}^{(n-c)\times(n-c)}$ represent block matrices formed from the remaining part of the matrix $\mathbf{S}^0(\omega)$.

Now, consider the eigen-decomposition of the intersection matrix $\mathbf{W}(\omega)$ as $\mathbf{\Psi}_w(\omega)\mathbf{\Lambda}_w(\omega)\mathbf{\Psi}_w^T(\omega)$,

where $\mathbf{\Psi}_w(\omega)$ is the orthogonal matrix of eigenvectors satisfying the requirement $\mathbf{\Psi}_w(\omega)\mathbf{\Psi}_w^T(\omega) = \mathbf{I}$, and $\mathbf{\Lambda}_w(\omega)$ represents a diagonal matrix containing the eigenvalues of $\mathbf{W}(\omega)$. Then, the Nyström based low-rank approximation of the matrix $\mathbf{S}^0(\omega)$ is given by (Williams and Seeger, 2001):

$$\mathbf{S}^0(\omega) \simeq \widetilde{\mathbf{S}}(\omega) = \mathbf{C}(\omega)\mathbf{W}^{\dagger}(\omega)\mathbf{C}^T(\omega), \tag{3.26}$$

where $\mathbf{W}^{\dagger}(\omega)$ denotes Moore-Penrose pseudo-inverse of the matrix $\mathbf{W}(\omega)$ and can be computed as,

$$\mathbf{W}^{\dagger}(\omega) = \mathbf{\Psi}_w(\omega)\mathbf{\Lambda}_w^{-1}(\omega)\mathbf{\Psi}_w^T(\omega) \tag{3.27}$$

in which $\mathbf{\Lambda}_w^{-1}(\omega)$ is the inverse of the diagonal matrix $\mathbf{\Lambda}_w(\omega)$ that can be computed very easily. Based on the Nyström method, the approximate eigenvalues $\widetilde{\mathbf{\Lambda}}(\omega)$ and eigenvectors $\widetilde{\mathbf{\Psi}}(\omega)$ of the CPSD matrix $\mathbf{S}^0(\omega)$ are given as (Kumar et al., 2009; Williams and Seeger, 2001):

$$\widetilde{\mathbf{\Lambda}}(\omega) = \left(\frac{n}{c}\right)\mathbf{\Lambda}_w(\omega) \quad \text{and} \quad \widetilde{\mathbf{\Psi}}(\omega) = \sqrt{\frac{c}{n}}\mathbf{C}(\omega)\mathbf{\Psi}_w(\omega)\mathbf{\Lambda}_w^{-1}(\omega). \tag{3.28}$$

In Eq. (3.28), the terms n/c and $\sqrt{c/n}$ are scaling factors introduced to compensate for the smaller sample size used to approximate the full CPSD matrix $\mathbf{S}^0(\omega)$. It should be noted that the matrices $\widetilde{\mathbf{\Lambda}}(\omega)$ and $\widetilde{\mathbf{\Psi}}(\omega)$ contain the approximate eigenvalues and eigenvectors only for the leading c modes, not for all modes. Therefore, $\widetilde{\mathbf{\Lambda}}(\omega)$ is a $c \times c$ diagonal matrix, while $\widetilde{\mathbf{\Psi}}(\omega)$ has a dimension of $n \times c$. This implies that the number of eigenmodes r that can be determined from Nyström method is always limited by the number of columns sampled i.e., $r \leq c \leq n$. Using the eigenvalues and eigenvectors determined from Eq. (3.28), the approximation to the matrix $\mathbf{D}(\omega)$ in Eq. (3.5) can therefore be constructed by:

$$\widetilde{\mathbf{D}}(\omega) = \mathbf{C}(\omega)\mathbf{\Psi}_w(\omega)\mathbf{\Lambda}_w^{-1/2}(\omega) \tag{3.29}$$

It can be shown theoretically that Nyström-based POD (NY-POD) approach can be very attractive to reduce the overall computing and memory cost for large-scale applications. As stated in Eq. (3.26), one of the main advantages of Nyström method is that we only need to compute the eigen-decomposition of the matrix $\mathbf{W}(\omega)$ which is relatively small compared to $\mathbf{S}^0(\omega)$. Furthermore, based on Eq. (3.29), to effectively compute the decomposed matrix $\widetilde{\mathbf{D}}(\omega)$, one only needs to evaluate and store a fraction of the CPSD matrix in $\mathbf{C}(\omega)$. To demonstrate this, if c is the number of columns sampled and the wind field simulation is performed using the first $r \leq c$ leading modes, the computational cost of performing eigen-decomposition of $\mathbf{W}(\omega)$ scales with $O(c^3)$ time, while the matrix multiplication part in Eq. (3.29) takes $O(nmr)$. Therefore, the total computational complexity of the Nyström method scales with $O(c^3 + ncr)$.

Provided that $c \ll n$, it is clear that the method offers a significant reduction in computational cost compared to factorizing the whole matrix at the cost of $O(n^3)$. In addition, the NY-POD method reduces the storage cost of the CPSD matrix from $O(n^2)$ to $O(cn)$.

Finally, the wind field simulation formula that uses the leading $r \leq c$ eigenvalues and the corresponding eigenvectors determined using the Nyström method can be derived by modifying Eq. (3.13) as

$$V_j(t) \simeq 2 \sum_{m=1}^{r} \sum_{l=1}^{N} \widetilde{D}_{jm}(\omega_l) \sqrt{\Delta\omega} \cos\left(\omega_l t + \phi_{ml}\right) \quad j = 1, 2, \ldots, n, \tag{3.30}$$

where $\widetilde{D}_{jm}(\omega_l)$ represents entries of the decomposed matrix $\widetilde{\mathbf{D}}(\omega)$ determined using Eq. (3.29). It should be noted that the simulation formula given in Eq. (3.30) asymptotically approaches the full eigen-decomposition when the number of columns sampled approaches the total number of columns. Similar to the standard POD-based formulation given in Eq. (3.13), the computation of the velocity time series by Eq. (3.30) can be significantly reduced by employing the FFT technique presented in Section 3.2. Hereafter, the proposed technique that uses Eq. (3.30) is termed as NY-POD method, as a shorthand for Nyström based POD approach. A step-by-step procedure for the proposed method is shown in algorithm 1. For faster execution, the proposed procedure is implemented in C++, and Appendix C provides details of the implemented numerical algorithm.

3.3.3 Error estimate of the Nyström method

It is important that approximation errors relative to eigen-decomposition are quantified in the form most relevant to the spectral representation method. Here, the approximation error associated with the Nyström method is characterized by comparing the proximity of the reconstructed matrix $\widetilde{\mathbf{S}}(\omega)$ to the target CPSD matrix $\mathbf{S}^0(\omega)$ in terms of matrix norm. The reconstructed approximate CPSD matrix $\widetilde{\mathbf{S}}(\omega)$ is given by:

$$\widetilde{\mathbf{S}}(\omega) = \begin{bmatrix} \mathbf{W}(\omega) & \mathbf{S}_{21}^{\mathsf{T}}(\omega) \\ \mathbf{S}_{21}(\omega) & \mathbf{S}_{21}(\omega)\mathbf{W}^{\dagger}(\omega)\mathbf{S}_{21}^{\mathsf{T}}(\omega) \end{bmatrix} \tag{3.31}$$

Comparing Eq. (3.31) with Eq. (3.25), it is evident that Nyström method perfectly reconstructs the three of blocks of the target CPSD matrix except for $\mathbf{S}_{22}(\omega)$ (Williams and Seeger, 2001). This can be physically interpreted as the method being capable of accurately representing the CPSD functions at the sampled points and all the CPSD functions between sampled and non-sampled points. Therefore, the error solely comes from the approximation of the CPSD matrix

Algorithm 1: Nyström based Spectral Representation Method

Input: Points $\mathbf{X}$, number of sample points c, number frequency steps N, number time steps N_T, cut-off frequency ω_u, CPSD function

Output: Velocity field $\mathbf{V}(t)$

$\mathcal{I} \leftarrow$ `SAMPLE-POINTS(`$\mathbf{X}, c, n$`)` `// select the representative points`

for $l \in [1 \dots N]$ **do**

  $\omega_l \leftarrow (l-1)\Delta\omega + \Delta\omega/2$

  $\mathbf{C}(\omega_l), \mathbf{W}(\omega_l) \leftarrow$ `CPSD(`$\mathcal{I}, \mathbf{X}, c, n, \omega_l$`)` `// sample columns from CPSD matrix`

  $\mathbf{\Psi}_w(\omega_l), \mathbf{\Lambda}_w(\omega_l) \leftarrow$ `EIG-DECOMP(`$\mathbf{W}(\omega_l)$`)` `// perform eigen-decomposition`

  $\widetilde{\mathbf{D}}(\omega_l) \leftarrow \mathbf{C}(\omega_l)\mathbf{\Psi}_w(\omega_l)\mathbf{\Lambda}_w^{-1/2}(\omega_l)$

$\mathbf{\Phi} \leftarrow$ `UNIFORM-DIST(`$0, 2\pi$`)` `// sample phase angles from uniform distribution`

for $j \in [1 \dots n]$ **do**

  for $m \in [1 \dots r]$ **do**

    for $l \in [1 \dots 2N]$ **do**

      if $l \leq N$ **then**

        $B_{jm}(\omega_l) \leftarrow 2\sqrt{\Delta\omega}\widetilde{D}_{jm}(\omega_l)\exp(\mathrm{i}\phi_{ml})$

      else

        $B_{jm}(\omega_l) \leftarrow 0$

    $A_{jm} \leftarrow$ `IFFT(`B_{jm}`)` `// perform inverse fast Fourier transform`

    for $p \in [1 \dots N_T]$ **do**

      $q \leftarrow p\%(2N)$

      $V_j(p\Delta t) \leftarrow V_j(p\Delta t) + \mathscr{R}\left\{A_{jm}(q\Delta t)\exp\left[\mathrm{i}\left(\frac{\Delta\omega}{2}\right)(p\Delta t)\right]\right\}$

formed using non-sampled points only i.e. $\mathbf{S}_{22}(\omega) \approx \mathbf{S}_{21}(\omega)\mathbf{W}^{\dagger}(\omega)\mathbf{S}_{21}^{\mathsf{T}}(\omega)$.

We measure the accuracy of the NY-POD method by calculating the reconstruction error in Frobenius norm defined as

$$\left\|\mathbf{S}^0(\omega) - \widetilde{\mathbf{S}}(\omega)\right\|_F = \sqrt{\sum_{j=1}^{n}\sum_{k=1}^{n}\left[S^0_{jk}(\omega) - \widetilde{S}_{jk}(\omega)\right]^2}. \tag{3.32}$$

However, a gross comparison using the Frobenius norm of the reconstruction error alone may not be enough. Hence, for a given modal truncation r, the performance of the NY-POD method and a full eigen-decomposition (POD) method are compared using a relative reconstruction error. Let $\mathbf{S}_r(\omega)$ and $\widetilde{\mathbf{S}}_r(\omega)$ represent the reconstructed CPSD matrices for the POD and NY-POD methods using the leading r eigenmodes. Then, the relative approximation errors for each method are defined as:

$$\begin{aligned}\epsilon_{\mathsf{pod}}(\omega) &= \frac{\|\mathbf{S}^0(\omega) - \mathbf{S}_r(\omega)\|_F}{\|\mathbf{S}^0(\omega)\|_F},\\ \epsilon_{\mathbf{nys}}(\omega) &= \frac{\|\mathbf{S}^0(\omega) - \widetilde{\mathbf{S}}_r(\omega)\|_F}{\|\mathbf{S}^0(\omega)\|_F}\end{aligned} \tag{3.33}$$

where $\epsilon_{\mathsf{pod}}(\omega)$ and $\epsilon_{\mathsf{nys}}(\omega)$ are relative reconstruction errors associated with the POD and NY-POD methods, respectively. These errors reflect the accuracy of each method to reproduce the CPSD matrix at a particular frequency ω using the leading r eigenmodes.

Considering the special structure of the CPSD matrix, it is possible to come up with a simple analytical bound to the reconstruction error of the NY-POD method. At the high-frequency end, the CPSD matrix becomes a diagonally dominated matrix. If we assume a homogeneous wind field, the reconstruction errors provided in Eq. (3.33) are bounded by:

$$\epsilon_{\mathbf{pod}}(\omega) \leq \epsilon_{\mathbf{nys}}(\omega) = \frac{\|\mathbf{S}^0(\omega) - \widetilde{\mathbf{S}}_r(\omega)\|_F}{\|\mathbf{S}^0(\omega)\|_F} \leq \sqrt{\frac{n-c}{n}} \tag{3.34}$$

We can deduce from this bound that the relative error approaches zero as we sample more columns (c) from the CPSD matrix. Derivation of the error limit expressed in Eq. (3.34) is demonstrated in Appendix B.

The relative errors given in Eq. (3.33) are important in evaluating the accuracy of each method at a particular frequency. Nevertheless, the CPSD functions for all frequencies do not contribute equally to the variance of the fluctuation. For most theoretical spectral models, such as von Karman spectra (Simiu and Scanlan, 1996), the contribution from the low-frequency fluctuations is usually large. Thus, to account for contribution from the full range of frequencies, by performing numerical integration, the areas under the CPSD functions (i.e., cross-covariances) are computed for each entry of the CPSD matrix, and the cumulative errors are

calculated as follows:

$$
\begin{aligned}
E_{pod} &= \frac{||\mathbf{\Gamma}^0 - \mathbf{\Gamma}_r)||_F}{||\mathbf{\Gamma}^0||_F} \\
E_{nys} &= \frac{||\mathbf{\Gamma}^0 - \mathbf{\Gamma}_r)||_F}{||\mathbf{\Gamma}^0||_F} \\
\Gamma_{jk} &= \sum_{l=1}^{N} 2\Delta\omega S_{jk}(\omega_l) \quad j,k = 1,2,\ldots,n
\end{aligned}
\tag{3.35}
$$

E_{pod} and E_{nys} can be considered as representative of the error in cross-covariance of the velocity field generated using the POD and NY-POD methods, respectively.

3.3.4 Column sampling schemes

Fundamental to the Nyström method is the choice of the matrix $\mathbf{C}(\omega)$. In addition to the number of columns sampled, the accuracy of the method greatly depends on the way this small subset of c-columns are sampled. Essentially, there are $\binom{n}{c}$ possible combinations to select c columns out of n. Hence, the main challenge here is to sample the best column subset that minimizes the reconstruction error $||\mathbf{S}^0(\omega) - \widetilde{\mathbf{S}}(\omega)||_F$. In the literature, different sampling techniques have been developed with varying levels of success. One of the most commonly used approaches is to sample the columns randomly from a uniform distribution. This technique has been predominantly used in computer science applications (Affandi et al., 2013; Wang and Zhang, 2013; Williams and Seeger, 2001). Figure 3.1a shows a schematic representation of the landmark points(solid dots) sampled randomly from uniform probability distribution on a two-dimensional setup.

For the current problem, however, because the CPSD matrix represents a velocity field on a grid of points spread in space, random sampling may not be well suited. In fact, for such kinds of applications, selecting columns corresponding to landmark points well-distributed in space is a better representative (De Silva and Tenenbaum, 2004; Liu et al., 2006; Silva and Tenenbaum, 2002). This type of sampling technique is referred to as the farthest point sampling (FPS) method. Figure 3.1b demonstrates the FPS technique with landmark points that are mutually far away from each other.

Another class of sampling methods particularly relevant to the current application is a "clustered" approach. For this method, first, the points are grouped into clusters, and then the centroid of each cluster is used as a landmark point (He and Zhang, 2018; Zhang and Kwok, 2010). It has been shown that Nyström approximation with clustering is highly effective and generally gives a very good empirical accuracy (Kumar et al., 2012). In the current study, the points are clustered on the basis of geometric proximity. We created the centers of the clusters using the

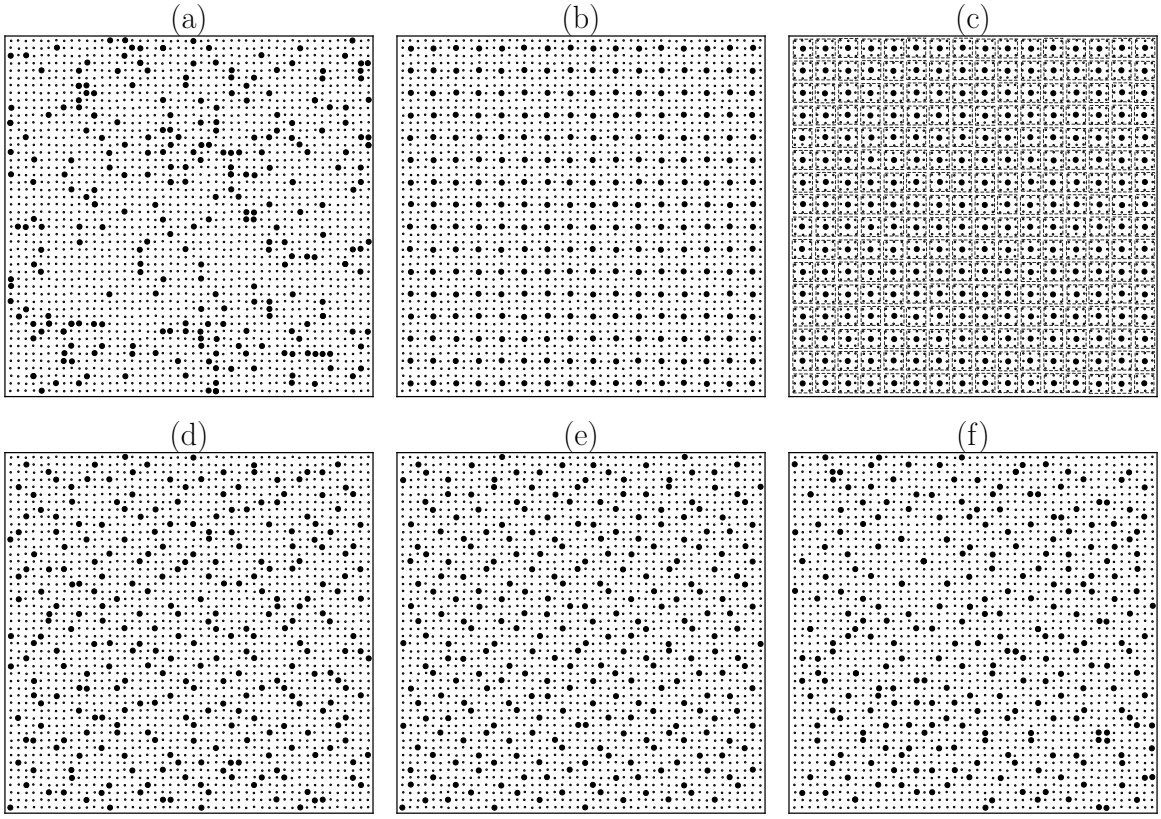


Figure 3.1: Sampling techniques used for points distributed over a plane: (a) pure random; (b) farthest point; (c) clustered; (d) Halton; (d) Hammersley; (d) Sobol

FPS method, and the neighboring points that are not sampled are assigned to the nearest center as shown in Figure 3.1c. Ultimately, an informative set of columns is calculated simply by taking the average of the columns corresponding to the points in each cluster.

In addition to the sampling techniques described above, three other sampling methods commonly used in Monte Carlo integration, which include Halton (Halton, 1960), Hammersley (Hammersley, 1960), and Sobol (Sobol, 1967) sequences, are investigated. Figure 3.1d-f depicts these sampling techniques for a two-dimensional space. Since these sampling methods are based on quasi-random (low-discrepancy) sequences, they show greater uniformity (Morokoff and Caflisch, 1994). As such, they cover the domain of interest more homogeneously and are expected to perform better than pure random sampling from a uniform distribution.

3.4 Numerical examples

To evaluate the accuracy and computational efficiency of the proposed method (NY-POD), two numerical examples are considered. The first example simulates a wind field on the deck of a

long-span bridge, assuming homogeneous wind velocity spectra. Then, in the second example, a non-homogeneous wind field is simulated on a 2D plane targeting application for boundary conditions of a transient CFD simulation.

3.4.1 Wind characteristics

For the numerical examples, the wind fields are generated based on the atmospheric boundary layer(ABL) profiles that characterize an open exposure condition with an aerodynamic roughness height of $z_0 = 0.03m$. Consider a Cartesian coordinate system with x, y and z representing the longitudinal, lateral, and vertical directions, respectively. The longitudinal component of the velocity field $\mathsf{V}(y, z; t)$ on the yz-plane can be represented as

$$\mathsf{V}(y, z; t) = \bar{V}(z) + V(y, z; t), \tag{3.36}$$

where $\bar{V}(z)$ represents the mean part of the velocity field and $V(y, z; t)$ refers to the fluctuating part described statistically by the CPSD functions. For any two velocity processes $V_j(t)$ and $V_k(t)$ at points j and k, their cross-spectral density functions are given by

$$S_{V_jV_k}(\omega) = \sqrt{S_{V_j}(\omega)S_{V_k}(\omega)}\mathrm{Coh}_{jk}(\omega). \tag{3.37}$$

Here, values of $S_{V_jV_k}(\omega)$ are used to form the target CPSD matrix $\mathbf{S}^0(\omega)$. The term $\mathrm{Coh}_{jk}(\omega)$ in Eq. (3.37) is a frequency dependent coherency function between $V_j(t)$ and $V_k(t)$, and defined as (Davenport, 1961b)

$$\mathrm{Coh}_{jk}(\omega) = \exp\left\{-\frac{|\omega|}{2\pi}\frac{\sqrt{\left[C_y\left(y_k - y_j\right)\right]^2 + \left[C_z\left(z_k - z_j\right)\right]^2}}{\frac{1}{2}\left[\bar{V}(z_j) + \bar{V}(z_k)\right]}\right\}, \tag{3.38}$$

where C_y and C_z refers to the coherency decay coefficients for the longitudinal component of the velocity in y and z direction, respectively. In the current study, $C_y = 10.0$ and $C_z = 16.0$ are specified based on the recommendation in Simiu and Scanlan (1996).

For the power spectral density function $S_V(\omega)$, the well-known von Karman model is used given by (Simiu and Scanlan, 1996):

$$\frac{S_V(f)}{\sigma_V^2(z)} = \frac{4\left[L_V(z)/\bar{V}(z)\right]}{\left[1 + 70.8\left[fL_V(z)/\bar{V}(z)\right]^2\right]^{5/6}}, \tag{3.39}$$

where $f = \omega/2\pi$ is the frequency; $\sigma_V(z)$ represent the standard deviation and $L_V(z)$ is the

Table 3.1: Summary of the main simulation parameters

Parameter	Values
Span	$L = 2000m$
Number of points	$n = 201$
Spacing between points	$\Delta y = 10m$
Height above the ground	$z = 50m$
Mean velocity	$\bar{V} = 40m/s$
Turbulence intensity	$I_V = 14.8\%$
Integral length scale of turbulence	$L_V = 421.4m$
Upper cut-off frequency	$\omega_u = 4\pi$ rad/s
Frequency intervals	$N = 4096$
Time step	$\Delta t = 0.25s$
Period of simulation	$T_0 = 4096s$
Number of time steps	$N_T = 16,384$

integral length scale of turbulence. For the current examples, the profiles of $\sigma_V(z)$ and $L_V(z)$ are determined based on the ESDU-85020 (2001) standard.

3.4.2 Example 1: Homogeneous wind field simulation over a line

For this example, the longitudinal component of the velocity field is simulated on points equally distributed over a linear domain. Since all the points are assumed to be on a horizontal line, the wind field is regarded as homogeneous turbulence. As such, the mean velocity, turbulence intensity and integral length scale are set to be uniform for all points on the deck. The parameters used in the simulation are summarized in Table 3.1.

Figure 3.2 shows the simulation points distributed over the line with landmark points used to form the matrix $\mathbf{C}(\omega)$. For farthest point sampling, since the simulation points are equally distanced, we get landmark points that are uniformly spaced at $\Delta y_s = (n/c)\Delta y$ as shown in Figure 3.2. Whereas, for clustered sampling, about the center of each cluster (see Figure 3.2), points within $\Delta y_s = (n/c)(\Delta y/2)$ distance are grouped together. The central points are the same as those used in the farthest point sampling arrangement. In the case of random sampling, however, the landmark points are arbitrarily assigned from a uniform probability distribution, and each landmark point is sampled at most once.

Before evaluating the statistics of the generated velocity field, first, let us compare the eigenvalues and eigenvectors found from the NY-POD method with the eigen-decomposition of the full CPSD matrix for this example. To do so, consider a case with a roughly 10%

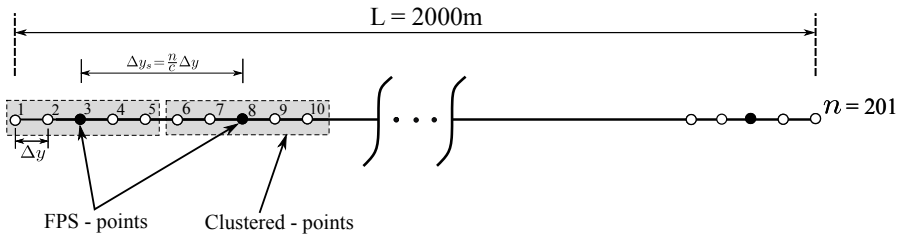


Figure 3.2: Distribution of simulation points and selected landmark points.

column sampling using a clustered approach that has $c = 20$ groups, each containing roughly 10 points. Figure 3.3 compares the eigenvalues approximated by the NY-POD method based on Eq. (3.28) with exact values determined using eigen-decomposition of the full CPSD matrix for the first four eigenvalues $\lambda(\omega)$. From Figure 3.3, it is evident that the first few eigenvalues are predicted with a high level of accuracy for the range of frequencies considered. Similarly, in Figure 3.4 the eigenvectors from NY-POD and POD methods are compared. The figure depicts the comparison of eigenvectors at three frequencies ($\omega = 0.01, 0.1, 1.0$) rad/s associated with the first eight eigenmodes. Considering the sinusoidal variation of the eigenvectors over the line, for ease of comparison, the curves shown in Figure 3.4 are compared by neglecting the differences in phase lags. Based on the comparison shown in Figure 3.4, the approximation from the NY-POD method is generally excellent for lower eigenmodes. The fact that the CPSD matrix has a low-rank structure (Di Paola, 1998) i.e., the first few eigenvalues carry a significant portion of the total energy, makes Nyström method clearly advantageous.

Now it is time to evaluate the performance of the proposed method in terms of the error metrics given in equations (3.33) and (3.35). Figure 3.5 illustrates this comparison. In this figure, the relative reconstruction error $\epsilon(\omega)$ determined using Eq. (3.33) is depicted for NY-POD and POD methods over the frequency range $\omega[0, \omega_u]$. The errors for the NY-POD method are calculated using 10%, 20%, and 30% of landmark points with clustered sampling scheme utilizing all the modes from the NY-POD approach (i.e., $r = c$). From Figure 3.5a, both NY-POD and POD methods give the highest accuracy in the low-frequency range as compared to the high-frequency end. However, it should be noted that, at the high-frequency end where the CPSD matrix becomes nearly diagonal, the accuracy of both methods approach the analytical error bound given in Eq. (3.34). Overall, for a given modal truncation r, the difference in reconstruction error between NY-POD and POD methods is very small over the frequency range considered.

Similarly, the reconstruction error determined using Eq. (3.35) is shown in Figure 3.5b. The figure shows the convergence of both methods by varying the number of modes r considered. As expected, when the percentage of columns sampled increases, the reconstruction error of

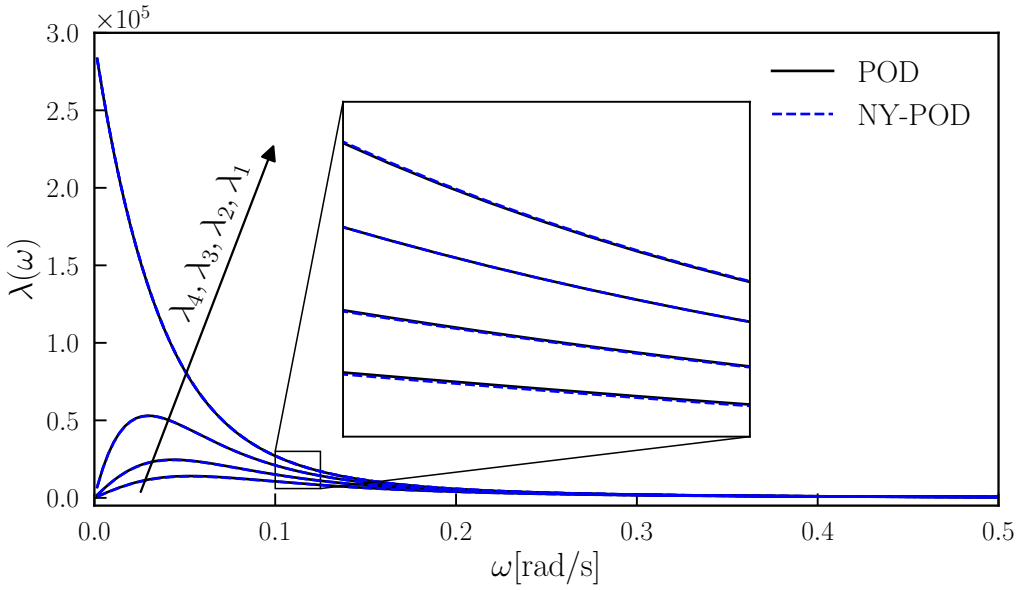


Figure 3.3: Comparison of the leading four eigenvalues from NY-POD with POD method.

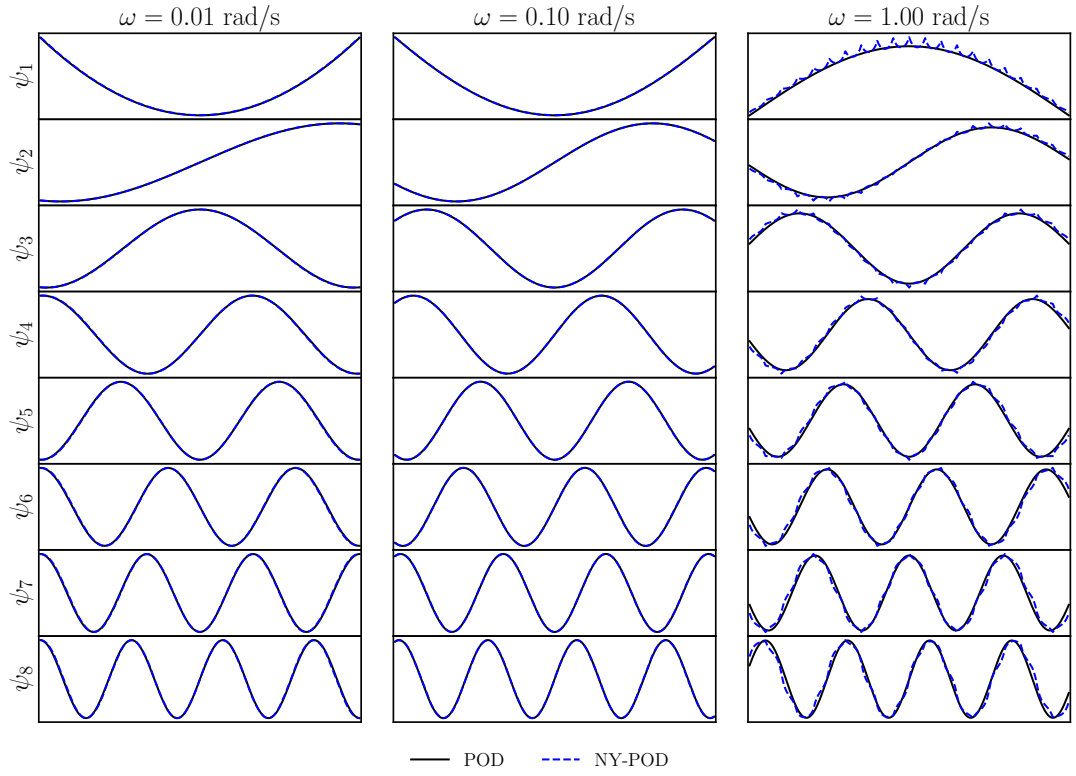


Figure 3.4: Comparison of the first eight eigenvectors (top to bottom) for NY-POD and POD methods at frequencies $\omega = 0.01, 0.1, 1.0$ rad/s.

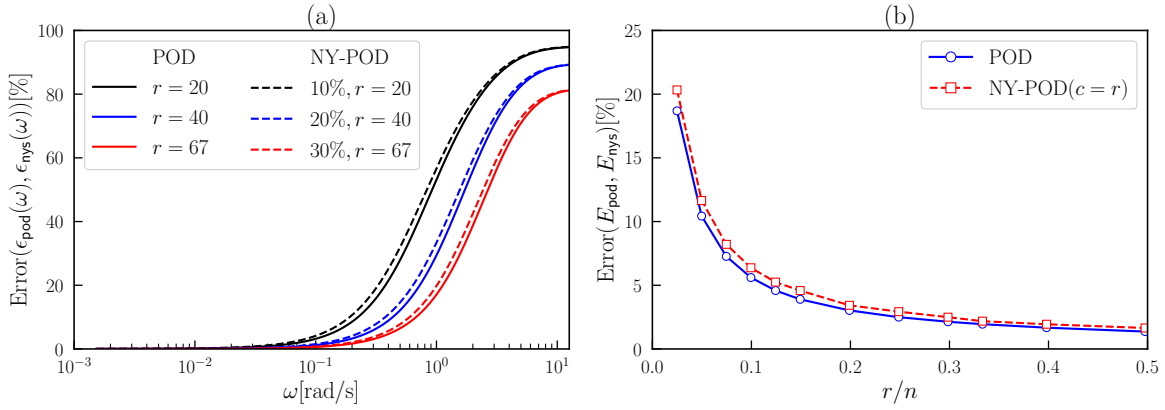


Figure 3.5: Comparison of the relative reconstruction error for NY-POD and POD methods: (a) $\epsilon_{nys}(\omega)$ and $\epsilon_{pod}(\omega)$; (b) E_{nys} and E_{pod}

the NY-POD method is generally reduced. However, sampling a large number of columns to construct a perfect Nyström approximation defies the main advantage of the method, making it computationally expensive. Therefore, it is important that an optimal number of landmark points c that produce the desired level of accuracy is determined.

Another important factor affecting the accuracy of the NY-POD method is the technique used to sample representative columns from the CPSD matrix. To demonstrate this, the performances of random, farthest point, and clustered sampling schemes are compared. Figure 3.6 shows this comparison using the same relative error metrics defined in Eqs. (3.33) and (3.35). From Figure 3.6, it is evident that the clustered sampling technique consistently outperforms both random and farthest point sampling methods for a wide frequency range. On the other hand, compared to random sampling, the results from farthest point sampling are more accurate, considering the fact that landmark points that are well spread in space tend to be more representative.

3.4.2.1 Comparison of the generated velocity field

Thus far, we have studied the efficacy of the NY-POD method primarily by assessing its accuracy to reconstruct the target CPSD matrix. Now, we evaluate its performance by comparing the statistics of the generated velocity field against the POD approach. To do so, the wind field on the 201 points shown in Figure 3.2 is simulated using both methods. For the NY-POD method, 20% of the columns were sampled using the clustered scheme as depicted in Figure 3.2.

The velocity field is generated utilizing the leading $r = c = 40$ modes for both methods. Figure 3.7 shows the time series of the generated velocity field at points 1, 2, 18 and 20 for the first 1000s. For the same set of points, Figure 3.8 shows the comparison of the estimated auto-

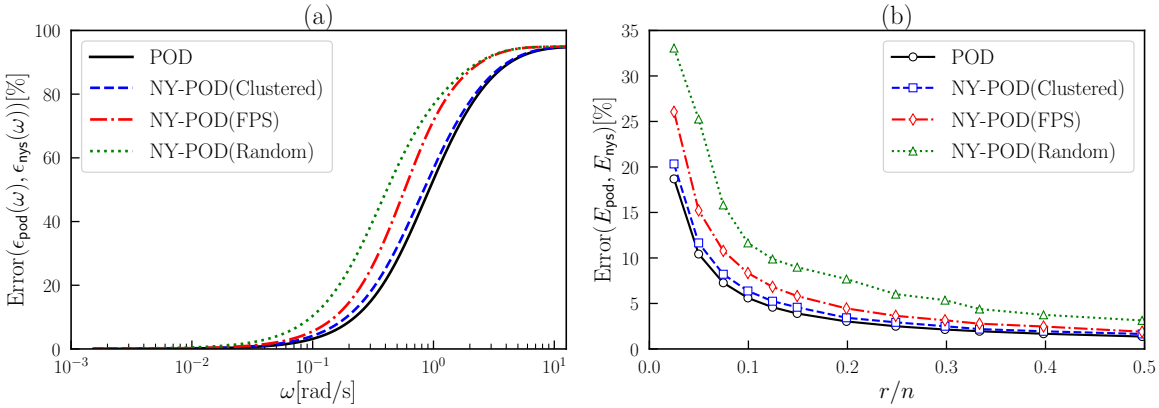


Figure 3.6: Effect of sampling technique on the relative reconstruction error of NY-POD method: (a) $\epsilon_{\text{nys}}(\omega)$ and $\epsilon_{\text{pod}}(\omega)$; (b) E_{nys} and E_{pod}

/cross PSD functions from NY-POD and POD methods against the target curves. Considering that the generated velocity process is not ergodic, the auto-/cross PSD functions shown in Figure 3.8 are statistical averages of 20 realizations. Figures 3.8c and 3.8d show the CPSD functions between points separated by 10m and 20m lateral distance, respectively.

As can be seen in Figure 3.8, the proposed method with clustered sampling scheme yields a spectrum that coincides with the one from the POD method and target function, especially in the low-frequency region. However, in the higher-frequency end, it appears that the estimated spectra from both NY-POD and POD methods exhibit loss. Such a compromise is primarily a consequence of truncating the higher modes that constitute high-frequency fluctuations. These differences were also manifested in the reconstruction error of the CPSD matrix presented in Figure 3.5 earlier. Nevertheless, the error introduced in the variance of the fluctuation as a result of sacrificing some portion of the high-frequency end of the spectra is usually small and can be tolerable depending on the application. Regarding the correlation functions, Figure 3.9 shows the estimated auto-/cross correlation functions in comparison with the target for the same points. The correlation functions shown in Figure 3.9 were determined using a biased correlation estimator. The target auto-/cross correlation functions were calculated by numerically integrating the Fourier transformation of the target CPSD functions given in 3.37. As can be seen from Figure 3.9, similar to the spectral comparison, the auto-/cross- correlation functions from NY-POD and POD methods are generally in satisfactory agreement.

An interesting observation from the proposed method lies in the accuracy of the generated time series at the landmark points sampled using the FPS and random schemes. Figure 3.10 shows the auto-PSD functions for the first two landmark points, 3 and 8. Notably, compared to the POD method, the auto-PSD functions $S_{3,3}$ and $S_{8,8}$ from the NY-POD method match

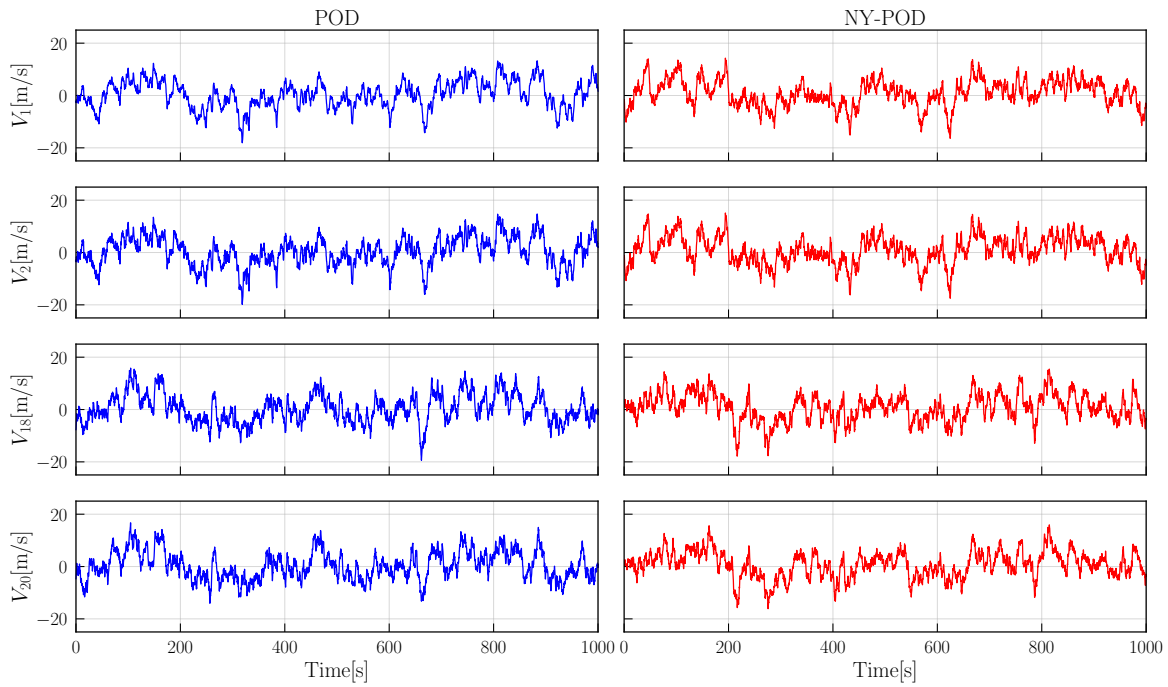


Figure 3.7: Generated sample velocity time-series for the first 1000 seconds at points $1, 2, 18$ and 20 (top to bottom): POD (left) and NY-POD (right).

the target auto-PSD accurately. This is because Nyström method perfectly reconstructs the intersection matrix $\mathbf{W}(\omega)$ of Eq. (3.25) that holds all the spectral information of the landmark points. Hence, if all the eigenmodes of the $\mathbf{W}(\omega)$ matrix are utilized for the landmark points, the results from the NY-POD method are equivalent to using the original POD formulation utilizing all the eigenmodes of the full CPSD matrix. This makes the NY-POD method particularly advantageous for applications where high accuracy is needed only at critical locations in the simulation domain. However, in the case of clustered sampling, since the representative columns are sampled as averages over each cluster, the accuracy at landmark points is compromised.

Finally, since the accuracies of both the POD and NY-POD methods greatly depend on the degree of modal truncation, it is worth investigating the effect of the number of modes used (r) on the statistics of the generated time series. Table 3.2 summarizes the comparison of the relative error in standard deviation averaged overall points using 20 realizations. The errors in the table are listed for random, farthest point, and clustered sampling schemes. The lowest error in standard deviation for the NY-POD method occurs for clustered sampling scheme. Overall, considering the significant reduction in computational cost, the relative error incurred by the NY-POD method is comparable with the original POD formulation for the same order of modal truncation.

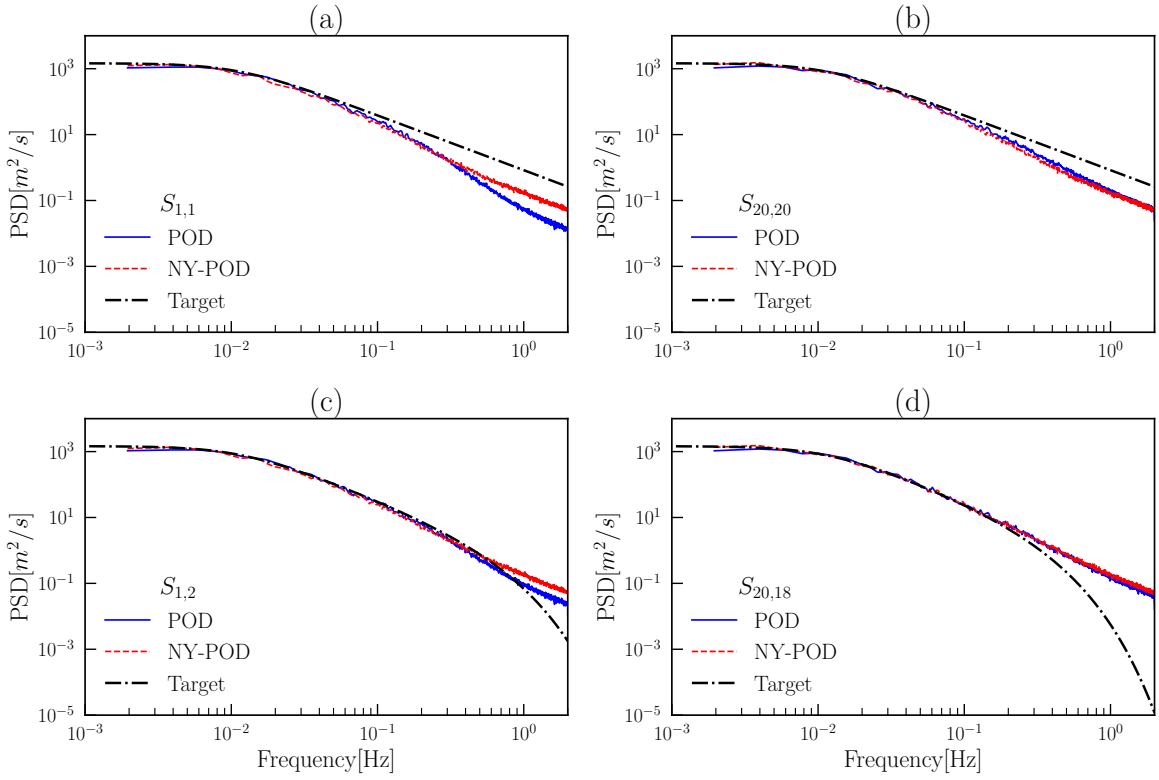


Figure 3.8: Estimated PSD functions of the velocity time-series generated by NY-POD and POD methods using the leading $r = 40$-modes: (a) auto-spectrum $S_{1,1}$; (b) cross-spectrum $S_{1,2}$; (c) auto-spectrum $S_{20,20}$; (d) cross-spectrum $S_{20,18}$

Table 3.2: Absolute relative error in standard deviation averaged over all points

	Error(%)					
Number of modes (r)	20	40	67	80	100	201
POD	10.91	7.12	4.98	4.35	3.59	1.23
NY-POD (Random)	16.96	12.27	8.21	7.05	5.73	1.23
NY-POD (FPS)	14.78	9.62	6.58	5.79	4.58	1.23
NY-POD (Clustered)	12.82	8.42	5.92	5.41	4.73	1.23

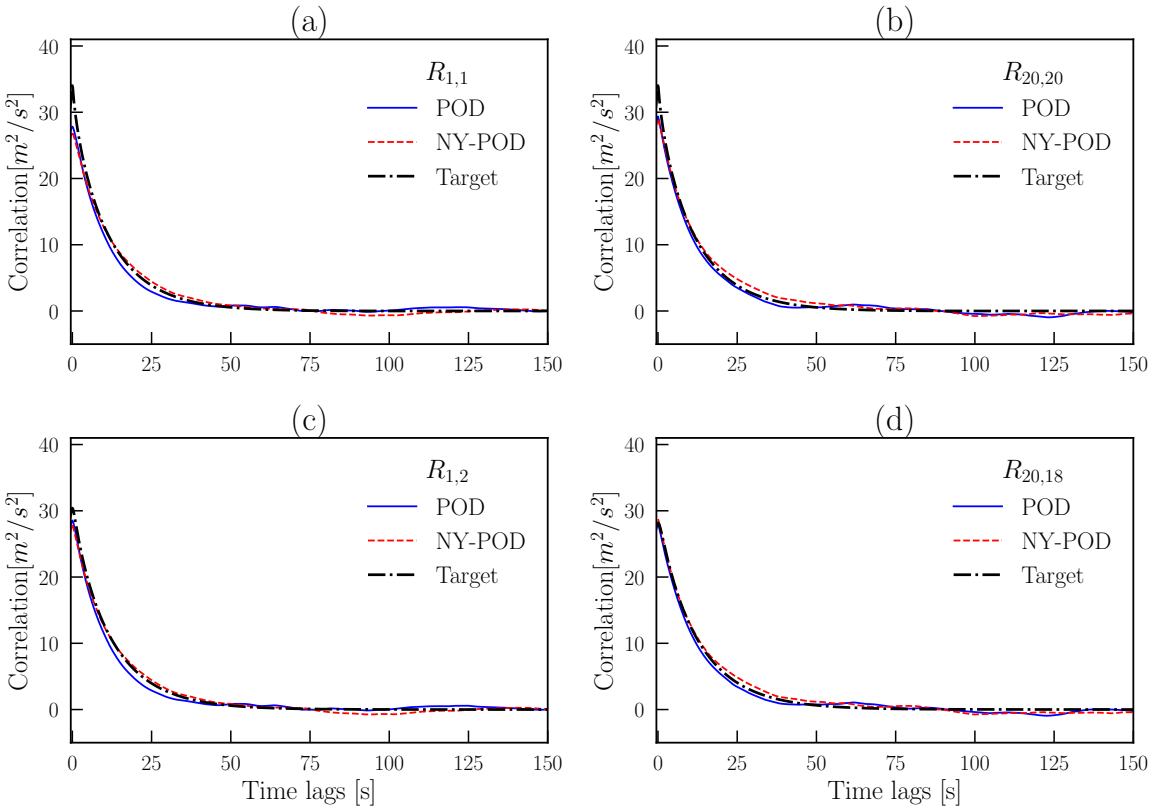


Figure 3.9: Estimated correlation functions of the velocity time-series generated by NY-POD and POD methods using $r = 40$-modes: (a) auto-correlation $R_{1,1}$; (b) auto-correlation $R_{20,20}$; (c) cross-correlation $R_{1,2}$; (d) cross-correlation $R_{20,18}$

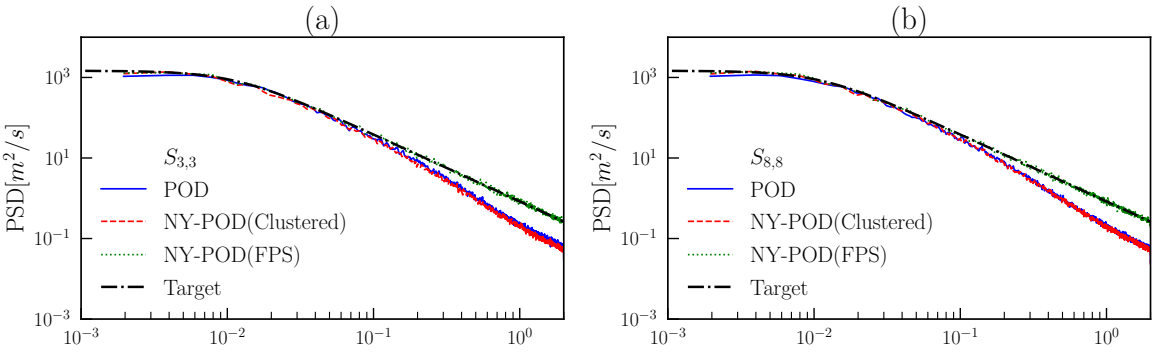


Figure 3.10: Comparison of sample auto-PSD functions at the landmark points: (a) Point 3; (b) Point 8

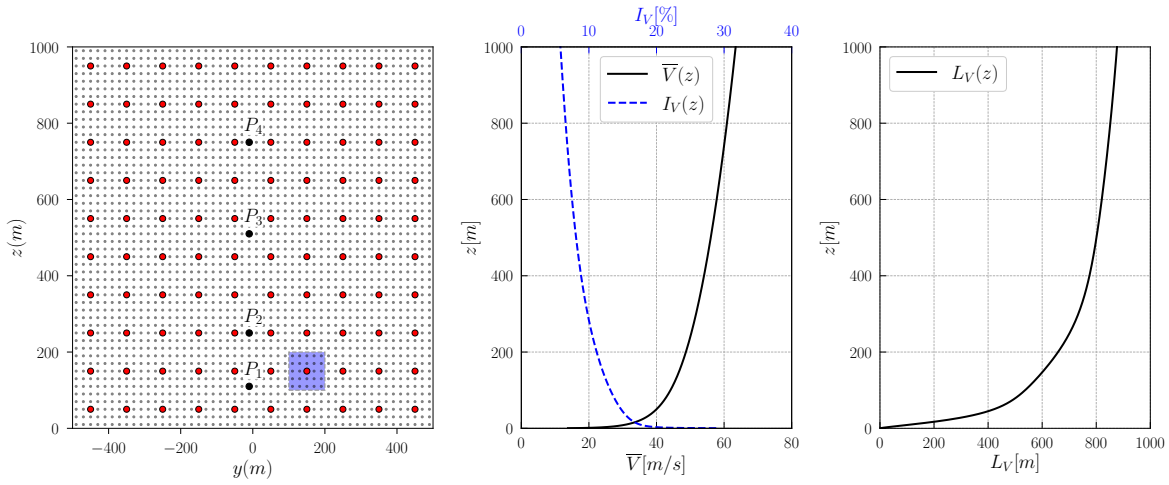


Figure 3.11: Simulation points and wind profiles: (a) grid of 50×50 simulation points with 100(4%) landmark points; (b) mean velocity and turbulence intensity profiles; (c) integral length scale profile

3.4.3 Example 2: Inflow generation for large-eddy simulation of ABL flow

In the previous example, we have demonstrated the application of the proposed method to simulate a homogeneous stochastic turbulent velocity field on a bridge deck. In this second example, the application for non-homogeneous turbulence on a two-dimensional plane is illustrated. This has interesting practical use in computational wind engineering studies to generate inflow turbulence for transient CFD methods such as large-eddy simulation (Melaku and Bitsuamlak, 2021). For such applications, one of the main challenges is the efficient generation of inflow turbulence on a large number of grid points, which can be improved using the proposed method.

The points used for simulation are defined by a grid on a square plane extending 1000 m in y- and z-directions. The grid consists of $50 \times 50 = 2500$ points as shown in Figure 3.11a. The wind profile is defined based on the ESDU-85020 (2001) standard for open exposure condition with a 10 m references wind speed $V_{10} = 37.1$m/s. Figures 3.11b-c show the mean velocity, turbulence intensity, and integral length scale profiles used for the simulation. Except for the distribution of grid points and wind profiles, the same simulation parameters used in the previous numerical example are adopted (see Table 3.1). For the NY-POD method, cases that represent 4%, 10%, and 25% column sampling using a clustered scheme were considered. To run all the simulations in this example, we used a device with 32-core Xeon(R) 2.2GHz E5-4620 CPUs, 256GB RAM, and Linux CentOS 7.6 operating system.

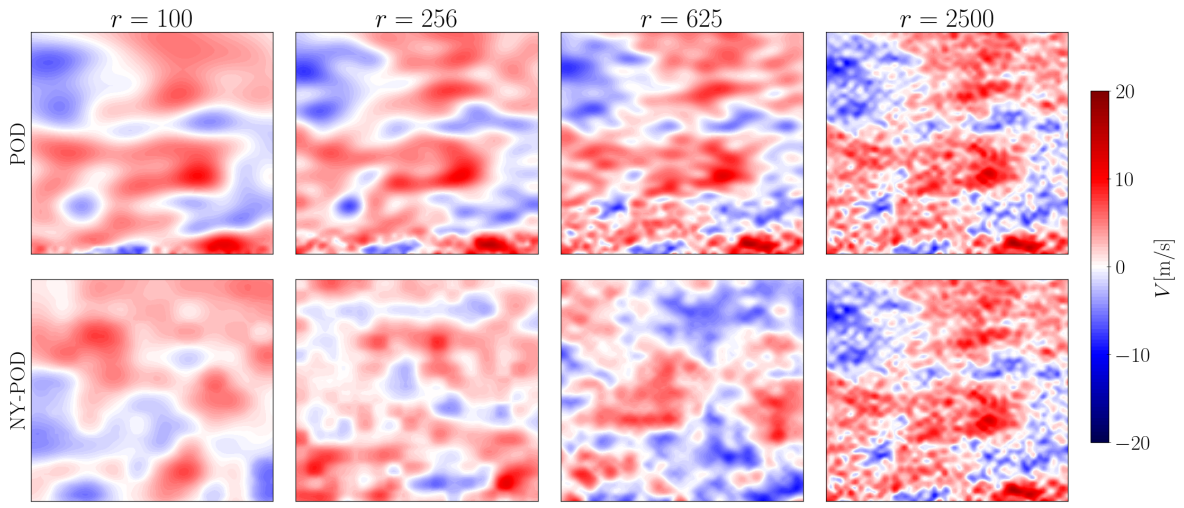


Figure 3.12: Snapshot of the generated instantaneous velocity field. The plots from the NY-POD method are based on clustered sampling with a gird of $c = 10 \times 10, 16 \times 16, 25 \times 25$, and 50×50 landmark points (left to right).

Next, the velocity time series on all points are simulated for different modal truncation values. Figure 3.12 shows the snapshot of the generated velocity field over the plane for NY-POD and POD methods using the leading $r = 100, 256, 625$, and 2500 modes. For the NY-POD approach, the plots correspond to landmark points with 4%, 10%, 25%, and 100% sampling, respectively. Note that for NY-POD, all the computed modes are used, i.e., $r = c$. Visual inspection of the velocity contours in Figure 3.12 indicates that the NY-POD method is able to generate a velocity field that has similar flow features to the POD approach. It can be seen that the eigenmodes are associated with different flow structures. For instance, lower modes are associated with fluctuations that cover large areas, while the higher modes give rise to small-scale fluctuations. This is in line with the classical view of turbulence as a superposition of coherent structures of various shapes and sizes.

Figure 3.13 shows the estimated auto-PSD functions of the velocity time-series at four points indicated in Figure 3.11a as P_1, P_2, P_3, and P_4, which are located at $z = 110$ m, 250 m, 510 m, and 750 m, respectively. The comparison of the two methods is based on the leading 625 modes that correspond to 25% sampling in the case of the NY-POD method. Similar observations noted in the first example were witnessed in this example as well. Except near the high-frequency region, which is affected by modal truncation, the auto-PSD function from NY-POD generally follows the one from POD. Figure 3.14 demonstrates the effect of modal truncation on the high-frequency end. The plots in Figure 3.14 are shown for point P_3 using the leading 100, 256, 625, and 2500 modes. Although the overall accuracy of the POD-based method is superior when compared to the NY-POD method, it was observed that the estimated

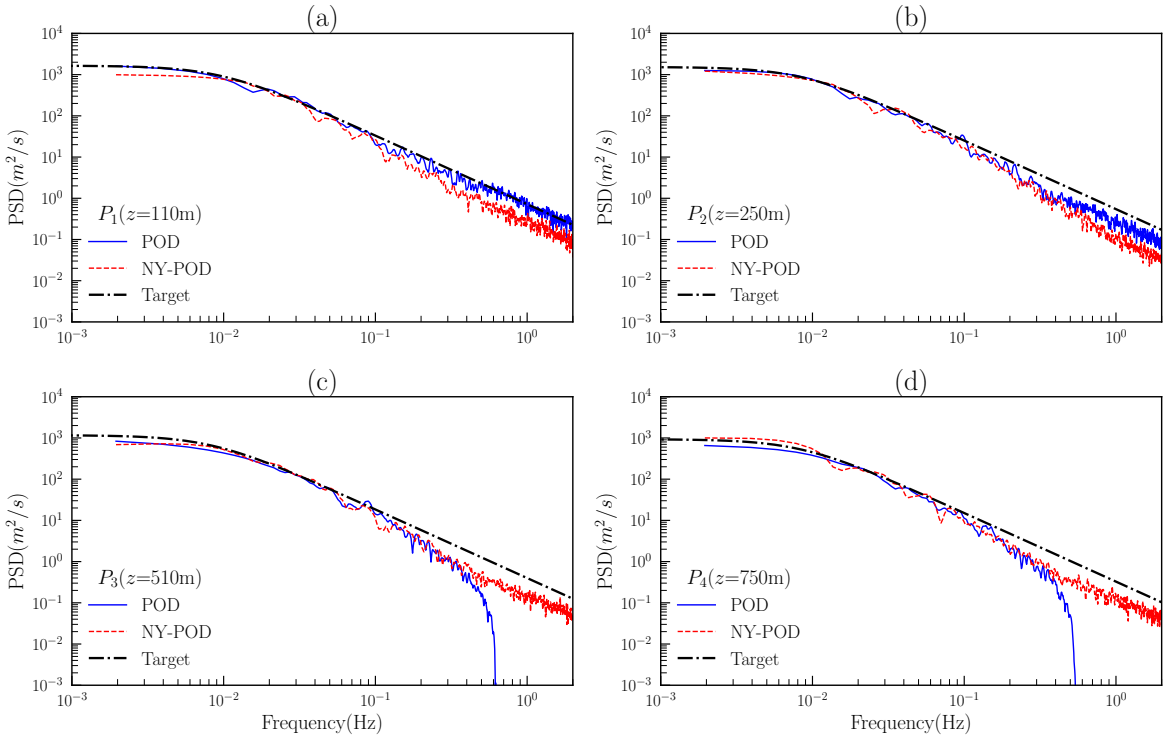


Figure 3.13: Comparison of auto-PSD functions from POD and NY-POD method using the leading 625 modes: (a) Point 1; (b) Point 2; (c) Point 3; (d) Point 4

spectra from the NY-POD method do not sharply drop near the high-frequency end (see Figure 3.14).

Finally, we demonstrate the numerical accuracy and computational efficiency of the proposed method. Figure 3.15a shows the mean relative error in the standard deviation of the generated time series averaged over all the simulation points. The errors are plotted as a function of the percentage of modes utilized. Also, the same figure compares the relative performance of all the sampling techniques used for the NY-POD method illustrated in Figure 3.1. Based on Figure 3.15a, over the range of columns sampled, clustered sampling showed the lowest error compared to the other sampling methods. The quasi-random approaches, including Halton, Hammersley, and Sobol sampling methods, generally performed better than pure random sampling. Also, for this example, the quasi-random methods performed as good as, in some cases, even better than the farthest point sampling method. This is attributed to the spatial uniformity of the quasi-random methods, which results in a sample more representative of the simulation domain. Overall, based on results from the first and the current numerical example, the clustered technique remains the best alternative among the sampling methods tested.

Figure 3.15b shows the normalized CPU time for the NY-POD method against the percentage of columns sampled. The CPU times shown in Figure 3.15b are normalized by the total

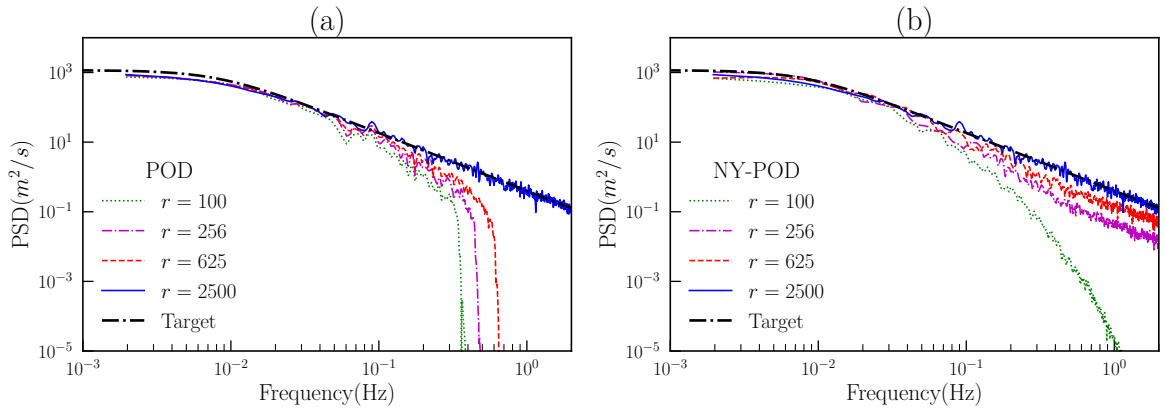


Figure 3.14: Comparison of estimated auto-PSD functions for different modal truncations(r): (a) POD; (b) NY-POD

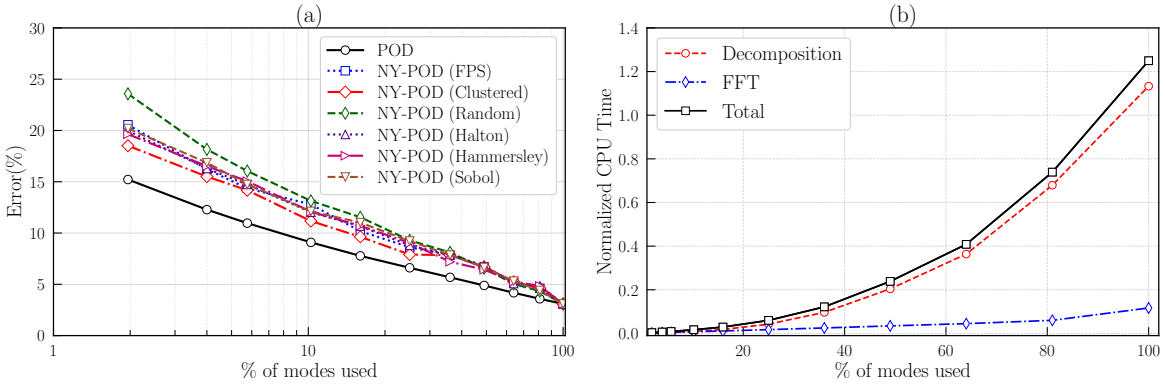


Figure 3.15: Performance comparison between NY-POD and POD methods: (a) relative error in standard deviation; (b) normalized CPU time for NY-POD method

time consumed by the POD-based approach utilizing all the eigenmodes. Also, in the same figure, the computational time for decomposing the CPSD matrix and generating the velocity field employing the FFT technique is shown. Looking into Figure 3.15b, it is not a surprise that for a large number of points, the decomposition of the CPSD matrix is the major computational bottleneck when compared to simulating the velocity time series. Recalling that the computational complexity of the NY-POD method scales with $O(c^3 + ncr)$, its performance is especially significant when compared to the decomposition of the full matrix at the cost of $O(n^3)$. For instance, as shown in Figure 3.15b, with 25% columns sampling, the proposed procedure takes only 6.65% of the total CPU time required for the POD-based method with the same order of modal truncation. Whereas the error for the NY-POD method is 7.90% for cluster sampling, which appears to be comparable to the 6.61% error of the POD-based method.

3.5 Summary and conclusion

A computationally efficient spectral representation method is introduced to simulate a multivariate wind velocity field on a large number of points using a low-rank representation of the cross-power spectral density (CPSD) matrix. The proposed method uses the Nyström approximation to compute the most pertinent eigenmodes of the CPSD matrix without having to decompose the entire matrix. By systematically sampling only a small portion of the CPSD matrix, it was demonstrated that the current method could achieve remarkable computational efficiency with a minimal compromise in numerical accuracy. Furthermore, it was observed that the performance of the proposed method, to a great extent, depends on the sampling technique used to construct informative columns from the CPSD matrix. Among the studied sampling techniques, the clustered scheme yielded the highest numerical accuracy compared to random, farthest-point, and other sampling schemes commonly used in quasi-Monte Carlo methods. We then used the proposed method to simulate homogeneous as well as non-homogeneous wind turbulence for inflow turbulence generation applications. For the same order of modal truncation, the auto-/cross- power spectral density functions estimated from the velocity time series generated using the proposed method and the standard POD-based approach are generally in good agreement. Thus, with the application of the NY-POD method, one can significantly reduce the CPU and memory cost of generating highly correlated inlet turbulence for large-eddy simulations of atmospheric boundary layer flows using the spectral representation method.

Chapter 4

LES for predicting wind loads and responses of a standard tall building: prospect for wind-resistant tall building design

4.1 Introduction

Over the past three decades, fueled by the burgeoning advancements in computing power, considerable progress has been made in using computational fluid dynamics (CFD) for wind engineering applications (Blocken, 2014a; Dagnew and Bitsuamlak, 2014; Murakami, 1990; Stathopoulos, 1997). The increased extreme wind-induced built-environment damage, which is multi-scale (i.e., ranging from components and buildings to neighborhood and cities) and multi-physics (i.e., wind-structure interaction, wind-driven-rain (Blocken and Carmeliet, 2004), snow drift (Tominaga and Stathopoulos, 2011), wind-born-debris (Kakimpa et al., 2010), and other climate stressors (Kahsay et al., 2021)), is demanding integrated numerical and experimental techniques that can handle the scale and physics complexities of the engineering problems. Furthermore, the need to have a preliminary assessment of wind-induced loads, responses, damage, and loss in a quick and approximate manner at the early stages of planning and designing a project is becoming far more beneficial than accurate results at the end of the project life cycle. The other driving factor for using computational methods like CFD stems from the need for iterative solutions in the design optimization process where numerical approaches coupled with efficient surrogate models are inherently suitable (e.g., aerodynamic optimization of tall buildings and long-span bridge sections) (Birhane et al., 2017; Elshaer and Bitsuamlak,

2018; Elshaer et al., 2017a). There are still challenges in CFD application, however, especially for wind load evaluation that require adequate modeling of lower ABL turbulence conditions for estimating peak loads and responses with reasonable accuracy.

For the wind-resistant design of structures, computational methods offer several advantages over conventional wind tunnel testing. Most importantly, because of their high versatility and shorter model development cycle, they can seamlessly be integrated into most design iterations where experimental testing cannot be achieved within a given time frame. Moreover, CFD provides enormous information about the relevant flow field without having virtually any physical constraints. These advantages make CFD a promising tool to serve as a complementary tool to wind tunnels for the foreseeable future before eventually becoming widely used by the industry when large-scale computing resources become more abundant and verified and validated CFD workflows and guidelines become available. Despite these advantages, the accuracy and reliability of the CFD for computational wind load evaluation is still a primary concern (Aboshosha et al., 2015c; Cochran and Derickson, 2011; Dagnew and Bitsuamlak, 2013; Melaku et al., 2022; Ricci et al., 2018) at the moment. The wind-tunnel modeling had undergone several validation efforts with full-scale measurements before it enjoyed industry-wide acceptance (Dalgliesh, 1975; Davenport, 1988; Holmes, 1975; Newberry, 1967). For CFD to become a reliable wind engineering tool, it needs comparable and extensive verification and validation against bench-mark experimental and field measurements focusing on turbulence modeling. Even more, stricter guidelines and streamlined validation schemes are required, especially for non-expert users. To handle these multi-faceted challenges, the development of CFD models of varying levels of fidelity, complemented with established experimental techniques for calibration and validation, and a standardized guideline is essential.

In the past, notable research efforts have been made to study wind loads on buildings using CFD by employing turbulence models of various levels of fidelity. Computational modeling of wind flow around bluff objects like tall buildings is challenging mainly because it is a high Reynolds number flow characterized by peculiar flow features like impingement, separation, reattachment, recirculation, and vortices shedding. Resolving the entire turbulent spectra for such type flows using direct numerical simulation (DNS) of the governing Navier–Stokes equation is simply computationally intractable even for the most simplified cases, let alone for a building located in a city center. Thus, resorting to some turbulence closure scheme remains the only viable alternative. The most commonly used turbulence models for computational wind engineering applications are the Reynolds-averaged Navier-Stokes equations (RANS) and large-eddy simulation (LES). The RANS-based models provide the statistical average of turbulence quantities, while LES captures the transient nature of the flow yielding higher-order statistics, such as standard deviation and peak. In LES, large scales of the turbu-

lent fluctuations are resolved up to the smallest grid scale (filter width), and only the effect of small scales of turbulence is modeled. For a detailed review of the turbulence closure schemes commonly utilized in computational wind engineering studies, the reader is advised to refer to the work of Murakami (1998). Among the available turbulence modeling methods, large-eddy simulation (LES) is now becoming a powerful tool because of its inherent capability to handle complex unsteady turbulent flows that are commonly encountered in wind engineering applications (Dagnew and Bitsuamlak, 2014; Tamura, 2008). Due to this merit, LES is now becoming a valuable tool in different subdomains of wind engineering, including structural loading, pedestrian-level wind, pollutant dispersion, and ventilation studies.

Since the current study focuses on modeling transient wind loading and responses of tall buildings, here we present a brief review of the related works, primarily focusing on LES-based wind load evaluation studies with turbulent atmospheric boundary layer (ABL) conditions. A more comprehensive review of the current progress in computational wind load evaluation of buildings can be found in Dagnew and Bitsuamlak (2014); Thordal et al. (2019). Previous studies have shown that the flow field around tall buildings can be simulated with satisfactory accuracy (Murakami, 1993; Tominaga et al., 2008a). However, predicting the unsteady surface pressure fluctuations, especially peak values, is often challenging and requires substantial mesh refinement near the building surface (Nozu et al., 2008, 2015). Despite the encouraging success of LES for many industrial flows (Fureby, 2008; Piomelli, 1999; Sagaut and Deck, 2009; Tucker and Lardeau, 2009), applications to wind load evaluation have been limited until quite recently. By performing an aerodynamic wind load evaluation on the CAARC building with LES, Huang et al. (2007) showed that the fluctuating pressure coefficients agreed with the experimental data better than those from RANS-based models. However, the power spectral densities of the integrated aerodynamics were not accurately captured mainly due to the limitation of the turbulent inlet boundary conditions used (Aboshosha et al., 2015c; Huang et al., 2007). Later on, developing an inflow turbulence generator suited for ABL flows, Huang et al. (2010) showed that surface pressure fluctuations and base aerodynamic loads could be predicted with better accuracy if inflow turbulence with prescribed target velocity spectra is specified at the inlet. Employing the method developed by Huang et al. (2010), different validation studies have been conducted mainly on the CAARC (Dagnew and Bitsuamlak, 2014; Li et al., 2015; Zhang et al., 2022, 2015). Other related studies that investigate surface pressure fluctuations can also be found in the works of Daniels et al. (2013); Lamberti and Gorlé (2020, 2021).

Several studies have compared the performance of different inflow turbulence generation methods for tall building aerodynamics (Melaku et al., 2017; Yan and Li, 2015). Aboshosha et al. (2015c) incorporated Davenport's coherency function (Davenport, 1961b) into the in-

flow proposed by Huang et al. (2010) improving its performance for CWE applications. They demonstrated that proper modeling of the frequency-dependent coherency function is crucial for accurately predicting tall building wind-induced response, especially if the building is dynamically sensitive. The inflow turbulence generator has been refined further by Castro et al. (2017) and Yu et al. (2018). Utilizing this inflow turbulence generation method, Elshaer et al. (2016) investigated surface pressure distribution and wind induced-response of isolated and surrounded CAARC model and reported a reasonable agreement with experimental measurements.

Most studies discussed so far focus on validating the mean and root-mean-square (RMS) of surface pressure fluctuations for a selected few wind directions using limited pressure points. Although validating these quantities is the initial step, it is not sufficient to establish the use of LES for evaluating cladding loads and overall structural loads and responses (Ricci et al., 2018). The accurate modeling of peak pressure is crucial for adequately predicting cladding loads. Typically this analysis is done over the 360° azimuth at an increment of 10°. Moreover, matching the mean and RMS statistics of local surface pressure fluctuations with good precision may not always result in a more accurate prediction of overall structural loads and responses. It is well understood that the overall structural loads and responses are highly influenced by the structure's mechanical properties, such as natural frequency, mass, stiffness, and damping (Davenport, 1971). Notably, the unsteady aerodynamic loads with frequencies near the natural frequency of the structure considerably affect its wind-induced response. Hence, during the LES modeling process, proper consideration must be given to adequately resolve the spectral content of these unsteady aerodynamic loads. It is well recognized that the accuracy of an LES model is highly dependent on the boundary conditions, the subgrid-scale model, and the quality of the computational grid used (Fureby, 2008; Tucker and Lardeau, 2009).

Among the LES parameters, the inflow boundary condition is the most influential parameter to accurately model the target statistical properties of the atmospheric boundary layer (ABL) for accurate wind load prediction. To this end, Melaku and Bitsuamlak (2021) proposed a new synthetic inflow turbulence generation method that satisfies one- and two-point statistics of ABL flows. The technique named the Divergence-free Spectral Representation (DFSR) method uses the spectral representation method (Deodatis, 1996b; Shinozuka and Jan, 1972) to generate spatially and temporarily correlated inflow turbulence. Although it showed improvement in modeling the characteristics of ABL flows compared to its predecessors (Aboshosha et al., 2015c; Huang et al., 2010), its broader applicability for wind load applications has not been demonstrated. This study aims to investigate the application of LES to predict the cladding loads and the wind-induced response of a standard tall building – CAARC (Commonwealth Advisory Aeronautical Research Council) – using the DFSR inflow generator (Melaku and

Bitsuamlak, 2021).

The remaining part of this paper is organized into five sections. Section 4.2 briefly describes the target boundary layer wind tunnel measurement conducted to validate the LES model. In Section 4.3, the details of the LES modeling process are described. The dynamic analysis procedure and structural properties used to calculate the wind-induced response of the building are presented in Section 4.4. Section 4.5 discusses the comparison of the LES results with experimental measurements. Finally, Section 4.6 provides a summary and conclusion of the present numerical study.

4.2 Boundary layer wind tunnel experiment for LES validation

A set of boundary layer wind tunnel tests were conducted on the CAARC building to generate aerodynamic data for validating the LES results. The experiment was conducted in Tunnel II of the Boundary Layer Wind Tunnel Laboratory (BLWTL) at Western University. Measurements of the approaching wind profiles and the surface pressure fluctuations on the CAARC building were conducted at a 1:400 geometric scale. The CAARC building is a rectangular prismatic building with a full-scale height H = 182.88 m, width B = 45.72 and depth D = 30.48 m as shown in Figure 4.1b.

4.2.1 Target atmospheric boundary layer flow

The test facility used is a close-circuit boundary layer wind tunnel with a working section of approximately 3.4 m wide and 2.5 m high, and 39 m long fetch length. The long working section of the tunnel allows the development of thick turbulent boundary layers over a rough tunnel floor covered with generic automated roughness blocks. The tunnel's inlet has a saw-tooth strip, three spires, and one solid stripping board to model the turbulence characteristics. These additional turbulence-generating features allow the thickening of the boundary layer, which increases turbulence intensity and integral length scale without significantly altering the mean velocity profile (Davenport and Isyumov, 1967).

The atmospheric boundary layer (ABL) flow is simulated at the same scale as the building model. The simulation of the exposure condition is based on the characteristics of the target ABL flow provided in Engineering Sciences Data Unit (ESDU, 2001a; ESDU-85020, 2001). The targeted upstream exposure condition for the present study is an open terrain characterized by an aerodynamic roughness height of z_0 = 0.025 m. The target mean velocity and turbulence

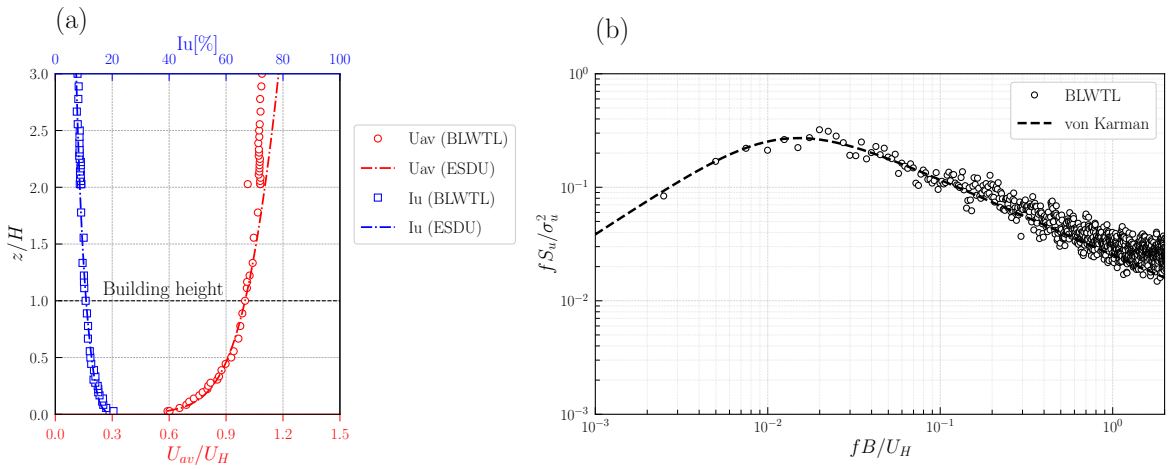


Figure 4.1: Characteristics of BLWT ABL flow: (a) mean velocity and stream-wise turbulence intensity profiles; (b) stream-wise velocity spectrum at the roof height in comparison to von Karman spectrum.

intensity profiles for the longitudinal component of the velocity were based on the ESDU-85020 (2001). The wind tunnel profiles are measured on the turntable just upstream of the turntable center. The velocity is recorded using hot-wire velocity probes placed on a vertically traversing rig for about 90 seconds at a 400 Hz sampling rate. Although the hot-wire probes provide high-resolution instantaneous flow velocity, the measurements are only limited to the stream-wise direction. Figure 4.1 compares the measured wind tunnel profiles against the targets. As shown in the figure, the mean wind speed and turbulence profiles simulated in the testing facility reproduce the turbulence characteristics of the natural wind given by ESDU-85020 (2001) standard. Similarly, the spectrum of the longitudinal component of the velocity is compared with the von Karman spectrum in Figure 4.1b.

4.2.2 High-frequency pressure integration model

Aerodynamic loads on the CAARC building were measured using a high-frequency pressure integration (HFPI) model. The HFPI model is 3D-printed from plastic material and equipped with pressure taps that are connected to pressure transducers to measure the wind-induced pressure. Figure 4.2a shows the close-up views of the pressure model with the upstream terrain simulation setup. Overall, eleven wind directions (θ) from 0° to 90° in increments of 10° and additional 45° were tested. The pressure distribution on the exterior surface of the building is synchronously measured using 367 pressure taps distributed over the building surface as shown in Figure 4.2b. The pressure time history was recorded at a sampling rate of 400 Hz. From previous similar experimental measurements, it was estimated that the pressure tubing

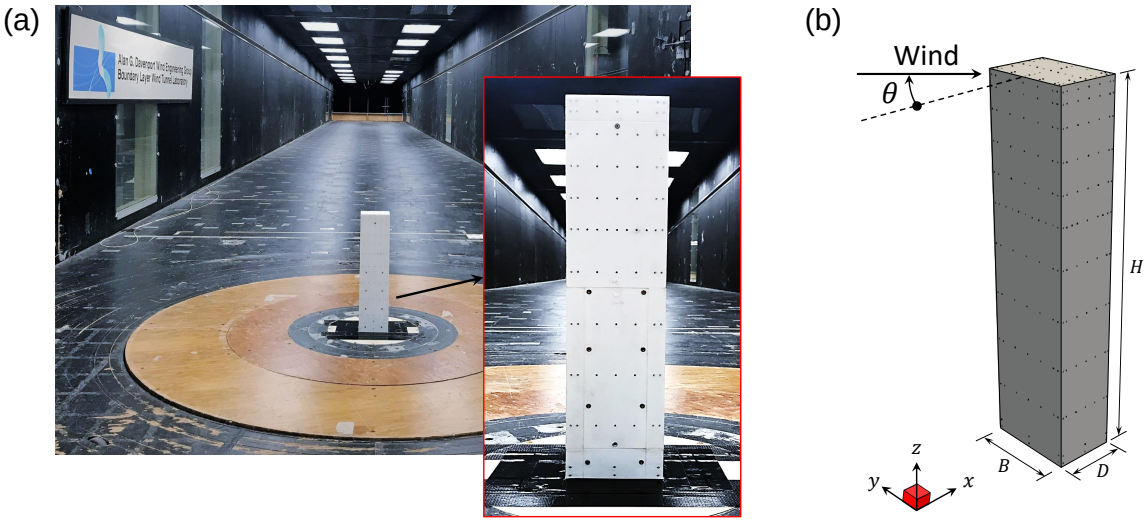


Figure 4.2: Wind tunnel model of the CAARC building: (a) picture of the HFPI model at the test section; (b) plan dimensions and definition of coordinate system

system used has negligible attenuation for frequencies only up to about 200 Hz (BLWTL, 2007; Ho et al., 2005). Therefore, a low-pass digital filter of 200 Hz is applied to the final pressure data. The pressure fluctuations are referenced to the free-stream mean static pressure recorded upstream of the turntable. Finally, the surface pressure coefficients are calculated by normalizing the measured pressures by the mean dynamic pressure at the building height. The pressure tests were conducted at a roof-height mean wind speed of $U_H = 12.35$ m/s and turbulence intensity of 10.6%. The Reynolds number based on the roof-height mean wind speed and building height is $Re = 3.76 \times 10^5$.

4.3 LES modeling

All the cases described in the BLWTL experimental test (presented in Section 4.2) were modeled using LES. The simulations were performed in model scale at 1:400 geometric scale, the same as the BLWT experimental setup. The simulation conditions of the LES models are set to match the experiments as much as possible. Table 4.1 summarizes the steps followed in wind load evaluation in boundary wind tunnels and the corresponding computational steps that needs to be followed to produce comparable results are also articulated in the same table. The ABL is assumed to be neutrally stratified, and the flow develops over rough, homogeneous terrain. Considering the high computational cost of wall-resolved LES for high Reynolds number flows, the simulations in the current study are conducted using a wall-modeled LES. All the simulations were conducted employing open source CFD toolbox OpenFOAM-8 (Greenshields and Weller, 2022a; Weller et al., 1998).

Table 4.1: Guiding the numerical procedure with experience from wind tunnel

Step	Wind tunnel procedure	Numerical procedure
1.	Test profile determination through upwind terrain roughness assessment using standard methods	Detailed upwind roughness and topography modeling in large-size computational domain simulation or proper inflow turbulence generation by using synthetic methods (Aboshosha et al., 2015c; Melaku and Bitsuamlak, 2021)
2.	Construction of physical aerodynamic study building model	Preparation of accurate three-dimensional study-building aerodynamic computer model
3.	Inner disc trace and immediate surrounding building construction within 500 m radius from the study site	Preparation of three-dimensional computer model for the immediate surrounding buildings and topographic elements within 1 km radius (Elshaer et al., 2016, 2017b)
4.	Upwind tunnel floor roughness and spire adjustment based on test profile requirement	Appropriate implicit or explicit roughness modeling method coupled with proper inflow turbulence generation method (Abdi and Bitsuamlak, 2014a; Aboshosha et al., 2015a; Melaku and Bitsuamlak, 2021)
5.	Wind tunnel testing for typically 36 wind directions by turning the turn table and different test configurations (un-sheltered, present, future configurations)	Numerical simulation for 36 wind directions by creating a computational domain and generating grids appropriate for each wind direction and different configurations (un-sheltered, present, future configurations)
6.	Wind tunnel data analysis to obtain overall drag (along-wind forces) and lift (across-wind forces) for design, detailed pressure coefficient (C_p) distributions on the faces of the buildings being studied, or on any required portion of the building for C&C design	Numerical output data analysis to extract overall drag (along-wind forces) and lift (across-wind forces) for design, detailed pressure coefficient (C_p) distributions on the faces of the buildings being studied, or on any required portion of the building for C&C design
7.	Integrate wind tunnel data with local meteorological information at the study site to account for directional effects (Warsido and Bitsuamlak, 2015)	Integrate numerical output data with local meteorological information at the study site to account for directional effects (Warsido and Bitsuamlak, 2015)
8.	Obtain design wind loads and other wind-induced responses as required	Obtain design wind loads and other wind-induced responses as required

4.3.1 Governing equations

Assuming the wind flow around the building as an incompressible fluid, the filtered Navier-Stokes equations that govern the motion of the large-scale eddies can be written as:

$$\frac{\partial \widetilde{u_i}}{\partial x_i} = 0, \tag{4.1}$$

$$\frac{\partial \widetilde{u_i}}{\partial t} + \frac{\partial}{\partial x_j}\left(\widetilde{u_i}\widetilde{u_j}\right) = -\frac{1}{\rho}\frac{\partial \widetilde{p}}{\partial x_i} + \nu\frac{\partial^2 \widetilde{u_i}}{\partial x_j \partial x_j} - \frac{\partial \tau_{ij}}{\partial x_j}\,, \tag{4.2}$$

where $\widetilde{u_i}$ is the i-th component of the filtered velocity, $\widetilde{p}$ represents the filtered pressure, ν is the kinematic viscosity, and ρ refers to the density of the fluid. The shear stress term τ_{ij} represents

the effect of small-scale eddies and is expressed in the form of a residual term as

$$\tau_{ij} = \widetilde{u_i u_j} - \widetilde{u}_i \widetilde{u}_j. \tag{4.3}$$

This expression is called the subgrid-scale (SGS) stress and is often modeled using the eddy-viscosity hypothesis. The fundamental premise of LES is that the SGS motions are spatially homogeneous and hence can be modeled universally. The primary role of the SGS model is to dissipate energy from the resolved scales (Nicoud and Ducros, 1999). For the current study, the standard Smagorinsky SGS model (Smagorinsky, 1963) is adopted as

$$\tau_{ij} - \frac{1}{3}\delta_{ij}\tau_{kk} = -2\nu_t \widetilde{S}_{ij}, \;\; \widetilde{S}_{ij} = \frac{1}{2}\left(\frac{\partial \widetilde{u}_i}{\partial x_j} + \frac{\partial \widetilde{u}_j}{\partial x_i}\right), \tag{4.4}$$

where δ_{ij} is the Kronecker delta, $\widetilde{S}_{ij}$ represents the strain rate tensor of the resolved velocities, and ν_t represents turbulent eddy-viscosity. For small-scale turbulent motions, assuming that the energy production and dissipation are in equilibrium, ν_t is obtained by

$$\nu_t = (C_s \Delta)^2 |\widetilde{S}|, \qquad |\widetilde{S}| = \sqrt{2\widetilde{S}_{ij}\widetilde{S}_{ij}}. \tag{4.5}$$

Here, the parameter Δ represents the filter width defined as the cube root of the cell volume. Whereas, the coefficient C_s is Smagorinsky's constant and can be expressed as (Sullivan et al., 1994)

$$C_s = \left(C_k \sqrt{\frac{C_k}{C_\epsilon}}\right)^{1/2}, \tag{4.6}$$

in which the coefficients C_k and C_ϵ are constants associated with the production and dissipation of sub-grid scale turbulent kinetic energy. Assuming the grid size lies within the inertial sub-range, the constants C_k and C_ϵ can be obtained from spectral analysis of Kolmogorov's $-5/3$ spectrum (Lilly, 1967; Moeng and Wyngaard, 1988). For ABL flows, the value of C_s usually used with the standard Smagorinsky SGS model range between 0.1 and 0.2 (Vasaturo et al., 2018). For the current study, we used $C_k = 0.094$ and $C_\epsilon = 1.048$ which yields a Smagorinsky constant $C_s = 0.1678$. In order to study the sensitivity of the numerical results to the Smagorinsky constant C_s used, by changing the value of C_k, three values, specifically $C_s = 0.1, 0.17$ and 0.2, were also investigated. Because of its simplicity and low computational cost, the standard Smagorinsky model is widely used in CWE literature with considerable success (Murakami, 1998; Tominaga et al., 2008a). However, it is well known that it produces excessive dissipation close to solid boundaries (Sullivan et al., 1994; de Villiers, 2006). To account for this effect, near the solid walls, the coefficient C_s is multiplied by the van Driest damping factor given as

(Van Driest, 1956):

$$D(y^+) = 1 - \exp\left\{-\frac{y^+}{A^+}\right\} \tag{4.7}$$

where y^+ is normalized wall coordinate and $A^+ = 26$ is the van Driest constant.

For comparison purposes, in addition to the standard Smagorinsky model, two other SGS modeling techniques, namely the Wall Adapting Local Eddy-viscosity (WALE) model (Nicoud and Ducros, 1999) and the one equation eddy-viscosity model developed by Yoshizawa (1986) were studied for 0° wind direction. The performance of the three SGS models for predicting wind loads is compared employing the same computational grid and numerical setup.

4.3.2 Dimensions of the computational domain

The choice of the computational domain is often dictated by the extent of the region of interest and the boundary conditions. The building needs to be sufficiently far away from the domain's boundaries to minimize the effects of the boundary conditions (Dagnew and Bitsuamlak, 2013; Franke et al., 2011; Mahaffy et al., 2007). Furthermore, the domain should be large enough to encompass the largest energy-carrying eddies in the atmospheric boundary layer. In contrast, having a smaller computational domain is always preferable to reduce computational costs. Therefore, the adequacy of the domain size needs to be decided based on a sensitivity analysis. In the current study, however, since comparison with experimental measurement is the primary interest, the dimensions of the computational domain were adapted from the wind tunnel where the validation measurements were carried out.

Figure 4.3 shows the extent of the computational domain employed with the boundary condition types. The cross-sectional width and height of the computational domain are set to 3.4 m ($7.5H$) and 2.5 m ($5.5H$), which are the same as the width and height of the wind tunnel at the turntable (see the description in Section 4.2). Since the inflow is generated synthetically (see Section 4.3.4.1), the length of the computational domain is set to 10 m ($22H$) instead of the long fetch length of the wind tunnel used to develop the flow. The size of the current computational domain satisfies the minimum requirements of the COST recommendations (Franke et al., 2011). The maximum blockage ratio of the model, which occurs at the oblique wind angle of attack, is approximately 0.75%. This value of blockage ratio is significantly smaller than the 3% limit recommended by Franke et al. (2011) as well as the maximum acceptable blockage ratio for wind tunnel studies of buildings and structures (e.g., 8% in ASCE-49-21 (2022)). The computational domain dimensions are kept the same for all simulated cases.

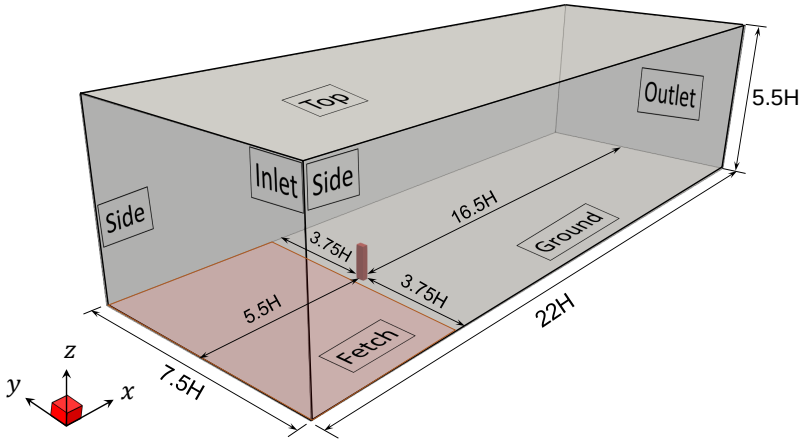
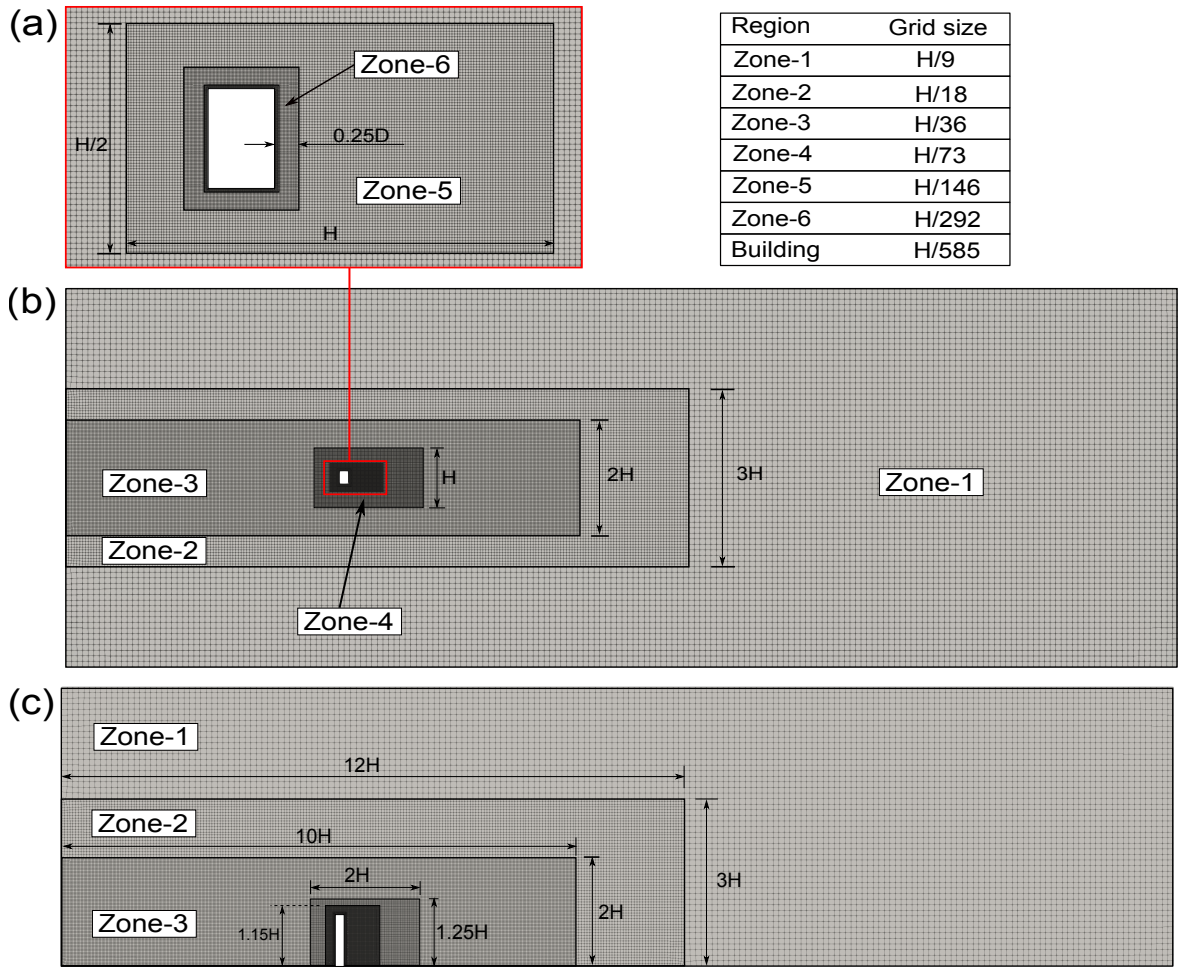


Figure 4.3: Dimensions of the computational domain relative to the building height(H) and naming of boundaries.

4.3.3 Mesh generation

For the computational grid, we used unstructured mesh with several refinement regions as shown in Figure 4.4. The computational grid is generated using NUMECA HEXPRESS® meshing tool with the hex-dominant meshing option. Because the flow features in the vicinity of the building are more critical in determining the wind loads, the computational grid is designed to have progressive refinement close to the building surface. The grid size used in each refinement region is shown in Figure 4.4. The first two additional refinement zones (Zone-2 and Zone-3) extend from the inlet into the wake of the building to sustain the inflow turbulence up to the building location. The extent of the refinement zones remains the same for all wind directions. However, the orientation of Zone-6 is changed together with the building, depending on the wind direction. On the surface of the building, an additional seven-cell thick surface refinement is applied. Overall, the total number of cells per case ranges between 9.85 – 10.7 million, depending on the wind direction. The y^+ values on the building surface were directly estimated from an initial test run using the computational grid generated for the 0° wind direction. In general, it was found that the y^+ values range between 0.4 to 35, depending on the location on the surface. The time-averaged maximum y^+ value on the surface of the building is approximately 35, which occurs near the building's upwind edges where high flow acceleration happens. However, the mean area averaged y^+ on the surface is approximately 7. Considering the unstructured nature of the mesh, some non-orthogonality is expected. Non-orthogonality of the grid measures the angle between the line connecting the centers of two adjacent cells and the normal of the face shared by them. It gauges how rectangular the mesh is and assumes a zero value for a fully rectangular structured mesh. For the current study, the maximum non-



Region	Grid size
Zone-1	H/9
Zone-2	H/18
Zone-3	H/36
Zone-4	H/73
Zone-5	H/146
Zone-6	H/292
Building	H/585

Figure 4.4: Design of the computational grid showing all the mesh refinement zones (sample case for 0° wind direction): (a) close-up sectional view near the building; (b) horizontal section; (c) longitudinal section. The mesh sizes in each zone are given relative to the building height.

orthogonality of the generated grid ranges between 46.5° and 53.7°. It should be noted that maximum non-orthogonality up to 65° is often deemed mild for the type of solver used in the current study (Greenshields et al., 2015).

The same computational domain with a slightly different mesh is also prepared for an empty domain configuration. For this domain, only refinement regions from Zone-1 up to Zone-4 (see Figure 4.4) are used with 4.9 million cells. The empty domain simulation is later used to measure the undisturbed incident wind profile without the effect of the study building.

Three additional meshes were generated to investigate the discretization error associated with the computational grid. The computational grid in each case was systematically refined to capture essential flow phenomena. Assuming that the flow dynamics near the building, i.e., impingement from upcoming flow, flow separation, and the turbulent wake, are primarily

Table 4.2: Computational grids used for mesh sensitivity study

Case	Mesh regions used	**Smallest gird size	No. cells
G1	Zone-1 to Zone-4	$H/146$	4.78×10^6
G2	Zone-1 to Zone-5	$H/293$	7.04×10^6
G3*	Zone-1 to Zone-6	$H/585$	9.85×10^6
G4	Zone-1 to Zone-6	$H/1170$	1.65×10^7

*G3 is used for the final wind load simulations.

** The size of mesh used on the building surface.

responsible for most of the wind loads, the meshes used for the grid sensitivity study only differ in the vicinity of the building. All the mesh sensitivity studies were conducted for 0° wind direction. Table 4.2 outlines the details of the test grids (G1, G2, and G4) and the grid used for the final simulations (G3), which is depicted in Figure 4.4. The mesh size given in Table 4.2 is the size of mesh used on the building surface. Cases G3 and G4 have the same mesh refinement zones; however, for G4, two levels of surface refinement, each having 7-cell thickens, are employed on the building surface.

4.3.4 Boundary Conditions

Boundaries of the computational domain should be carefully defined with appropriate boundary conditions. Especially, validation studies require the formulation of boundary conditions consistent with the experimental setup. Generally, for ABL flows, the transient inflow boundary condition at the inlet and the rough wall boundary condition on the ground surface usually need special consideration (Melaku and Bitsuamlak, 2021). Furthermore, appropriate wall treatment on the surface of the building is required, especially if all the flow features up to the viscous region are not resolved. These boundary conditions are separately discussed in Sections 4.3.4.1, 4.3.4.2 and 4.3.4.3.

On the side and top faces of the domain, slip, and zero-gradient boundary conditions are adopted for velocity and pressure fields, respectively. It is important to note that for most computational wind load applications (Daniels et al., 2013; Elshaer et al., 2016; Huang et al., 2010), at the side and top faces, a symmetric boundary condition is commonly chosen for velocity, which resembles the slip boundary condition used in the current study. Considering that the top and side boundary conditions are sufficiently far from the area of interest, they are expected to have very little effect on flow near the building even if they do not perfectly reproduce the actual boundary of the experimental facility used for the testing. At the outlet, a Neumann boundary condition with zero gradients is applied for the velocity, while a fixed

value of zero pressure is specified for the pressure field.

4.3.4.1 Inflow turbulence generation

At the inlet of the computational domain, a time-dependent turbulent velocity field is specified for the velocity. The turbulence is generated using the DFSR method developed by Melaku and Bitsuamlak (2021). The method uses a computationally efficient implementation of the spectral representation technique to synthesize a velocity field for a given target wind profile and velocity spectra. Here, only a brief description of the DFSR method is given. For a detailed description of the numerical procedure, the reader is advised to refer to the cited publication.

The specified target mean velocity, turbulence intensity, and integral length scale profiles were taken from an empty tunnel measurement with the same test setup (see Section 4.2). In addition to the turbulence intensity profile, shear stress profiles were modeled by imposing a correlation between the longitudinal and vertical wind velocity components. The target shear stress profile used in the DFSR method is taken from the experimental measurements.

The turbulent inlet wind field is generated as a three-dimensional stochastically stationary process that varies with position and time. The procedure starts by first defining the target cross-power spectral density (CPSD) functions for the three components of the velocity. The CPSD functions are determined from the velocity spectrum and frequency-dependent coherency function. For this study, the velocity spectra are defined based on the well-known von Karman model, while for the coherency function, Davenport's exponential model (Davenport, 1961b) is adopted. The velocity spectra is discretized using $N = 8192$ frequency segments. At the inlet of the computational domain, there are about 1.15×10^4 points, and decomposing the CPSD matrix formed from all these points for each discrete frequency is computationally expensive. Therefore, the target CPSD matrix is decomposed only using 25 systematically sampled frequencies, and for intermediate frequencies, spline interpolation is employed. The maximum theoretical cut-off frequency of the simulated velocity spectra is set to $f_{\max} = 200$ Hz, which corresponds to a time step of $dt = 0.0025$ s based on Nyquist sampling theorem. Note that the sampling frequency used for the inflow is in line with the experimental measurements. Finally, the velocity time series is computed using the Fast Fourier Transform (FFT) technique for a duration of 38 s.

For incompressible flows, it is required that the mass flow rate entering the computational domain should be constant to avoid undesired pressure fluctuations (Gungor et al., 2012; Kim et al., 2013; Patruno and de Miranda, 2020; Poletto et al., 2011). However, due to finite sampling of the inflow points at the inlet and uneven contribution from low-frequency velocity fluctuations, the mass flow rate of the generated inflow varies with time. Thus, the instantaneous stream-wise velocity field generated using the DFSR method is re-scaled to have a

constant flow rate defined by the mean velocity profile.

It is worth noting that the incident profiles measured near the building often differ from the one specified at the inlet because of the downstream evolution of the flow. This results in a noticeable decay of turbulence downstream. Therefore, for this study, the target turbulence intensity profiles used at the inlet are adjusted until the desired level of turbulence is achieved at the location of incidence (Lamberti et al., 2018; Melaku and Bitsuamlak, 2021). The adjustment is made iteratively by a scaling factor determined from the ratio between the target to measured turbulence intensity profiles at the location of the study building.

Finally, to save computational cost, the inflow turbulence is generated once and stored in a database, and the wind load simulations (for all wind directions) are run using the same inflow data.

4.3.4.2 Ground surface roughness modeling

The effect of upstream terrain roughness is represented using a wall modeling technique. As shown in Figure 4.3, the bottom patch of the computational domain is divided into two parts. The upstream part is labeled "fetch" and stretches $5.2H$ from the inlet. A rough-wall boundary condition is applied for this patch, while a no-slip boundary condition is used for the remaining part of the bottom surface. This is analogous to most boundary-layer wind tunnels (including the one used in the current study), where the roughness blocks are typically placed upstream of the test section, and the remaining part of the tunnel floor is a smooth surface. Furthermore, this makes it easy to use a bottom boundary condition based on the logarithmic wind profile where the flow can be treated as relatively homogeneous (Cheng and Porté-Agel, 2013).

The rough wall boundary condition used for this study is based on the Schumann–Grötzbach model (Grötzbach, 1987; Schumann, 1975). The model works by specifying the instantaneous surface shear stress determined from the log-law relationship and the filtered velocity at the wall-adjacent cell center. For a neutrally stratified ABL flow, the wall shear stress $\tau_{i3}^{w}(x, y, t)$ is calculated as (Churchfield et al., 2010; Porté-Agel et al., 2011)

$$\tau_{i3}^{w}(x, y, t) = -\left[\frac{\kappa \widetilde{U}(z_p, t)}{\log[(z_p + z_0)/z_0]}\right]^2 \frac{\widetilde{u}_i(x, y, z_p, t)}{\widetilde{U}(z_p, t)}, \qquad (i = 1, 2) \tag{4.8}$$

where z_0 represents the aerodynamic roughness height, κ is the von Karman constant ($\kappa = 0.41$), $\widetilde{U}(z_p, t) = [\langle\widetilde{u}_1(x, y, z_p, t)\rangle^2 + \langle\widetilde{u}_2(x, y, z_p, t)\rangle^2]^{1/2}$. Here, $\widetilde{u}_1$ and $\widetilde{u}_2$ are filtered velocities at the center of the wall-adjacent cell in the stream-wise and span-wise directions, respectively. The symbol $\langle\cdot\rangle$ denotes a span-wise average, and z_p represents the mid-height of the wall adjacent cell. Note that because of the regional refinements used, the mid-height of the wall-adjacent

cells varies over the ground surface. Therefore, z_p used in Eq.(4.8) is determined by taking the area-weighted average in the fetch region.

4.3.4.3 Building surface

Considering the high computational cost of resolving the wall boundary layer up to the viscous regime, a boundary condition based on a smooth-wall function is specified on the building surface. This boundary condition works by providing a constraint on the turbulent viscosity using Spalding's formula (Spalding, 1961). Spalding's law provides a universal formulation for the velocity profile in laminar, buffer, and turbulent core regions by

$$y^+ = u^+ + \frac{1}{E}\left[\exp\left(\kappa u^+\right) - 1 - \kappa u^+ - \frac{(\kappa u^+)^2}{2!} - \frac{(\kappa u^+)^3}{3!}\right] \tag{4.9}$$

where $y^+ = y_p u_\tau/\nu$ is the normalized wall coordinate, $u^+ = u_p/u_\tau$ represents near-wall velocity in wall units, $\kappa = 0.41$ is the von Karman constant and E is a wall function constant approximately equal to 9.8 for smooth walls. Here, y_p is the mid-height of the wall-adjacent cell, and u_p is the component of the cell center velocity parallel to the boundary. The main advantage of Eq.(4.9) is that the wall relationship can be computed from a single expression without switching between different formulations for each region of the flow (Tominaga et al., 2008b; de Villiers, 2006). As a result, the center of the wall adjacent cell does not need to be in the logarithmic region.

Now, we can expressed the wall shear stress τ_w using effective kinematic viscosity $\nu_{\text{eff}} = \nu_t + \nu$ as (Vuorinen et al., 2015),

$$\tau_w = \rho\left(\nu_t + \nu\right)\left.\frac{\partial u}{\partial y}\right|_{y=y_p} \approx \rho(\nu_t + \nu)\frac{u_p}{y_p}, \tag{4.10}$$

where ν represents the kinematic viscosity of the fluid and ν_t is the turbulent eddy viscosity. Rearranging Eq.(4.10) and substituting $\tau_w = \rho u_\tau^2$, we get:

$$\nu_t = \max\left(0, \frac{u_\tau^2}{(u_p/y_p)} - \nu\right). \tag{4.11}$$

Finally, the wall law is indirectly specified using the relationship in Eq.(4.11) and Eq.(4.9). Since Eq.(4.9) is non-linear, u_τ is computed iteratively using the Newton-Raphson method. Although Spalding's law is theoretically valid for the mean flow, it has also been demonstrated to be useful for LES with instantaneous filtered velocity (Cheng and Porté-Agel, 2013; Wang and Chen, 2020). For the current study, implementation of this boundary condition provided in OpenFOAM-v8 as `nutUSpaldingWallFunction` is adopted.

4.3.5 Numerical setup

The simulations were conducted using a transient solver implemented in OpenFOAM-8 called `pimpleFoam`, which is partly based on the Pressure-Implicit with Splitting of Operators (PISO) algorithm developed by Issa (1986). The `pimpleFoam` solver essentially works by running multiple PISO loops per each time step. The choice of `pimpleFoam` solver for the current study is mainly due to its stability at high Courant-Friedrichs-Lewy (CFL) numbers. The solver is configured with one velocity predictor computation followed by two pressure corrector loops. In addition, to account for skewed mesh, one non-orthogonal pressure corrector loop is added. Since `pimpleFoam` solver allows a dynamic time advancement, for all the wind load simulations, the time step is dynamically adjusted based on a maximum CFL number of 10, which was noticed to significantly reduces the total computational time. The maximum CFL number of 10 is decided based initial test run simulation. For spatial discretization, a second-order central differencing scheme with linear interpolation is used. The time discretization is done using a second-order backward scheme. All the simulations were run over 38 s, and the first 2 s duration was truncated to avoid the initial transient stage of the simulation. Hence, effectively only 36 s duration is used for the analysis, which translates to a full-scale duration of nearly 1 hour. While the simulations are running, the pressure data on the surface of the building is monitored using probes placed at the exact location as the experimental measurements (see Figure 4.1a). The pressure time series is recorded at a sampling rate of 400 Hz (identical to the wind tunnel study). All the LES cases were simulated in parallel using 128 AMD Rome 7502 @ 2.50 GHz CPU cores. On average, each wind load simulation case took approximately 7.5 days to run.

4.4 Structural model

The responses of the structure can be evaluated using either the time or frequency domain approach. In time-domain analysis, the structure's response is computed directly by executing step-by-step time integration of the governing equation of motion. On the other hand, when a frequency-domain approach is used, the root-mean-square (RMS) of the response is estimated using a generalized force spectrum and the theory of random vibration without solving the governing equation of motion (Davenport, 1965, 1967). The main advantage of the frequency domain approach over the time domain method lies in its computational efficiency, which often requires simplifying approximations (Aas-Jakobsen and Strømmen, 2001; Yeo and Simiu, 2011). Nevertheless, the time domain approach provides a more accurate and direct estimate of the responses, alleviating most of the limitations of the conventional frequency domain tech-

nique (Simiu et al., 2008). Moreover, with the currently available computing resource, the cost of executing time domain analysis is relatively small, especially if we compare it to the cost of running wind load simulation using LES. Thus, for the current study, the responses of the building to dynamic wind excitation are evaluated by employing a time-domain approach. Section 4.4.1 presents the time domain analysis procedure used to represent the structure as a multi-degree-of-freedom (MDOF) dynamic system. In Section 4.4.2, the dynamic properties of the building used for the analysis are briefly described.

4.4.1 Equations of motion

The structure is represented by a lumped-mass system, assuming that the floors behave like a rigid diaphragm. At the center of each floor, we have two translational(x and y) and one rotational(ϑ) degree of freedom. Hence, for a multi-story building with n floors, we have a total of $3n$ degrees of freedom. The dynamic equilibrium equation governing the motion of a building can be written as follows:

$$\mathbf{M}\ddot{\mathbf{d}}(t) + \mathbf{C}\dot{\mathbf{d}}(t) + \mathbf{K}\mathbf{d}(t) = \mathbf{F}(t) \tag{4.12}$$

where $\mathbf{M}$, $\mathbf{C}$, and $\mathbf{K}$ are $3n \times 3n$ matrices representing the mass, damping, and stiffness of the structure, respectively. The vector $\mathbf{d}(t)$ contains the Lagrangian displacements of the floors in x, y, and ϑ directions. Whereas $\mathbf{F}(t)$ holds the wind load vectors acting at the center of each storey. The vectors $\mathbf{d}(t)$ and $\mathbf{F}(t)$ can be expressed in expanded form as

$$\mathbf{d}(t) = \begin{bmatrix} \mathbf{d}_x(t) \\ \mathbf{d}_y(t) \\ \mathbf{d}_\vartheta(t) \end{bmatrix} = \begin{bmatrix} d_{x1}(t) \\ \vdots \\ d_{\vartheta n}(t) \end{bmatrix} \quad \text{and} \quad \mathbf{F}(t) = \begin{bmatrix} \mathbf{F}_x(t) \\ \mathbf{F}_y(t) \\ \mathbf{F}_\vartheta(t) \end{bmatrix} = \begin{bmatrix} F_{x1}(t) \\ \vdots \\ F_{\vartheta n}(t) \end{bmatrix}, \tag{4.13}$$

where $d_{xi}(t)$, $d_{yi}(t)$ and $d_{\vartheta i}(t)$ are the x-direction, y-direction and torsional displacements at $i-$th floor for a time t. Similarly, $F_{xi}(t)$, $F_{yi}(t)$ and $F_{\vartheta i}(t)$ represent $i-$th storey loads in x, y and ϑ directions, respectively.

Using modal analysis with modal basis functions $\mathbf{\Phi} = [\mathbf{\Phi}_x, \mathbf{\Phi}_y, \mathbf{\Phi}_\vartheta]$, where $[\mathbf{\Phi}_s]_{ji} = \phi_{sji}(s = x, y, \vartheta)$ denotes the $j-$th mode shape at $i-$th floor. Utilizing the first N modes, the displacement vector $\mathbf{d}(t)$ can be expanded by superposing modal contributions as

$$d_{si}(t) = \sum_{j=1}^{N} q_j(t)\phi_{sji} \quad \text{with} \quad s = x, y, \vartheta, \tag{4.14}$$

where $q_j(t)(j = 1, 2, \ldots N)$ is the generalized response of $j-$th mode (Chopra, 2007). Assuming

the structure is classically damped, the governing equations in Eq.(4.12) can be simplified to N decoupled modal equations. Thus, the equation of motion in the modal coordinate is given by:

$$m_j^* \ddot{q}_j(t) + c_j^* \dot{q}_j(t) + k_j^* q_j(t) = F_j^*(t) \tag{4.15}$$

The parameters m_j^*, c_j^*, k_j^*, and F_j^* represent generalized mass, damping, stiffness, and force associated with j-th mode, respectively. These generalized parameters can be computed following the form:

$$\begin{aligned} m_j^* &= \sum_{i=1}^{n} \left(m_{xi} \phi_{xji}^2 + m_{yi} \phi_{yji}^2 + m_{\vartheta i} \phi_{\vartheta ji}^2 \right) \\ c_j^* &= 2 \omega_j m_j^* \xi_j \\ k_j^* &= \omega_j^2 m_j^* \\ F_j^*(t) &= \sum_{i=1}^{n} \left(F_{xi} \phi_{xji} + F_{yi} \phi_{yji} + F_{\vartheta i} \phi_{\vartheta ji} \right) \end{aligned} \tag{4.16}$$

where ω_j is $j-$th mode natural frequency of the building and ξ_j is the modal damping ratio. The natural frequencies ω_j and the mode shapes ϕ_j are obtained from eigenvalue analysis of a freely vibrating structure as $[\mathbf{K} - \omega^2 \mathbf{M}]\mathbf{\Phi} = \mathbf{0}$. These structural properties are often conveniently extracted from the building's Finite Element Model (FEM) in software programs like ETABS and SAP2000. Since the pressure measurements are not available at each storey level in the case of the experimental data, the generalized forces $F_j^*(t)$ in Eq. (4.16) are evaluated using the mode shape vector and pressure integration scheme described in Section 4.4.3.

After computing the generalized properties using Eq.(4.16), each modal equation in Eq.(4.15) is numerically solved as an ordinary differential equation. In the current study, a 4^{th} order Runge–Kutta method is adopted to perform time integration of Eq.(4.15). Finally, once the generalized displacements $q_j(t)$ are solved, the structural response of the building per each floor is determined from the mode shape vector and generalized responses using Eq.(4.14).

4.4.2 Structural properties of the CAARC building

For the building structure, we adopted the structural properties of the 60-story reinforced concrete structure used in the work of Park et al. (2018). The building was designed based on ASCE 7-16 (2017) specifications as a moment-resisting frame system (see Figure 4.5). The dimensions of the building structure are identical to the geometry of the CAARC model used in the wind load simulations. The structure consists of 7 bays with span lengths of 6.53m along the width and 5 bays with a 6.10 m span along the depth of the building, as shown in

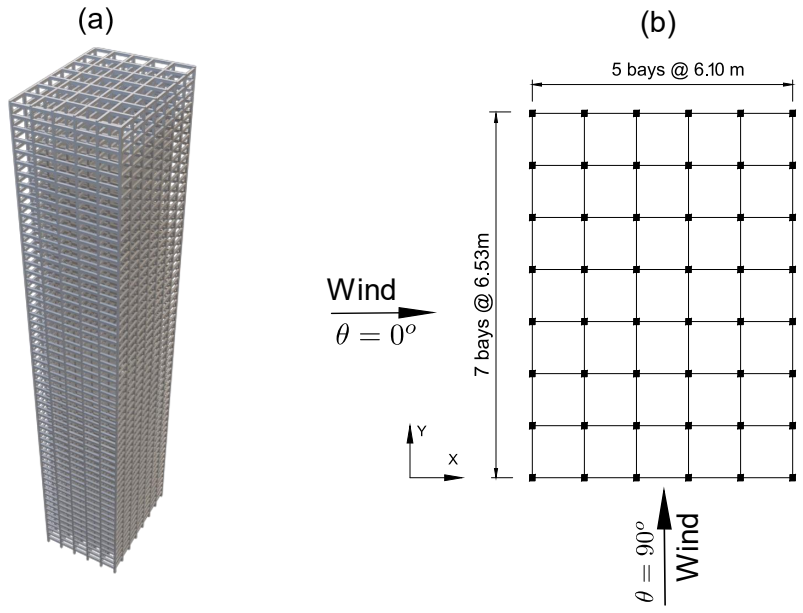


Figure 4.5: Structural model of the 60-story reinforced concrete building: (a) 3D view; (b) structural layout plan

Table 4.3: Summary of the dynamic properties

Modes	1st	2nd	3rd	4th	5th	6th
Natural frequency (Hz)	0.156	0.167	0.192	0.450	0.459	0.512
Damping ratio (%)	2.0	2.0	2.0	2.0	2.0	2.0

Figure 4.5b. All the floors have inter-story heights of 3.05 m and are assumed to be rigid diaphragms. A detailed description of the sectional properties of the structural members, including reinforcement details, can be found in Park et al. (2018). The structural properties relevant to the wind-induced response calculation are discussed below.

The first six mode shapes reported from the modal analysis of the FEM are shown in Figure 4.6. As seen in Figure 4.6, both flexural and torsional modes of vibration are decoupled. Table 4.3 gives a summary of the dynamic properties used for the study. For the structural damping ratios, 2.0% of critical is used for all six modes of vibrations.

4.4.3 Wind load transfer scheme

The total dynamic loads are evaluated by integrating point pressure measurements over the surface of the building. The same approach is used for both the LES and experimental aerodynamic data. Although high-resolution pressure measurement is available for the LES cases, the numerical pressure measurements at the exact tap location were used instead for ease of com-

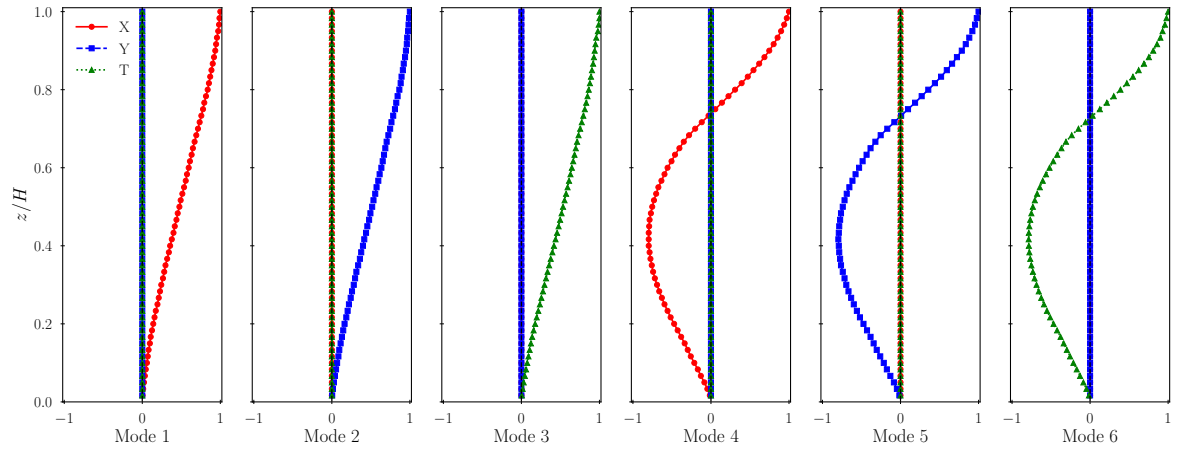


Figure 4.6: The first six vibration mode shapes for lateral (X),transversal (Y) and torsional (T) directions

parison with the experimental data. This entails the error introduced due to pressure integration to be the same for both LES and experimental data.

For each wind direction, the instantaneous integrated wind forces acting in x, y, and ϑ degrees of freedom are evaluated by superposing tributary force from each pressure tap as

$$\begin{aligned} f_{xi}(t) &= p_i(t)a_{xi} \\ f_{yi}(t) &= p_i(t)a_{yi} \\ f_{\vartheta i}(t) &= F_{xi}(t)r_{xi} + F_{yi}(t)r_{yi} \end{aligned} \tag{4.17}$$

where $f_{xi}(t)$, $f_{yi}(t)$ and $f_{\vartheta i}(t)$ represent the tributary force in x, y, and ϑ directions. p_i is the point pressure, while a_{xi} and a_{yi} represent the tributary area of the tap projected in x and y directions. r_{xi} and r_{yi} denote moment arms of the tributary forces $f_{xi}(t)$ and $f_{yi}(t)$ about the center of the building. Finally the instantaneous generalized forces $F^*_j(t)$ for j-th mode is determined from:

$$F^*_j(t) = \sum_{i=1}^{N_{\text{tap}}} \left[f_{xi}\phi_{xj}(z_i) + f_{yi}\phi_{yj}(z_i) + f_{\vartheta i}\phi_{\vartheta j}(z_i) \right], \tag{4.18}$$

in which $\phi_{xj}(z_i)$, $\phi_{yj}(z_i)$ and $\phi_{\vartheta j}(z_i)$ are the mode shapes in x, y, and ϑ directions interpolated at tap height z_i. It is worth mentioning that the pressure integration procedure assumes that the aeroelastic effects (e.g., the effect of aerodynamic damping) are negligible, which is the case for most buildings that are not very flexible under practical wind speeds. For the structural properties and test speed used for the current study, no significant aeroelastic feedback is expected. However, for dynamically sensitive structures that experience pronounced aeroelastic

effects, a fully coupled fluid-structure interaction might be required. For these types of structures, Chapter 6 presents a fully coupled high-fidelity fluid-structure interaction framework.

4.5 Results and discussion

To assess the accuracy of the LES model, the numerical simulations were validated against the BLWT experimental tests both at incident flow and aerodynamic forces level, step by step. A necessary procedure at this early stage of LES for wind load evaluation. First, the incident flow characteristics from the LES are compared to the wind profiles measured in the wind tunnel as presented in Section 4.5.1. Second, the building surface pressure coefficients, including mean, RMS, and peak values, are compared with those obtained from the BLWT experiment, Section 4.5.3. Then, in Section 4.5.4, the aerodynamic base loads calculated from the LES, such as base shear forces and overturning moments, are extensively compared with those calculated from the experiment. Finally, Section 4.5.5 presents the accuracy of LES based estimation of the overall wind-induced response of the structure.

4.5.1 Incident flow characteristics

The aerodynamic forces on the building are sensitive to the upcoming turbulence, hence the statistics of the incident flow are first investigated using empty domain simulation. The empty domain simulation allows us to directly measure the characteristics of the wind the building will experiences as opposed to the inflow used at the inlet boundary. Since the empty domain simulation is conducted using the same numerical setup as computational domain used for the wind load simulations, the undisturbed roof-height reference wind speed can be conveniently measured without the effect of the building aerodynamics. This reference velocity is later used to normalize aerodynamic data such as pressure and force coefficients. It is worth emphasizing that, analogous to the wind tunnel procedure, where we first calibrate the wind profile in the empty tunnel configuration, it is equally vital for LES to run an empty domain simulation before running the wind load simulations (Melaku and Bitsuamlak, 2021; Ricci et al., 2018).

Figure 4.7 shows the comparison of the incident wind profile from LES against the target BLWT experimental measurement. The LES profile is measured over a vertical line located $0.25H$ upstream of the intended location of the study building model. In this figure, the curves shown as DFSR correspond to the wind profile inlet. The absolute relative error of the LES (downstream) profiles against the experiment averaged over $3H$ height above the ground is used to measure the accuracy. It was observed that the characteristics of inlet turbulence undergo some change downstream as the flow adjusts to an actual turbulent boundary layer structure.

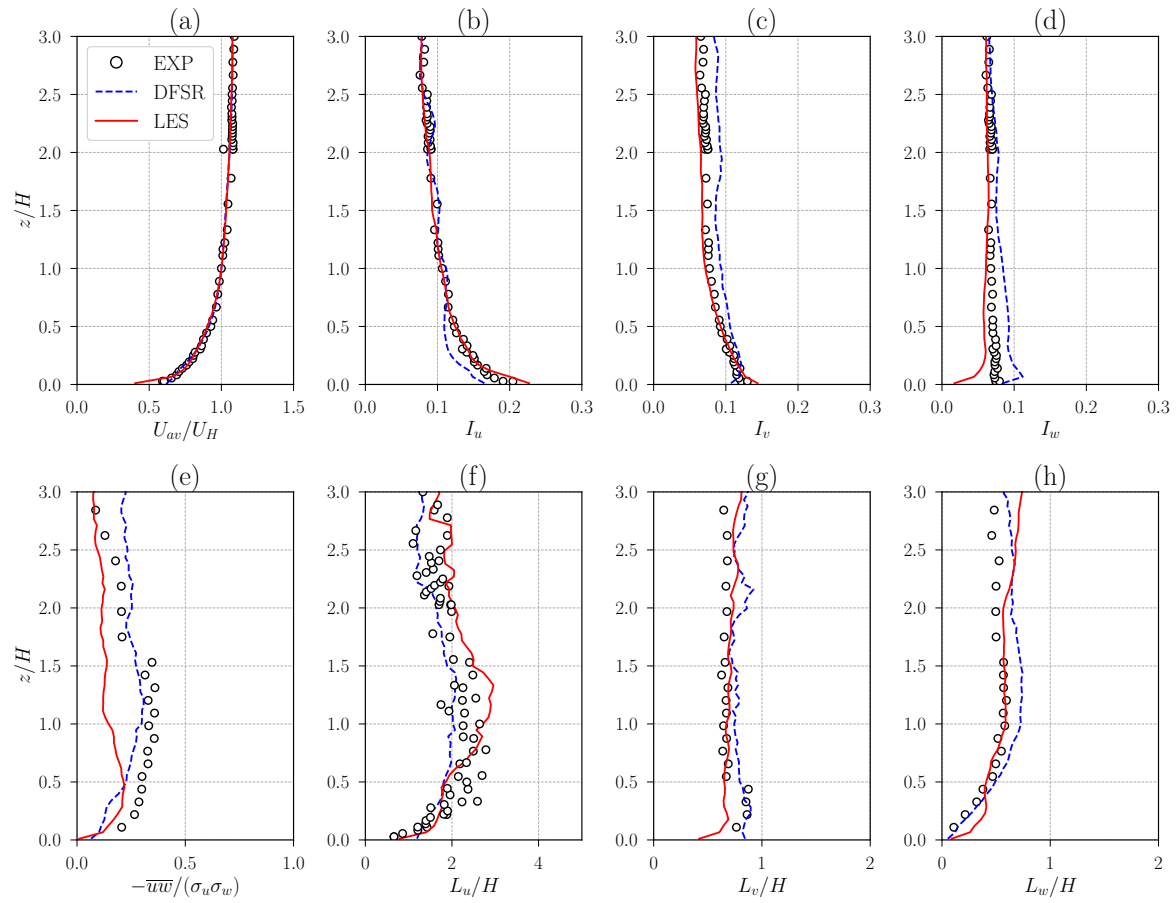


Figure 4.7: Comparison of the incident wind profile with the target BLWT experimental measurements: (a) mean velocity; (b-d) turbulence intensity profiles for u, v, and w components, respectively; (d) normalized Reynolds shear stress profile $\overline{uw}$; (e-h) integral length scale profiles ${}^xL_u$, ${}^xL_v$ and ${}^xL_w$.

However, the downstream evolution in the mean velocity profile is generally very small compared to the turbulence intensity profiles (see Figure 4.7a). The mean velocity profile is in excellent agreement with the experiment having only 1.09% error over $3H$ height. This success is mainly attributed to the rough wall boundary condition applied on the ground surface (the Schumann–Grötzbach model), which maintained the mean velocity profile well up to the incident location.

The shape and magnitude of turbulence intensity profiles have a close agreement with some appreciable difference in the v and w components. The deviations of the incident turbulence intensity profiles from the experiment are 2.50%, 8.19%, and 11.17% for I_u, I_v, and I_w, respectively. The I_u component, which is the most important for the longitudinal components of the load, has the least error among the three. Turbulence intensity profiles, especially the v and w components, show appreciable decay, as can be noted from the plots in Figure 4.7c-d. This

is mainly due to the nature of the synthetic turbulence used; since the phase information of the flow is generated randomly, some form of turbulence decay is unavoidable (Melaku and Bitsuamlak, 2021). Similarly, the normalized Reynolds shear stress profile, $\frac{\overline{uw}}{\sigma_u \sigma_w}$ is shown in Figure 4.7e. From the figure, it can be seen that the DFSR method was able to generate a shear stress profile comparable to the experimental measurement. Considering that the turbulence intensity (in the mean wind direction) and mean velocity profiles predominately affect the wind loads for tall buildings, no further optimization was applied for the Reynolds shear stress profile. For low-rise structures, the incident flow corrections such as those suggested by Lamberti et al. (2018) can be used.

Another important quantity to be compared is the turbulence integral length scale, which measures the average size of the turbulent eddies (Simiu and Scanlan, 1996). The integral length scale profile is computed from the velocity time-series applying Taylor's frozen turbulence hypothesis (Taylor, 1938) as,

$$^{x}L_{u_i} = U_{av} \int_0^{\infty} \rho_{u_i}(\tau) d\tau, \tag{4.19}$$

where $u_i (i = 1, 2, 3)$ is the fluctuating velocity component; U_{av} is the mean velocity in the stream-wise direction and ρ_{u_i} represents the auto-correlation function for u_i components of the velocity. The three turbulence integral length scale profiles($^{x}L_u$, $^{x}L_v$, $^{x}L_w$) computed by Eq. (4.19) are shown in Figure 4.7f-h. Overall, the integral length scale profiles monitored at the incident location are in reasonable agreement with the target experimental data showing an average deviation of 10.33%, 11.22%, 25.92% for $^{x}L_u$, $^{x}L_v$, and $^{x}L_w$, respectively.

Figure 4.8 shows the reduced velocity spectra at the roof height for the three velocity components. The figure depicts the comparison of the spectra at the inlet (DFSR), the incident flow (LES), and the one from the wind tunnel (EXP). The figure also shows the von Karman model used as target spectra for generating the inflow turbulence. Examining Figure 4.8a, the spectrum of the longitudinal velocity is in good agreement over most of the frequency range with the BLWTL measurement. For the lateral and vertical velocity components, the LES spectra closely follow that of the von Karman. At the high-frequency end, the spectral plots from LES show a sharp drop-off due to the grid resolution limit below which LES filters the turbulence fluctuations. By utilizing refined grids, an attempt was done to limit the effect on the wind loads. Since the small-scale fluctuations are localized, they can be captured by additional mesh refinement near the building surface. The benefit of LES lies in resolving the large-scale fluctuations that produce a significant portion of the wind loads (Shah and Ferziger, 1997).

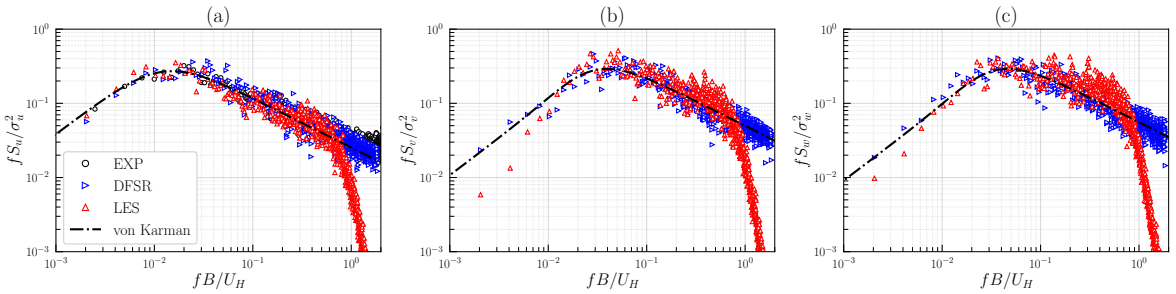


Figure 4.8: Reduced velocity spectra at the building height:(a) u-component; (b)v-component; (c) w-component;

4.5.2 Flow structure around the building

The wind field around the building is marked by various highly three-dimensional features such as flow separation, reattachment, vortex shedding, down-wash on the leeward side, and wake re-circulation region. Coupled with the turbulent characteristics of the incoming flow, these body-generated flow features are responsible for the significant variations of the wind loads on the building with changes in the wind direction. Figure 4.9 shows the flow structures around the building when the wind is perpendicular to the wider face of the building ($\theta = 0°$). The turbulent flow structures shown in the figure are computed from iso-surfaces of the second invariant of the velocity gradient tensor. The flow features are generally topologically similar to the widely studied wall-mounted finite-length square cylinder (Wang and Zhou, 2009). The figure depicts the shedding of span-wise vortices rolling from separated shear layers at the building corners. These vortices are coherent along the building height and are responsible for the lateral loads that the building experiences in the cross-wind direction (i.e. lift). At the front of the building, the horseshoe vortex can also be observed encircling the bottom of the building.

To examine the details of the flow structure, in Figure 4.10, the mean streamlines around the building are shown on cross-sectional plans. The figure illustrates the mean flow field for $\theta = 0°, 45°$ and $90°$ wind directions across $xy-$ and $xz-$ planes passing through the center of the building. The $xy-$plane sections are taken at $2/3H$ height above the ground, where the flow stagnation point is expected to occur. When the wind is normal to a face (i.e., for $0°$ and $90°$ wind directions), as expected, the flow separation is initiated at the upwind edges of the building. In both cases, the flow remains separated, and no reattachment is observed downstream. Considering that the depth-to-width ratio for the CAARC building is 1.5, a notable difference in the flow structure of $0°$ and $90°$ cases is the size of the separation bubble and the wake region. As expected, for $0°$, the flow separation region and the wake are wider than the $90°$. For the $45°$ case, unlike $0°$ and $90°$ wind angles, the flow remains attached over the faces exposed to the

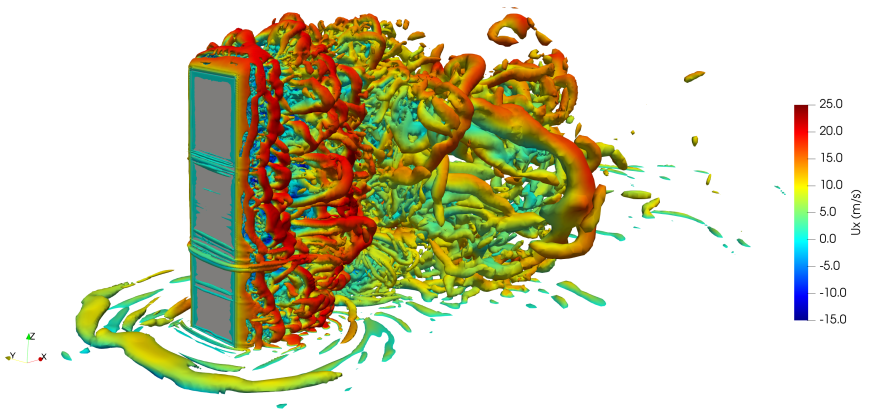


Figure 4.9: Turbulent flow structure around the building for wind direction $\theta = 0°$ determined based on the Q-criterion and colored by the longitudinal component of the velocity.

wind. As a result, the flow separation occurs at the end of the faces inclined to the incoming flow. Compared to both 0° and 90° cases, it experiences a relatively larger separation region.

4.5.3 Pressure coefficients

Validation of pressure distribution on the building surface is the first important step in obtaining accurate overall structural loads and responses. Furthermore, assessing the performance of LES in estimating surface pressure fluctuations is crucial for cladding load predictions. The pressure coefficient on the building surface is defined as

$$C_p(t) = \frac{p(t) - p_0}{\frac{1}{2}\rho U_H^2}, \tag{4.20}$$

where $p(t)$ is the measured pressure, p_0 represents a static reference pressure, U_H is the roof-height mean velocity measured in the empty domain setup, and ρ represents the air density. For the LES simulation, the static reference pressure p_0 is set to zero, considering that we used a zero-pressure outlet boundary condition at the outflow.

In Figure 4.11, the time series of the pressure fluctuation recorder on the building's windward, side, and leeward faces are compared for $\theta = 0°$. The C_p time-series are taken from taps located at the centers of each face $\frac{2}{3}H$ high above the ground. Initial qualitative inspection of the time series indicates that the result from LES has a similar aerodynamic signature to that of the experimental data. The spectra of the C_p fluctuations at the same tap locations are shown in Figure 4.12. Overall, the spectral energy contents of the C_p fluctuations from LES are in excellent agreement with those of the wind tunnel data. On the windward face of the building, the pressure spectra resemble that of the incident longitudinal velocity. On the side

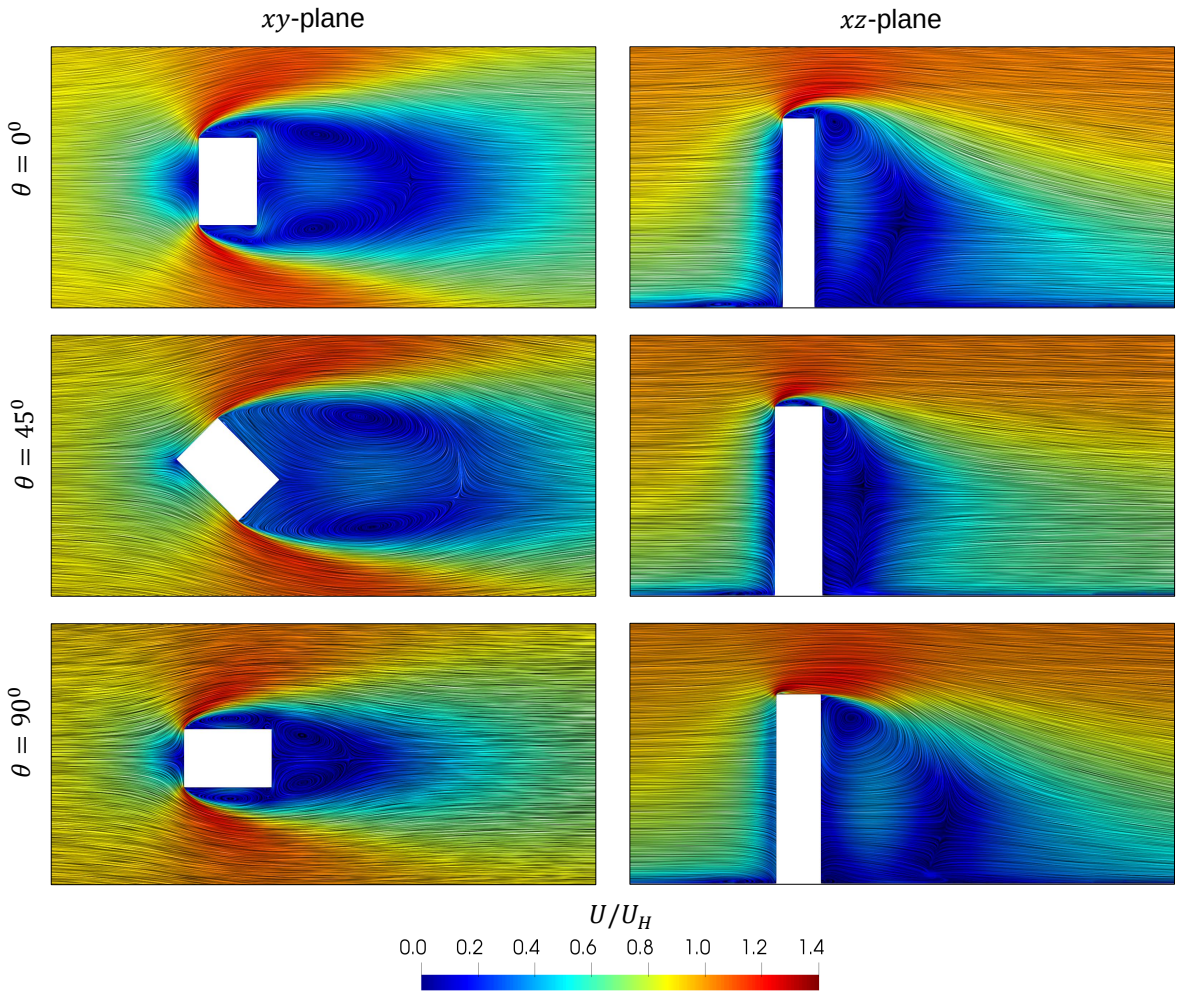


Figure 4.10: Streamlines around the building colored by the magnitude of mean velocity normalized by roof-height wind speed: (rows) wind directions $(0°, 45°, 90°)$; (columns) cross-sectional views on xy and xz planes.

face, the effect of vortex shedding is depicted with a peak in energy spectra close to the shedding frequency (see Figure 4.12b). Also, the Strouhal number from the sidewall tap is in good agreement with those reported in the literature. For the leeward face, the spectrum is characterized by fluctuations with a wide frequency range due to the highly turbulent nature of the wake flow. It is worth noting that, with additional local mesh refinement around the building, it was possible to capture the building-generated small-scale fluctuations well up to the sampling rate of the experimental data. As such, it was possible to cut significant computational costs while resolving relevant small-scale flow features near the building with localized mesh refinements.

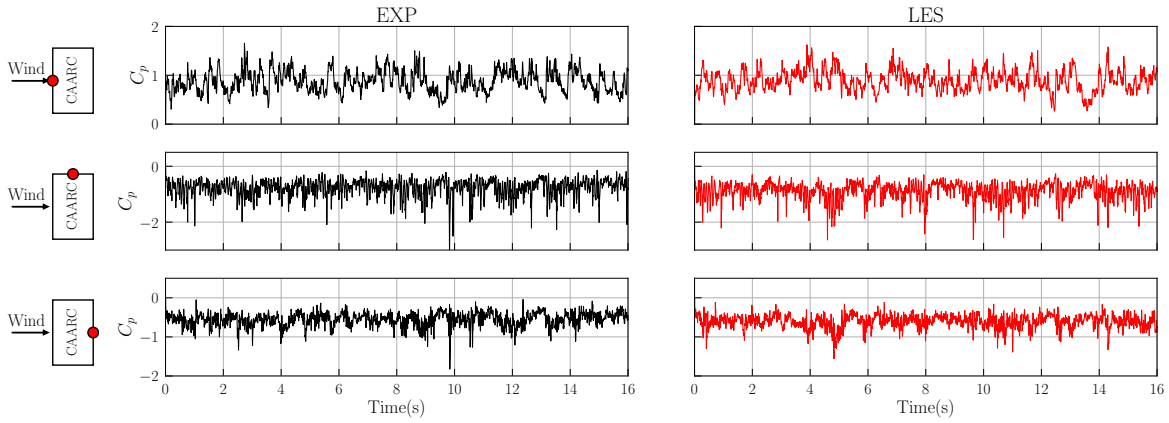


Figure 4.11: Time histories of pressure coefficient at the center of windward, side and leeward faces of the building measured at $\frac{2}{3}H$ for 0° wind direction: (left) experiment; (right) LES

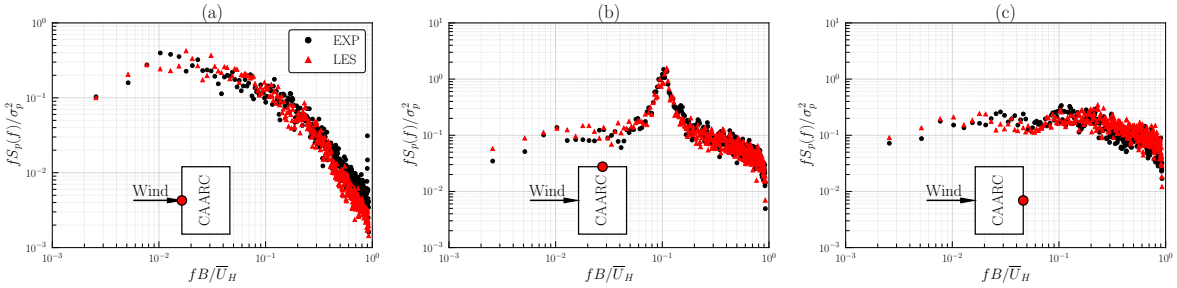


Figure 4.12: Reduced power spectral density of pressure at $\frac{2}{3}H$ for 0° wind direction: (a) windward face; (b) side face; (c) leeward face

4.5.3.1 Comparison of mean, RMS and Peak

Here, we present comparisons of the mean, RMS, and peak pressure coefficient with the experimental data. Figure 4.13 compares the mean C_p for 0° and 90° wind directions. As seen from Figure 4.13, on the windward face, the mean C_p is predicted well for both 0° and 90° cases.

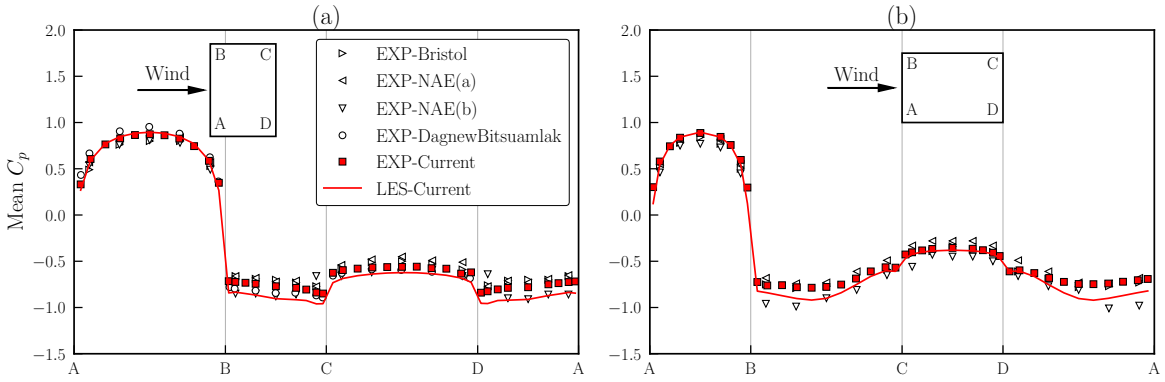


Figure 4.13: Mean pressure coefficients measured at $\frac{2}{3}H$ height of the building: (a) $\theta = 0°$; (a) $\theta = 90°$

However, on the side and leeward faces, the results from LES show relatively more negative C_p (higher suction). This may be due to differences related to approaching flow characteristics which can affect the behavior of the separated shear layers and wake regions. Figure 4.14 presents the comparison of the root-mean-square (RMS) C_p values obtained from the LES with the experiment for 0° and 90° wind directions, respectively. Again, as shown in Figure 4.14, the RMS C_p from the LES generally compares well with the experimental data.

For comparison purposes, the results from the current study are shown together with previous C_p measurements on the CAARC building reported in the literature (see Figures 4.13 and 4.14). Aerodynamic data measured at the University of Bristol and National Aeronautical Establishment(NAE(a) and NAE(b)) reported in Melbourne (1980) and more recent measurements conducted by Dagnew and Bitsuamlak (2014) are used for the comparison. Although the testing facilities used for these measurements have some differences, the reported roof-height longitudinal turbulence intensities vary between 8% to 12%, which are comparable to the ones used in the current study (10.6%). Overall, the mean C_p values from all studies are in reasonable agreement with our experimental data. However, the RMS C_p from the current study are relatively higher than the values in the literature. These variations are associated with the differences in the turbulence profiles used by each testing facility. Deviations of turbulence characteristics in the lateral and vertical directions could also exist that are not reported in those wind-tunnel studies.

Comparing mean and RMS pressure coefficients is not enough to describe all the relevant wind load characteristics on the building surface. Estimating peak C_p values is also important, particularly when evaluating wind loads for the cladding design of the building. Figure 4.15 shows these peak pressure coefficients at the $\frac{2}{3}H$ height of the building for 0° and 90° wind directions. The peak pressure coefficients shown in Figure 4.15 are determined by fitting se-

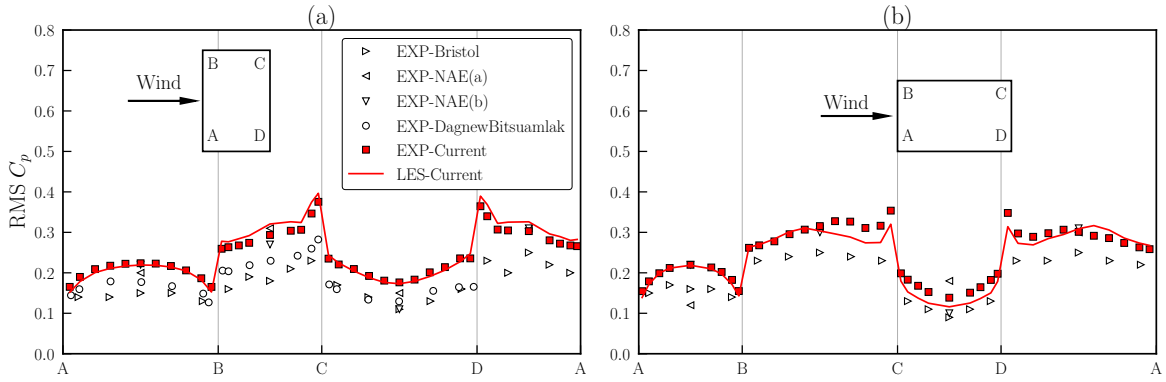


Figure 4.14: Standard deviation of pressure coefficients measured at $\frac{2}{3}H$ height of the building: (a) $\theta = 0°$; (a) $\theta = 90°$

lect maxima/minima values to Type I (Gumbel) distribution using the Best Linear Unbiased Estimator (BLUE) Lieblein (1976) method. The maxima/minima are collected from individual minima/maxima of each segment after dividing the time history into ten epochs. The epochal peak is converted into the full duration value using Cook and Mayne's method (Cook and Mayne, 1979). It is worth noting that using the Gumbel fitted with the Lieblein-BLUE method instead of absolute worst values results in statistically reliable peak estimation. In Figure 4.15, the peak pressure ($+ve$) and suction ($-ve$) coefficients are separately shown. As seen from the figure, the peaks predicted from the LES show satisfactory agreement with the wind tunnel measurement. Especially on the windward face of the building, the positive peak pressures are predicted well compared to the negative peak pressure on the side and leeward faces of the building. This is expected because the peaks on the side and leeward faces of the building are mainly dominated by flow separation, re-circulation, and turbulent wake regions that often require a highly dense computational grid to resolve accurately with LES.

Thus far, we have presented a systematic comparison of the pressure coefficients only along the perimeter of the building near the stagnation point for 0° and 90° cases. The contour plots in Figure 4.16 present the distribution of the mean, RMS, and peak C_p distributions over all the faces of the building for 0° wind direction. Comparing these contour plots, it is evident that the distribution from LES depicts similar patterns as the BLWT measurements. Further statistical comparison of the numerical and experimental measurements using all the pressure taps and wind directions is presented in Figures 4.17, 4.18, and 4.19. These figures provide a more comprehensive validation to gauge the accuracy of the numerical modeling process. In Figure 4.17, scatter plots of experimental and LES data are shown for mean pressure coefficients. Similarly, Figure 4.18 and Figure 4.19 depict the comparison for RMS and peak C_p values, respectively. The scatter plots in Figure 4.19 show both positive and negative peaks.

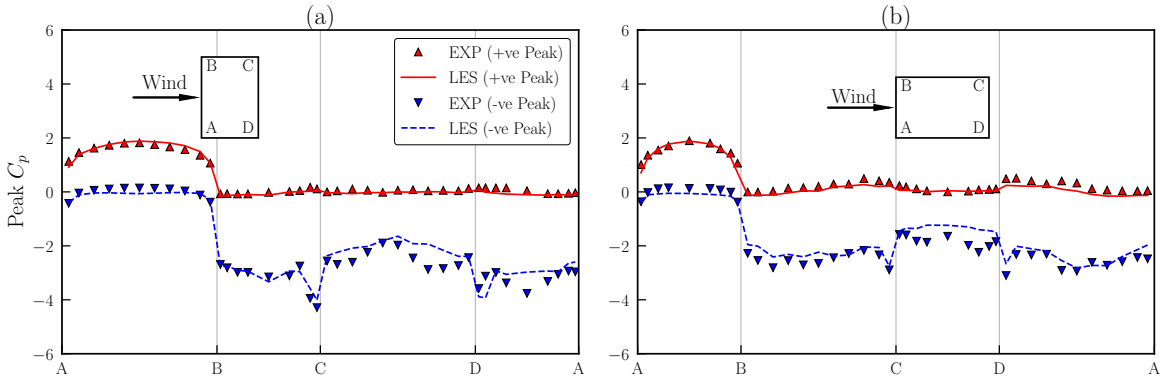


Figure 4.15: Positive and negative peak pressure coefficient measured at $\frac{2}{3}H$ height of the building: (a) $\theta = 0°$; (a) $\theta = 90°$

In all the figures, the line of a perfect match between LES and the experiment is shown with a dotted trend line. Considering all the wind directions simulated, the agreement between the experimental and LES data presented in Figures 4.17, 4.18 and 4.19 is generally satisfactory. For a given wind direction, the accuracy of the LES relative to the experiment is estimated using Normalized Mean Absolute Error (NMAE), defined as

$$\text{NMAE} = \frac{1}{N_{\text{tap}}} \sum_{i=1}^{N_{\text{tap}}} \frac{\left|Q_{\text{EXP}}^{(i)} - Q_{\text{LES}}^{(i)}\right|}{(Q_{\text{EXP}}^{\max} - Q_{\text{EXP}}^{\min})} \times 100, \tag{4.21}$$

where Q_{EXP} and Q_{LES} represent statistics of pressure coefficients from experimental and LES data. Such type of error expression offers two advantages. First, using the absolute value for the error permits both positive and negative differences to accumulate instead of offsetting each other (Oberkampf and Trucano, 2002). Second, it avoids the infinite error that arises if the normalization was done by actual experimental Q_{EXP} quantities that have close to zero values (e.g., C_p near the upwind edges of the face normal to the wind). Thus, the normalization based on the min-max values in Eq. (4.21) gives more weight to critical locations such as high pressure or suction regions and penalizes low absolute C_p regions that are often less important from wind loading perspective. Furthermore, Eq. (4.21) measures the accuracy of LES relative to the range of variation in the experimental data.

Figure 4.20 shows the error (NMAE) related to each wind direction determined using the expression in Eq. (4.21) for the mean, RMS and peak C_p distributions. The mean C_p predictions from LES show a 4.6% error when averaged over all the wind directions. Whereas for the RMS C_p, the average error is roughly close to 4%. For the peak values, the smallest error occurs for the positive (pressure) C_p values, which is approximately below 4.7%. In contrast, for the

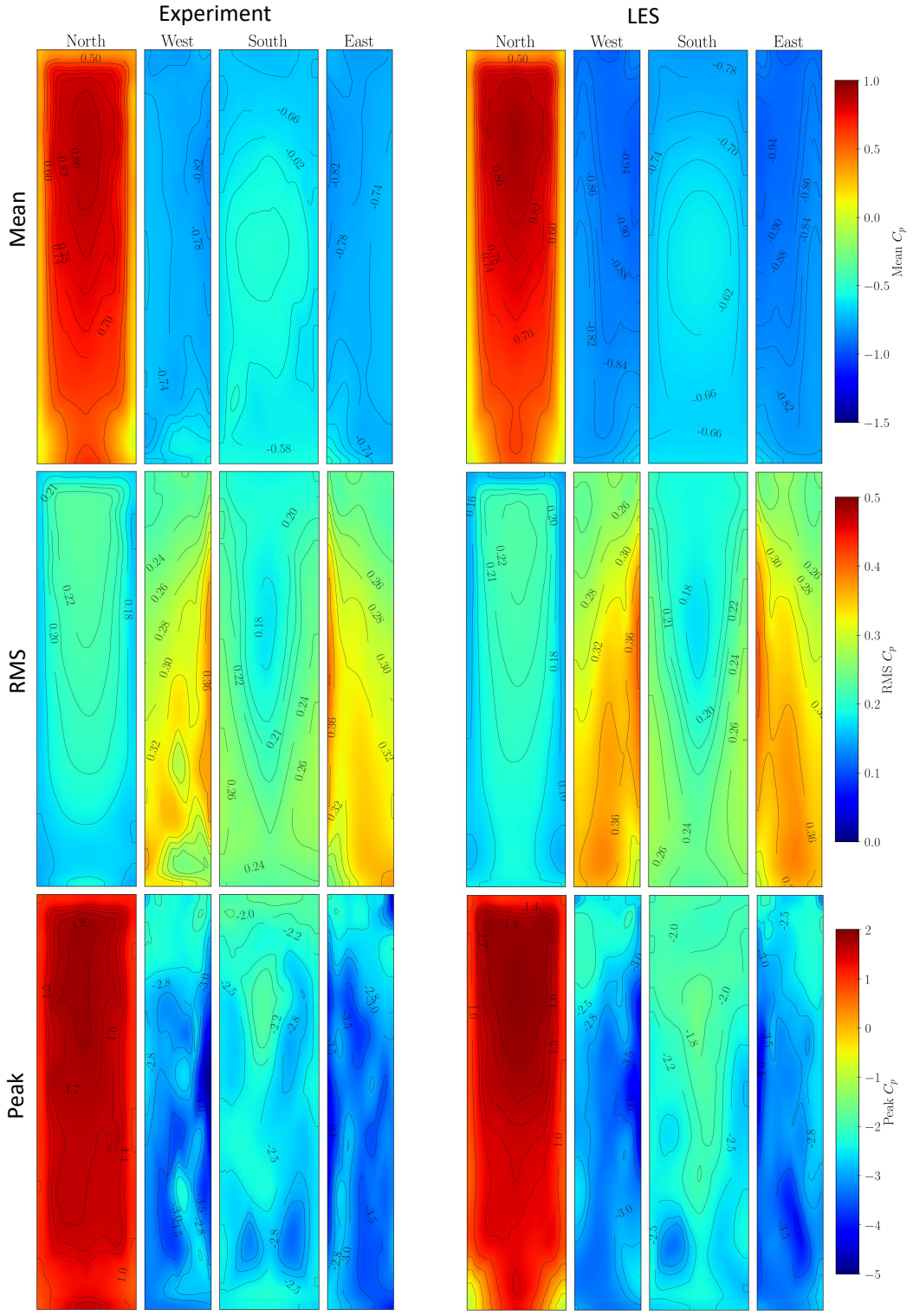


Figure 4.16: Contour plots of C_p statistics for $\theta = 0°$: (columns) Experiment and LES; (rows) Mean, RMS and Peak

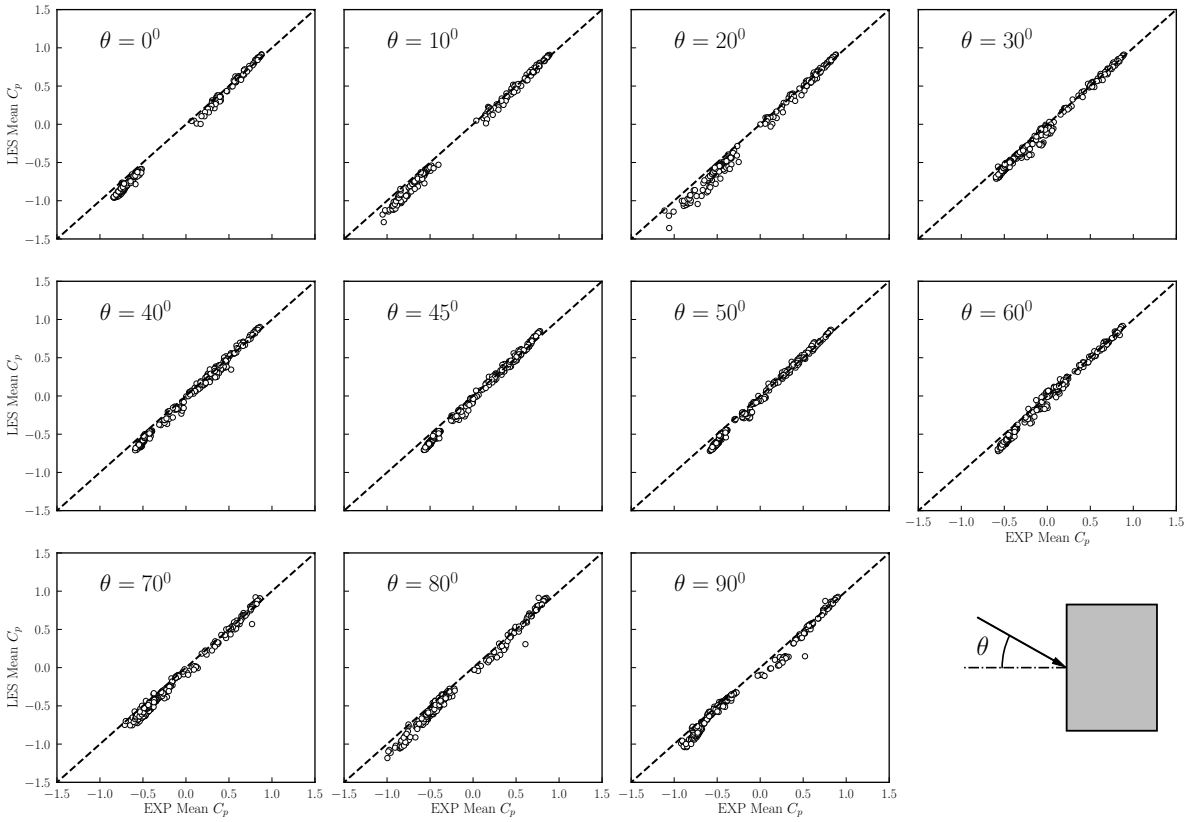


Figure 4.17: Scatter plots comparing mean C_p from LES with experiment for all wind directions

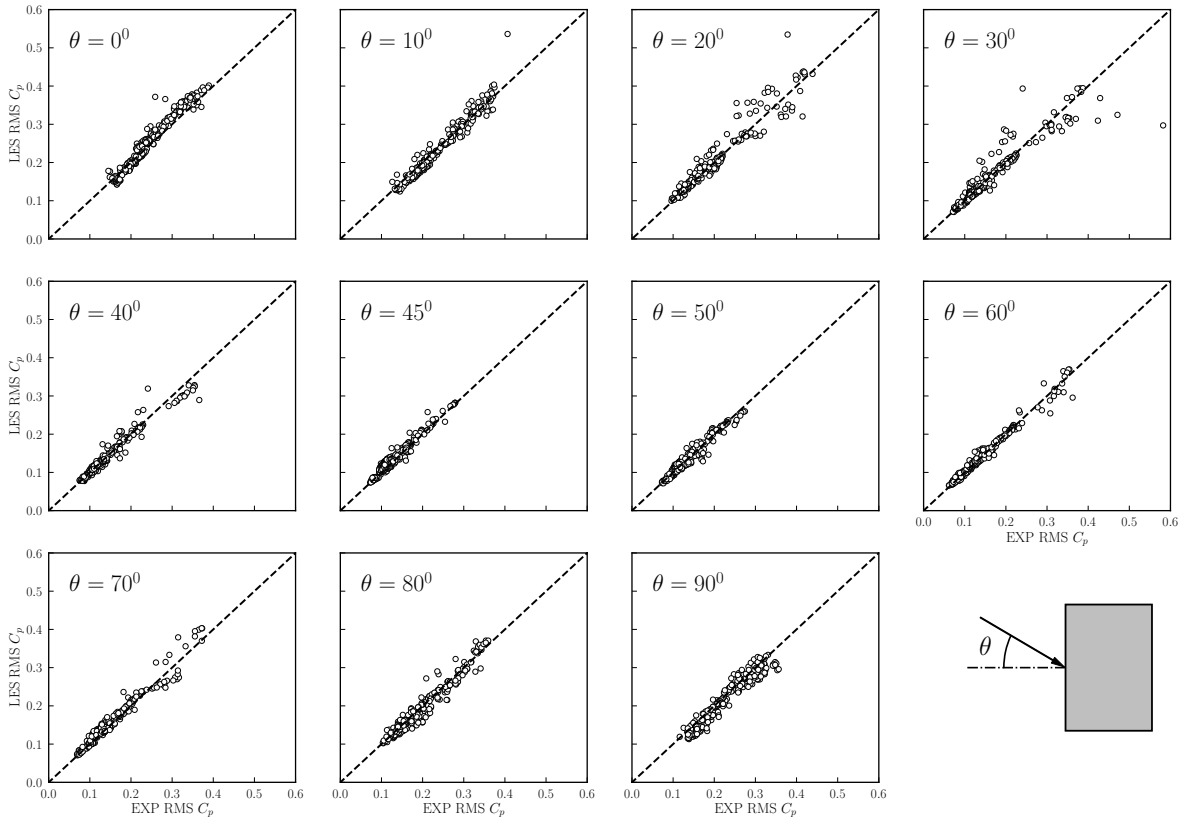


Figure 4.18: Scatter plots comparing RMS C_p from LES with experiment for all wind directions

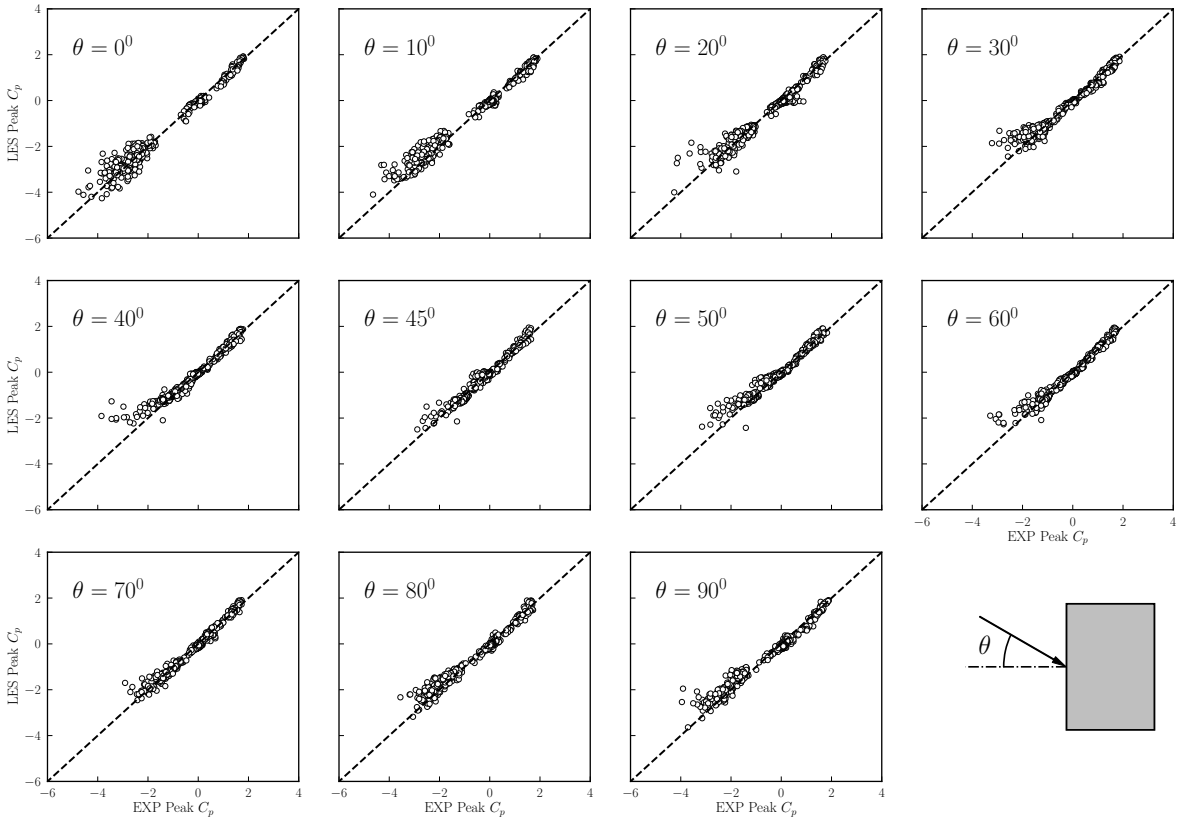


Figure 4.19: Scatter plots comparing peak C_p from LES with experiment for all wind directions

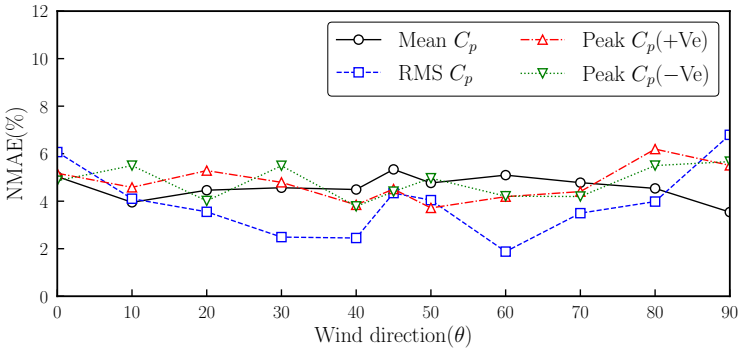


Figure 4.20: Estimated error(NMAE) for mean, RMS, and peak values. The error for each wind direction is calculated based on Eq. (4.21).

negative (suction) peaks, the LES predictions show a 4.8% deviation (under-predicted in most cases) compared to the wind tunnel data.

4.5.3.2 Comparison of Skewness and Kurtosis

The mean and standard deviation do not completely describe the probability distribution of the pressure on the surface of the building. Thus, higher-order moments of the probability distribution of C_p data such as Skewness and Kurtosis are computed and compared for both the experimental and LES data. Skewness is the third central moment and measures the symmetry of the probability distribution. For a normally distributed (Gaussian) measurement, the Skewness is zero. Whereas, Kurtosis is the fourth-order moment and measures the flatness of the probability distribution. For normally distributed data, Kurtosis becomes 3. Figure 4.21 compares the Skewness and Kurtosis of C_p at $2/3H$ of the building along its perimeter for 0° and 90° wind directions. As expected, for the face directly exposed to the wind, the pressure field is close to Gaussian, and LES predictions agree with the experiments having Skewness and Kurtosis close to zero and three. On the side and leeward faces, the C_p becomes non-Gaussian, and the numerical results generally show a good agreement. Comparison of Skewness and Kurtosis for all wind directions using all the pressure taps are shown in Figure 4.22 and Figure 4.23. Overall, it was observed that compared to the Kurtosis, the LES gave a better prediction for Skewness. This possibly is due to Skewness being one level lower order statistics compared to Kurtosis and hence less difficult to match.

4.5.3.3 Grid sensitivity study

The effect of the errors introduced due to spatial discretization are investigated using grid-sensitivity analysis. The sensitivity analysis uses three additional meshes for 0° wind direc-

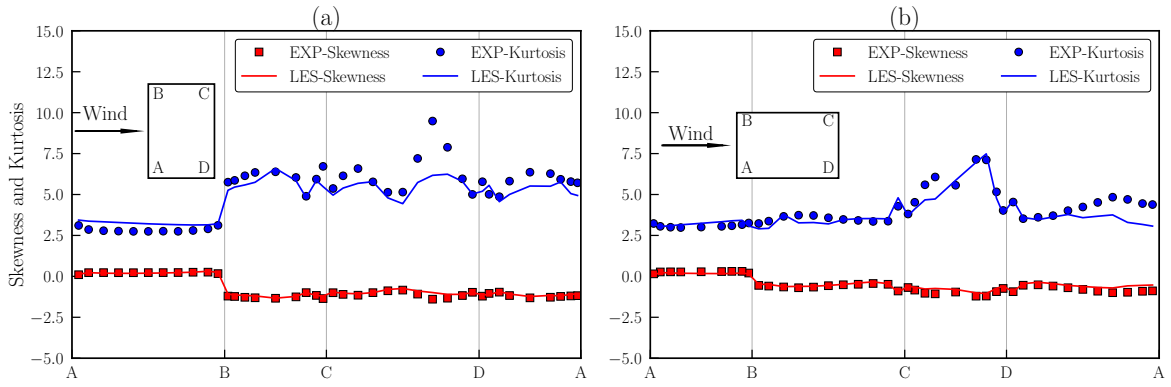


Figure 4.21: Comparison of Skewness and Kurtosis for experimental and LES data at $\frac{2}{3}H$ height of the building: (a) $\theta = 0°$; (a) $\theta = 90°$

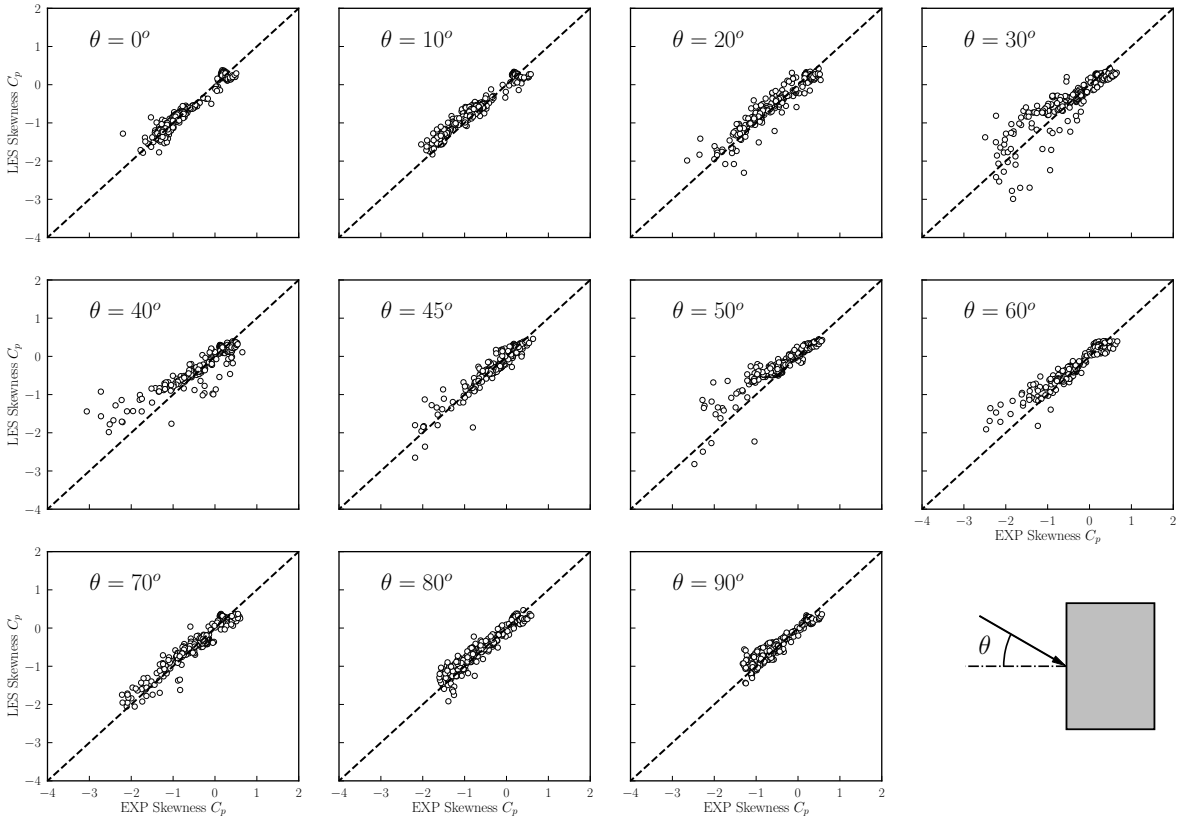


Figure 4.22: Comparison of Skewness for experimental and LES data using all pressure taps and wind directions.

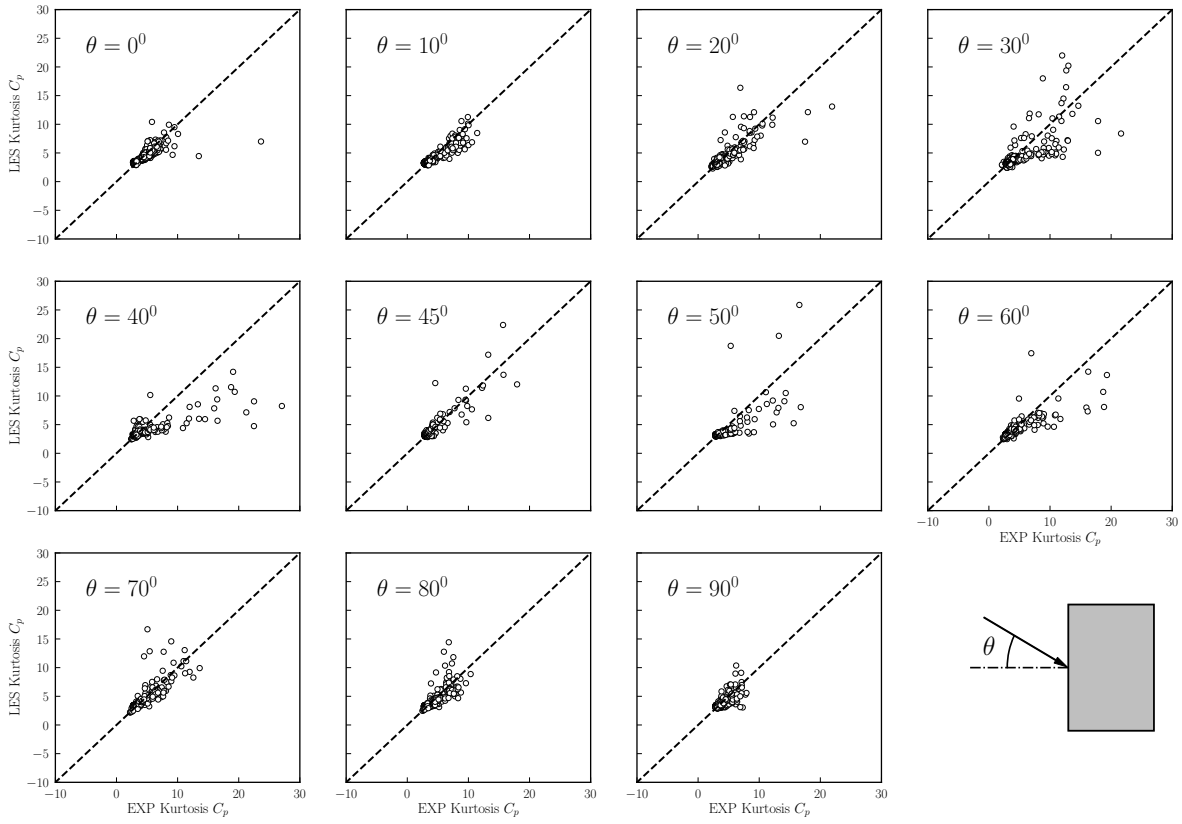


Figure 4.23: Comparison of Kurtosis for experimental and LES data using all pressure taps and wind directions.

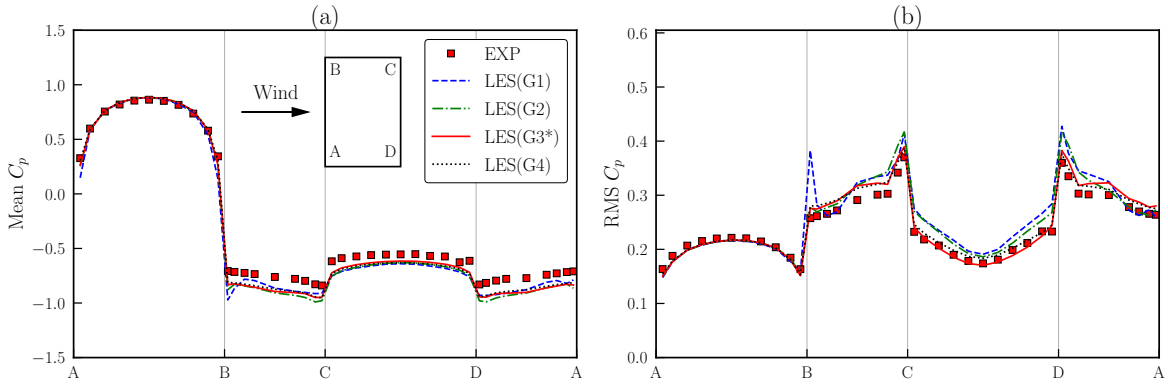


Figure 4.24: Grid sensitivity for distribution of C_p at $2/3H$ of the building for 0° wind direction: (a) mean; (b) RMS. See Table 4.2 for the details of the grids tested.

tion. The details of each grid used for the test runs are provided in Table 4.2. In Figure 4.24 the mean and RMS of pressure coefficients for the four grids used, including the final grid used for the wind load simulation (G3), are depicted. The mean C_p is generally less sensitive to the grid resolution as seen in Figure 4.24a. Except very close to the upwind edge where the flow separation initiates, further grid refinement has a marginal effect on the mean C_p distribution. Similarly, for the face of the building normal to the wind, the grid refinement has almost no effect on the RMS C_p too (see Figure 4.24b). This is because the primary loading mechanism on the front face is quasi-steady by nature (i.e., gust buffeting from the upcoming turbulence); hence, increasing the grid refinement level does not improve the accuracy of resolving an important flow phenomenon. However, on the side and back faces of the building that are in strong flow separation and turbulent wake regions, the RMS C_p distribution shows some variation with the grid refinement level. Considering the complex nature of the flow around the building, the results in Figure 4.24 do not change monotonically with the grid refinement level. Nevertheless, based on Figure 4.24b, it can be seen that further grid refinement beyond the mesh used in G3 seems to have a negligible effect on simulation accuracy.

4.5.3.4 Sensitivity to SGS modeling

Here we study the sensitivity of the LES results to the SGS model used. Figure 4.25 shows the comparison of the C_s distribution found from the three SGS models tested, namely, the Smagorinky, the WALE, and kEqn models described in Section 4.3.1. For the mean C_p distribution, as depicted in Figure 4.25a, the sensitivity to the SGS model used is minimal. Averaged over taps at $2/3H$, the difference between the three models is well below 2%. For RMS C_p distribution, however, as shown in Figure 4.25b, noticeable differences exist, especially near strong flow separation regions (i.e., side faces of the building). Compared to the standard

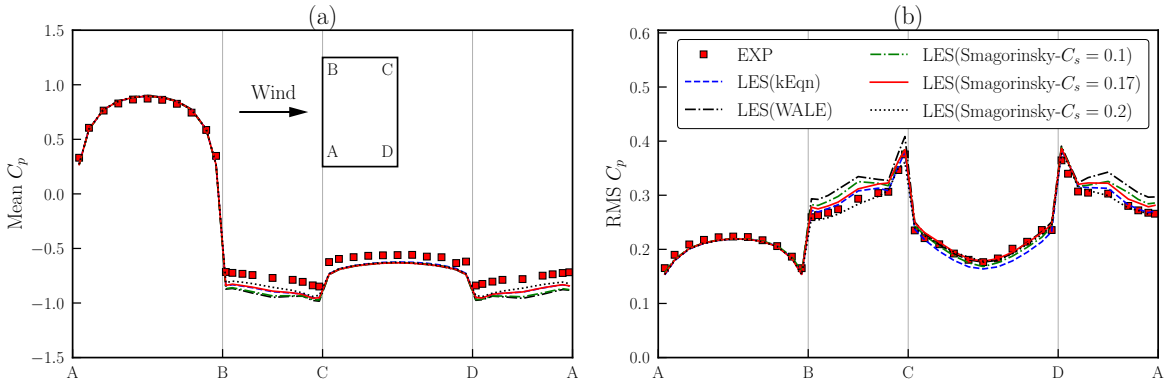


Figure 4.25: Sensitivity of C_p distribution to SGS model used. Comparison for taps located at $2/3H$ of the building for 0° wind direction: (a) mean; (b) RMS

Smagorinky and kEqn SGS models, the WALE mode seems to produce relatively higher C_p fluctuations on the side and leeward faces of the building. Nevertheless, the differences between different SGS models can be tolerable from a practical perspective. For instance, when averaged over all the tap locations, the maximum disparity for RMS C_p among the three modes is close to 4%. Thus, for this study, considering its simplicity and low computational cost, the standard Smagorinsky model was employed. In the same figure, for the Smagorinsky model, the sensitivity of the LES results to the different model constant (C_s) is shown. As expected, for lower C_s values(i.e., 0.1), the model becomes less dissipative, and more turbulence is observed in the flow-separated regions (see Figure 4.25b). The results from both $C_s = 0.168$ and $C_s = 0.2$ are relatively closer to the experimental measurement. Therefore, we adopted a Smagorinsky constant $C_s = 0.168$, as it is an optimal value configured in the CFD solver used in the current study.

4.5.4 Global wind loads

In Section 4.5.3, the C_p distributions on the surface of the building have been validated with experimental data. However, to demonstrate the full potential of LES for predicting the unsteady wind loads on tall buildings, validation of the integrated aerodynamic forces is more relevant than surface pressure distribution, as the former determines the overall structural loads and responses. Furthermore, combined with climate data and structural properties of the building, the integrated aerodynamic forces are ultimately used to determine the design loads on the main structural system. Therefore, validating these quantities is of paramount importance from a practical standpoint. This section compares these global aerodynamic loads obtained from LES against the wind tunnel measurements.

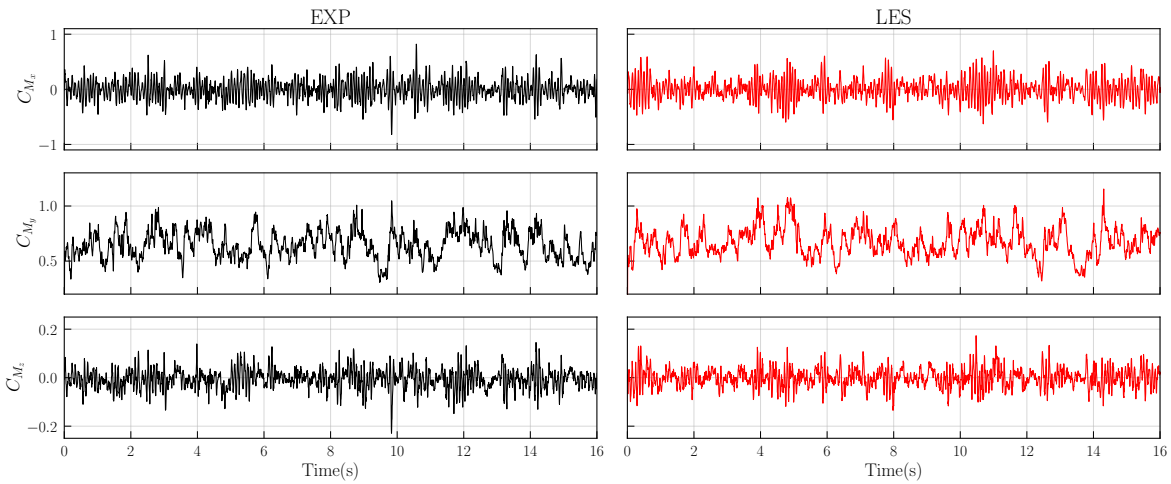


Figure 4.26: Time-series of the base moment coefficients C_{M_x}, C_{M_y} and C_{M_z} (top to bottom) for $\theta = 0°$: (left) experiment; (right) LES

4.5.4.1 Base force and moment coefficients

The aerodynamic base loads are evaluated using the pressure integration procedure described in Section 4.4.3. For ease of comparison, for both LES and experimental data, these aerodynamic forces and overturning moments are expressed in non-dimensional form as:

$$C_{F_x} = \frac{F_x}{\frac{1}{2}\rho\overline{U}_H^2 BH}, \qquad C_{F_y} = \frac{F_y}{\frac{1}{2}\rho\overline{U}_H^2 DH}, \tag{4.22a}$$

$$C_{M_x} = \frac{M_x}{\frac{1}{2}\rho\overline{U}_H^2 DH^2}, \qquad C_{M_y} = \frac{M_y}{\frac{1}{2}\rho\overline{U}_H^2 BH^2}, \qquad C_{M_z} = \frac{M_z}{\frac{1}{2}\rho\overline{U}_H^2 BDH} \tag{4.22b}$$

where ρ is the air density; B and D are the widths of the wider and narrower faces of the building, respectively; H is the height of the building, and $\overline{U}_H$ represents the roof-height mean velocity. The global loads F_x and F_y represent the integrated forces in x and y direction, while M_x, M_y and M_z denote base moments about x, y and z (torsional) directions, respectively.

Figure 4.26 shows sample time-series of the base moment coefficients C_{M_x}, C_{M_y} and C_{M_z} for 0° wind direction. A qualitative comparison of the time series indicates that the loads from LES and the experiment exhibit similar behavior. For instance, the fluctuations of the along-wind load (C_{M_y}) contain a wide range of frequencies resembling the fluctuations in the approaching flow. In contrast, the moment created by cross-wind force (C_{M_y}) is a narrow-band process predominantly produced by vortex shedding.

The spectral content of the base aerodynamic loads is shown in Figure 4.27. The figure depicts the reduced power spectral density of M_x, M_y, and M_z for the three wind directions,

i.e., 0°, 45°, and 90°. In the along-wind direction, results agree well with the experiment for a broad range of frequencies, especially for 0° and 90° wind directions. It is worth noting that when the wind is normal to the faces of the building, the along-wind loads are directly related to the quality of the inflow turbulence generator. Recalling that the pressure fluctuations on the windward face are less sensitive to the fidelity of the LES model (e.g., grid resolution) as seen in Figure 4.24, the accuracy of the LES prediction in the along-wind direction is expected to be good, provided that the spatiotemporal characteristics of the approaching flow are adequately reproduced. On the other hand, the cross-wind and torsional moment spectra are relatively more challenging to match because accurate modeling of the body-generated turbulence around the building and its complex interaction with the incoming turbulence needs a finer grid resolution coupled with more accurate SGS model.

As seen in Figure 4.27, essential flow phenomena such as vortex shedding are well captured. This is distinctively seen as peaks at the shedding frequency in the cross-wind moment spectra of 0° and 90° cases (e.g., see the M_x plots in Figure 4.27). Compared to 0°, the peak for 90° occurs over a broader frequency range mounting to the fact that the vortex shedding is less organized for the 90° case (Obasaju, 1992). However, the LES cross-wind overturning moment spectrum for the 90° case shows a subtle difference around the shedding frequency. For 0° wind angle, the cross-wind moment spectrum M_x from the LES and experimental data show a narrow-band peak near a reduced frequency of $fB/\overline{U}_H \approx 0.10$ which corresponds to a Strouhal number ($S_t = fB/U = 0.1$) widely reported in the literature for CAARC building in atmospheric boundary layer flow. Similarly, the base shear forces (F_x and F_y) are in satisfactory agreement with the experiment. However, since the base shears are less significant design metrics for high-rise buildings compared to the overturning moments, they are omitted in this comparison for brevity.

In Figures 4.28 and 4.29, the mean and RMS of the force coefficients (C_{F_x} and C_{F_y}) and base moment coefficients (C_{M_x}, C_{M_y} and C_{M_z}) are compared for all the wind directions studied (0° to 90°). Note that the base load coefficients in Figures 4.28 and 4.29 are only aerodynamic loads and do not include the resonant contribution from the building. As shown in the figures, the mean and RMS base load coefficients are estimated with reasonable accuracy. In Table 4.4, a summary of the errors of the LES results relative to the experiment is reported. These errors are calculated based on the expression in Eq. (4.21) (NMAE) and averaged over all the wind directions. Overall the mean base load coefficients are predicted within a 6% error range, while for the RMS values, the maximum deviation is well within 10% range.

In addition to base aerodynamics loads, the generalized loads calculated using the expression in Eq. (4.18) are compared for each mode shape considered in the analysis. Since the generalized loads take into account the effect of the mode shape, they are more representative

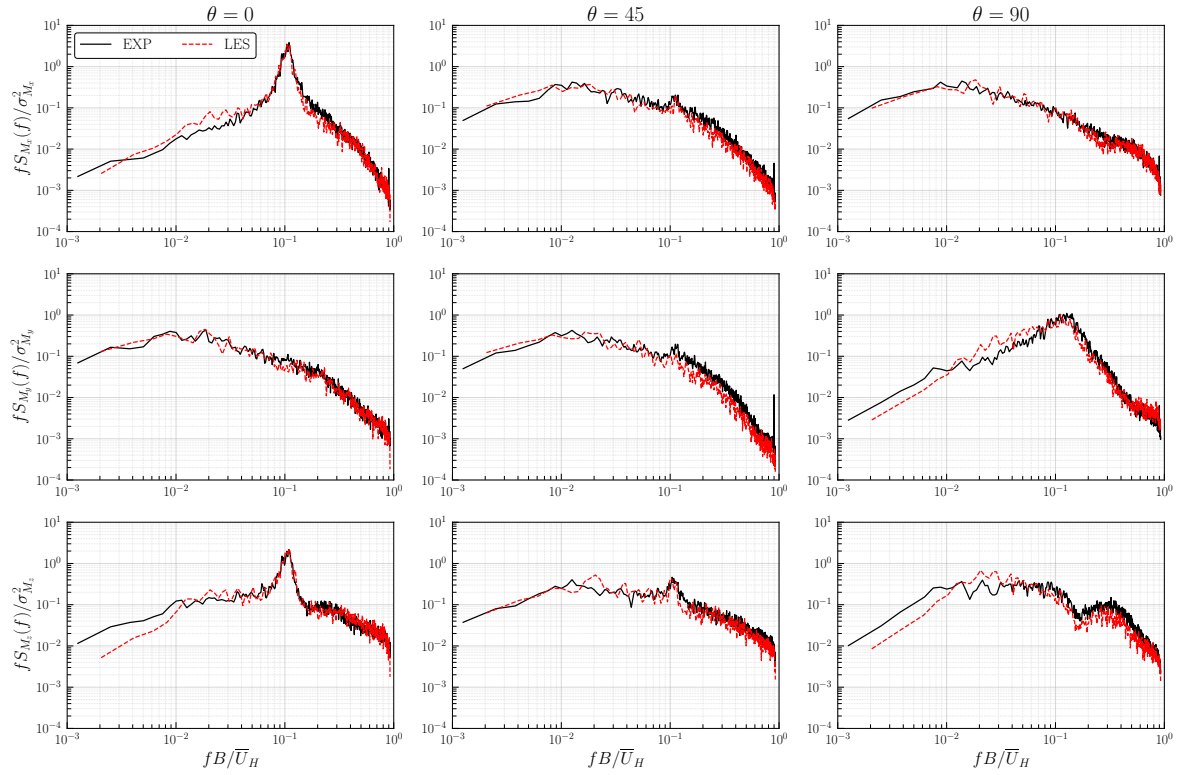


Figure 4.27: Reduced power spectral density of base moments: (rows) C_{M_x}, C_{M_y} and C_{M_z}; (columns) wind directions, 0°, 45° and 90°

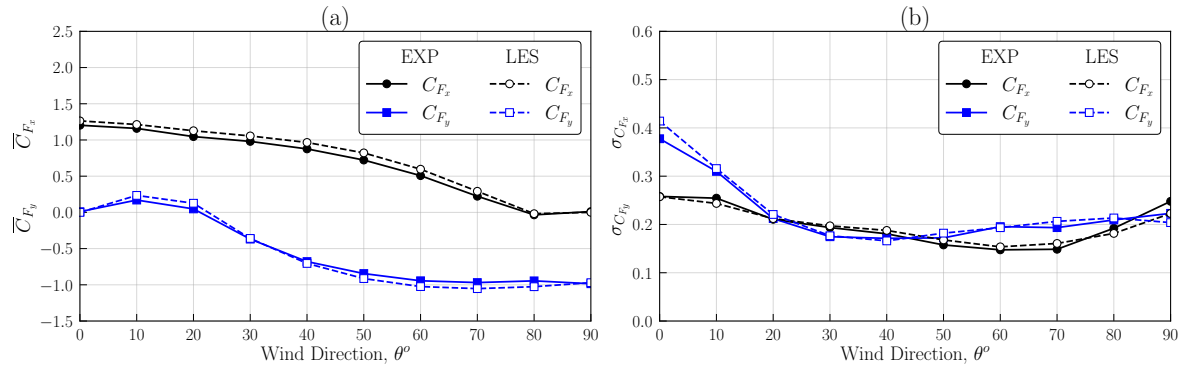


Figure 4.28: Comparison of force coefficients (C_{F_x} and C_{F_y}) per wind direction: (a) mean; (b) RMS

Table 4.4: Errors for mean and RMS base load coefficients

Error(%)	C_{F_x}	C_{F_y}	C_{M_x}	C_{M_y}	C_{M_z}
Mean	5.19	4.45	4.79	5.71	3.61
RMS	7.97	5.16	5.44	9.14	2.75

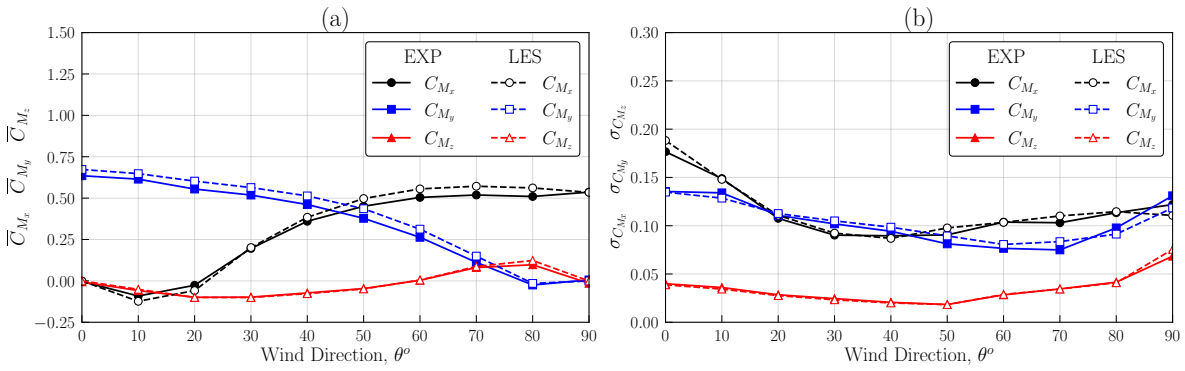


Figure 4.29: Comparison of base moment coefficients (C_{M_x}, C_{M_y} and C_{M_z}) per wind direction: (a) mean; (b) RMS

of the dynamic loads exciting the building. In Figure 4.30, the spectra of the generalized loads for 0° wind direction are depicted for all six modes. The first three modes are fundamental modes of vibration representing two flexural (x and y) and one torsional mode, whereas the last three represent higher modal contribution in x, y, and torsional directions, respectively. Similar to the spectra moment coefficients presented in Figure 4.27, the generalized load spectra from the LES appear to agree with the experiment well, rendering the magnitudes and distributions of storey loads are accurately estimated.

4.5.5 Structural responses

This section investigates the responses of the structure, specifically displacement and acceleration responses, using the aerodynamic loads evaluated from the LES and experimental aerodynamic data. The dynamic analysis is performed using the theory of random vibration in the time domain, following the procedure presented in Section 4.4.

For calculating the structural response, aerodynamic and structural properties in full scale using the first 6 modes of vibration are adopted. The responses are calculated using a roof-height service design wind speed of 30 m/s. It should be noted that for comparing the structural responses, no climate data and wind directionality effect is considered. Therefore, considering the main objective of this study is to assess the accuracy of the LES results, for all the wind directions studied, the same wind speed is used for calculating the responses.

4.5.5.1 Displacement response

Figure 4.31 shows the time series of the top-floor structural displacements in x, y, and rotational directions for both the LES and experiment. For ease of comparison, the rotational displace-

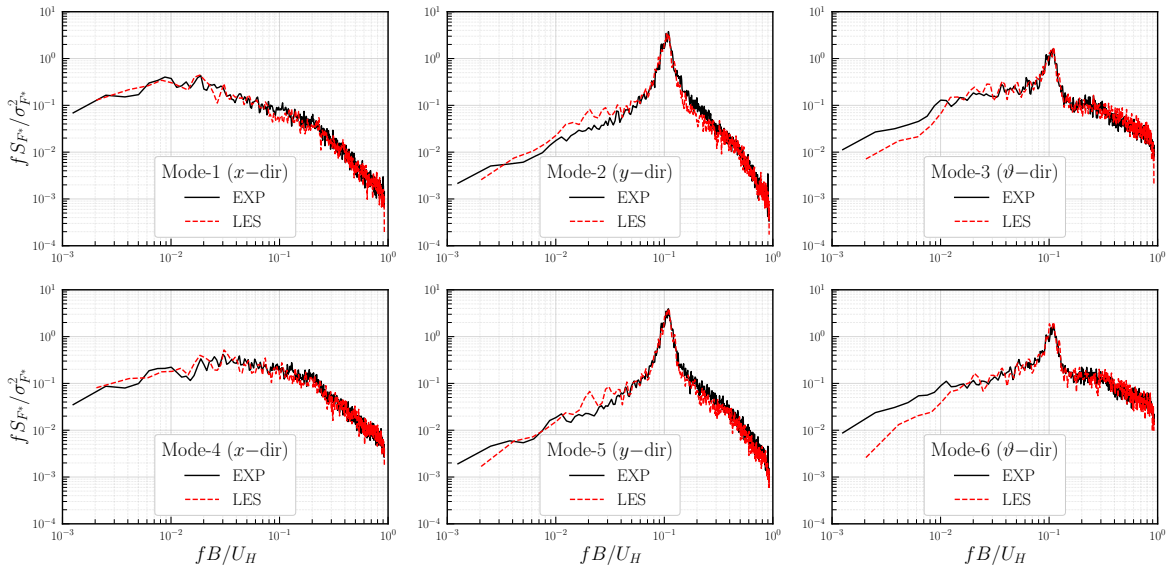


Figure 4.30: Generalized wind load spectra for the first six modes of vibration.

ments shown in the figure are multiplied by the radius of gyration at the rooftop level. Next, we examine the response spectrum of the structural displacements as shown in Figure 4.32. The figure compares the reduced power spectral density of the top-floor displacement determined from LES against the experiment for 0°, 45°, and 90° wind directions. It is well-recognized that the structure's response has a background and resonant contributions (Davenport, 1961a, 1967). As depicted in Figure 4.32, the response spectra show distinct resonant peaks concentrated around the natural frequency of the structure. On the other hand, the background contributions are observed mainly in the along-wind direction distributed over the low-frequency end of the spectra (see the spectrum of $d_x(t)$ for the 0° case).

The accuracy of the LES in predicting the contribution of the resonant peaks depends significantly on how well the generalized force spectra near the natural frequencies of the building are resolved. This becomes especially critical if the structure is dynamically sensitive and the resonant contributions are much higher than the excitation due to the low-frequency background turbulence. To investigate this further, the structure's response is broken down into background and resonant contributions. The RMS of the background response (σ_{d_B}), which contains contributions from slowly varying forces, is treated as quasi-static and can easily be calculated from the generalized force as:

$$\sigma_{d_B} = \frac{\sigma_{F^*}}{k^*} \tag{4.23}$$

where σ_{F^*} is the RMS of the generalized force, and k^* represents the generalized stiffness of

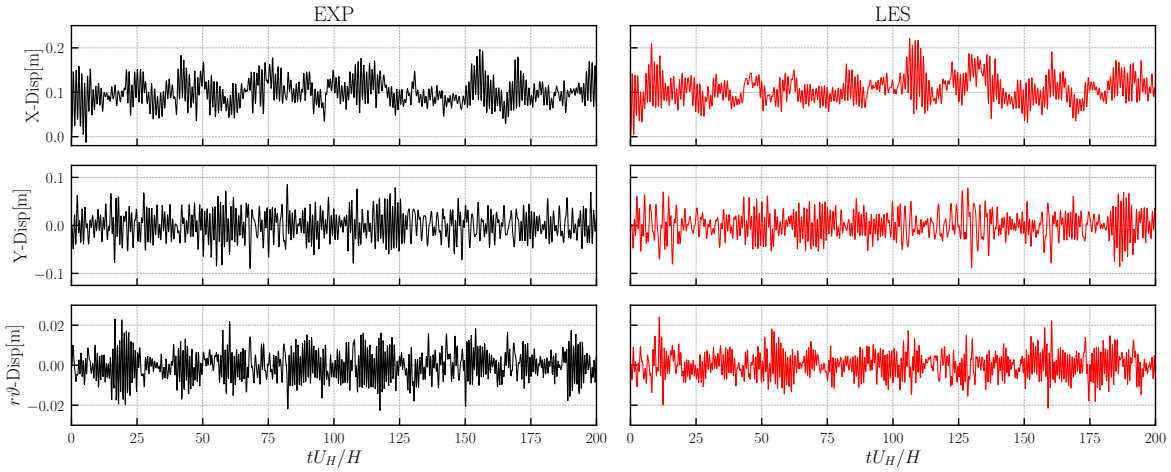


Figure 4.31: Comparison of top floor displacement time histories in x, y and rotational directions(top to bottom) for 0° wind direction for $\xi = 2\%$ damping: (left) experiment; (right) LES

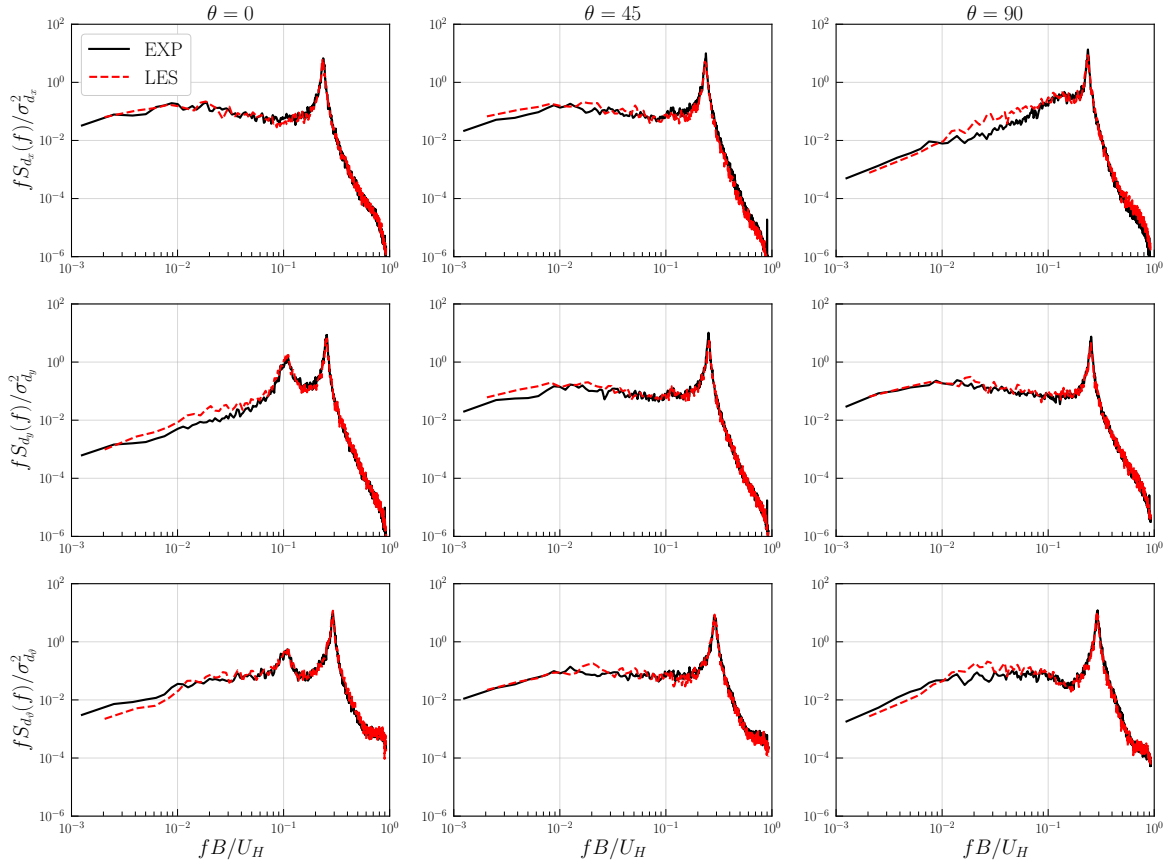


Figure 4.32: Reduced power spectral density of top floor displacement for $\xi = 2\%$ damping: (rows) d_x, d_y and d_ϑ; (columns) wind directions, 0°, 45° and 90°

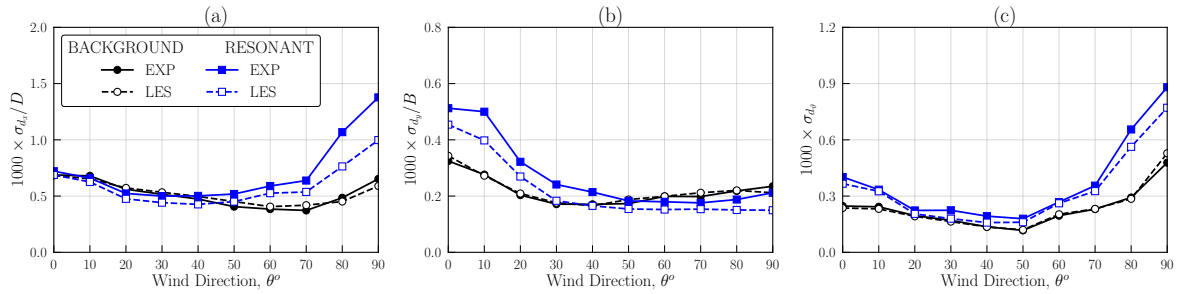


Figure 4.33: Comparison of the background and resonant displacement responses from LES and with experiment for $\xi = 2\%$ damping: (a)$x-$direction; (b)$y-$direction; (c)$\vartheta-$direction

the structure (Davenport, 1961a, 1967). Since we used time domain analysis, the RMS of the resonant part (σ_{d_R}) is conveniently taken as the remaining part of the total mean square fluctuating response, $\sigma_{d_R} = \sqrt{\sigma_d^2 - \sigma_{d_B}^2}$.

In Figure 4.33, the contribution of the background and resonant parts of the displacement responses are compared over the range of wind directions considered. The main observation from the comparison is that LES predicted the background responses with high accuracy compared to the resonant part. Considering its quasi-static behavior, the accuracy of LES in predicting the background responses mainly depends on how precisely the RMS of the generalized force is estimated. Since the LES model generally predicted the mean and RMS of the aerodynamic forces well, the background responses from LES show only minor deviations from the experiment, as depicted in Figure 4.33. However, this is not the case for the resonant response. Estimating the resonant component of the response employing LES requires capturing the high-frequency content of the generalized force properly; most importantly, the power spectrum of the generalized force at the natural frequency of the structure, $S_F(f_0)$ as opposed to its RMS value. Thus, it is worth emphasizing that to accurately predict the wind-induced response of tall buildings using LES, turbulent fluctuations and the aerodynamic mechanism that produces dynamic wind loads with frequencies near the natural frequency of the building need to be adequately resolved. This needs to be considered when designing the computational grid, selecting the SGS model, and generating inflow turbulence.

4.5.5.2 Acceleration response

For tall buildings, top-floor accelerations are evaluated to insure compliance with serviceability criteria. Hence, validation of the LES and dynamic structural analysis generated top-floor acceleration is critical. Figure 4.34 depicts the power spectral density of a_x, a_y and a_ϑ components of the top-floor acceleration for 0°, 45° and 90° wind directions. As expected, the top-floor acceleration of the building is dominated by the resonant response, which is captured

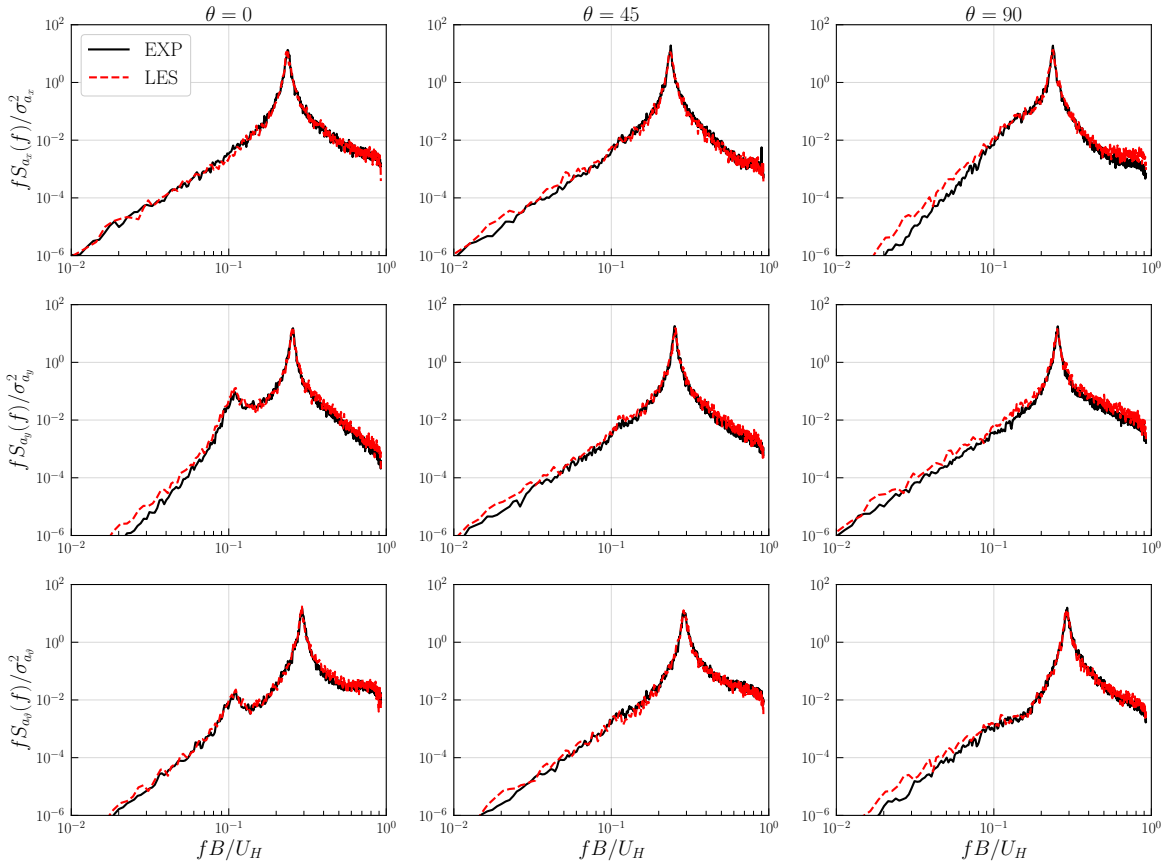


Figure 4.34: Reduced power spectral density of top floor acceleration with $\xi = 2\%$ damping: (rows) a_x, a_y and a_ϑ; (columns) wind directions, 0°, 45° and 90°

by both the LES and BLWT experiment.

In Figure 4.35, the RMS of sway and torsional accelerations obtained from the LES and BLWT experiment are compared for all the wind directions. The torsional accelerations shown in Figure 4.35c are expressed as linear accelerations at a distance furthest away from the center of the building. Figure 4.35 shows that the cross-wind response of a tall building is dominant compared to the along-wind response. The largest RMS acceleration occurs in $x-$direction for 90° wind angle when the wind is perpendicular to the narrower face of the building (see Figure 4.35a). The main reason being, the structure has a lower frequency and generalized stiffness in the $x-$direction.

The resultant peak acceleration is calculated by combining the sway and torsional accelerations are compared in Figure 4.36a. To calculate the expected peak acceleration ($\hat{a}_r$), a simple yet conservative combination approach based on the Square Root of the Sum of Squares (SRSS) rule (Rosenblueth, 1951) and Davenport peak factor, (Davenport, 1964) are employed

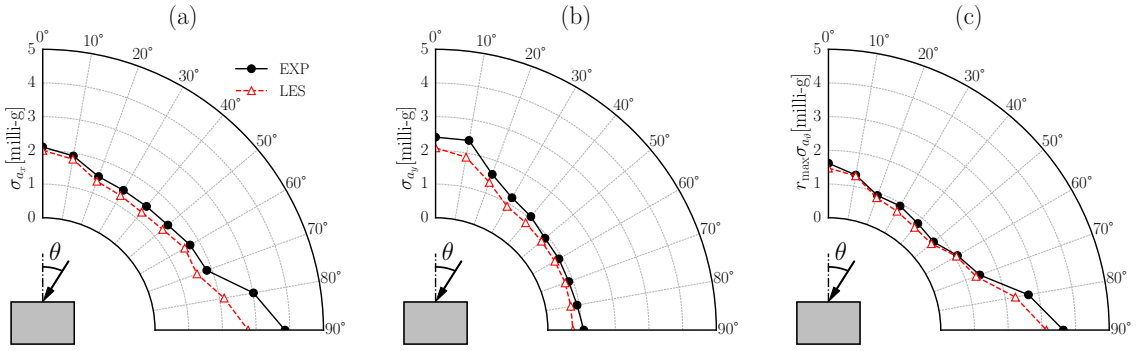


Figure 4.35: RMS of the top floor acceleration for different wind directions using $\xi = 2\%$ damping: (a) x-acceleration; (b) y-acceleration; (c) torsional acceleration

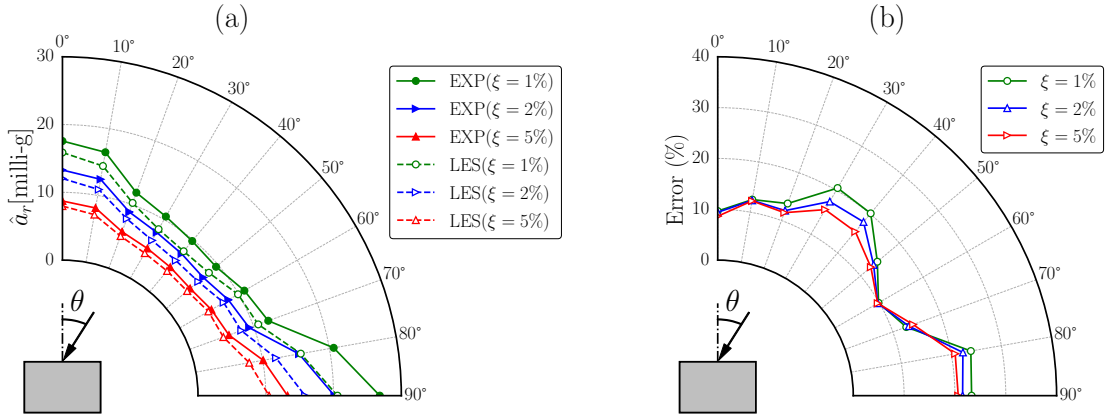


Figure 4.36: Comparison of top floor acceleration for different wind directions: (a) peak acceleration; (b) relative deviation of the LES from experiment

as:

$$\hat{a}_r = g_p \sqrt{\sigma_{a_x}^2 + \sigma_{a_y}^2 + (R_x^2 + R_y^2)\sigma_{a_\theta}^2} \tag{4.24}$$

where (R_x, R_y) are the coordinates of the building's corner from the mass center of the top floor. For the present study, the peak factor g_p is about 3.72.

The relative errors in peak acceleration are shown in Figure 4.36b. The same figure also shows the variation of the peak acceleration with structural damping. These responses are calculated using 1%, 2%, and 5% damping ratios. From Figure 4.36, it can be seen that LES gave a reasonable estimate of the resultant peak acceleration for most wind directions. It was observed that the resultant peak acceleration shows the highest deviation for the 90° case, which is roughly 22%. Overall, averaging over all the wind directions and damping ratios used, the LES predictions in the current study are well within 15% tolerance.

4.6 Conclusion and summary

The wind loads and responses of the well-known benchmark tall building (the CAARC model) were evaluated numerically to demonstrate the capabilities of LES for the wind-resistant design of tall buildings. The key challenges while using LES for wind load evaluation were addressed by employing the recent developments in realistic turbulent inflow generation and wall treatment techniques for ABL flows. For this purpose, a synthetic inlet turbulence generator and an implicit ground roughness model were implemented in the open-source CFD toolbox. For validating the numerical models, experimental measurements were conducted in the boundary layer wind tunnel laboratory. The results obtained from LES were thoroughly validated in comparison with the wind tunnel measurements stage by stage.

Statistical characteristics of the approaching wind flow in terms of the mean velocity, turbulence intensity, integral length, and velocity spectra at the incident location compared well with the experimental measurements. The capabilities of LES for estimating cladding loads were demonstrated by statistical comparison of the mean, RMS, and peak pressure distributions on the building surface with those estimated from the BLWT experiments. Furthermore, integrated aerodynamic loads such as base shear and overturning moments showed satisfactory agreement with the experimental data. Overall, the base aerodynamic loads are generally well predicted by the LES. The mean base load coefficients are estimated within a 6% error range, while for the RMS values, the maximum deviation is approximately below 10%. Finally, using the integrated aerodynamic loads from LES and the experiment, the dynamic response of the structure is evaluated by performing a time-domain analysis of the structure. The top-floor structural displacements and peak accelerations were compared against the experimental data. It was observed that the LES predicted the low-frequency background displacement response with high accuracy compared to the resonant part for all wind directions considered. Also, the numerical estimate of the resultant peak acceleration is encouraging. On average, the peak acceleration from the LES roughly showed a 15% deviation with the experiment.

Considering the encouraging level of agreement between LES and the experiment, the current study has demonstrated the potential of LES for estimating wind loads and wind-induced responses for the wind-resistant design of tall buildings. With the increasing availability of cutting-edge research and cloud computing platforms, the capabilities of LES to accurately resolve turbulent separated flows in wind engineering applications are progressively improving. However, it is to be noted that, especially for wind-induced response estimation, LES still needs further validation using more accurate aeroelastic experimental models. The computational modeling used for the current study assumed that the building undergoes small displacement. For flexible structures, however, the aeroelastic effect should be incorporated using

high-fidelity fluid-structure interaction simulations. Given the current state of LES capability, it offers numerous advantages to structural engineers and architects at the preliminary stage to address the fundamental design challenges earlier in the design process.

Chapter 5

LES-based wind load evaluation on a tall building located in a city center: comparison with experimental data

5.1 Introduction

In recent years, in the research community, we have witnessed significant developments in the use of CFD tools to estimate wind loads on buildings (Abdi and Bitsuamlak, 2014b; Dagnew and Bitsuamlak, 2013; Elshaer et al., 2016; Huang et al., 2010; Lamberti et al., 2018; Lamberti and Gorlé, 2020; Melaku and Bitsuamlak, 2021; Ricci et al., 2017). Particularly, for tall building aerodynamics, several CFD-based studies were conducted to predict wind loads employing different types of turbulence models. Noting the importance of validation, the numerical results from the aforementioned studies were compared with experimental data in most cases. Especially the Commonwealth Advisory Aeronautical Council (CAARC) standard tall building (Melbourne, 1980) has served as an important benchmark for validation. Nevertheless, the vast majority of CFD-based studies are focused on the validation of wind loads on isolated tall buildings. Although this is a key step towards establishing the required confidence in the CFD model, it is not enough from a practical standpoint. In practice, most tall buildings are designed and constructed in a complex urban setup. Therefore, the accuracy, as well as the computational cost of CFD models for wind load evaluation, must be assessed by taking these complexities into consideration.

In the past, few attempts have been made to study wind loads on a tall building located in complex urban environments using large-eddy simulation (LES) (Elshaer et al., 2016; Nozu et al., 2008, 2015; Tamura et al., 2015, 2017, 2010; Yan and Li, 2016; Yoshikawa and Tamura,

2015). In these studies, for accurate transient wind load estimation on buildings in urban areas, authors emphasized the importance of (1) proper modeling of atmospheric boundary layer(ABL) flow conditions, (2) generation of an optimum unstructured grid in the proximity of the target region, and (3) adequate turbulence modeling with wall-treatment as the grid resolution requirement is severe. In addition, considering the current challenges of LES to predict wind load in urban areas, it is clear that the numerical simulations need to be accompanied by reliable wind tunnel measurements for validation. Thus, it is imperative that more research needs to be done to assess the accuracy of LES for dense urban settings with more realistic simulation scenarios that parallel a typical wind tunnel study.

In the current study, wind load on a tall building with a complex geometry located in a city center with a realistic urban setup is studied using large-eddy simulation (LES). The approaching wind characteristics measured in a boundary layer wind tunnel were properly reproduced using a synthetic inflow turbulence generation method recently developed by Melaku and Bitsuamlak (2021). For modeling the roughness of the surrounding terrain, an implicit roughness modeling technique was employed. The numerical simulations were conducted using OpenFOAM on an unstructured grid, adequately resolving the surrounding urban environment. The global wind loads from LES were validated using experimental measurements from High-Frequency Pressure Integration (HFPI) model. The challenges related to accurate wind load simulation in complex urban environments using LES are addressed. From a practical application standpoint, this study also aims to demonstrate how properly validated CFD models can give designers the ability to generate site-specific, reasonably accurate preliminary design loads, especially at early design stages. Moreover, with abundant flow field data from CFD, it is possible to bring helpful design insight by examining complex aerodynamic loading mechanisms responsible for the variation of wind loads.

This paper is organized into five sections. In Section 5.2, we briefly describe the boundary layer wind tunnel data utilized for validating the LES model. Section 5.3 provides the details of the numerical modeling process. In Section 5.4, the wind loads estimated using the LES model are compared against the experimental data. Finally, Section 5.5 summarizes the main findings from the present study.

5.2 Reference wind tunnel measurement

This section briefly describes the aerodynamic wind tunnel data used to validate the LES model in a realistic urban setup. The experimental measurement is conducted at the Boundary Layer Wind Tunnel Laboratory (BLWTL) of Western University. The aerodynamic data is measured using a High-Frequency Pressure Integration (HFPI) model of a tall building located in a city

center. The pressure model is constructed in a 1:400 scale using ABS plastic material and instrumented with pressure taps. Figure 5.1 shows a photograph of the rigid model in the wind tunnel with the upstream terrain model employed. The study building used for testing has a complex geometry. The main roof has a full-scale height(H) of 277.82 m above the ground level. The effective plan width B and depth D of the building in full scale are 78.49 m and 47.70 m, respectively. Figure 5.1b and c depict the elevation and plan views of the building together with the coordinate system used to define the wind loads. The surrounding city is represented using a detailed proximity model extending approximately a 500 m radius from the center of the building. The surrounding model is built in blocks from Styrofoam material as shown in Figure 5.1a.

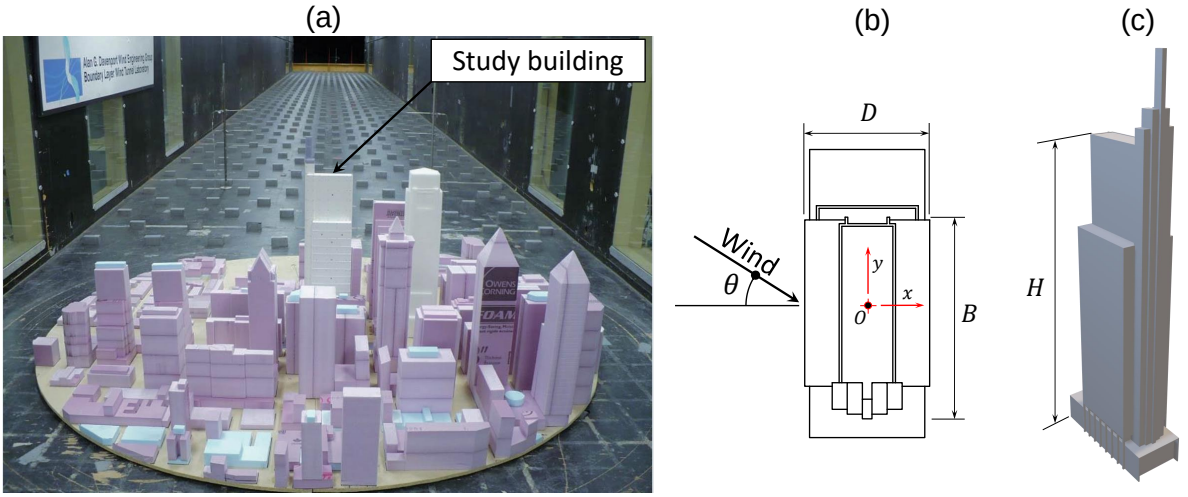


Figure 5.1: Aerodynamic model used for the validation: (a) picture of the model in the wind tunnel; (b) roof-plan view and dimensions of the building; (c) isometric view of the study building

The original experimental measurement was conducted for a total of 36 wind directions (θ) from 0° to 360° in increments of 10°. However, considering the high computational cost of running LES for all the wind directions, only a few representative wind directions were chosen for the validation. Thus, for the current study, four different wind directions, specifically 0°, 90°, 180°, and 270° are chosen for the validation task (see Figure 5.1b). These wind directions are chosen such that they represent cases with complex flow phenomena peculiar to the urban environment, such as impingement, sheltering effect, and interference from adjacent tall buildings.

5.2.1 Characteristics of the simulated terrain

The testing facility used for the simulation is a closed-circuit wind tunnel having a long section that stretches 39 m long with a test cross-section of 3.4 m wide and 2.5 m high. The experiment simulated a fully developed turbulent atmospheric boundary layer flow employing turbulence-generating devices such as barriers and spires placed near the inlet. On the wind tunnel floor, generic roughness blocks are used to represent the ground upwind terrain roughness (see Figure 5.1a). The simulated longitudinal mean velocity and turbulence intensity profiles measured just upstream of the center of the turntable are shown in Figure 5.2a. The roof-height mean wind speed measured from the tunnel is 12.68 m/s, while the turbulence intensity at the building height is 13%. The model scale Reynolds number of the flow calculated based on the incoming roof-height mean wind speed and building height H is approximately 5.9×10^5.

The roof-height reduced velocity spectrum for the stream-wise component of the velocity is shown in Figure 5.2b. The spectrum for the experiment is estimated using measurement data from Hot-wire anemometer, hence only the stream-wise flow component is measured. For comparison purposes, the von Karman spectrum is also shown in the same plot. As seen in Figure 5.2b, the spectrum from the wind tunnel matches the von Karman model reasonably well for a wide frequency range. Thus, the Karman model will be utilized for later comparison and to generate synthetic inflow turbulence.

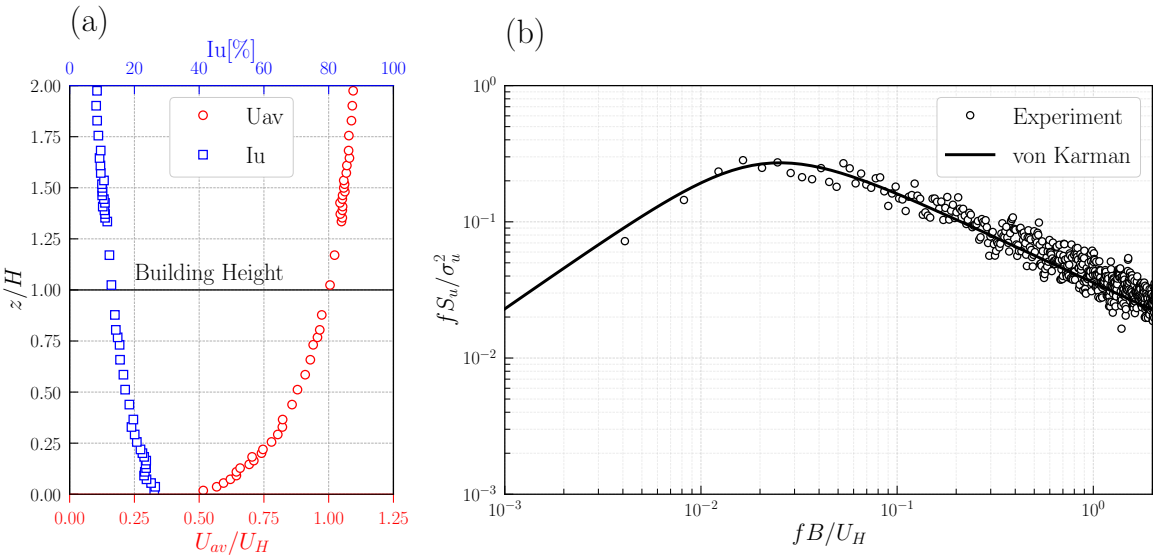


Figure 5.2: Approaching flow characteristics from the experimental measurement: (a) stream-wise mean velocity and turbulence intensity profiles; (b) roof-height velocity spectra

5.3 Numerical model

This section describes the details of the LES model employed for the wind load simulation. The LES modeling is done in 1:400, the same as the experiment. Although it is possible to conduct the CFD simulation at full scale, the same geometric scale used in the experimental study is chosen for ease of comparison with experimental data. The CFD simulations are carried out using wall-modeled large-eddy simulations replicating simulation conditions similar to the boundary layer wind tunnel test. The approaching wind characteristics were modeled using synthetically generated inflow turbulence. The computation grid for the complex urban environments is generated as an unstructured hex-dominated mesh with several refinement regions. All the simulations were conducted employing an open-source CFD toolbox called OpenFOAM-8 (Weller et al., 1998). The overall CFD modeling procedure is summarized in Figure 5.3.

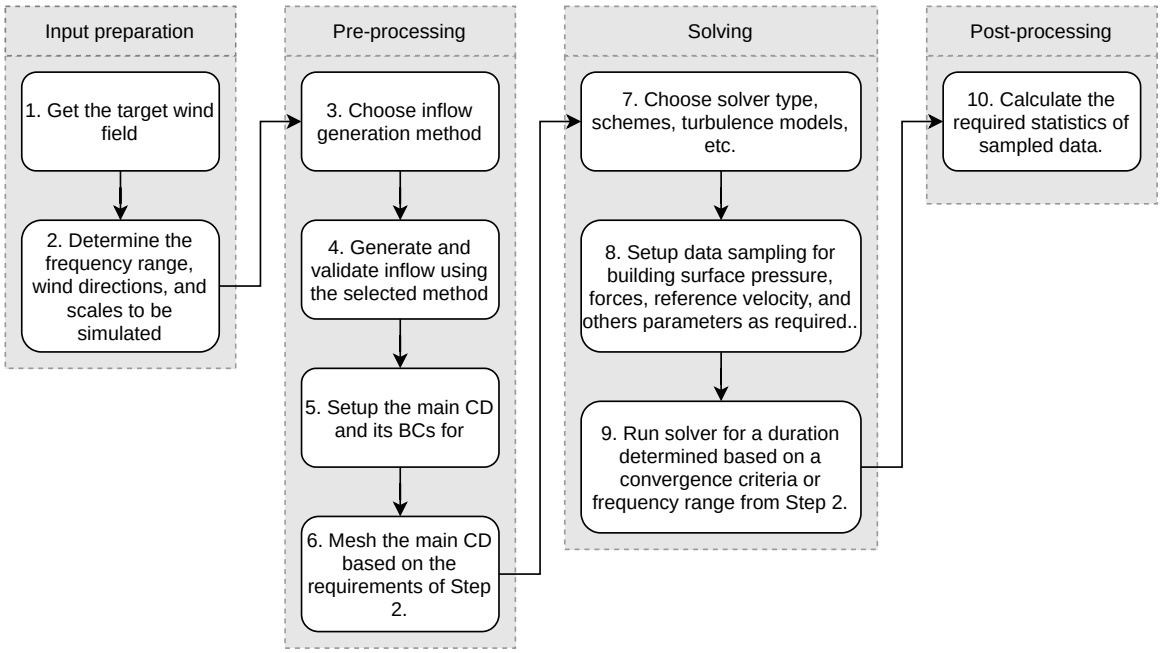


Figure 5.3: Description of the CFD modeling procedure: input preparation, pre-processing, solution, and post-processing (left to right)

5.3.1 Size of the computational domain

The computational domain is designed based on the recommendation provided in COST (Franke et al., 2011). Considering that the current study primarily deals with the comparison of LES with experiments, it is reasonable to use a computational domain that has a comparable size

to the test section of the boundary layer wind tunnel. The experimental HFPI model of the building combined with the proximity model has a blockage ratio approximately equal to 11%, which is slightly higher than the recommended maximum blockage ratio of 8% for wind tunnel studies (ASCE-49-21, 2022). For the LES model, however, to avoid possible blockage effects and minimize the influence of the domain boundaries, we adopted a computational domain that has a cross-section dimension much larger than the test section of the boundary layer wind tunnel. The height and width of the computational domain are set to $5m(7.2H)$ and $10m(14.4H)$, respectively, compared to 2.5 m high and 3.4 m wide test section of the wind tunnel. This results in a maximum blockage ratio of 1.86% that happens for 0° wind direction. It should be noted that this level of blockage ratio is much smaller than the 3% limit normally recommended for CFD studies (Franke et al., 2011).

In the longitudinal direction, the domain is extended to 20 m ($28.8H$). The center of the study building is located 5 m ($7.2H$) away from the inlet. Note that a long fetch length in boundary layer wind tunnels is often required to naturally develop a turbulent boundary layer. However, for LES-based studies, a long fetch distance is not required, provided that the inflow turbulence specified at the inlet satisfies the characteristics of the ABL flow. Figure 5.4 shows the extent of the computational domain used with dimensions expressed relative to the height of the study building. For all the simulated LES cases, the size of the computational domain remains the same.

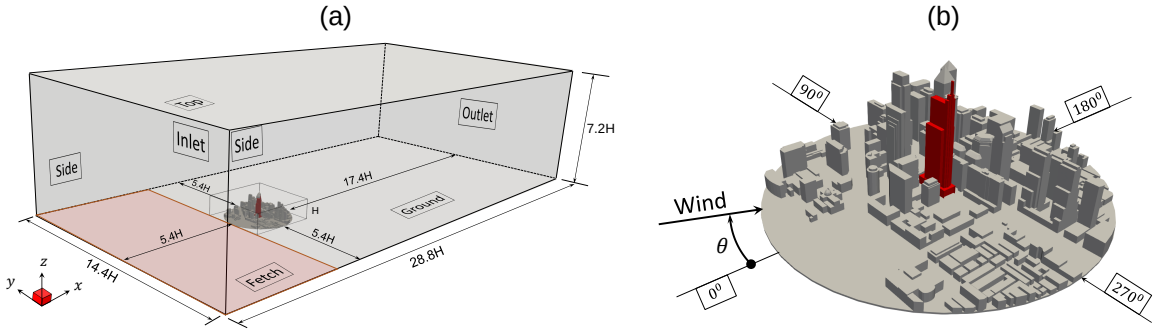


Figure 5.4: Computational domain and boundaries: (a) domain size relative to building height; (b) geometry of the numerical model

5.3.2 Geometric modeling

Before generating a computational grid, the first task is to prepare reasonably accurate geometric representations of the study building and the surrounding urban environment. A complete representation of all the geometric details beyond the resolution capability of the local mesh refinements is not practically relevant. Especially for the surrounding buildings, locations far

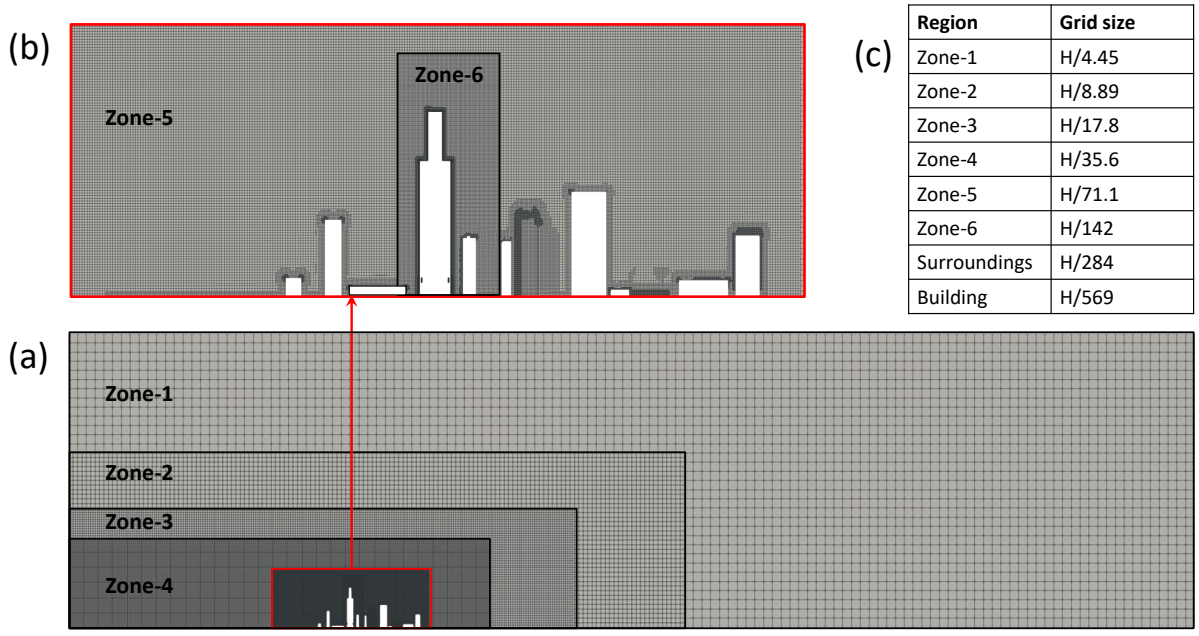
from the area of interest can be represented with fewer geometric details without affecting the expected flow field. Hence, the geometry of the building model and the surroundings are cleaned, in some cases moderately simplified, to generate an optimal computational grid with acceptable mesh quality. Finally, the CAD geometry of the whole model is made water-tight and exported to STL format for mesh generation. The geometry of the computational model containing the study building with the 500m radius proximity model is shown in Figure 5.4b.

5.3.3 Computational grid generation

Generating a computational grid for wind load evaluation of a tall building in a complex urban environment is very challenging compared to an isolated tall building. Normally, wind flows around an isolated building in itself (i.e., without the interference of the surrounding buildings) has a complex flow field marked by a wide range of spatial and temporal scales of turbulent eddies. When considering the effect of the surrounding buildings, the flow field becomes even more complex, with the nearby buildings significantly altering the local flow field and, consequently, the wind loads on the target building. For an LES-based study, this essentially necessitates the design of an optimal computational grid that resolves important flow features responsible for most of the wind loads on the building with a satisfactory level of accuracy. Thus, in addition to the local grid refinements near the target building, the surrounding buildings, specifically those building upstream of the target building, need to have additional regional refinements to adequately capture the aerodynamics. As a result, when designing the computational grid used for LES models, these challenges need to be taken into consideration.

Because of the complex geometry of the CFD model, the grid is generated as a non-uniform unstructured mesh using the `snappyHexMesh` tool of OpenFOAM software. Figure 5.5 shows a sampled computational grid generated for 0° wind direction simulation. The mesh is divided into six refinement zones. Progressively finer mesh is used as we get close to the study building. Before placing additional refinement zones, the computational domain is first provided with a uniform mesh having 32, 64, and 128 cells in x, y, and z directions, respectively (see Zone-1 in Figure 5.5). To sustain the inflow turbulence applied at the inlet up to the region of interest and provide further local mesh refinements, zones 2, 3, and 4 are employed. An additional mesh refinement region shown as Zone-5 in Figure 5.5 is provided to capture the effect of the upstream buildings that may impact the target building. In the vicinity of the study building, Zone-6 has finer mesh to better resolve body-generated turbulence, such as vortex shedding and turbulence at the wake. In addition, 10-cell thick surface refinements are provided in the wall-normal direction on the surface of all the buildings. Figure 5.5c shows the grid size adopted in each refinement region. Overall, the total number of cells for the wind load simulations for

each case is roughly 20 million.



Region	Grid size
Zone-1	H/4.45
Zone-2	H/8.89
Zone-3	H/17.8
Zone-4	H/35.6
Zone-5	H/71.1
Zone-6	H/142
Surroundings	H/284
Building	H/569

Figure 5.5: Sample view of the computational grid for 0° wind direction case: (a) *xz*-sectional view; (b) close-up view near the target region; (c) gird size used in each region expressed relative to the building height.

The topology of the mesh used close to the surface of the buildings is predominantly non-orthogonal, as we have complex urban geometry. A common issue with such type of grid is the mesh quality, which affects the speed and stability of the CFD solver. The criteria normally used to measure the quality of a computational grid for a finite volume-based solver is the degree of non-orthogonality. The orthogonality of a grid is measured by the angle between the normal vector of a cell surface and the line connecting the cell center of two neighboring cells (Ferziger et al., 2002). Non-orthogonal grids compromise the accuracy and efficiency of the solution algorithm. For instance, for transient solvers available in OpenFOAM, non-orthogonal grids usually require additional internal iterations at each time step. Whereas a highly non-orthogonal grid may cause nonphysical solutions and affect the stability of the solver. Therefore, in the current study, for all simulated cases, the maximum non-orthogonality of the grid is kept below 65°.

In addition to the computational grid shown in Figure 5.5, a slightly different mesh is prepared for an empty domain configuration using the same domain size and boundary conditions to study the incident wind profiles. For this domain, only refinement regions from Zone-1 up to Zone-5 are utilized (see Figure 5.5). The total number of cells for the empty domain simulation is approximately 8.74 million. Note that because the empty domain simulation does not have

the building model, the approaching wind profiles that are used for all the wind directions are extracted from a single run.

5.3.4 Inflow boundary condition

The critical step when conducting a transient simulation of ABL flows using LES is the generation of inflow turbulence that characterizes the approaching wind condition (Melaku and Bitsuamlak, 2021). This can be seen as an analogous technique used in the boundary layer wind tunnels, where turbulence generators such as spires and a strip-board at the inlet of the tunnel are used to inject turbulence to simulate the natural wind in ABL flows. For this study, the inflow turbulence is generated synthetically using the Divergence-free Spectral Representation (DFSR) method developed by Melaku and Bitsuamlak (2021). The method applies the spectral representation technique proposed by Shinozuka and Jan (1972) to artificially generated inflow turbulence as a multivariate stochastic stationary process. The procedure of generating inflow turbulence using the DFSR method involves first defining the cross-power spectral density(CPSD) matrix of the velocity field over points at the inlet. These points are the face center of the cells at the inlet of the domain. The CPSD matrix for each velocity component is calculated using the well-known von Karman model and a frequency-dependent coherency function of Davenport (1961b). Then, the velocity field for each component is generated independently using the inverse Fourier transform from the CPSD matrix. The procedure typically requires the definition of target wind profiles at the inlet. The required target profiles include mean velocity, turbulence intensity, and integral length scale profiles of the three components of the velocity. For the details of the DFSR method, the reader is advised to refer to Melaku and Bitsuamlak (2021).

For the current study, the target wind profiles fed to the DFSR method are based on the experimental data described in Section 5.2. Since the wind profiles reported in Section 5.2 are calculated from a hot-wire measurement, we have the statistics of the velocity in the streamwise direction only. However, the DFSR method normally requires profiles specifying all three velocity components. Therefore, turbulence intensity and integral length scale profiles in the lateral and vertical directions are taken from another set of wind tunnel measurements conducted in the same testing facility for suburban exposure conditions. These profiles are already reported in Chapter 2. One notable drawback of most synthetic inflow generation methods, including the DFSR method, is the generated turbulence does not reproduce the correct phase information of actual physical turbulence. This often leads to a noticeable deviation of the incident flow characteristics from the targets specified at the inlet. Thus, to account for the downstream evolution of the flow, it is common to adjust the wind profiles at the inlet until

the incident wind profiles match the target wind profiles (Lamberti et al., 2018). In the current study, using a simple iterative approach, the turbulence intensity profile specified at the inlet is re-scaled until we match the target value at the location of the study building.

Since we have the same exposure condition for all the wind directions, the inflow turbulence is generated and stored in a database for multiple uses. The generated velocity field is stored at a sampling rate of 400 Hz for a duration of 36 seconds. This combination of sampling rates and duration is chosen to get a duration comparable to values used in the experimental HFPI study. Finally, for the wind load simulations, for any solver time step, the inflow database is interpolated linearly between adjacent time steps and applied at the inlet.

5.3.5 Other boundary and initial conditions

Here we described the boundary conditions specified for velocity and pressure field on the boundaries of the computational domain shown in Figure 5.4. On the inlet face of the domain, the synthetic turbulence generated using the procedure in Section 5.3.4 is applied as a time-dependent velocity boundary condition. For the pressure field, a zero-gradient Neumann boundary condition is used. On the side and top faces of the domain, a slip boundary condition is used for velocity, while a zero-gradient Neumann boundary condition is adopted for the pressure. At the outlet, a mixed Neumann and Dirichlet boundary condition is applied for velocity, where a zero gradient is specified for the outflow condition with the inflow set to zero (i.e., no reverse flow was permitted). A constant static zero pressure is applied at the outlet for the pressure field. It is worth noting that this static pressure can be set to any desired reference value. However, when calculating the pressure coefficients on the building surface, the static reference pressure at the outlet must be deducted to get the gauge pressure.

The treatment of the ground surface roughness representing the target exposure condition is crucial for LES ABL flows. Particularly the effect of the upstream terrain needs to be modeled adequately for a realistic turbulent boundary layer to develop. As depicted in Figure 5.4a, the bottom surface of the computational domain is divided into two parts. The upstream part of the ground surface is separately labeled as “fetch” in Figure 5.4a. It stretches approximately $5.4H$ from the inlet of the domain up to the edge of the proximity model (or just upstream of the turn table in the experimental model). On this surface, a homogeneous rough-wall boundary condition is applied based on Schumann–Grötzbach model (Grötzbach, 1987; Schumann, 1975) effectively representing the effect of roughness blocks are typically placed upstream of the test section in the wind tunnel test. The numerical details of the Schumann-Grötzbach model used in the current study can be found in Melaku and Bitsuamlak (2021). On the remaining part of the bottom surface, a no-slip boundary condition is adopted.

Another important boundary condition that needs special treatment is the smooth wall boundary of the surface of the building and the proximity model. Resolving the entire boundary layer up to the viscous regime is computationally demanding on these surfaces. Thus, we used a boundary condition based on a smooth-wall function. This boundary condition works by providing a constraint on the turbulent viscosity using Spalding's formula (Spalding, 1961) that provides an analytical formulation for the velocity profile in the whole surface layer (including the viscous sub-layer, the buffer layer, and inertial sub-layer). For the current study, implementation of this boundary condition provided in OpenFOAM-v8 as `nutUSpaldingWallFunction` is adopted. One of the main advantages of this boundary condition is that the center of the wall adjacent cell does not always need to be in the logarithmic region, giving more freedom to choose mesh sizes independent of the wall model employed. Even though Spalding's law is strictly defined in the mean sense, it has also been frequently used in LES with instantaneous spatially filtered velocity (Cheng and Porté-Agel, 2013; Wang and Chen, 2020).

As initial conditions, in the whole computational domain, the velocity field is initialized with a stream-wise velocity of $10 m/s$, and the pressure field is set to zero static pressure.

5.3.6 Numerical setup

The Navier–Stokes equations governing the motion of wind were solved using the PIMPLE algorithm implemented as `pimpleFoam` in OpenFOAM-8. This solver operates by combining a transient PISO (Pressure Implicit with Splitting of Operator) algorithm with steady-state SIMPLE (Semi-Implicit Method for Pressure-Linked Equations) solver. The PIMPLE algorithm offers better numerical stability at higher Courant numbers (i.e., higher than 1) compared to the PISO solver. Thus, for the current study, a maximum Courant number is set to 7, and the solver time step is incremented dynamically to match the desired Courant number. This made it possible to save significant computational time while maintaining numerical stability. It should be noted, however, that this value of the Courant number is normally very high but occurs only in a few grid points near the building edges where the flow gets highly accelerated.

For the time integration of the temporal derivatives, we used a fully implicit second-order accurate backward differencing scheme. The backward differencing is preferred because of its low computational cost compared to relatively more accurate but expensive schemes like the Crank–Nicholson method (de Villiers, 2006). For spatial discretization, a second-order accurate central differencing scheme is used. The face center quantities were interpolated from their cell center values utilizing linear interpolation. Considering that the computational grid is highly orthogonal near the buildings' surface, additional non-orthogonal corrections are applied while solving the pressure equation to improve numerical accuracy and stability.

The wind load simulations were run for well over an equivalent 1-hour full-scale duration, assuming a time-scale of approximately 1 : 100. To avoid the initial transient period, the first few seconds were truncated. Effectively, only 36 s duration (in model scale) is used for the analysis. Surface pressure fluctuations, as well as integrated aerodynamic base loads, were monitored from the numerical simulations. The pressure data on the surface of the study building is recorded by placing probes on the LES model at the exact location of the HFPI experimental model. The base load time histories were recorded at a sampling rate of 400 Hz (identical to the wind tunnel study) at the center of the study building. The empty domain simulation used for examining the undisturbed incident wind profiles was conducted using the same numerical setup as the wind loads simulations. The velocity is monitored on a vertical line just in front of the study building at a 400 Hz sampling rate.

Finally, to run the simulations in parallel, the domain is distributed to 256 processors using Scotch decomposition (Pellegrini, 1994). The computations are performed using AMD Rome 7532 @ 2.40 GHz CPU processors. On average, each wind load simulation took 127 hours (approximately five days) to run on 256 processors. Using the same number of processors, the inflow turbulence generation with the DFSR method and the empty domain simulations took roughly 2 and 10 hours to finish, respectively.

5.4 Results and discussion

The key step towards establishing the use of CFD for wind loading applications involves detailed validation and verification of the numerical model. This section presents a systematic comparison of the results from the LES model against the experimental measurements. The validation is performed stage by stage for selected representative wind directions. First, the characteristics of the upcoming flow measured in an empty domain simulation are compared with the wind-tunnel profiles in Section 5.4.1. Then, the base aerodynamic loads monitored from LES are compared with the integrated overall forces calculated from the HFPI model in Section 5.4.3.

5.4.1 Incident flow characteristics

After generating the inflow turbulence using the DFSR method, an empty domain simulation (i.e., without the study building and the proximity model) is run to investigate the upcoming flow characteristics. For the LES model, the wind profiles were measured just in front of the study building, approximately a $7H$ distance away from the inlet. The LES wind profiles were monitored for about 36 seconds after the flow achieved an equilibrium boundary layer flow. We

examined the mean velocity and turbulence intensity profiles to evaluate how well the target wind-tunnel profiles are reproduced. Furthermore, the energy content of the incident flow is investigated by comparing the roof height velocity spectra against the targets.

In Figure 5.6, the incident wind profiles are compared against the target experimental measurements. Figure 5.6a depicts the comparison of the mean velocity profile. As seen in Figure 5.6 the incident mean velocity wind profile generally compares well with the experiment. However, close to the ground surface, approximately below $0.25H$, the shape of the velocity profile found from LES shows a slight deviation from the experiment. This difference could be due to the effect of the wall boundary on the ground surface downstream of the fetch region (see Figure 5.4). For the empty domain simulation, the region corresponding to the "turntable" in the wind tunnel model is modeled as a smooth wall boundary condition. It is expected that this boundary condition causes the mean flow to readjust such that the profile near the ground seems to be altered compared to the shape of the profile reported in the wind tunnel study. Nevertheless, when used with the proximity models for the final wind loads simulation, this is expected to have minimal effect on the wind loads since the wind profile just upstream of the "turntable" is more important. The mean velocity profile from LES generally has an absolute relative deviation of 2.7% averaged over $2H$ height. Next, the turbulence intensity profile in the stream-wise direction is compared in the same figure. For the most part of the profile, as seen in Figure 5.6a, the turbulence intensity profile from the LES also shows a good agreement with the experiment. The deviation from the experimental data averaged over $2H$ height is approximately 6.75%.

For both the experiment and LES data, a sample time series of the velocity at the building height is shown in Figure 5.7b. As can be seen from Figure 5.7, using the inflow turbulence generated by the DFSR method, it was possible to replicate the characteristics of velocity fluctuations observed in the experimental measurement. To inspect the energy content of the approaching flow from the LES, the spectrum of the stream-wise velocity is illustrated in Figure 5.6b. In the same figure, for comparison purposes, the target von Karman spectrum that was used to generate the synthetic turbulence is shown. As shown in Figure 5.6b, the velocity spectra from the LES follow the target spectra for a wide range of frequencies. The spectral plot from the LES shows a sharp drop-off at the high-frequency end, which is related to the grid resolution limit. However, this limitation will be compensated in the final wind load simulations, where the zonal and surface refinements near the study building permit the resolution of small-scale turbulence features with high-frequency content compared to that of Figure 5.6b.

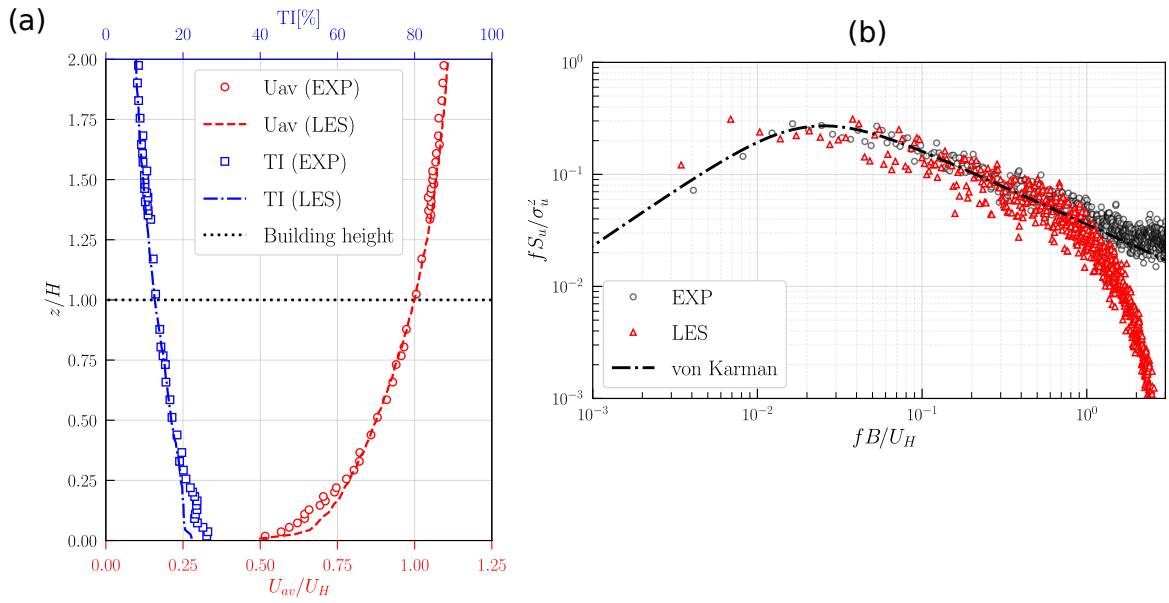


Figure 5.6: Comparison of the incident flow characteristics from LES with the experimental target measurements: (a) stream-wise mean velocity and turbulence intensity profiles; (b) reduced velocity spectra at the building height.

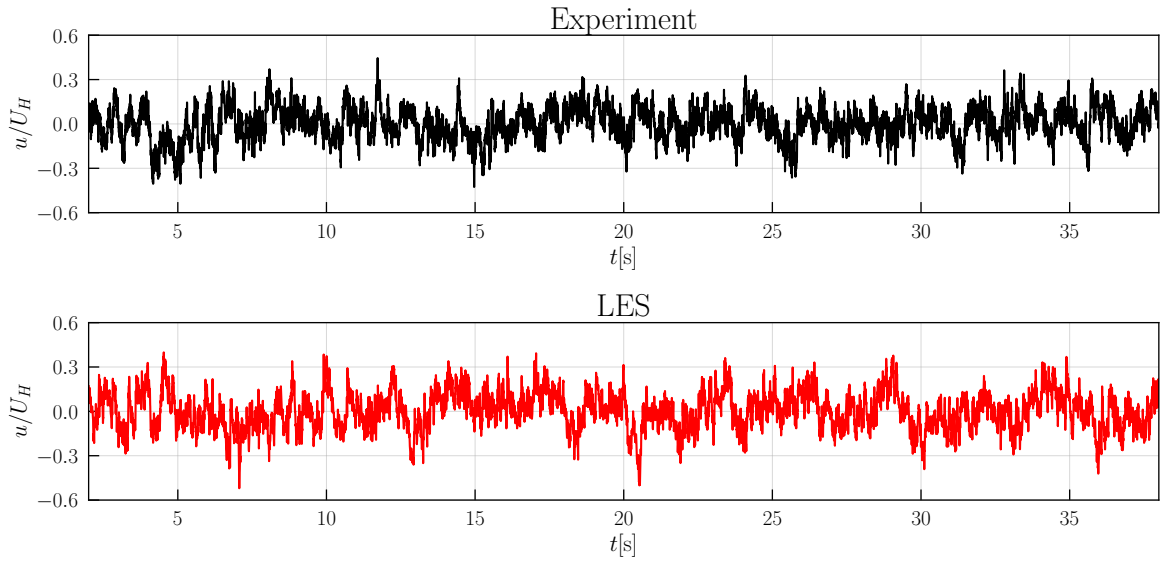


Figure 5.7: Time series of the velocity at the building height for the first 36s in model scale: (top) experimental; (bottom) LES.

5.4.2 Wind flow field

The wind flow around the study building is generally complex and is marked by different flow phenomena depending on the wind direction. Figure 5.8 shows the instantaneous velocity

contour and streamline paths for 0° wind direction. The turbulent nature of the upcoming flow and the turbulence generated by the study building and the surroundings can be clearly seen in the figure. To investigate the aerodynamic loading mechanisms for different wind directions, in Figure 5.9, the magnitude of instantaneous velocity contour for the longitudinal section is shown for all the wind angles considered (i.e., 0°, 90°, 180° and 270°). For instance, for 0° wind direction, the wider face of the study building is more or less directly exposed to the impingement from the upcoming flow (see Figure 5.9). Whereas, for 90° wind direction, the study building is in weak of a nearby tall building of almost equal height. In the case of 180° wind direction, the study building experiences sheltering effect from relatively short buildings upstream of it. On the other hand, the 270° case is the most exposed direction as there are no upstream buildings.

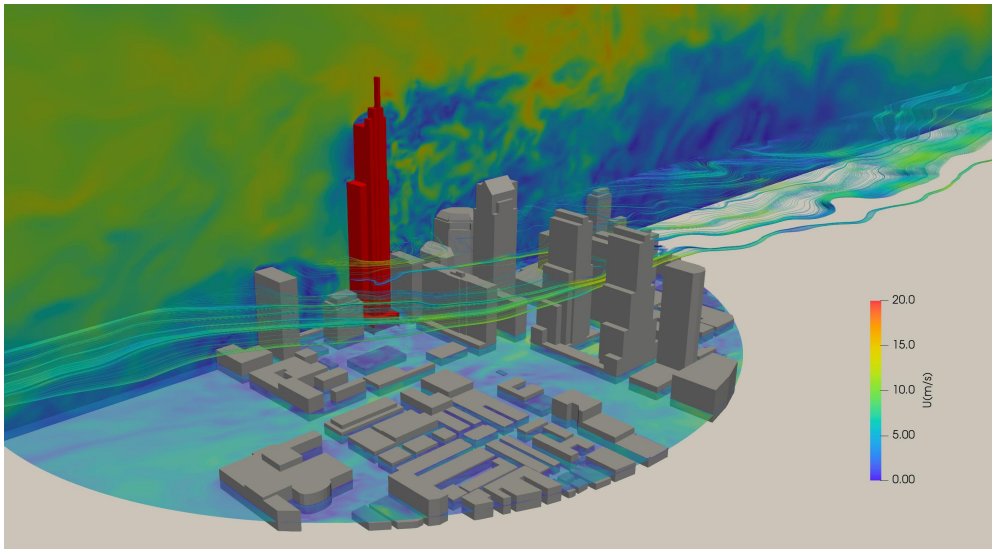


Figure 5.8: Isometric view of instantaneous wind velocity contour and streamline paths calculated from LES for 0° wind direction.

5.4.3 Comparison of global aerodynamic loads

The global aerodynamic loads monitored during LES runs were compared with the corresponding values from the experimental data obtained by pressure integration. The coordinate system used to define the base shears and moments can be referred from Figure 5.1b. The base shear

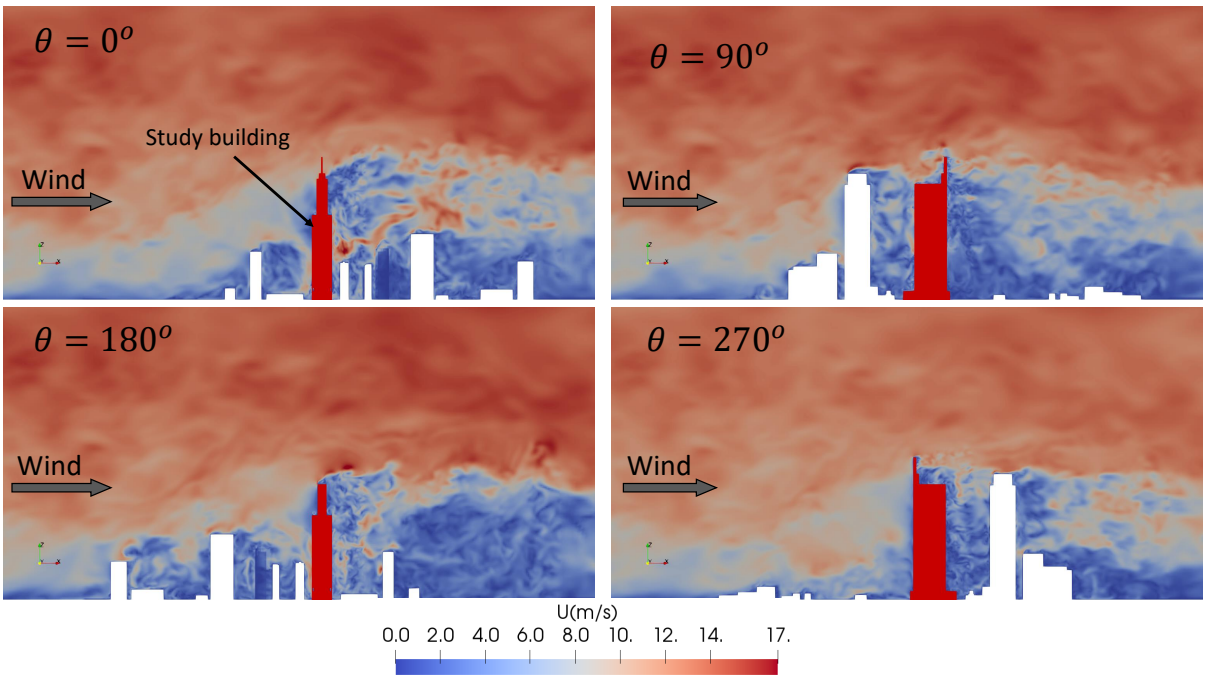


Figure 5.9: Magnitude of instantaneous velocity contour taken on xz-plane for all wind directions simulated.

and moment coefficients are defined as follows:

$$C_{F_x} = \frac{F_x}{\frac{1}{2}\rho \overline{U}_H^2 BH}, \qquad C_{F_y} = \frac{F_y}{\frac{1}{2}\rho \overline{U}_H^2 B_x H}, \tag{5.1a}$$

$$C_{M_x} = \frac{M_x}{\frac{1}{2}\rho \overline{U}_H^2 DH^2}, \qquad C_{M_y} = \frac{M_y}{\frac{1}{2}\rho \overline{U}_H^2 BH^2}, \qquad C_{M_T} = \frac{M_T}{\frac{1}{2}\rho \overline{U}_H^2 BDH}, \tag{5.1b}$$

where B, D, and H, respectively, are the study building's effective width, depth, and height, which are defined in Section 5.2. $\rho = 1.227$ kg/m^3 denotes the air density, and $\overline{U}_H$ is the roof-height mean wind speed. Here, C_{F_x} and C_{F_y} are the force coefficients in x and y directions, and C_{M_x}, C_{M_y} and C_{M_T} denote moment coefficients about x, y and torsional directions, respectively.

5.4.3.1 Validation metric

The difference between the LES and experiment for a given quantity Q (e.g., C_{F_x}, C_{M_x}) in a particular wind direction θ is measured using normalized absolute percentage deviation (E_θ) as

$$E_\theta = \frac{\left|Q_{\text{EXP}}^{(\theta)} - Q_{\text{LES}}^{(\theta)}\right|}{\max(|Q_{\text{EXP}}|)} \times 100, \tag{5.2}$$

where $Q^{(\theta)}_{\text{EXP}}$ and $Q^{(\theta)}_{\text{LES}}$ represent mean or RMS base loads from LES and experiment, respectively. In Eq. (5.2), for the denominator, we normalized the deviation by directionally enveloped experimental base load value. For this study, the expression $\max(|Q_{\text{EXP}}|)$ is simply calculated from experimental data by taking the maximum absolute base load quantity over all wind directions considered (i.e., 0°, 90°, 180°, and 270°). The advantage of such normalization is that it permits the error to be measured relative to the most unfavorable wind direction, which is often critical from a design perspective. Furthermore, it avoids infinite error that results when comparing base load quantities with values close to zero, e.g., mean cross-wind loads, if otherwise the normalization is done by the actual experimental value $Q^{(\theta)}_{\text{EXP}}$.

5.4.3.2 Base force coefficients

The force coefficients computed using Eq. (5.1) accordingly for both experimental and LES data are compared. Figure 5.10 shows a sample time history of C_{F_x} and C_{F_y} for 0° wind direction. As shown in Figure 5.10, a graphical qualitative inspection of the time histories of the force coefficients indicates that the results from LES are marked by a similar aerodynamics signature with the experimental measurements. The reduced force spectra for all wind directions considered are shown in Figure 5.11. The spectral plots from LES and experimental are generally in good agreement, indicating the capability of LES to capture complex flow around the study building and the resulting aerodynamic loads.

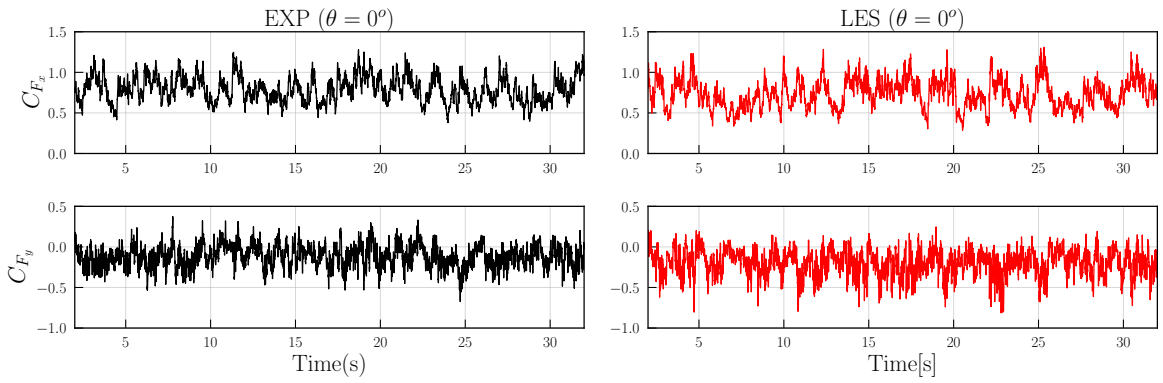


Figure 5.10: Time series of the force coefficients C_{F_x} and C_{F_y} for 0° wind direction: (left) Experiment; (right) LES

In Table 5.1, the mean and RMS force coefficients from LES are compared with the experimental values for all wind directions considered. The error for the mean and RMS force coefficients are shown in Figure 5.12. The deviations shown in the figure are calculated based on the expression in Eq. (5.2). To make the comparisons between LES and experiment fair,

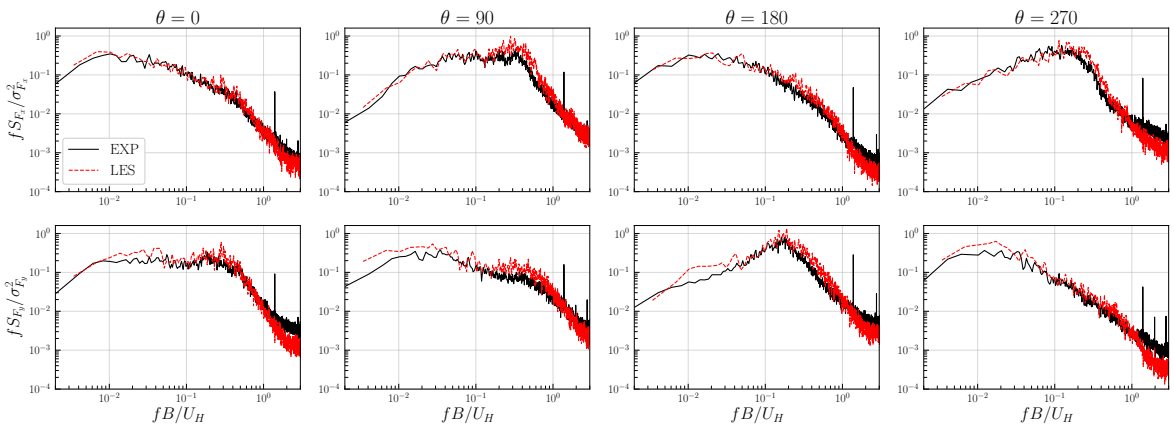


Figure 5.11: Reduced power spectral density of the base forces for all wind direction: (rows) components C_{F_x} and C_{F_y}; (columns) wind directions 0°, 90°, 180°, and 270°

Table 5.1: Comparison of mean and RMS force coefficients

	Mean				RMS			
	C_{F_x}		C_{F_y}		C_{F_x}		C_{F_y}	
θ	Exp.	LES	Exp.	LES	Exp.	LES	Exp.	LES
0°	0.799	0.725	-0.131	-0.197	0.153	0.169	0.136	0.152
90°	-0.060	-0.054	-0.282	-0.330	0.080	0.093	0.071	0.094
180°	-0.667	-0.574	-0.039	-0.166	0.141	0.150	0.076	0.089
270°	0.118	0.090	0.560	0.599	0.094	0.097	0.131	0.169

the same duration is used to compute the errors. For the mean value, the maximum deviation between LES and experiment is 22.7% and happens for C_{F_y} when the wind direction is 180° (see Figure 5.12a). Whereas, for the RMS quantities, it is 24.2% and happens for C_{F_y} with 270° wind direction (see Figure 5.12b). On average, the mean force coefficients are estimated with ±9.6% accuracy, while the RMS values have ±10.9% deviation.

5.4.3.3 Base moment coefficients

The base moment coefficients were also compared in the same manner as the force coefficients. In Figure 5.13, a sample time history of C_{M_x}, C_{M_y} and C_{M_T} are shown for 0° wind direction. The base moment spectra for each component and wind directions are depicted in Figure 5.14. The mean base moment coefficients from LES and the experiment are provided in Table 5.2. Similarly, Table 5.3 gives the RMS base moment coefficients. The deviation of LES from the experiment is presented in Figure 5.15. Overall, it was observed that for the mean base moments, the values from LES are ±8.7% far from the experiment when averaged over the

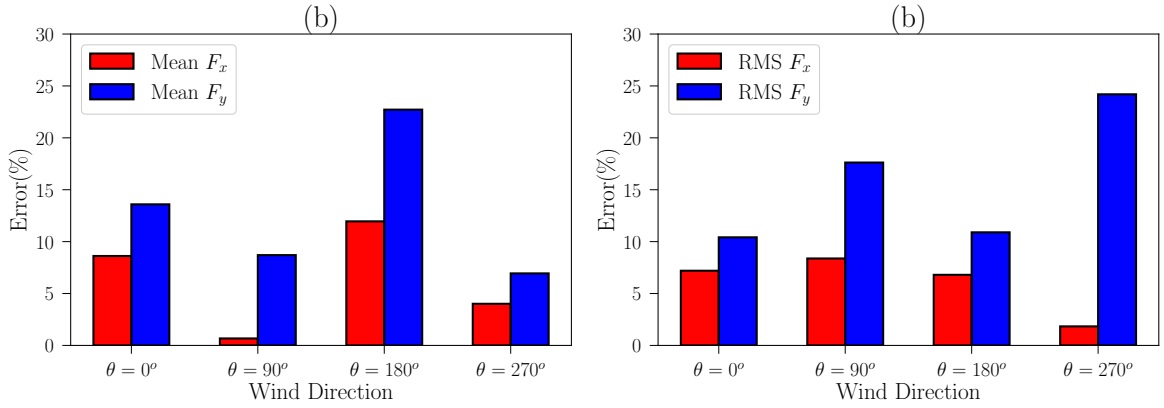


Figure 5.12: Deviation of force coefficients estimated using LES from the experimental values: (a) Mean ; (b) RMS

four wind directions considered. Similarly, for the RMS base moments, on average, the LES predictions have a ±9.0% difference.

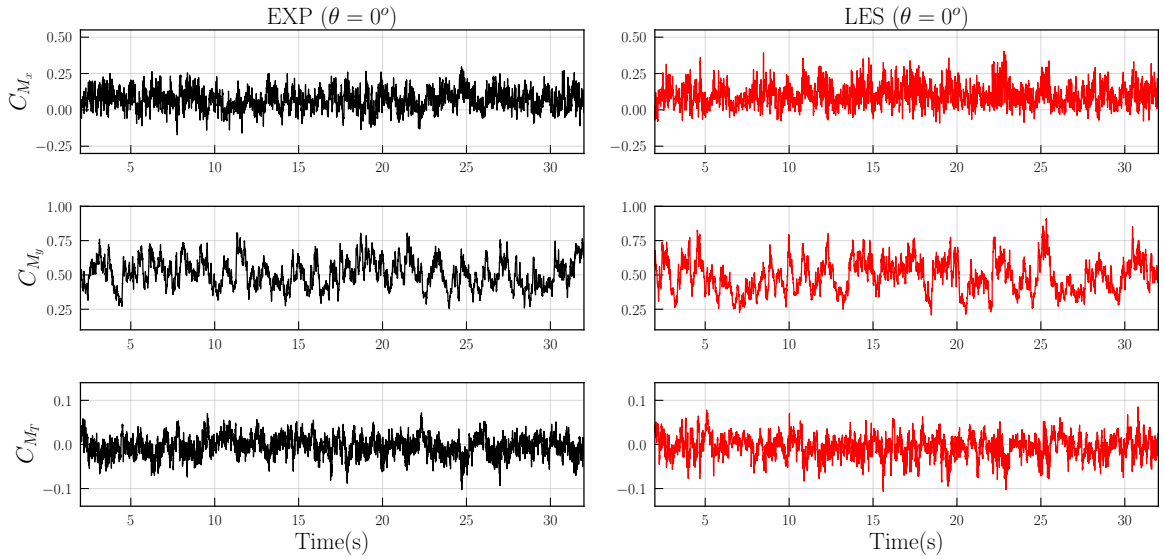


Figure 5.13: Time series of the base moment coefficients C_{M_x}, C_{M_y} and C_{M_T} for 0° wind direction: (left) Experiment; (right) LES

5.5 Summary and conclusion

Large-eddy simulation of wind flow around a tall building located in a city center was conducted to assess the accuracy of LES for predicting transient wind loads. The results from LES

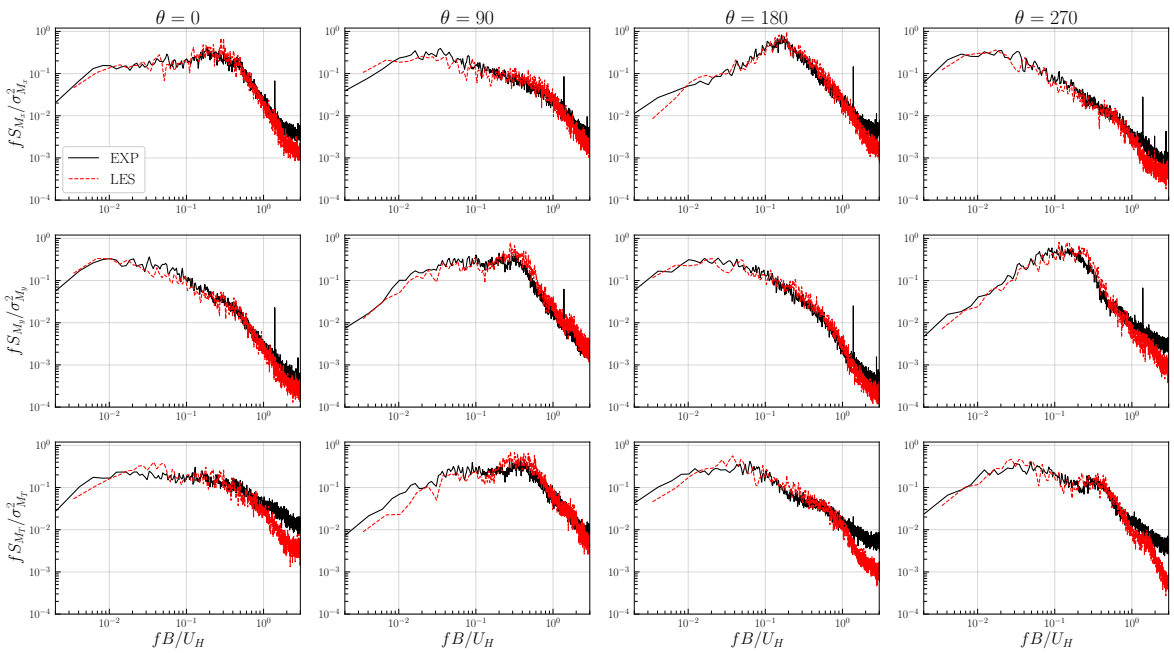


Figure 5.14: Reduced power spectral density of the base moments for all wind directions: (rows) components C_{M_x}, C_{M_y} and C_{M_T}; (columns) wind directions 0°, 90°, 180°, and 270°

Table 5.2: Comparison of mean base moment coefficients

	C_{M_x}		C_{M_y}		C_{M_T}	
θ	Exp.	LES	Exp.	LES	Exp.	LES
0°	0.075	0.100	0.518	0.483	-0.007	-0.005
90°	0.164	0.199	-0.036	-0.031	0.014	0.008
180°	0.014	0.079	-0.443	-0.386	0.051	0.039
270°	-0.324	-0.336	0.036	0.025	0.008	0.006

were validated thoroughly using experimental data measured in a boundary layer wind tunnel. The incident wind profiles obtained from empty domain LES runs were compared to the target wind tunnel profiles. It was shown that wind profiles were accurately modeled. Also, the spectral content of the incident flow was shown to be well resolved in the relevant frequency ranges. Then, the aerodynamic wind loads, including base shears and overturning moments, were compared with the experimental for representative wind directions. A comparison of the base load spectra indicated that, for the most part, the LES cases have adequately captured the energy content of the base loads observed in the experimental data. Next, statistical quantities of base loads, such as mean and RMS, are compared for different wind directions. Only in a few cases, it was observed that the maximum difference between the LES and experimental becomes 24%. However, when averaged over all the wind directions simulated, the mean

Table 5.3: Comparison of RMS base moment coefficients

	C_{M_x}		C_{M_y}		C_{M_T}	
θ	Exp.	LES	Exp.	LES	Exp.	LES
0°	0.066	0.068	0.099	0.113	0.022	0.023
90°	0.044	0.055	0.056	0.062	0.023	0.024
180°	0.047	0.054	0.094	0.102	0.029	0.031
270°	0.072	0.090	0.056	0.058	0.027	0.030

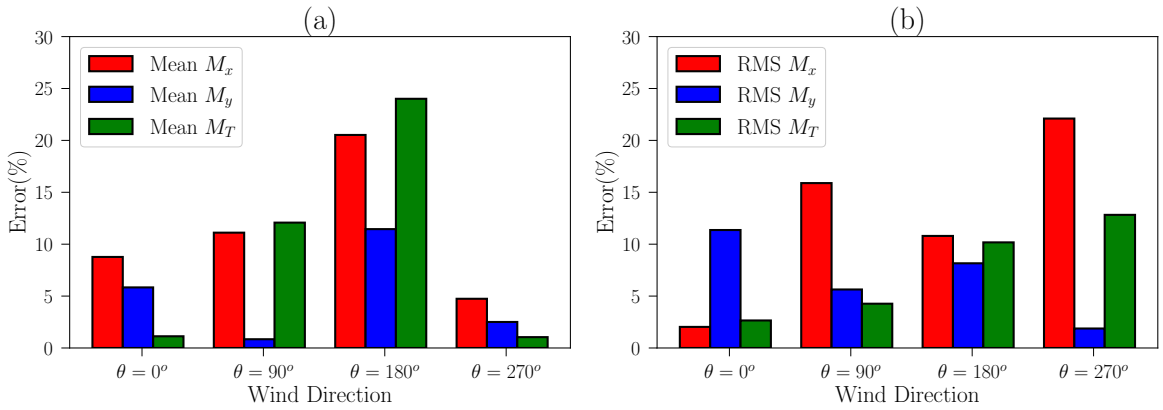


Figure 5.15: Deviation of base moment coefficients estimated using LES from the experimental values: (a) Mean ; (b) RMS

and RMS of base loads predicted using LES showed approximately ±10% deviation from the corresponding experimental measurement.

Overall, the agreement between the LES and the experiment is generally encouraging. The present work has successfully demonstrated the use of LES to predict dynamic loads on a tall building in realistic urban configurations. Furthermore, the numerical procedure demonstrated in this study is of practical importance for wind engineers, structural engineers, and architects — it provides them with a versatile tool at their disposal to incorporate wind-informed decisions early in the project life cycle.

Chapter 6

Fluid-structure interaction framework for computational aeroelastic modeling of tall buildings

6.1 Introduction

In recent years, the application of computational methods to multi-physics problems has expanded rapidly in many engineering disciplines benefiting from advances in new software technologies. Multi-physics simulation frameworks enable the integration of established techniques in different domains to address more complex and coupled engineering problems such as fluid-structure interaction (Farhat et al., 2006; Felippa and Park, 1980; Piperno and Farhat, 2001), soil-structure interaction (Romero et al., 2013; Yazdchi et al., 1999; Zhang et al., 1999), and thermomechanical simulation (Armero and Simo, 1992; Farhat et al., 1991), to name a few. The wind-induced excitation of flexible structures (e.g., tall buildings, chimneys, long-span bridges, and flexible roof systems), often described as *aeroelastic phenomenon* in wind engineering studies (Dowell et al., 2021; Irwin, 1982; Isymov, 1982; Vickery, 1990), represents a specific class of multi-physics (i.e., fluid-structure interaction) problems, which involves a coupled solution of a moving or deformable structure immersed in a turbulent flow. In the past, computational Fluid-Structure Interaction (FSI) modeling techniques have been successfully used in several applications, including, but not limited to, lightweight membrane structures (Gallinger et al., 2009; Wüchner et al., 2007), turbomachinery (Marshall and Imregun, 1996; Willcox et al., 1999) arterial blood flows (Bazilevs et al., 2008; Gerbeau et al., 2005) and parachute dynamics (Stein et al., 2000; Tezduyar et al., 2008). However, despite the success in these areas, there are only limited applications of FSI methods for modeling the aeroelas-

tic behavior of civil structures such as tall buildings (Braun and Awruch, 2009; Zhang et al., 2012). The development of accessible frameworks tailored to wind engineering applications holds tremendous potential for modeling aeroelastic structures in more efficient and versatile ways.

In practice, there are several methods to evaluate the wind-induced response of structures employing wind tunnel procedures. The most commonly used approach involves measuring the aerodynamic loads on a geometrically scaled rigid model of the structure and evaluating the responses incorporating the mechanical properties of the structure (Boggs, 1991; Irwin, 2009). This approach neglects the effect of motion-induced forces (aeroelastic feedback). Currently, the most accurate experimental technique to model the wind-induced response of dynamically sensitive structures is to use aeroelastic models that appropriately scale the structure's mechanical properties in addition to its geometry (Isymov, 1982; Vickery, 1990). However, aeroelastic modes are often expensive and time-consuming to fabricate. Furthermore, the manufacturing challenges resulting from their intricacy might lead to violation of similitude conditions (Isymov, 1982). On the other hand, in recent years, alternative to experimental techniques, Computational Wind Engineering (CWE) tools, particularly Computational Fluid Dynamics (CFD), have gained favor for wind load evaluation as it alleviates most of the limitations of experimental techniques (Blocken, 2014b; Dagnew and Bitsuamlak, 2014; Stathopoulos, 1997). Several CFD-based wind load evaluation studies have been conducted (Elshaer et al., 2016; Huang et al., 2010, 2007; Ricci et al., 2018) in recent years, albeit most of these studies assume the building is rigid and simulate only the aerodynamic loads. However, for dynamically sensitive structures, accurate response estimation requires the fluid-structure interaction phenomenon to be aptly simulated.

The numerical procedures to solve FSI problems can essentially be classified into two categories: the monolithic and the partitioned approaches (Hou et al., 2012). In the monolithic approaches (Hübner et al., 2004; Ryzhakov et al., 2010), the fluid and structural dynamics equations are casted in a single system of linear equations, and the solution is advanced by a unified algorithm. The obvious benefit of these approaches is that no load/deformation transfer is needed as it is implicitly handed. In addition, monolithic approaches can achieve more accuracy and robustness, making them suitable for strongly coupled problems (Yamada et al., 2016). However, from a software development and maintenance perspective, such codes require high expertise as they are very specialized. Conversely, in partitioned methods (Felippa and Park, 1980; Felippa et al., 2001), the fluid and structural dynamics equations are solved separately using their respective mesh and solution algorithm. At the fluid-structure interface, explicit communication is necessary for transferring load and deformation. This makes the partitioned method modular, and one can use off-the-shelf codes separately developed for the

fluid and structure parts and couple them. Based on the literature, the partitioned approach is by far the most extensively used technique as compared to the monolithic approaches.

Nowadays, several software packages exist for solving general FSI problems. The most commonly used commercial software packages, such as STAR-CCM+, ANSYS, and ADINA, have already incorporated FSI functionalities. However, these software packages are generally black-box, making them less suited for further improvement or to incorporate new techniques tailored to specific applications. As a result, in the research community, we now see more momentum towards open-source packages like OpenFOAM. Previously, other researchers have developed FSI frameworks for different applications by coupling structural solvers with OpenFOAM's transient solvers (Cesur et al., 2014; Hewitt et al., 2019; Jasak and Tuković, 2010). Nevertheless, in these implementations, the structure is represented as a continuum medium. This makes it challenging to represent a structure like a tall building made from discrete structural elements such as columns, beams, and slabs in a computationally efficient and convenient manner using these frameworks. Considering that wind-structure interaction is of practical significance for the wind-resistant design of civil structures, open-source FSI frameworks specifically suited for this purpose are of paramount importance to research and practice in the wind engineering discipline.

This paper proposes a high-fidelity FSI framework for computational aeroelastic modeling of slender structures. The framework is developed by coupling OpenFOAM with a modal structural solver implemented in C++ using the Object-Oriented Programming (OOP) paradigm. The proposed FSI framework employs a partitioned approach, where fluid and structural solvers exchange deformation and load data every time step. The fluid subsystem is solved in an Arbitrary Lagrangian-Eulerian (ALE) frame of reference on a moving mesh. The deformation of the mesh around the structure is computed using Spherical Linear Interpolation (SLERP), which follows the modal basis functions of the structure in three dimensions. For coupling the fluid and structural subsystems, two algorithms that represent *weak* and *strong* coupling, namely Conventional Serial Staggered (CSS) and fixed point iteration algorithms, respectively, were implemented. In order to reduce the communication overhead between the fluid and structural subsystems, the structural solver is directly integrated into the CFD solver architecture. The proposed framework is named `windFSI`, considering its potential application to the target computational *wind* engineering community. Having designed the framework's architecture in OOP paradigm, the implementation of the code is versatile and can easily be extended to simulate various wind-structure problems involving structures with complex configurations and non-linear material properties.

The paper is organized into five sections. Section 6.2 describes the formulation of the FSI problem for aeroelastic application. Software implementation of the proposed framework

using OpenFOAM and modal structural solver is described in Section 6.3. In Section 6.4, the proposed FSI framework is demonstrated using two numerical examples, namely, the vortex-induced oscillation of a circular cylinder and the wind-induced vibration of a tall building immersed in turbulent atmospheric boundary conditions. Finally, Section 6.5 provides the summary and future outlook for the current work.

6.2 Formulations of the fluid-structure interaction framework

This section presents the mathematical foundations of the proposed FSI algorithm for modeling the aeroelastic behavior of flexible structures. The field equations governing the flow of the wind and the equations of motion representing the deformation of the structure are described. Most of all, the coupling algorithm synchronizing the fluid (wind) and structure domains at each time step is presented.

Besides the fundamental difference in the governing physical laws, the fluid and structure domains are commonly simulated in different coordinate systems (Hughes et al., 1981). The fluid flow is best studied in Eulerian co-coordinate systems, while the structural deformations can be conveniently represented in the Lagrangian frame of reference. However, in the fluid-structure interaction problems, the motion of the structure causes the boundaries of the fluid to move, hence requiring handling of the deforming mesh for the fluid domain (Jasak, 2009). Thus, for the current study, we adopted Arbitrary Lagrangian-Eulerian (ALE) formulation widely used in the FSI studies with dynamic mesh capability. ALE permits the solution of the Navier-Stokes equations in a domain of arbitrarily changing shape (Demirdžić and Perić, 1990). This makes the fluid-structure interaction problem rather a three-field coupling problem that involves three computational entities: fluid flow, structural motion, and mesh deformation (Felippa and Park, 1980; Piperno and Farhat, 2001; Piperno et al., 1995).

6.2.1 Governing equations

The fluid and structure regions are non-overlapping domains with their respective boundaries. Figure 6.1 shows the schematic representation of the fluid and structure domain denoted by $\Omega_{\mathcal{F}}$ and $\Omega_{\mathcal{S}}$, respectively. The two domains share a common interface represented by $\Gamma_{\mathcal{F}/\mathcal{S}}$. Unlike other boundaries of the fluid and the structure domains, the fluid-structure interface is where the information between the two domains is exchanged and needs to satisfy additional compatibility and equilibrium requirements.

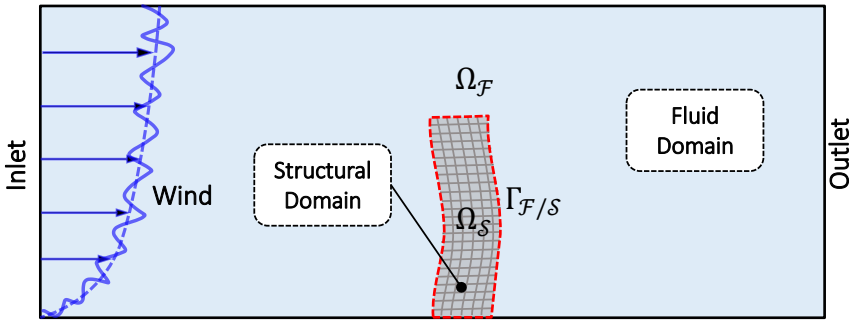


Figure 6.1: Schematic representation of the fluid and structure domains

6.2.1.1 Fluid domain

In the fluid domain $\Omega_{\mathcal{F}}$, the flow of wind is governed by the well know Navier–Stokes equation. Considering that the displacements and rotations of the structure will be significant, the governing equations are re-casted in the ALE frame of reference over a moving grid. In ALE framework, for an incompressible flow with a fluid velocity $\mathbf{u}$ and the pressure p, the continuity and Navier-Stokes equations take the form:

$$\nabla \cdot \mathbf{u} = 0, \tag{6.1}$$

$$\frac{\partial \mathbf{u}}{\partial t} + [(\mathbf{u} - \mathbf{u}_m) \cdot \nabla]\mathbf{u} = -\frac{1}{\rho}\nabla p + \nu \nabla^2 \mathbf{u}, \tag{6.2}$$

where $\mathbf{u}_m$ is the mesh velocity vector; ν and ρ refer to the kinematic viscosity and density of the fluid. Note that the ALE formulation of Navier–Stokes in Eq. (6.2) is similar to the standard form used in the Eulerian frame of reference. The only difference is the advection term, which is modified simply by substituting a relative convective velocity $(\mathbf{u} - \mathbf{u}_m)$ into the equation. However, the viscous term as well as the continuity equation in Eq. (6.1) are not altered as both involve only spatial derivatives (Carmo et al., 2011).

6.2.1.2 Structural domain

The structure is described in the Lagrangian frame of reference. For aeroelastic modeling of tall buildings, the structure is best represented as a discrete system built from individual structural elements (i.e., beams, columns, slabs, and shear walls). However, for FSI modeling, we do not need the whole FE model of the building. The structural deformations can be effectively described only using generalized properties. Hence, the deformation of the building is computed in the time domain using these properties and integrated instantaneous aerodynamic

loads over the building surface. The generalized equations of motion for the first m modes can be written as

$$[M]\{\ddot{q}(t)\} + [C]\{\dot{q}(t)\} + [K]\{q(t)\} = \{F(t)\} \tag{6.3}$$

$$\{d(x, y, z, t)\} = \sum_{i=1}^{m} q_i(t)\{\phi_i(x, y, z)\} \tag{6.4}$$

where $\{d(x, y, z, t)\}$ represents the structural displacement of the building at any location and time. Here, $\{q(t)\}$ and $\{F(t)\}$ respectively denote the generalized displacement and force vectors. The matrices $[M]$, $[C]$, and $[K]$ are the generalized mass, damping, and stiffness matrices, respectively. Whereas, $\phi_i(i = 1, 2, \ldots, m)$ is the modal basis function for $i-$th mode. The modal properties of the structure can be extracted from a free vibration analysis of the detailed FE model of the building.

It is worth mentioning that for linear structural analysis, the generalized mass, damping, and stiffness of the structure do not change with time and the deformation of the structure. However, if one wants to incorporate a nonlinear effect, only the formulation in Eq. (6.3) will change while the overall procedure remains the same. For example, if the structure undergoes significant deformation, the second-order effects can be accounted by formulating a geometric stiffness correction matrix (Wilson and Habibullah, 1987).

6.2.1.3 Dynamic mesh

The fluid mesh around the structure must deform following the motion of the structure. The deformed grid needs to fulfill compatibility constraints while preserving optimal mesh quality to reduce discretization error (Jasak and Tukovic, 2006). Most importantly, for the ALE system, the mesh motion must satisfy the Geometric Conservation Law (GCL) stated as (Demirdžić and Perić, 1988; Thomas and Lombard, 1979):

$$\frac{\partial V^{ce}}{\partial t} + \nabla \cdot \mathbf{u}_m = 0, \tag{6.5}$$

where V^{ce} is the cell volume. The GCL requires that the change in cell volume in any time interval Δt must be equivalent to the volume swept by all the faces of the cell during that time interval (Slone et al., 2002).

To maintain a fairly regular grid, the mesh deformation around the structure needs to be computed such that the displacements at the moving boundaries are smoothly distributed over the surrounding grid of the fluid domain (Chen and Christensen, 2018). For this study, we used a computationally efficient mesh morphing technique implemented in OpenFOAM that preserves grid quality. The fluid mesh is divided into three regions. Near the structure, the mesh

undergoes only rigid body motion following the deformation of the structure, and this region is controlled by specifying inner morphing distance d_i. The mesh far from the structure does not move at all, and this region is controlled by providing outer morphing distance d_o. The third part is the intermediate region; in this zone, the mesh deformation is smoothly interpolated between the other two regions. In this region, the mesh deformation is calculated using Spherical Linear Interpolation (SLERP). For a smooth transition, the displacement of the mesh points in this region is scaled by the function:

$$s(d) = \frac{1}{2}\left[1 - \cos\left(\pi \frac{d_o - d}{d_o - d_i}\right)\right], \tag{6.6}$$

where d is the shortest Euclidean distance of the mesh point from the structure's surface. The plot in Figure 6.2 shows the non-dimensional scaling factor $s(d)$ over the three regions. Compared to other mesh deformation techniques, e.g., Laplacian and pseudo-solid mesh smoothing techniques (Jasak and Tuković, 2010), SLERP is very efficient since it does not involve solving any differential equation.

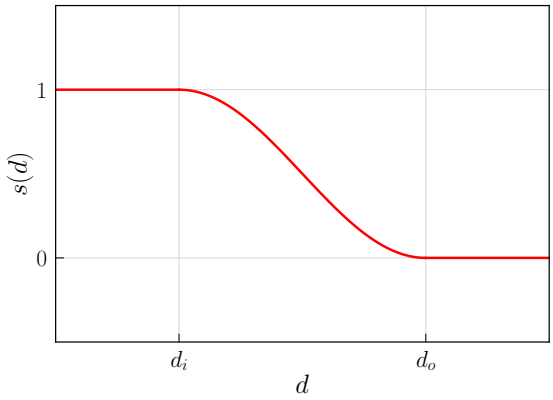


Figure 6.2: Mesh displacement scaling factor $s(d)$ over different morphing regions.

6.2.2 Compatibility requirements

On the fluid/structure interface $\Gamma_{\mathcal{F}/\mathcal{S}}$, kinematic and equilibrium conditions must be satisfied. The structural displacement needs to match the mesh displacement at the solid boundary. Similarly, on $\Gamma_{\mathcal{F}/\mathcal{S}}$, the velocity of the structure and fluid needs to be the same. Actually, this is a kinematic requirement which is the consequence of a no-slip boundary condition applied on the walls of the structure. Thus, the above compatibility requirements can be expressed

mathematically as:

$$\mathbf{u}_m = \frac{\mathrm{d}}{\mathrm{d}t}\mathbf{d} \qquad \text{on} \quad \Gamma_{\mathcal{F}/\mathcal{S}} \tag{6.7a}$$

$$\mathbf{x}_m = \mathbf{d} \qquad \text{on} \quad \Gamma_{\mathcal{F}/\mathcal{S}} \tag{6.7b}$$

In which $\mathbf{d}$ represents a Lagrangian displacement vector of the structure. On the other hand, if the structure was modeled as a continuum media, in addition to the kinematic requirements in Eq. (6.7), the stresses at the fluid/structure interface must be in equilibrium. However, for the current study, since the structure is modeled employing a simple lumped-mass approach, only area-integrated loads from the fluid are transferred to the structure.

6.2.3 Numerical schemes

Before discussing the FSI coupling algorithm, the numerical methods used for solving the governing fluid flow and structural dynamics equations are presented briefly.

6.2.3.1 Fluid solver

For the fluid solver, we used OpenFOAM-8.0, an open-source finite volume-based CFD toolbox. The incompressible Navier-Stokes equations are solved in a segregated manner using the PIMPLE algorithm, which essentially works by running multiple PISO loops per time step. The PISO (Pressure-Implicit with Splitting of Operators) algorithm is a pressure-velocity coupling algorithm first developed by Issa (1986) for solving Navier-Stokes equations. Thus, here we present a brief description of the PISO algorithm using the discretized form of the Navier-Stokes equations.

Writing the momentum equation in Eq. (6.2) in a semi-discretized form, we have:

$$a_P\mathbf{u}_P + \sum_f a_N\mathbf{u}_N = \mathbf{b} - \nabla p. \tag{6.8}$$

In which P denotes the index of an arbitrary cell while index N represents the neighboring cells. The coefficient a_P and a_N represent diagonal and off-diagonal elements of the assembled system of equations, respectively. The summation $\sum_f$ represents contributions from all bounding faces, and $\mathbf{b}$ contains the source terms. The pressure gradient term in Eq. (6.8) is left to be discretized in the final velocity-coupling stage. Following the notation in Jasak (1996), the momentum equation can be rewritten as:

$$a_P\mathbf{u}_P = \mathbf{H}(\mathbf{u}) - \nabla p, \tag{6.9}$$

where the term $\mathbf{H}(\mathbf{u})$ holds the contribution of neighboring matrix coefficients and all the source terms.

In the same manner, discretizing the continuity equation and applying Gauss's theorem, Eq. (6.1) becomes

$$\int_V (\nabla \cdot \mathbf{u}) \mathrm{d}V = \int_{\partial V} \mathbf{u} \cdot \mathrm{d}\mathbf{s} \simeq \sum_f \mathbf{u}_f \cdot \mathbf{s}_f = 0. \tag{6.10}$$

Here, $\mathbf{s}_f$ is the outward pointing surface area vector, and $\mathbf{u}_f$ is the face-center velocity. The pressure equation can be formed substituting $\mathbf{u}_P$ from Eq. (6.9) into Eq. (6.10). However, this normally requires obtaining face-center velocities from cell-centered values by some form of interpolation. Usually, using a simple linear interpolation with refined mesh near high-velocity gradient regions has shown to work reasonably well (de Villiers, 2006). This approach is equivalent to using a central differencing scheme and renders a second-order accurate approximation (Ferziger and Peric, 2012). Thus, after the interpolation, the final pressure equation reads as

$$\sum_f \mathbf{s}_f \cdot \left(\frac{1}{a_P} \nabla p \right)_f = \sum_f \mathbf{s}_f \cdot \left(\frac{1}{a_P} \mathbf{H}(\mathbf{u}) \right)_f \tag{6.11}$$

The left-hand side of Eq. (6.11) can be understood as the Laplacian of the pressure field.

Equations (6.9) and (6.11) represent the discretized Navier–Stokes equations. From these equations, it is evident that pressure and velocity fields are coupled. Although there are numerical algorithms (Caretto et al., 1972; Van Leer, 1979) that can solve the complete Navier–Stokes equations simultaneously, they are not often economical and incur high memory cost (de Villiers, 2006). An economical way of solving Eqs. (6.9) and (6.11) coupled is to use iterative methods such as the PISO algorithm. The PISO algorithm comprised one predictor and, at most, two corrector steps. In the predictor step, the momentum equation (6.8) is used to obtain an initial guess of the velocity field using the pressure gradient from a previous time step. Then, in the corrector steps, the pressure field is solved from Eq. (6.11) multiple times, followed by an explicit update to the velocity using Eq. (6.9) (Greenshields and Weller, 2022b).

For FSI application, since the simulation involves back-and-forth iteration between the fluid and structure domains, several PISO iterations per time step might be required until both the fluid and structural solutions converge. This makes the PIMPLE algorithm the solver of choice for this type of application. Furthermore, the PIMPLE procedure generally has better numerical stability and accuracy.

6.2.3.2 Structural solver

To compute the structural responses, the current study primarily uses Newmark's time integration method. For comparison purposes, however, in addition to Newmark's method, the 4-th order Runge–Kutta method, which has superior accuracy and stability (with a modest increase in computational cost), is also implemented.

The Newmark's method is a second-order implicit technique developed by Newmark (1959) for numerical integration of the dynamic structural equation of motion. Here, a brief description of the method for solving the modal equation of motions in Eq. (6.4) is presented. For the current study, the structural responses are solved in a generalized coordinate system. Based on Newmark's time integration method, the generalized displacement and velocity for the current time step t_{n+1} are:

$$q_{n+1} = q_n + (\Delta t)\dot{q}_n + \left[(0.5-\beta)(\Delta t)^2\right]\ddot{q}_n + \left[\beta(\Delta t)^2\right]\ddot{q}_{n+1} \tag{6.12a}$$

$$\dot{q}_{n+1} = \dot{q}_n + \left[(1-\gamma)\Delta t\right]\ddot{q}_n + (\gamma\Delta t)\ddot{q}_{n+1} \tag{6.12b}$$

with time step Δt and parameters β and γ control the stability and accuracy of the method. As seen in Eq. (6.12), the method is implicit, requiring iteration within a time step. However, for linear structure, by setting specific values of β and γ, responses can be calculated without iteration. The choice of β and γ determines the variation of acceleration within a time step (Chopra, 2007). Choosing $\beta = 1/4$ and $\gamma = 1/2$ results in average-acceleration approximation. For linear acceleration (trapezoidal) approximation, a value of $\beta = 1/6$ and $\gamma = 1/2$ are used. Once the modal displacement (q_{n+1}) and velocity ($\dot{q}_{n+1}$) of the structure are calculated from Eq. (6.12), the acceleration ($\ddot{q}_{n+1}$) is solved applying equilibrium condition. A step-by-step procedure for Newmark's method employing average and linear acceleration approximations can be found in Chopra (2007).

6.2.4 FSI coupling algorithm

To solve the FSI problem, the current study employs a partitioned approach. The primary reason for using this approach is that it preserves software modularity, which makes it simple to implement using existing CFD and structural solvers. Two coupling algorithms belonging to *weak* and *strong* classes of coupling algorithms were investigated. These algorithms are the Conventional Serial Staggered (CSS) and Fixed-Point Iteration (FPI) methods. Next, each coupling algorithm is briefly discussed with its implementation detail.

6.2.4.1 Conventional serial staggered algorithm

The Conventional Serial Staggered (CSS) is the most basic and popular coupling algorithm commonly used for partitioned approach (Piperno and Farhat, 2001; Piperno et al., 1995). The method is depicted graphically in Figure 6.3. Assuming the fluid and structure have the same time step ($\Delta t_F = \Delta t_S$), the CSS procedure goes as follows: (1) transfer the structural deformation ($\mathbf{d}_n, \dot{\mathbf{d}}_n$) to the fluid mesh; (2) advance the fluid system and compute the flow field ($p_{n+1}, \mathbf{u}_{n+1}$); (3) transfer the flow-induced loads to structure; (4) advance the structural subsystem to the next time step. Then, the same procedure is repeated for the next time step (see Figure 6.3). Note that when the fluid and structure have unequal time steps, the computational efficiency of the CSS method can be improved by applying sub-cycling for the fluid computation with a factor equivalent to $n_{\Delta t} = \Delta t_S / \Delta t_F$.

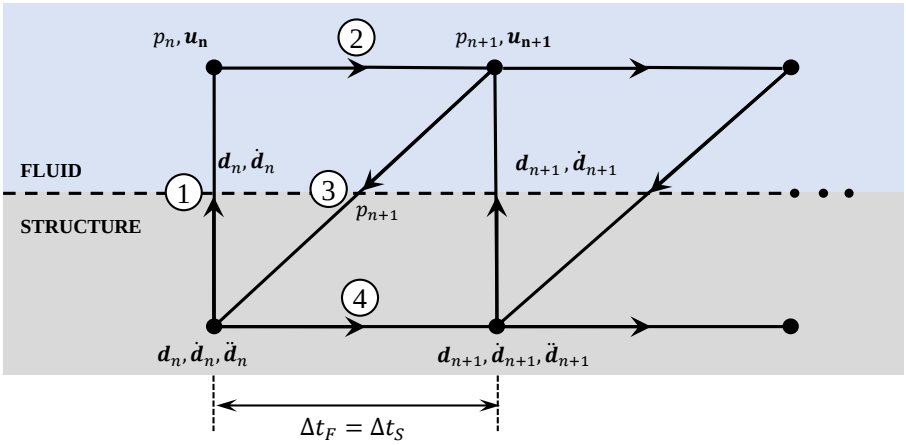


Figure 6.3: Conventional Serial Staggered(CSS) fluid-stricture coupling algorithm.

Although the CSS method is simple to implement and computationally inexpensive, the method has some drawbacks. For example, it does not guarantee convergence and unconditional numerical stability of the fluid-structure interaction (Farhat and Lesoinne, 2000). Furthermore, regardless of the accuracy of the numerical methods used for solving the fluid and structure domains, the CSS procedure itself is well-known to be only first-order time-accurate (Farhat et al., 2006). One of the primary reasons the CSS method lacks accuracy and stability is that it does not check for convergence before continuing to the next time step, i.e., if the predicted structural displacements and flow field stop changing with additional iterations for the same time step. For this reason, the CSS algorithm is regarded as a *weak* or *loose* coupling algorithm. These limitations especially become pronounced for FSI problems when the densities of the fluid and the structure are of the same order (Deparis et al., 2003) compared to aeroelastic applications where the fluid and structure have significant density differences.

6.2.4.2 Fixed-point iteration coupling algorithm

Figure 6.4 shows the flow chart of the fixed-point iteration (FPI) based FSI coupling algorithm implemented. Unlike the CSS method, which solves the fluid and structure domains once for each time step, in the fixed-point FSI algorithm, both domains are solved multiple times before marching to the next time step. Essentially, the FPI-based coupling algorithm treats the FSI problem as a root-finding problem. As such, solving the mesh, fluid, and structural subsystems is considered as evaluating abstract non-linear algebraic expressions denoted by $\mathcal{M}(\mathbf{d}, \dot{\mathbf{d}})$, $\mathcal{F}(\mathbf{x}, \dot{\mathbf{x}})$ and $\mathcal{S}(\mathbf{u}, p)$, respectively. For a given sub-iteration $k \geq 1$, omitting the time index, the three subsystems are interrelated as (Deparis et al., 2003):

$$\begin{aligned} (\mathbf{x}_{k+1}, \dot{\mathbf{x}}_{k+1}) &= \mathcal{M}(\mathbf{d}_k, \dot{\mathbf{d}}_k), \\ (\mathbf{u}_{k+1}, p_{k+1}) &= \mathcal{F}(\mathbf{x}_{k+1}, \dot{\mathbf{x}}_{k+1}), \\ (\mathbf{d}_{k+1}, \dot{\mathbf{d}}_{k+1}) &= \mathcal{S}(\mathbf{u}_{k+1}, p_{k+1}), \end{aligned} \tag{6.13}$$

where $(\mathbf{x}_0, \dot{\mathbf{x}}_0)$, $(\mathbf{u}_0, p_0)$ and $(\mathbf{d}_0, \dot{\mathbf{d}}_0)$, respectively represent mesh deformation, flow fields, and structural responses variables from the previous time step. For each sub-iteration, the fluid domain, therefore, is solved as $\mathcal{F}(\mathbf{x}_{k+1}, \dot{\mathbf{x}}_{k+1})$ using the PIMPLE algorithm. The sub-iteration is deemed converged when the difference between two successive iterations is within a certain error of tolerance. As a result, the method is globally implicit and usually unconditionally stable (Küttler and Wall, 2008). In the literature, such types of coupling algorithms are often referred to as *strong* or *tight* coupling methods.

To improve the convergence and stability of the FPI algorithm, the structural deformation fed into the mesh solver is often relaxed by some factor (Küttler and Wall, 2008). For the current study, we adopted a constant factor ω to relax the intermediate structural displacement and velocity as:

$$\begin{aligned} \mathbf{d}_{k+1} &= \mathbf{d}_k + \omega(\mathbf{d}^*_{k+1} - \mathbf{d}_k), \\ \dot{\mathbf{d}}_{k+1} &= \dot{\mathbf{d}}_k + \omega(\dot{\mathbf{d}}^*_{k+1} - \dot{\mathbf{d}}_k), \end{aligned} \tag{6.14}$$

where $\mathbf{d}^*_{k+1}$ and $\dot{\mathbf{d}}^*_{k+1}$ are responses directly computed from equation of motion. For $\omega < 1$ we have under-relaxation and when $\omega > 1$ the system is over-relaxed. But if the value of ω is set to 1, there will be no relaxation. The exact value of ω is often problem specific and can not be known apriori. Finally, the convergence of the method is decided based on the residual norm of the deformation of the structure. The stopping criterion for the sub-iterations can be based on

$$\frac{\|\mathbf{d}_{k+1} - \mathbf{d}_k\|}{d_{\text{ref}}} \leq \epsilon_{\text{tol}}. \tag{6.15}$$

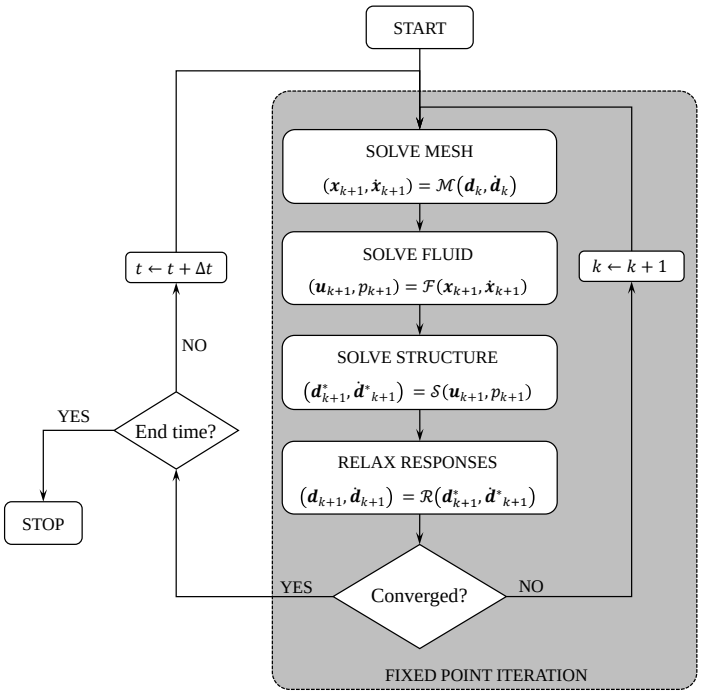


Figure 6.4: Fixed-point iteration (FPI) fluid-stricture coupling algorithm.

In which ϵ_{tol} represents a relative residual tolerance small enough such that the fluid-structure iteration is deemed converged. The parameter d_{ref} is a reference dimension characteristic of the structure. For wind-induced vibration building, d_{ref} can be taken as the width of the building. In Eq. (6.15), the reference dimension d_{ref} is used instead of the norm of structural displacement $\|\mathbf{d}_{k+1}\|$ to avoid the possibility of infinite error that would have resulted when the structure is nearly close to its rest configuration (i.e., $\|\mathbf{d}_{k+1}\| \approx 0$).

6.3 Software implementations

This section presents the implementation of the proposed FSI framework integrating high-level programming libraries of OpenFOAM® and an in-house structural solver. The framework is implemented using C++ programming language and employs the OOP paradigm. For seamless implementation and to reduce the communication overhead of the fluid and structural solvers, the developed structural solver is directly integrated into OpenFOAM® architecture. From a practical perspective, this seems a rational choice, especially for computational aeroelastic simulations, because most of the complexities starting from modeling the physics up to implementing the software architecture, result from the CFD part, not from the structure. Thus, migrating the structural software capabilities to the CFD framework greatly reduces the

software development effort. Furthermore, this permits the most demanding part of the FSI modeling (i.e., turbulent wind flow simulation) to be run in its most optimized native environment, ultimately making the overall FSI simulation computationally efficient.

In addition to considerations related to computational efficiency, by taking advantage of the object-oriented implementations, the framework is designed to offer greater flexibility in representing the fluid and structural subsystems. Though the proposed framework is tailored to the aeroelastic modeling of slender structures, its overall software architecture is organized such that it can easily be extended to simulate other FSI problems. For instance, by just simply changing the way the modal properties of the structure are fed into the current framework, one can seamlessly simulate the aeroelastic behavior of complex civil structures such as long-span bridges and flexible roof systems that are normally more susceptible to the aeroelastic phenomenon than tall buildings are (Vickery, 1990).

6.3.1 Overall software architecture

The core classes, data structure, and field manipulation functions are designed in a modular fashion to represent the entities and operations involved in the actual physical process. Following the outline of the FSI framework presented in Section 6.2, the implemented software architecture contains an assembly of classes that store and manipulate data related to each subsystem: the fluid, dynamic mesh, and structure. Furthermore, accessory (helper) data manipulation and monitoring classes were implemented. Figure 6.5 shows the high-level abstraction of the main software components in the `windFSI` framework. For storing and manipulating field data, as much as possible, the framework uses existing low-level data structures of OpenFOAM such as `scalarField` and `vectorField`.

Based on the existing OpenFOAM's transient solver `pimpleFoam`, a new solver called `pimpleFsiFoam` is developed to handle the fluid-structure coupling (see Figure 6.5). For the fluid subsystem, the implementations in `pimpleFsiFoam` solver are the main modification to the original OpenFOAM's solver capability. Note that `pimpleFsiFoam` is an executable application that contains `main` function. For the structural subsystem, several classes were implemented to compute fluid loads, solve modal equations of motion, and map the motion of the structure to its exterior surface. Thus, most of the software development effort was dedicated to the implementation of this component. Also, in order to manage dynamic mesh, the current framework implemented a new displacement-based motion solver class that can run with `pimpleFsiFoam` or any top-level flow solver. In Figure 6.6, the class diagram of all the objects implemented in the `windFSI` framework. For ease of presentation, the diagram shows only the most relevant properties and methods of each class. The details of the implemented

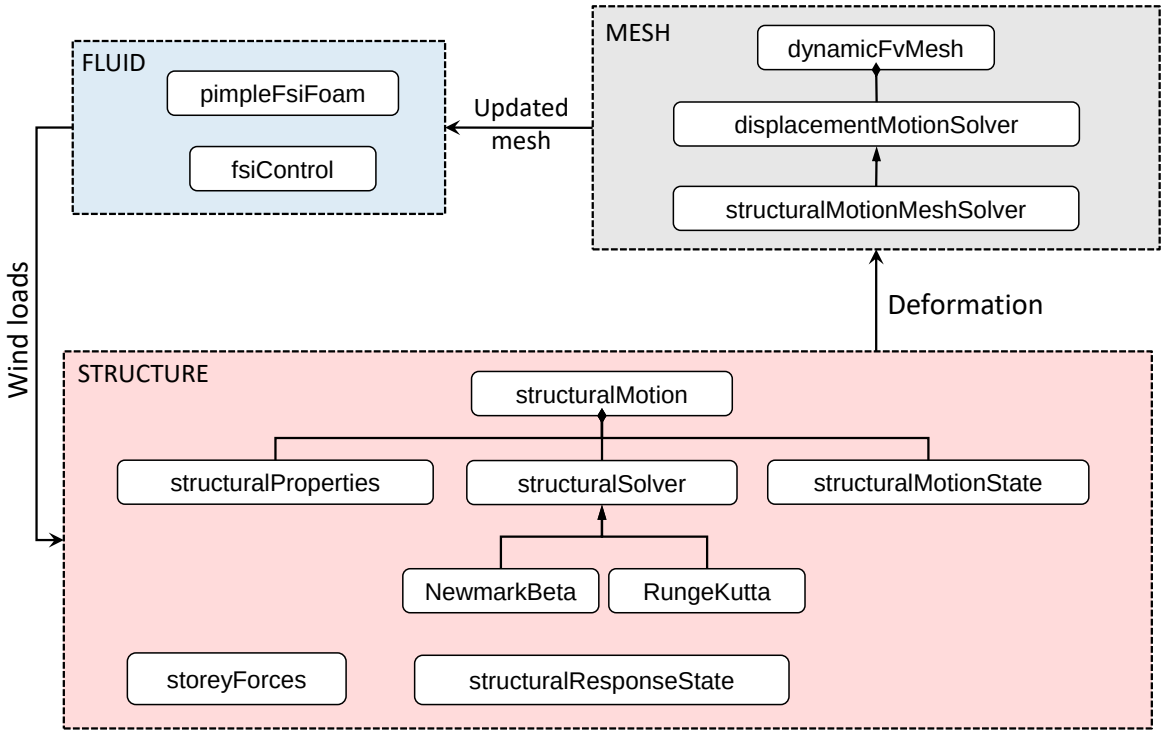


Figure 6.5: Core components of the implemented software architecture in `windFSI` framework

code can be found in Appendix D. The following subsections describe the implementation of each component and the framework's main features.

6.3.2 Implementation of the structural subsystem

The class structure of the structural software component is designed to be flexible and offer different features. It is worth noting that the existing libraries in OpenFOAM do have structural solvers (Jasak and Tuković, 2010). However, these solvers only handle rigid body motion with six degrees of freedom (3 translational and 3 rotational); hence cannot be used to describe the deformation of flexible structures with complex mode shapes (e.g., tall buildings). Thus, developing a new class library to handle structural motions with a multi-degree-of-freedom system prescribed using 3D mode shape and complex mass distribution was necessary.

Now, we introduce the high-level structure of the implemented C++ classes to demonstrate the architecture's capability. The structural component contains three main classes, namely `structuralMotion`, `structuralProperties`, and `structuralSolver`. Each of them is briefly described below. Furthermore, additional classes that assist these classes are described at the end.

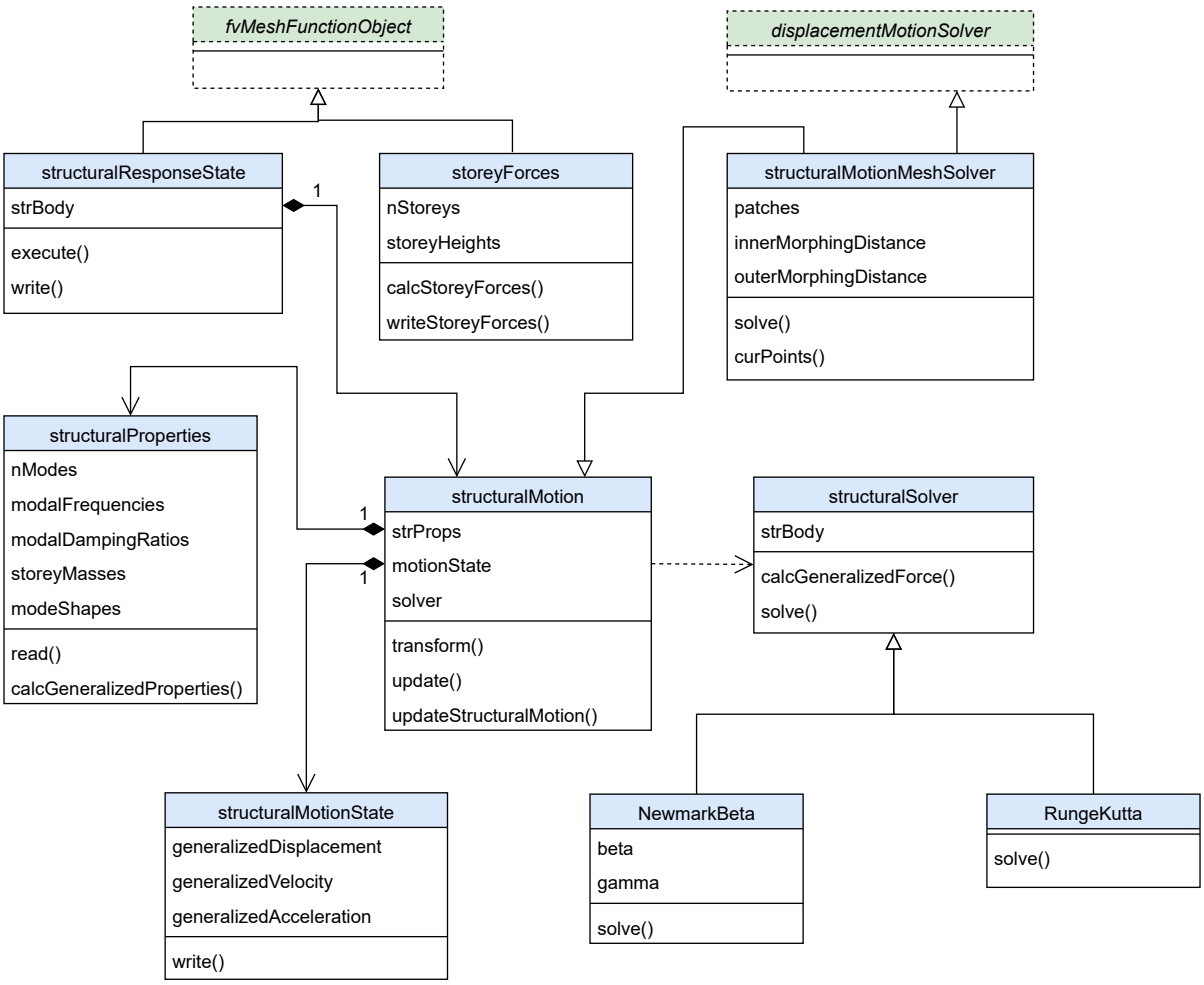


Figure 6.6: Class diagram showing the general structure of the implemented code. Classes shown with dotted borders are native OpenFOAM classes.

structuralMotion

The class `structuralMotion` is the core class of the component that coordinates most of the operations related to the structural subsystem and communicates with the moving mesh. To achieve this array of tasks, this class registers instances of the `structuralProperties` and `structuralSolver` classes. The `structuralMotion` object is responsible for updating the deformation of the structure at each time step based on the structural properties specified. More importantly, using the computed storey forces at each floor level, the `update()` method triggers the `structuralSolver` object to solve the modal equations of motion. Once the modal equations are solved, the `updateStructuralMotion()` method calculates the displacement, velocity, and acceleration of the structure at each floor level using the mode shape vector and instantaneous deformation of the structure in the generalized coordinate system. The `structuralMotion` class also implements a method called `transform()`, which maps

the deformation of the structure to the surrounding mesh given the initial undeformed state of the dynamic mesh points. Furthermore, parallel communication among processors is carried out by this class. Considering that only a few modes are sufficient to describe the motion of the structure, the modal equations of motion are solved only on the master processor, and the results in the generalized coordinate system are scattered to the remaining processors at each time step.

structuralSolver

The `structuralSolver` class is an abstraction of the numerical solver (time integrator) for the dynamic equations of motion. This class is the base class for all structural solvers. It provides methods to calculate generalized forces and solve the modal equations of motion. Furthermore, the class registers reference to `structuralMotion` object for accessing the current state of structural deformations. By driving the `structuralSolver` class, two time-integrator, namely `NewmarkBeta` and `RungeKutta`, which represent Newmark's method and the 4-th order Runge-Kutta scheme, were implemented. Note that these time integrators are run-time selectable by virtue of being derived from the base class `structuralSolver`. For `NewmarkBeta` method, coefficients β and γ are read from the dictionary; hence can be adjusted to specify constant or linear acceleration variants of the algorithm.

structuralProperties

Class `structuralProperties` holds the structural properties of the model. It reads dynamic properties of the structure, such as mode shapes, mass distribution, natural frequencies, and damping ratios. It also provides methods to calculate the generalized mass, damping, and stiffness of the structure from specified dynamic properties. For the application in this paper, the implementation of `structuralProperties` class assumes that the structure is just a multi-story building with the masses concentrated on each floor, and the floors are assumed to be rigid diaphragms. The class stores only properties that are relevant for modal analysis. However, if one wants to change the type of structure, mainly the implementation of this class needs to be modified.

Helper classes

Additional helper classes were also implemented for performing miscellaneous tasks. One of them is `storeyForces` which calculates the loads acting on the structure at each storey level. This is achieved by integrating pressure acting over the tributary area attributed to each floor. The class `structuralMotionState` holds the motion state of the structure in generalized

coordinates. It is mainly used to scatter the motion state among processors and to save the structural motion state if the FSI simulation gets interrupted. The fact that it stores only generalized information (much smaller than storing deformation of the whole structure) makes it memory efficient and takes less time to distribute across processors. The other attendant class is `structuralResponseState`, which monitors the structural response during runtime for postprocessing calculations. As shown in Figure 6.6, both the `structuralResponseState` and `storeyForces` are derived from `fvMeshFunctionObject` class of OpenFOAM. As a result, they are equipped with optimized runtime input-output and can be conveniently specified in `system/controlDict` file.

6.3.3 Implementation of the dynamic mesh

For treating the moving mesh, the current framework implemented a new mesh solver that can be used with the existing dynamic mesh capability of OpenFOAM. The implemented mesh solver is `structuralMotionMeshSolver` class shown in Figure 6.6. In order to take advantage of already developed functionalities, `structuralMotionMeshSolver` is derived from OpenFOAM's `displacementMotionSolver` class which only can handle rigid-body motions. The main features of `structuralMotionMeshSolver` are its ability to transfer the motion of a structure with complex deformation modes into the grid points of the fluid mesh. This class also inherits `structuralMotion` class and triggers the methods to update the motion of the structure.

Furthermore, by switching `twoWayCoupled` control on and off, it is possible to perform two-way or one-way coupling between the fluid and structure. For example, if `twoWayCoupled` is turned off, only the loads are transferred to the structure, but the mesh is not updated. This is actually equivalent to simply running an aerodynamic simulation and post-processing the response of the structure using the recorded aerodynamic loads. Although the primary purpose of performing FSI simulation is to do a two-way coupled simulation, having this option offers a systematic way to test whether the load transferring operation in the FSI simulation works accurately or not by comparing it with aerodynamic simulation data. Also, this type of coupling option can be advantageous for structures with negligible aeroelastic feedback.

6.3.4 Implementation of the FSI Solver

The FSI solver is implemented by modifying `pimpleFoam` transient solver, which is already equipped with dynamic mesh capabilities. The solver is compiled as an executable application with the name `pimpleFsiFoam`. Its interaction with the remaining components of the software architecture is inscribed in Figure 6.5. Essentially, the `pimpleFsiFoam` application has the

"main" function of the program and loops over the fluid and structural subsystems. The code implemented for solving the structure and the moving mesh components are integrated into one library called `libwindFSI.so`, which is dynamically linked to the FSI solver during the compilation. In order to coordinate the fluid-structure coupling, a new helper class called `fsiControl` is implemented. This class controls the FSI iterations and supplies convergence information/checks for the `pimpleFsiFoam` solver.

The high-level C++ code of `pimpleFsiFoam` solver is shown in Listing 6.1. The most basic difference between `pimpleFsiFoam` and the `pimpleFoam` solvers is that the former has one additional loop that runs over the structure, dynamic mesh, and fluid subsystems until the desired convergence criterion is met (see line 4 of Listing 6.1). Furthermore, in the `pimpleFsiFoam` solver, the structure and mesh are updated once the fluid is solved using a PIMPLE algorithm. However, in `pimpleFoam` solver, the dynamic mesh is updated after completing the PISO loops. The added FSI loop in `pimpleFsiFoam` permits better control over convergence while providing versatility in implementing different fluid-structure coupling algorithms (e.g., weak and strong coupling). For instance, if we set the maximum FSI iteration to one, the code shown in Listing 6.1 normally reduces to the CSS coupling algorithm, where both the fluid and structure are solved only once. On the other hand, when the function `fsi.loop()` computes Eq. (6.15) to check the convergence of the FSI iteration; the procedure becomes a fixed point iteration coupling algorithm. As such, by programming stopping criteria such as maximum FSI iteration and error tolerance in the `fsi.loop()` function (a method implemented in the `fsiControl` class), one can readily control the FSI coupling procedure. Note that the first `while` loop in Listing 6.1 increments the time step. Regarding the geometric conservation laws, the calculations in lines 12-16 of Listing 6.1 adjust the face fluxes such that the geometric continuity equation stated in Eq. (6.5) is satisfied.

The `pimpleFsiFoam` solver assumes that the fluid and structure subsystems use the same constant time step. For aeroelastic application, it is clear that this time step is governed by the fluid flow and should satisfy the maximum Courant–Friedrichs–Lewy (CFL) number (C_o) required for the accuracy and stability of the numerical solutions. For moving mesh, in addition to the flow Courant number, the mesh Courant number must be checked. The mesh Courant number is analogous to the flow Courant number but defined based on mesh velocity. The ratio between the mesh and flow Courant number is recommended to be kept below unity to guarantee high-quality results (Benjamin et al., 2021).

```
1  while (pimple.run(runTime))
2  {
3      runTime++;
4      while(fsi.loop())
5      {
```

```
6           //Solve the structure and move the mesh
7           mesh.update();
8           //Correct flux if the mesh is moving
9           if (mesh.changing())
10          {
11              MRF.update();
12              if (correctPhi)
13              {
14                  phi = mesh.Sf() & Uf();
15                  #include "correctPhi.H"
16                  fvc::makeRelative(phi, U);
17              }
18          }
19          //Solve the fluid domain
20          while (pimple.loop())
21          {
22              #include "UEqn.H"
23              // --- Pressure corrector loop
24              while (pimple.correct())
25              {
26                  #include "pEqn.H"
27              }
28              if (pimple.turbCorr())
29              {
30                  laminarTransport.correct();
31                  turbulence->correct();
32              }
33          }
34      }
35      runTime.write();
36  }
```

Listing 6.1: Sample high-level code of the `pimpleFsiFoam` solver

Regarding the turbulence modeling, similar to `pimpleFoam` solver, the `pimpleFsiFoam` can be used with any turbulence model, including laminar, RANS, and LES. For this study, however, the framework is tested and validated using LES, considering applications related to wind-induced vibration of structures, where the flow around the structure is massively separated and the response of the structure in the time domain is required. In Appendix E we briefly demonstrate how the user can set up an aeroelastic simulation utilizing the developed FSI framework.

6.4 Numerical examples and validation

This section demonstrates the capabilities of the developed framework using two numerical examples. The first numerical example involves a vortex-induced vibration of an elastically mounted circular cylinder in the cross-flow direction. Then, in the second example, the framework is used to simulate the aeroelastic response of a tall building with a relatively complex three-dimensional mode shape.

6.4.1 Vortex induced oscillation of a circular cylinder

Flow-induced vibration of a circular cylinder is widely studied in the literature and has broad applications in many engineering fields. For instance, many civil structures, including chimneys, towers, transmission-line conductors, etc., have circular cross-sections and are subjected to wind-induced vibration involving several fluid-structure interaction phenomena. Here, the vortex-induced vibration of a long circular cylinder in the cross-wind direction is chosen as a simple yet practically important bench-marking case to test the capabilities of the proposed framework. Furthermore, previous studies have shown that an elastically mounted circular cylinder undergoes substantial cross-wind oscillation with significant motion-induced (aeroelastic) effects compared to the wind-induced vibration of a tall building. Thus, testing the overall FSI framework using this example demonstrates the importance of modeling the fluid-structure interaction phenomenon.

The numerical model is prepared based on the experimental work published in Belloli et al. (2012). The experimental study presented the vortex-induced vibration of a long circular cylinder of diameter $D = 0.2m$ with a maximum Reynolds number $Re = 5.4 \times 10^5$. The parameters used in the current simulation are summarized in Table 6.1.

6.4.1.1 Numerical model

The extent of the computational domain adopted for the FSI simulation relative to the diameter of the cylinder D is shown in Figure 6.7. The domain has a dimension of $50D \times 20D \times 10D$, and the cylinder is placed $10D$ away from the inlet of the domain. The computational grid used has several refinement regions with denser mesh provided in the vicinity of the cylinder (see Figure 6.8). In total, the grid consists of 10.3 million hex-dominated cells. Over the surface of the prism, in addition to surface refinement, ten prism layers were introduced with a maximum aspect ratio of 5. On average, the center of the first off-wall cell is located approximately $y^+ < 3$ wall units away from the surface; hence, no wall treatment was required. At the inlet, a smooth uniform inflow is applied, while a zero pressure boundary condition is used at the outlet. For

Table 6.1: Details of the model used for FSI simulation

Parameters	Values
Diameter,	$D = 0.2\ m$
Length of the cylinder,	$L = 2\ m$
Aspect ratio,	$L/D = 10$
Mass per unit length,	$m_L = 6.5\ kg/m$
Mass ratio,	$m^* = 170$
Natural frequency,	$f_s = 3.2\ Hz$
Structural damping,	$\xi_s = 2.5 \times 10^{-3}$
Mass damping ratio,	$m^*\xi_s = 0.43$
Strouhal number,	$St = 0.18$
Strouhal velocity,	$U_{St} = 3.56\ m/s$
Air density,	$\rho_{\text{air}} = 1.225\ kg/m^3$
Nominal Reynolds number (at U_{St}),	$Re = 4.8 \times 10^4$

all other faces of the domain, a slip boundary condition is specified.

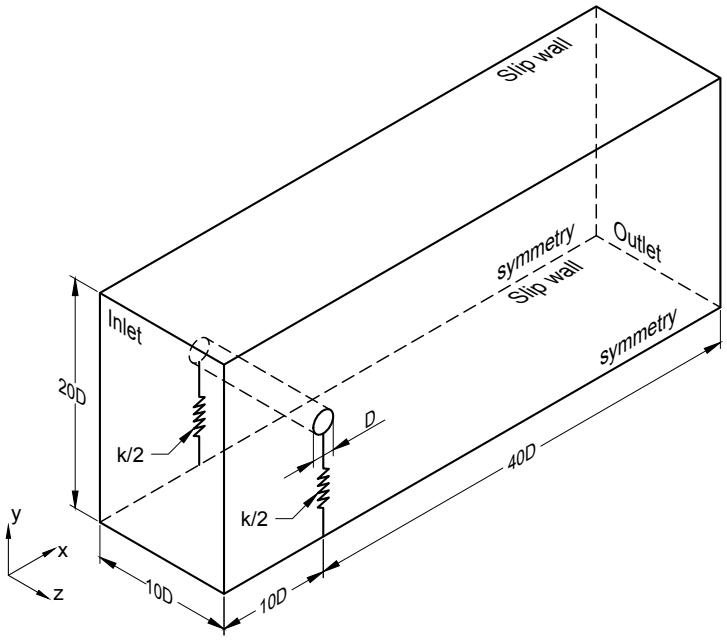


Figure 6.7: Dimensions of the computational domain and definition of boundary conditions

Considering that the flow is inherently unsteady, for the turbulence modeling, large-eddy simulation (LES) is employed with the standard Smagoronsky subgrid-scale model. A second-order accurate backward differencing scheme is used for time integration of the fluid domain. The fluid domain is solved using a single PIMPLE iteration based on the procedure described

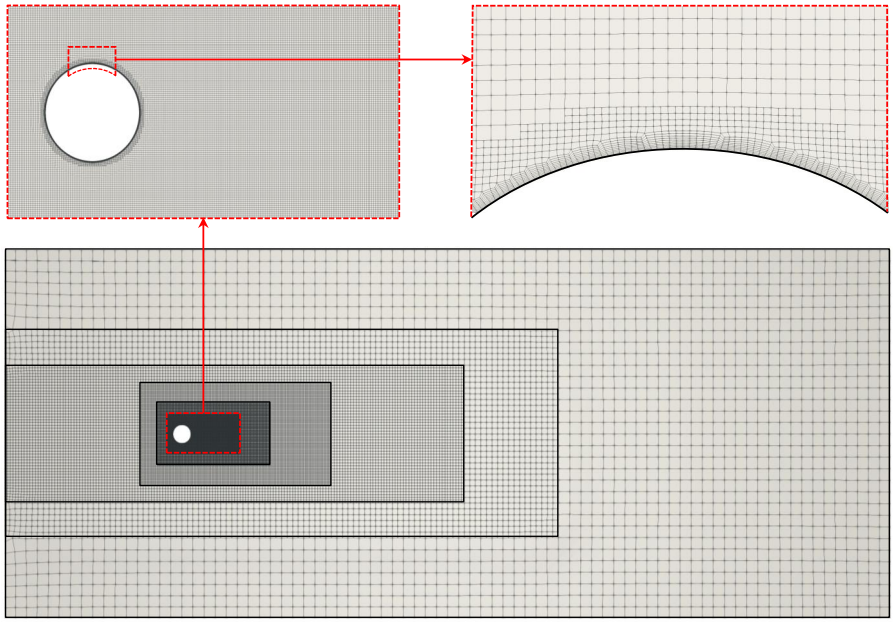


Figure 6.8: Computational grid used: cross-section view along xy-plane.

in Section 6.2.3.1. The simulations are conducted with a time step of 2×10^{-4} s for over 60 s duration.

For time integration of the structural domain, Newmark's method is utilized with $\beta = 0.25$ and $\gamma = 0.5$, with no relaxation of the structural displacements. Mechanical properties of the oscillating cylinder are provided in Table 6.1. In order to test the overall load/displacement transfer mechanism, the cylinder is divided into five segments having a constant mode shape in a cross-wind direction. The oscillation of the cylinder is restricted to be only in the $y-$direction.

For the fluid-structure coupling, CSS and fixed point iteration algorithms were investigated. For the inner and outer distance of the dynamic mesh, D and $4D$, respectively, were specified. For the tolerance of the FSI error given in Eq. (6.15), $\epsilon_{\text{tol}} = 1 \times 10^{-6}$ is used with a reference length equivalent to the diameter of the cylinder. For the fixed point iteration-based coupling algorithm, the maximum FSI iteration is limited to 5. The FSI simulations are initialized with a developed flow field mapped from prior aerodynamic simulations with the fixed cylinder. Finally, the FSI simulation was conducted in parallel utilizing a total of 256 processors, each having 2.50 GHz CPU core.

6.4.1.2 Results

The cylinder is first kept rigid, and the unsteady turbulent flow past the cylinder is simulated using LES. Figure 6.9 shows the flow structure around the cylinder determined from the second invariant of the velocity gradient. For a high Reynolds number flow ($Re \simeq 5 \times 10^4$) considered

in the present work, the flow is turbulent, containing a wide spectrum of length and time scales. As seen in Figure 6.9, the fine computational grid has captured the rolling up of the separated shear layer that forms the shedding of vortexes. Spectral analysis of the velocity at the wake of the rigid cylinder gave Strouhal a number approximately equal to $St = 0.2$, compared to a value of 0.18 reported in the experimental work of Belloli et al. (2012).

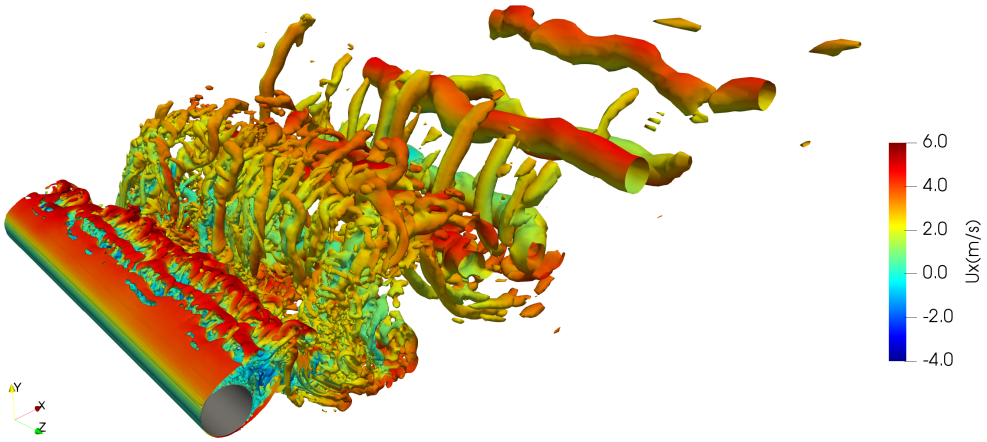


Figure 6.9: Iso-surface of the second invariant of the velocity gradient, $Q = 100$ colored by the stream-wise component of instantaneous velocity.

Now, we investigate the dynamics response of the cylinder. Here, by performing the FSI for different velocity ratios U/U_{St}, the cross-wind oscillation of the cylinder is investigated. Figure 6.10 shows the time series of the displacement response for different U/U_{St} values. It can be noted that the cylinder experiences significant oscillation for $1.1 < U/U_{St} < 1.25$. A similar range ($1.11 < U/U_{St} < 1.31$) is also reported in the experimental work of Belloli et al. (2012). This region is commonly referred to as the "lock-in" region in the literature, and the shedding frequency synchronizes with the natural frequency of the structure.

The first stage of validation involves comparing the responses predicted using the current FSI framework with the ones found using OpenFOAM's built-in dynamic mesh solvers for rigid body motion. Specifically, for this example, we used OpenFOAM's `sixDoFRigidBodyMotion` solver as a benchmark for `windFSI` framework. Thus, employing this solver, the vortex-induced vibration of the cylinder is simulated using the same numerical setup. Figure 6.11 compares the time series of the displacement responses from the two cases as a function of non-dimensional time $t_n = t/T_s$, where T_s is the period of oscillation. It is worth noting that OpenFOAM's rigid body motion solver implements a weak coupling scheme, and for the comparison in Figure 6.11, a staggered coupling algorithm is adopted for `windFSI` framework. As

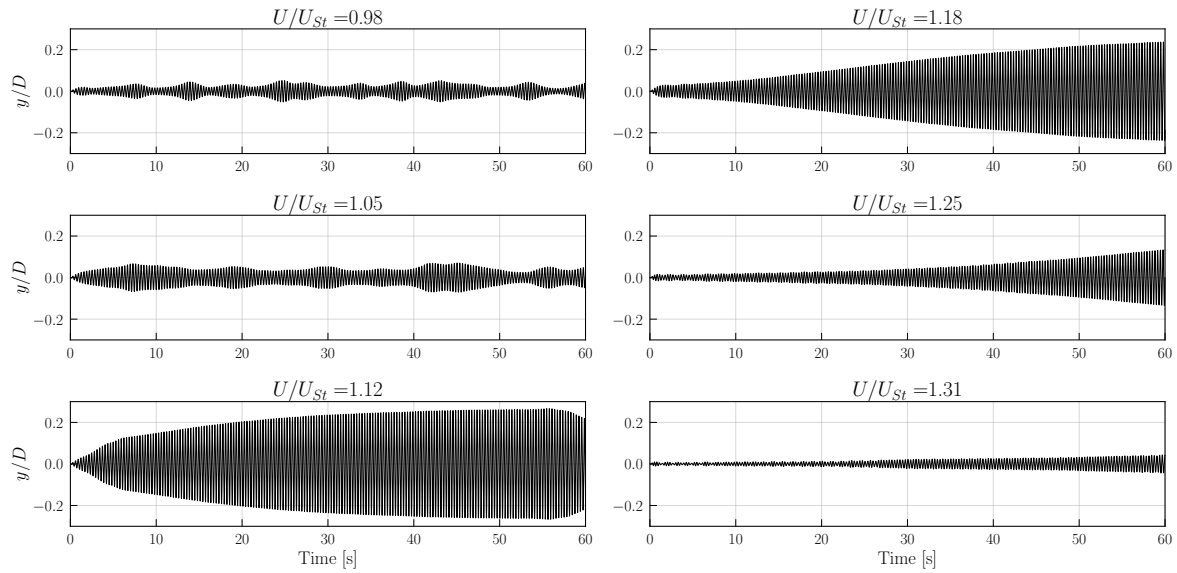


Figure 6.10: Time histories of the non-dimensional displacement of the cylinder for different velocity ratios U/U_{st}.

seen in Figure 6.11, despite the implementation difference, the responses predicted from both cases are nearly identical. Further statistical comparison of the results `windFSI` framework with OpenFOAM is shown Figure 6.14a for the lift force coefficient(C_L).

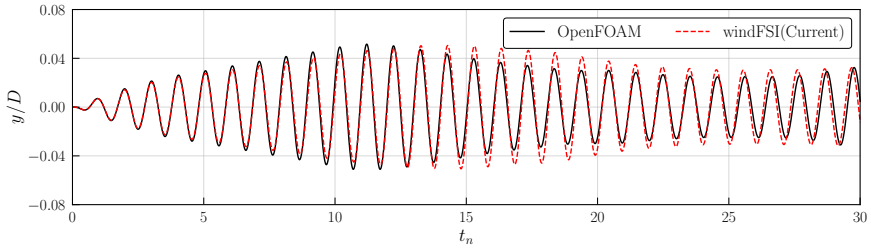


Figure 6.11: Comparison of the displacement time history from `windFSI` framework and OpenFOAM's rigid body motion solver for $U/U_{St} = 1.0$.

For FSI problems, one of the most important factors that determine the accuracy of the simulation is the coupling algorithm used. Here, using the same numerical example, we compare the performance of CSS and FPI coupling algorithms. Figure 6.12 shows the displacement of the cylinder predicted using the CSS and FPI coupling techniques. For the FPI coupling algorithm, it was observed that only two FSI iterations were sufficient to reduce the relative error below the tolerance limit. As shown in Figure 6.12, the difference between the CSS and FPI coupling algorithms is generally small. This is also reflected in the power spectral density (PSD) functions of the lift force coefficients from CSS and FPI as depicted in Figure 6.14a. Cit-

ing that the CSS method is a loosely coupled method and does not enforce convergence within the one-time step, it is normally expected to be less accurate compared to the fixed point iteration method. However, for this numerical example, mainly constrained by the requirements of the fluid domain, the time step used for the structural computation is very small compared to the period of oscillation of the structure, and the aerodynamic loads do not change significantly over a single time step. Hence, from a practical point of view, the accuracy of the CSS and FPI methods are comparable, ultimately making the CSS method preferable because of its reduced computational cost. This is also true for most aeroelastic applications, where the structure is usually heavy and stiff (Degroote et al., 2009).

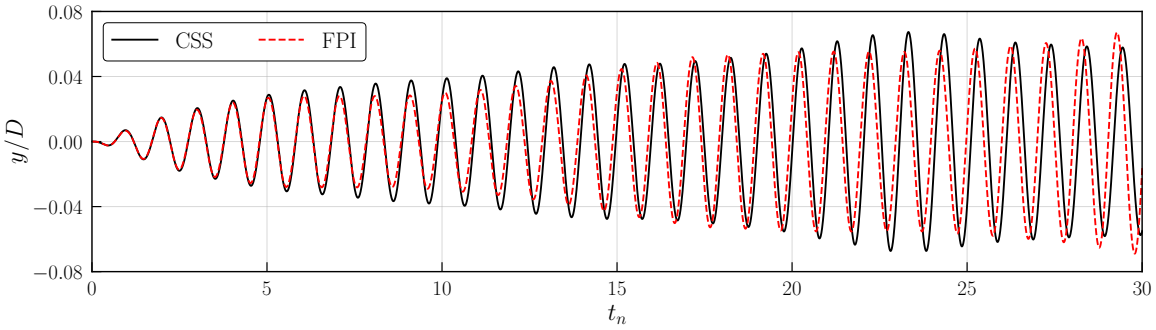


Figure 6.12: Displacement response computed using Conventional Serial Staggered (CSS) and Fixed-point Iteration (FPI) coupling algorithms for $U/U_{St} = 1.0$.

Furthermore, the performance of two different time integration methods, namely Newmark's constant acceleration method and a more accurate fourth-order Runge–Kutta method, are compared in Figure 6.13. As seen in the figure, the responses from both methods are very close and do not significantly alter the fluid-structure coupling. Thus, in the rest of this study, Newmark's scheme is employed as it is computationally less expensive.

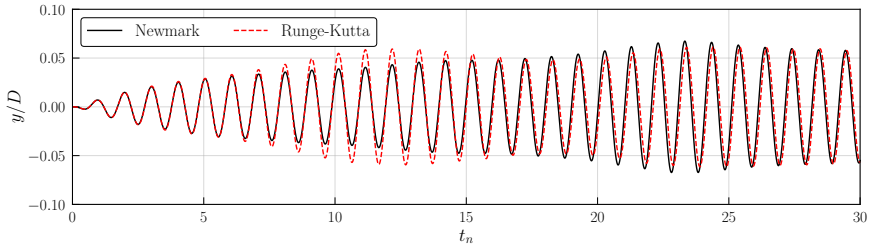


Figure 6.13: Comparison of the structural responses predicted using Newmark's constant acceleration method and a fourth-order Runge–Kutta scheme.

Finally, to investigate the significance of motion-induced forces (aeroelastic effect), the lift force coefficient (C_L) on the cylinder is studied with and without FSI simulation. Figure 6.14b

compares the PSD of the lift force coefficient for the rigid and oscillating cylinder. The spectrum shows a district peak at the vortex shedding frequency ($f/f_s = 1$). As expected, due to the motion-induced effects, the lift force FSI simulation results are higher than the rigid cylinder, which seems to be captured by the FSI simulation. Overall, in this numerical example, it was demonstrated that `windFSI` framework is capable of capturing the aeroelastic phenomenon of flexible structure.

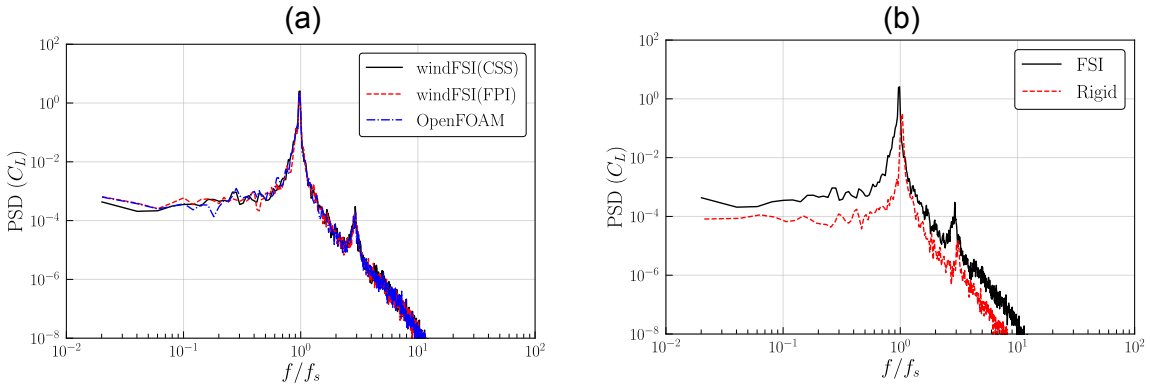


Figure 6.14: Power spectral density(PSD) of the lift force coefficient (C_L) for $U/U_{St} = 1.0$: (a) comparison of `windFSI` with OpenFOAM; (b) FSI vs. Rigid cylinder

6.4.2 Wind induced vibration of a tall building

In this second numerical example, employing the `windFSI` framework, the dynamic response of a tall building to a turbulent atmospheric boundary layer (ABL) flow is simulated. Considering the framework is specially developed for slender and flexible structures, this numerical example aims to demonstrate the framework's most important capabilities. The flow around the building is modeled using LES and assuming the building is immersed in ABL flow. For the building structure, we used the Commonwealth Advisory Aeronautical Research Council (CAARC) model. The CAARC model is a rectangular building with height $H = 182.88$ m, width $B = 45.72$ m, and depth $D = 30.48$ m (Melbourne, 1980). The simulation is conducted for two wind directions that represent cases where the wind is perpendicular to the wider face (0° wind direction) and the narrower face (90° wind direction). For comparison purposes, for both wind directions, the response of the structure is evaluated using a simple pressure integration using one-way coupling.

6.4.2.1 Modeling of the wind flow

The wind flow is simulated by specifying ABL flow conditions characterizing open exposure conditions with aerodynamic roughness height of $z_0 = 0.025$m. The computational domain used for the simulation is shown in Figure 6.15, where dimensions are expressed relative to the building height. At the inlet of the domain, we used inflow turbulence generated using the Divergence-free Spectral Representation (DFSR) method (Melaku and Bitsuamlak, 2021). A slip velocity boundary condition is specified for the side and top faces. At the outlet of the domain, a zero-pressure boundary condition is applied. Considering the computational cost of fully resolving the turbulent wall boundary layer for the building surface, a smooth wall function based on Spalding's formula (Spalding, 1961) is used. On the ground surface, assuming a log-law relationship, a rough wall boundary condition based on the Schumann–Grötzbach model (Grötzbach, 1987; Schumann, 1975) is adopted. For the Schumann–Grötzbach model, an aerodynamic roughness height representative of the exposure condition simulated (i.e., $z_0 = 0.025$m) is supplied. For the fluid, properties of air with density $\rho = 1.225 kg/m^3$ and kinematic viscosity $\nu = 1.5 \times 10^{-5} m^2/s$ are specified.

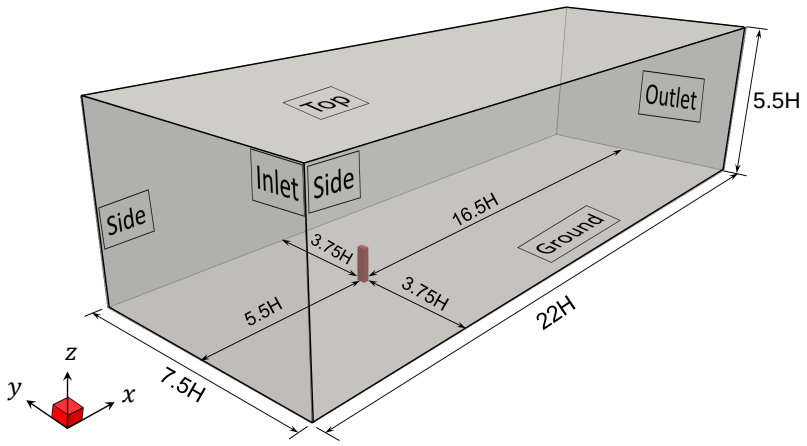


Figure 6.15: Extent of the computational domain and naming of the boundaries

The computational grid employed for the simulation is shown in Figure 6.16 for 0° wind direction. To resolve important flow features near the building, the mesh is progressively refined as we get close to the surface of the building. The grid size used in each refinement region is given in Figure 6.16. Overall, the computational grid consists of approximately 10 million cells.

In the fluid domain, a fully implicit second-order accurate backward differencing scheme is used to perform time integration of the temporal derivatives. For spatial discretization, a

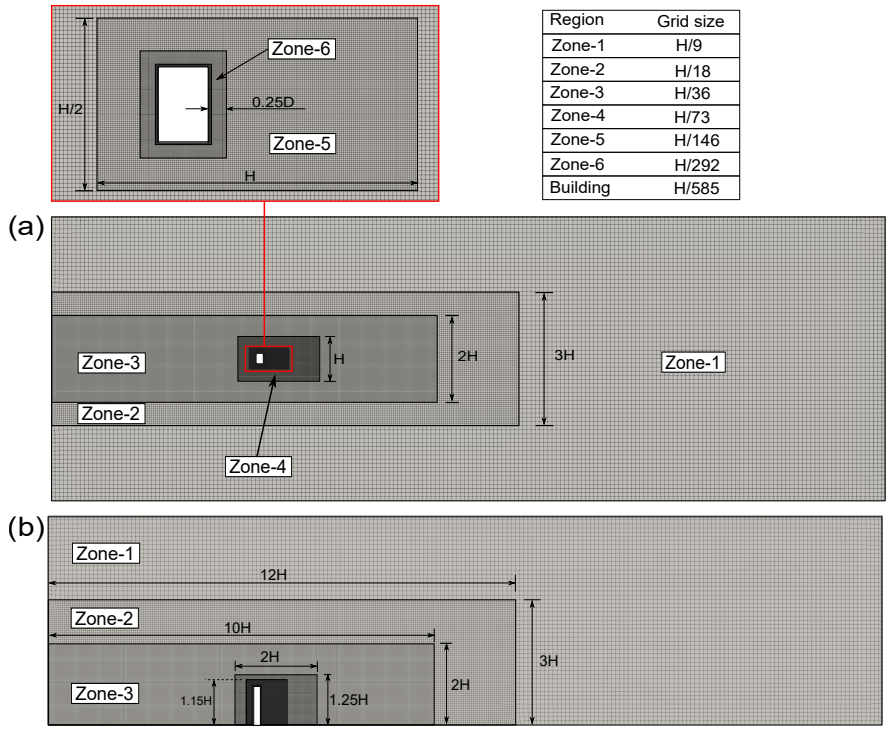


Figure 6.16: Computational grid used for FSI simulation with mesh size specified in each refinement zones: (a) horizontal section; (b) longitudinal section

second-order accurate central differencing scheme is adopted. The simulations were run for a duration that is roughly equivalent to a full-scale one-hour duration. The fluid subsystem governs the time step, and a constant value of $dt = 5 \times 10^{-}5$ s is adopted for this numerical example. The maximum Courant-Friedrichs-Lewy (CFL) number in all the simulations is kept well below 7.0.

6.4.2.2 Modeling of the building structure

For the building structure, the dynamic properties of a 60-storey reinforced concrete building are used. Each storey has a constant inter-storey height of 3.05 m. The building uses a moment-resisting frame structural system that constitutes columns, beams, and rigid diaphragm slabs. The structural system is shown in Figure 6.17. A detailed description of the sectional properties of the structural members, including reinforcement details, can be found in Park et al. (2018). The distribution of lumped masses over the building height is shown in Figure 6.17c.

For the FSI simulations, however, the building is represented by a lumped-mass system using only the modal properties of the structure extracted from a finite-element model of the building. This permits the FSI simulation to be computed in a computationally efficient manner. The simulations were conducted using the first six modes of vibration of the building. Fig-

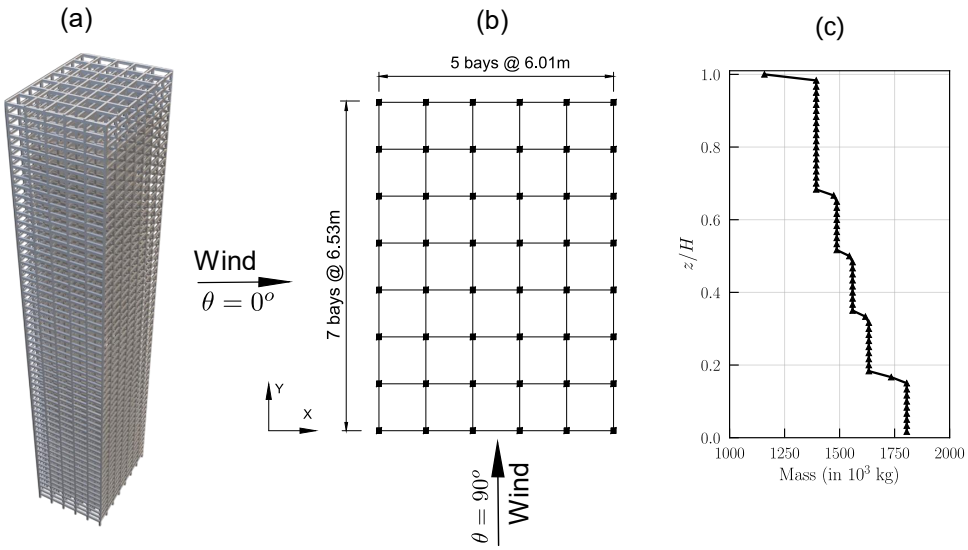


Figure 6.17: Structural system used for CAARC building: (a) 3D view ; (b) plan view; (c) mass distribution per each floor

ure 6.18 shows the mode shape of the building for longitudinal(X), lateral(Y) and torsional(T) degrees of freedom. For structural damping, a constant modal damping ratio of 2.0% (of the critical) is used for all modes. For the time integration of the structural subsystem, Newmark's

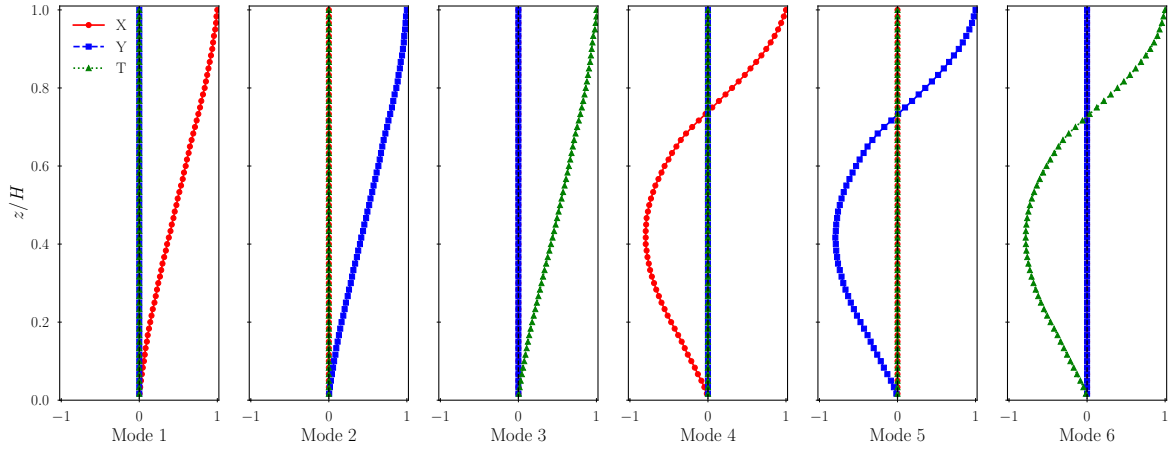


Figure 6.18: The first six vibration mode shapes of the building in full-scale: Mode 1(0.156Hz); Mode 2(0.167Hz); Mode 3(0.192Hz); Mode 4(0.450Hz); Mode 5(0.459Hz); Mode 6(0.512Hz)

method is employed. The same time step used in the fluid solver is also specified for the structural subsystem. The response of the building, including the displacement and acceleration of each storey, is monitored at a sampling frequency of 400 Hz.

6.4.2.3 Results

Before evaluating the performance of the FSI framework, the characteristics of the approaching flow simulated using LES and synthetic inflow turbulence generated using the DFSR method are compared against experimental target measurements. This is important because the wind profiles, as well as the spectral content of the approaching flow, significantly affect the response of the structure. Thus, in Figure 6.19, these flow characteristics are inspected. For the LES, the incident wind profiles and velocity spectra are measured in an empty domain simulation without the building. The comparison of the mean velocity and turbulence intensity profiles are depicted in Figure 6.19a and Figure 6.19b, respectively. Similarly, the roof-height velocity spectrum for the stream-wise component is compared in Figure 6.19c. From Figure 6.19, it can be seen that the wind profiles and velocity spectra are in excellent agreement with the experimental data.

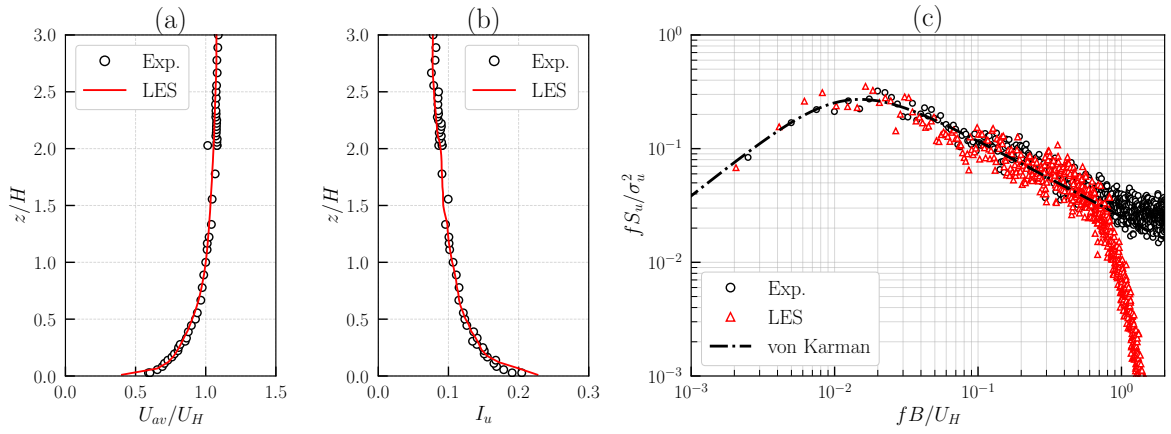


Figure 6.19: Characteristics of the incident flow used in the FSI simulations measured in an empty domain simulation: (a) mean velocity profile; (b) stream-wise turbulence intensity profile; (c) roof-height velocity spectra

Next, after running the FSI simulations, we investigated the wind-induced response of the building. Figure 6.20 shows the time history of the displacements (longitudinal, lateral, and torsional) monitored at the topmost floor of the building for 0° wind direction. The torsional displacements shown in the figure are converted into linear displacement by multiplying with the radius of gyration at the top floor. The displacements determined from fully coupled and one-way coupled (rigid) simulations are plotted together for comparison purposes. The spectral content of the displacement responses for the two wind directions considered (0° and 90°) is presented in Figure 6.21. The response spectra show that the FSI simulations performed using `windFSI` framework have captured the fundamental (first) and higher mode effects well. Similarly, in Figure 6.22, the spectra of the top floor acceleration responses are compared.

Comparing Figures 6.21 and 6.22, it can be noted that the contribution of higher modes is more pronounced in acceleration compared to displacement responses, which is also commonly observed in experimental studies (Warsido, 2013). Considering that the structure used for this numerical example does not show significant aeroelastic effects, the responses from fully coupled and one-way coupled (rigid) simulations are nearly identical, as shown in Figures 6.21 and 6.22.

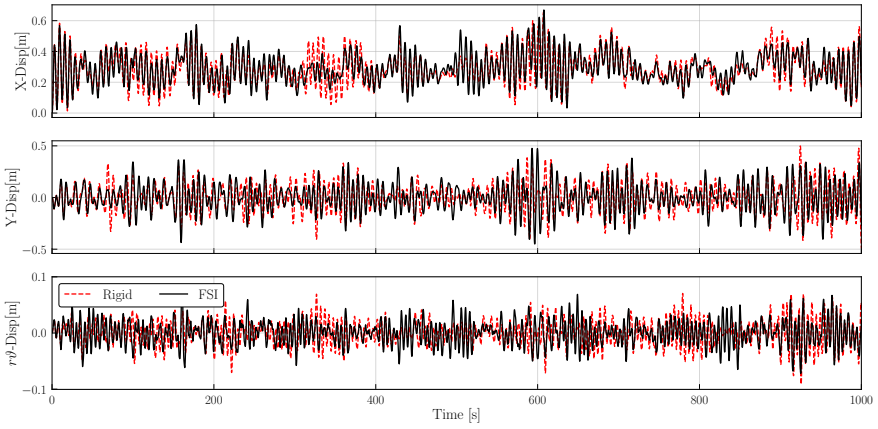


Figure 6.20: Time history of the top floor displacement for 0° wind direction.

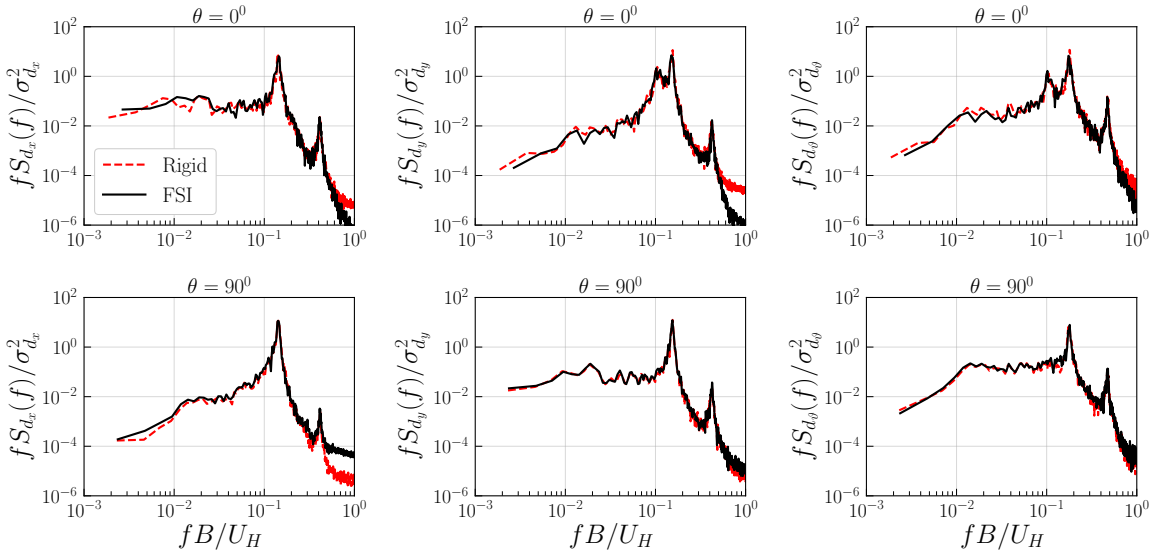


Figure 6.21: Reduced spectra of the top floor displacement response in x, y and ϑ directions for 0° and 90° wind angle of attack.

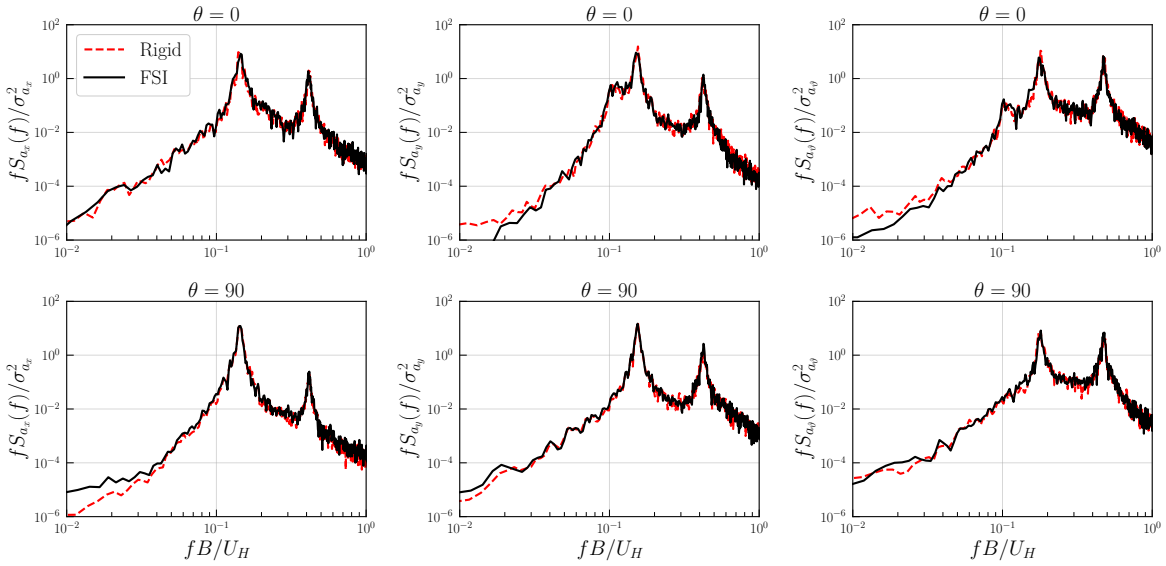


Figure 6.22: Reduced spectra of the top floor acceleration response in x, y and ϑ directions for 0° and 90° wind angle of attack.

6.5 Conclusions and outlook

In this paper, we presented the development of a high-fidelity fluid-structure interaction framework called `windFSI` for simulating the aeroelastic behavior of flexible structures. The framework is well-suited to applications related to wind engineering, specifically for simulating overall wind-induced loads and responses of slender flexible structures. The mathematical formulations of the framework, including the governing equations and coupling algorithms, were described in detail. The framework uses a partitioned approach that solves the Navier–Stokes equations for fluid domain using Arbitrary Lagrangian–Eulerian (ALE) formulation. The governing equations of motion for the structural subsystem were transformed and solved in a generalized coordinates system to improve computational efficiency. Both strong and weak coupling algorithms were built-in for coupling the fluid and structure subsystems. The `windFSI` framework was implemented using C++ with an object-oriented programming paradigm and is shown to operate seamlessly with the existing software architecture of OpenFOAM.

Finally, we demonstrated the application of the proposed framework for simulating wind-induced vibration of flexible structures using two validation examples. The first example demonstrated the accuracy of `windFSI` framework for predicting the vortex-induced cross-wind oscillation of a circular cylinder. In the second numerical example, we successfully showed how the proposed framework could be used for predicting the wind-induced vibration of a tall building with a complex mode shape and mass distribution immersed in turbulent

atmospheric boundary layer flow. In the future, we plan to extend the framework by incorporating features for the aeroelastic modeling of structures with nonlinear material behavior. Being in the age of high-performance computing, computational frameworks of this kind enable us to tackle complex problems of a multi-physics nature in an efficient and integrated manner.

Chapter 7

Summary, conclusions, contributions, and future research directions

7.1 Overview

In this chapter, the main findings, contributions, and limitations of the current research work are summarized. This thesis presented five studies that team towards the development of a CFD-based high-fidelity computational framework for aerodynamic and aeroelastic simulations of wind loads on tall buildings. The main objective of this thesis has been to improve the current capability of CFD, particularly that of large-eddy simulation (LES), by addressing some of the critical challenges in the numerical simulation of wind loads and wind-structure interaction. In addition to presenting these key enabling developments, this study demonstrated the framework using a series of validation studies that use benchmark boundary layer wind tunnel measurements. Based on results from the validation studies, it can be argued that CFD-based wind load evaluation is maturing, where wind and structural engineers can drive utility from the developments long before it reaches perfection to compete with wind tunnel testing.

The proposed framework addressed the challenges of CFD-based wind load evaluation on tall buildings in three key stages. The first stage of the research presented the development and implementation of computationally efficient methods needed to make LES an accurate tool for simulating Atmospheric Boundary Layer (ABL) flows. In the next stage, neglecting the flexibility of the building, the developed framework is applied to two case studies that investigate aerodynamic wind loads in generic and complex building configurations. At the final stage of research, we expanded the framework's capability by incorporating a methodology to accurately model the aeroelastic response of flexible tall buildings coupling the state-of-the-art LES model with the structural model of the building.

The overall thesis is summarized in three prominent themes that were pursued in Chapters 2 to 6. These themes were *modeling of approaching ABL turbulence*, *LES-based wind load and response evaluation on tall buildings*, and *fluid-structure interaction for aeroelastic modeling*. Summaries, conclusions, and contributions of the core chapters under each theme are laid out in the following subsections.

7.2 Modeling of approaching ABL turbulence

This research work started by addressing the challenges of LES-based modeling of atmospheric boundary layer (ABL) turbulence, which constitutes the first important step in accurately modeling wind loads on tall buildings. Chapter 2 presented the development and validation of inflow turbulence and ground roughness modeling techniques for ABL flows. Then, Chapter 3 improved the computational efficiency of the developed inflow turbulence generation method.

7.2.1 Chapter 2: Synthetic inflow turbulence generator for large-eddy simulation of ABL flows using spectral representation method

This chapter began by presenting a brief literature review of methods used to generate transient inflow boundary conditions for large-eddy simulation (LES) of the atmospheric boundary layer (ABL) flows. The advantages and limitations of the different inflow turbulence generation methods were revisited. It was emphasized that the synthetic methods provide greater flexibility and are computationally less expensive than wind tunnel replication and precursor methods. Most importantly, synthetic methods give tight control over the generated turbulence, as they take known statistical descriptions of the flow as input. This makes them particularly attractive for LES-based wind load evaluation, where the first step is to match the target approaching flow characteristics taken from experimental or field measurements.

Considering that two-point flow statistics are essential in the development of upstream ABL turbulence and consequently determine the accuracy of wind load simulation, in Chapter 2, a new synthetic inflow turbulence generation method is proposed. The method used the spectral representation method to generate inlet turbulence with explicitly defined two-point statistics of ABL turbulence. The two-point statistic of the flow is imposed by prescribing the target cross-power spectral density (CPSD) matrix at the inlet of the domain. The generation procedure involves the decomposition of the CPSD matrix followed by simulation of the velocity-time field using the Fast Fourier Transformation (FFT) technique. The developed technique is named the "divergence-free spectral representation" (DFSR) method.

For the LES of ABL flows, as important as the inflow boundary condition, a special wall

treatment is required to represent the effect of ground roughness. Thus, in this chapter, an implicit roughness modeling technique is implemented to simulate any desired exposure condition for a given aerodynamic roughness length. This boundary condition was found to reduce the resolution requirement needed near the ground surface. The method was implemented to work seamlessly with the CFD solver and the developed inflow boundary condition.

Finally, the developed methods were applied to the LES of ABL flow, and the results were validated using experimental data thoroughly. Also, the performance of the proposed inflow-generation method is compared with existing techniques in the literature. The main findings from this study are noted below:

- Comparison of one-point and two-point statistics of the turbulence generated using the proposed method with the experimental data reveals that the proposed approach renders an accurate representation of velocity spectra, coherency function, and spatial correlation. Taking the experimental data as a target, the method properly captured the coherency function and spatial correlation of the lateral and vertical components of the velocity owing to the robust capability of the spectral representation method to model two-point statistics accurately.

- Applying the DFSR method and the implemented ground roughness model to LES of ABL flow in open, suburban, and urban exposure conditions, it was shown that the mean velocity, turbulence intensity, and integral length scale profiles are in satisfactory agreement with the experimentally measured wind profiles. Similarly, the incident velocity spectra show reasonable agreement with the target von Karman spectra as well as experimental measurements. Compared to a commonly used method in the literature, it has been shown that accurate modeling of spatial correlation has resulted in a reduction of turbulence decay downstream of the inflow plane, particularly in the lateral and vertical directions.

- Furthermore, due to divergence-free operation, the DFSR method also showed a reduction in the artificial pressure fluctuation in the computational domain. This makes the developed method particularly useful for LES-based wind load evaluation studies, which are sensitive to pressure fluctuations.

One of the main contributions of this work is the development of a new inflow turbulence with explicitly defined two-point statistics that is well-suited to many CWE applications. Particularly, the method can be of practical use to LES-based wind load evaluation on high-rise and low-rise buildings, as well as other studies requiring modeling of ABL turbulence. The proposed method was implemented in OpenFOAM and distributed open-source to foster collaboration with other researchers in the CWE community.

7.2.2 Chapter 3: Computationally efficient inflow turbulence generation using a low-rank matrix decomposition

One of the challenges of generating flow turbulence using the spectral representation method (SRM) is the computational cost of decomposing/factorizing the CPSD matrix at multiple frequencies. This particularly becomes a significant bottleneck for a computational domain with a large number of points at the inlet because the cost of factorizing the CPSD matrix inherently scales cubically with the number of points used in the simulation. In this chapter, first, the fundamentals of the spectral representation method were revised. Then, a computationally efficient method that employs a low-rank representation of the CPSD matrix is developed. The proposed technique employs Nyström method to efficiently factorize the CPSD matrix by sampling only a small subset of informative points from the inlet plane. Factors affecting the compromise between accuracy and computational efficiency, which includes the sampling scheme used and the number of points sampled, were systematically investigated using numerical examples. The following specific observations were noted from this study:

- It was evident that the proposed method can predict the first few eigenvalues with a high level of accuracy for the range of frequencies studied. Considering that the first few eigenvalues carry a significant portion of the total energy makes Nyström method clearly advantageous in reducing the computational cost of the spectral representation method.

- The accuracy of the proposed method greatly depends on the sampling technique used to construct informative points for the low-rank representation of the CPSD matrix. From all six sampling methods tested, the clustered scheme yielded the highest numerical accuracy.

- The trade-off between accuracy and computational efficiency is marginal. For instance, to generate inflow turbulence, using only 25% of the points sampled at the inlet, the proposed procedure takes less than 7% of the total CPU time required to decompose the CPSD matrix formed using all points. While the average error introduced in the standard deviation of the generated turbulence is only 7.9%, compared to the 6.6% error of the original formulation.

The most notable contribution of this study is that the proposed method can achieve remarkable computational efficiency with a minimal compromise in numerical accuracy. This will speed up the generation of inflow turbulence for high-resolution LES of ABL flows by significantly lowering the CPU and memory cost associated with the spectral representation method. Furthermore, the application of the Nyström method to the SRM proposed by Shi-

nozuka and Jan (1972) is unique. To the best of the authors' knowledge, it was the first time the method was applied to speed up the stochastic simulation of any random field.

7.3 LES-based wind load and response evaluation on tall buildings

Under this theme of the study, the capabilities of the techniques developed in previous chapters are demonstrated using two validation studies. First, in Chapter 4, the potential of the LES for estimating wind loads and response is illustrated using an isolated generic tall building located downstream of open country exposure. Then, in Chapter 5, the LES-based wind load evaluation procedure is demonstrated using a more realistic case involving a tall building located in a complex urban environment.

7.3.1 Chapter 4: LES for predicting wind loads and responses of a standard tall building: prospect for wind-resistant tall building design

This chapter investigates cladding and overall loads and responses of a standard tall building using LES. In addition, the study addresses the main challenges of LES-based wind load prediction, including mesh generation, specification of boundary conditions, wall treatment, and sub-grid scale modeling. Using the dynamic properties of a 60-storey reinforced concrete building, this study also investigates the capabilities and limitations of LES for the wind-resistant design of tall buildings. Finally, the LES predictions are validated in detail using wind tunnel measurements stage by stage. A summary of the main findings from this study is provided below:

- Using the DFSR method coupled with an implicit ground roughness model, it was possible to reproduce the experimentally measured approaching flow characteristics in terms of the mean velocity, turbulence intensity, integral length, and velocity spectra at the incident location.
- Overall, the cladding loads predicted using LES were in satisfactory agreement with the wind tunnel measurement. Averaging across all wind directions, the normalized mean absolute error of the mean, RMS, and peak pressure coefficients are roughly within 5% deviation when compared to the experimental data.
- The mean integrated base aerodynamic loads are estimated within a 6% error range, while for the RMS values, the maximum deviation is well within a 10% margin.

- The LES models predicted the low-frequency background displacement response with high accuracy compared to the resonant part for all wind directions considered. Also, the numerical estimate of the resultant peak acceleration is encouraging. On average, the peak acceleration from the LES roughly showed a 15% deviation from the experiment.

The main contribution of this study is the development and application of a high-fidelity LES model for estimating wind loads and wind-induced responses of a standard tall building. The challenges of modeling turbulent separated flows for wind load prediction were addressed, and practical recommendations for LES were provided. Also, this study establishes the most important step towards the development of a fluid-structure interaction framework for a more accurate and direct estimation of the wind-induced response of tall buildings.

7.3.2 Chapter 5: LES-based wind load evaluation on a tall building located in a city center: comparison with experimental data

This chapter aims to demonstrate and further validate the LES-based wind load evaluation procedure using a 278m tall building located in a city center with complex surroundings. The approaching wind characteristics for suburban exposure conditions were simulated using the DFSR method. Important considerations for simulating wind loads in a complex urban environment, such as geometric modeling and computational grid design, were addressed. Finally, the results from the LES model were compared with wind tunnel data for four wind directions. Here are some of the observations from this study:

- The wind profiles and velocity spectra were reproduced in empty domain simulation. Compared to the wind tunnel profiles, the stream-wise mean velocity and turbulence intensity profiles from LES showed an absolute relative deviation of approximately 2.7% and 6.75%, respectively.

- In most cases, the base load spectra estimated from LES models are in satisfactory agreement with the experimental measurements, indicating that the LES model was able to capture various aerodynamic loading mechanisms.

- Averaging over all wind direction, the mean and RMS base force and moment coefficients predicting by LES deviate by about ±10% from the experiment. Whereas, in some of the simulated cases, the maximum difference between the LES and experimental becomes 24%.

A notable contribution of this study is the illustration of the LES-based wind-load evaluation framework using a realistic validation case. Furthermore, the procedure developed in this

study is of practical importance to wind engineers, structural engineers, architects, and contractors to address the fundamental wind engineering challenges of tall building design early in the project life-cycle. Given the current capability of LES, it is getting ever closer to fulfilling its promise to become a practical wind engineering tool.

7.4 Fluid-structure interaction for aeroelastic modeling

In this part of the thesis, a high-fidelity fluid-structure structure (FSI) framework tailored to *wind* engineering applications named `windFSI` was developed. The mathematical formulation, software implementation, and application examples are detailed in Chapter 6. Here the brief summary, main findings, and contribution of the proposed FSI framework are highlighted.

7.4.1 Chapter 6: Fluid-structure interaction framework for computational aeroelastic modeling of tall buildings

In this study, a partitioned fluid-structure structure framework is developed for computational aeroelastic modeling of tall buildings. For modeling the wind flow, the Navier–Stokes equations were solved in an Arbitrary Lagrangian–Eulerian (ALE) frame of reference on a moving mesh. The building structure is represented using the multi-degree-of-freedom system. Two coupling algorithms, namely Conventional Serial Staggered (CSS) and Fixed-point Iteration (FPI) methods were investigated. The framework was implemented employing an object-oriented programming paradigm with C++ language. Since the FSI simulations are generally demanding, the overall software architecture is designed to support parallel execution.

At the end of the chapter (see Section 6.4), the FSI framework is demonstrated using two numerical examples. The first numerical example involves a low-damped vortex-induced cross-wind oscillation of a long circular cylinder with uniform flow conditions. This example is intended to show the importance of incorporating the motion-induced (aeroelastic) effects using FSI simulation. The second numerical example investigates the wind-induced vibration of a standard tall building. In this second example, the most important capabilities of the proposed framework are demonstrated. For the building structure, the dynamic properties of 60-storey building with non-linear mode shapes were employed. For the wind flow simulation, the LES model described in Chapter 4 was adopted. The main findings from this study are:

- For aeroelastic simulations that involve rigid-body structural motion, the developed FSI solver produces similar results to the benchmark FSI solver, as demonstrated using the first numerical example.

- The spectral analysis of the dynamic response of the structure showed that both the CSS and FPI coupling algorithms have comparable performance. Noting that the FPI coupling algorithm is expensive, the CSS method becomes an economical alternative for aeroelastic applications with heavy and stiff structures.

- Comparing the lift forces coefficients estimated using one-way and two-way coupled simulations, it was observed that the motion-dependent forces (aeroelastic effects) were significant for the structures vibrating substantially, as demonstrated in the first numerical example.

- The proposed framework is successfully applied to simulate the aeroelastic response of 60-storey standard tall building. The FSI simulations were able to capture the response of the structure in its fundamental and higher modes of vibration. Compared to the cross-wind oscillation of a circular cylinder, the dynamic response of the building does not show significant aeroelastic effects. However, this study clearly showed the capabilities of the developed FSI framework for simulating the wind-induced response of tall buildings directly, including aeroelastic effects such as aerodynamics damping.

The key contribution of this study was the development of a high-fidelity fluid-structure interaction framework for the computational aeroelastic modeling of tall buildings. The developed framework is named `windFSI` and operates seamlessly with the existing software architecture of OpenFOAM. The software architecture is versatile and can easily be extended to simulate various wind-structure interaction phenomena with complex vibration modes and non-linear material properties, which can be challenging to do experimentally. Also, the input to `windFSI` framework is highly flexible to enable the user to easily modify properties of the wind and structure sub-systems for parametric study. Overall, being in the age of high-performance computing, computational frameworks of this kind enable us to tackle complex wind-structure interaction problems in an efficient and integrated manner.

7.5 Future research directions

Although the proposed framework is primarily developed and validated for tall buildings, it can generally be applied to simulate wind loads on other structures. Especially the inflow turbulence generation method presented in Chapters 2 and 3 is broadly applicable to many CWE studies that require modeling of ABL turbulence. Also, the FSI framework developed for tall buildings in Chapter 6 can be extended (with little effort) for the aeroelastic modeling of structures that experience significant motion-induced forces (e.g., long-span bridges, flexible roof systems, and cooling towers).

After all, it is important to note that the studies presented in this thesis are not without limitations. The most noteworthy limitations, potential improvements, and possible extensions of the present study include the following:

- One of the main challenges of using synthetic inflow generation methods is related to turbulence decay associated with the downstream evolution of the flow. Since reproducing the phase information (eddy structure) of the target ABL flow is challenging for synthetic methods, the turbulence applied at the inlet experiences some decay. Especially for lateral and vertical components of the velocity, the decay can be relatively significant. This challenge is noted in Sections 2.5 and 4.5.1. In the current study, to compensate for this decay, the turbulence intensities at the inlet were iteratively adjusted by a height-dependent factor until the target incident profiles were matched. The iterative adjustment is made using a trial-and-error approach. In the future, more efficient inflow optimization methods can be incorporated to address this problem. This will reduce the wind profile optimization iterations that require running empty domain simulations multiple times.

- Furthermore, it is expected that the eddy structure of the upcoming turbulent flow can influence the wind loads on the building. Future research efforts that compare higher-order velocity correlations and coherent structures of the approaching flow may be required to determine whether the flow structures observed in the boundary layer wind tunnels are accurately reproduced by the LES model. Note that validation of the LES at this level of detail always needs to be accompanied by advanced experimental flow measurement techniques often used in the field of experimental fluid dynamics.

- In Chapters 4 and 5, for the large eddy simulation of the wind loads, the standard Smagorinsky subgrid-scale stress (SGS) model is primarily used. For bluff body aerodynamics, the behavior of the separated shear layers and attendant reattachment depend on small-scale turbulence structures, which can be affected by the SGS model employed. Thus, future research needs to systematically evaluate the performance of different SGS models for wind load evaluation. This investigation must also be supported by detailed experimental flow field measurements such as Particle Image Velocimetry (PIV) in a boundary layer wind tunnel.

- In Section 6.2.1.2 of the proposed FSI framework, the structure is modeled employing simplified modal representation. For buildings with complex structural configurations, the structural model used in the FSI framework should be upgraded with a full finite element model. Thus, incorporating more sophisticated open-source structural solvers

routinely used in the structural engineering community like OpenSees (McKenna, 1997) constitutes future research direction.

- For FSI validation cases presented in Section 6.4, the time step of the aeroelastic simulation is often dictated by the fluid subsystem. This entails the time step for the structural solver to be very small; hence, the structure undergoes only small deformation in a single time step. As a result, solving the structure and computing the dynamic mesh around it at every time step increases the overall computational cost of the FSI simulation. Thus, the computational efficiency of the proposed framework can be improved using unequal time steps (i.e., setting sub-cycling option) for the fluid and structure subsystems according to their respective numerical stability and accuracy requirements.

Bibliography

Aas-Jakobsen, K., Strømmen, E., 2001. Time domain buffeting response calculations of slender structures. Journal of Wind Engineering and Industrial Aerodynamics 89, 341–364.

Abdi, D., Bitsuamlak, G.T., 2014a. Numerical evaluation of the effect of multiple roughness changes. Wind and Structures 19, 585–601.

Abdi, D.S., Bitsuamlak, G.T., 2014b. Wind flow simulations on idealized and real complex terrain using various turbulence models. Advances in Engineering Software 75, 30–41.

Aboshosha, H., Bitsuamlak, G., El Damatty, A., 2015a. Les of abl flow in the built-environment using roughness modeled by fractal surfaces. Sustainable Cities and Society 19, 46–60.

Aboshosha, H., Bitsuamlak, G., El Damatty, A., 2015b. Turbulence characterization of downbursts using les. Journal of Wind Engineering and Industrial Aerodynamics 136, 44–61.

Aboshosha, H., Elshaer, A., Bitsuamlak, G.T., El Damatty, A., 2015c. Consistent inflow turbulence generator for les evaluation of wind-induced responses for tall buildings. Journal of Wind Engineering and Industrial Aerodynamics 142, 198–216.

Adamek, K., Vasan, N., Elshaer, A., English, E., Bitsuamlak, G., 2017. Pedestrian level wind assessment through city development: A study of the financial district in toronto. Sustainable cities and society 35, 178–190.

Affandi, R.H., Kulesza, A., Fox, E., Taskar, B., 2013. Nystrom approximation for large-scale determinantal processes, in: Artificial Intelligence and Statistics, PMLR. pp. 85–98.

Anderson, E., Bai, Z., Bischof, C., Blackford, L.S., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., et al., 1999a. LAPACK Users' guide. SIAM.

Anderson, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., Sorensen, D., 1999b. LAPACK Users' Guide. Third ed., Society for Industrial and Applied Mathematics, Philadelphia, PA.

Arcolano, N., Wolfe, P.J., 2010. Nyström approximation of wishart matrices, in: 2010 IEEE International Conference on Acoustics, Speech and Signal Processing, IEEE. pp. 3606–3609.

Armero, F., Simo, J., 1992. A new unconditionally stable fractional step method for non-linear coupled thermomechanical problems. International Journal for numerical methods in Engineering 35, 737–766.

ASCE-49-21, 2022. Wind tunnel testing for buildings and other structures, in: ASCE-7, American Society of Civil Engineers. pp. 48–49.

Baker, C.T., 1977. The numerical treatment of integral equations. Oxford University Press.

Bazilevs, Y., Calo, V.M., Hughes, T.J., Zhang, Y., 2008. Isogeometric fluid-structure interaction: theory, algorithms, and computations. Computational mechanics 43, 3–37.

Belabbas, M.A., Wolfe, P.J., 2009. Spectral methods in machine learning and new strategies for very large datasets. Proceedings of the National Academy of Sciences 106, 369–374.

Belloli, M., Giappino, S., Muggiasca, S., Zasso, A., 2012. Force and wake analysis on a single circular cylinder subjected to vortex induced vibrations at high mass ratio and high reynolds number. Journal of wind engineering and industrial aerodynamics 103, 96–106.

Benjamin, T.K.J., Hesse, H., Wang, P.C., 2021. Wing-tail interaction under forced harmonic pitch, in: AIAA AVIATION 2021 FORUM, p. 2517.

Benowitz, B.A., Deodatis, G., 2015. Simulation of wind velocities on long span structures: A novel stochastic wave based model. Journal of wind engineering and industrial aerodynamics 147, 154–163.

Birhane, T., Bitsuamlak, G., King, J., 2017. A computational framework for the aerodynamic shape optimization of long-span bridge decks, in: Structures Congress 2017, pp. 223–239.

Blocken, B., 2014a. 50 years of computational wind engineering: past, present and future. Journal of wind engineering and industrial aerodynamics 129, 69–102.

Blocken, B., 2014b. 50 years of computational wind engineering: past, present and future. Journal of Wind Engineering and Industrial Aerodynamics 129, 69–102.

Blocken, B., Carmeliet, J., 2004. A review of wind-driven rain research in building science. Journal of wind engineering and industrial aerodynamics 92, 1079–1130.

BLWTL, 2007. Wind tunnel testing: a general outline. Technical Report. The Boundary Layer Wind Tunnel Laboratory, The University of Western Ontario, Faculty of Engineering Science. London, Ontario, Canada. URL: www.blwtl.uwo.ca.

Boggs, D.W., 1991. Wind loading and response of tall structures using aerodynamic models. Colorado State University.

Bou-Zeid, E., Meneveau, C., Parlange, M., 2005. A scale-dependent lagrangian dynamic model for large eddy simulation of complex turbulent flows. Physics of fluids 17, 025105.

Braun, A.L., Awruch, A.M., 2009. Aerodynamic and aeroelastic analyses on the caarc standard tall building model using numerical simulation. Computers & Structures 87, 564–581.

Cao, Y., Xiang, H., Zhou, Y., 2000. Simulation of stochastic wind velocity field on long-span bridges. Journal of Engineering Mechanics 126, 1–6.

Carassale, L., Solari, G., 2002. Wind modes for structural dynamics: a continuous approach. Probabilistic Engineering Mechanics 17, 157–166.

Carassale, L., Solari, G., 2006. Monte carlo simulation of wind velocity fields on complex structures. Journal of Wind Engineering and Industrial Aerodynamics 94, 323–339.

Caretto, L., Curr, R., Spalding, D., 1972. Two numerical methods for three-dimensional boundary layers. Computer Methods in Applied Mechanics and Engineering 1, 39–57.

Carmo, B.S., Sherwin, S.J., Bearman, P.W., Willden, R., 2011. Flow-induced vibration of a circular cylinder subjected to wake interference at low reynolds number. Journal of Fluids and Structures 27, 503–522.

Castro, H.G., Paz, R.R., 2013. A time and space correlated turbulence synthesis method for large eddy simulations. Journal of Computational Physics 235, 742–763.

Castro, H.G., Paz, R.R., Mroginski, J.L., Storti, M.A., 2017. Evaluation of the proper coherence representation in random flow generation based methods. Journal of Wind Engineering and Industrial Aerodynamics 168, 211–227.

Cesur, A., Carlsson, C., Feymark, A., Fuchs, L., Revstedt, J., 2014. Analysis of the wake dynamics of stiff and flexible cantilever beams using pod and dmd. Computers & Fluids 101, 27–41.

Chen, H., Christensen, E.D., 2018. Simulating the hydrodynamic response of a floater–net system in current and waves. Journal of Fluids and Structures 79, 50–75.

Chen, J., Song, Y., Peng, Y., Spanos, P.D., 2018. Simulation of homogeneous fluctuating wind field in two spatial dimensions via a joint wave number–frequency power spectrum. Journal of Engineering Mechanics 144, 04018100.

Chen, X., 2008. Analysis of alongwind tall building response to transient nonstationary winds. Journal of structural engineering 134, 782–791.

Chen, X., Kareem, A., 2005. Proper orthogonal decomposition-based modeling, analysis, and simulation of dynamic wind load effects on structures. Journal of Engineering Mechanics 131, 325–339.

Cheng, W.C., Porté-Agel, F., 2013. Evaluation of subgrid-scale models in large-eddy simulation of flow past a two-dimensional block. International journal of heat and fluid flow 44, 301–311.

Chopra, A.K., 2007. Dynamics of structures. Pearson Education India.

Churchfield, M.J., Vijayakumar, G., Brasseur, J.G., Moriarty, P.J., 2010. Wind energy-related atmospheric boundary layer large-eddy simulation using OpenFOAM. Technical Report. National Renewable Energy Lab.(NREL), Golden, CO (United States).

Cochran, L., Derickson, R., 2011. A physical modeler's view of computational wind engineering. Journal of Wind Engineering and Industrial Aerodynamics 99, 139–153.

Cook, N., Mayne, J., 1979. A novel working approach to the assessment of wind loads for equivalent static design. Journal of Wind Engineering and Industrial Aerodynamics 4, 149–164.

Cook, N.J., 1997. The deaves and harris abl model applied to heterogeneous terrain. Journal of wind engineering and industrial aerodynamics 66, 197–214.

Dagnew, A., Bitsuamlak, G.T., 2013. Computational evaluation of wind loads on buildings: a review. Wind Struct 16, 629–660.

Dagnew, A.K., Bitsuamlak, G.T., 2014. Computational evaluation of wind loads on a standard tall building using les. Wind and Structures 18, 567–598.

Dalgliesh, W.A., 1975. Comparison of model/full-scale wind pressures on a high-rise building. Journal of Wind Engineering and Industrial Aerodynamics 1, 55–66.

Daniels, S.J., Castro, I.P., Xie, Z.T., 2013. Peak loading and surface pressure fluctuations of a tall model building. Journal of wind engineering and industrial aerodynamics 120, 19–28.

Davenport, A., 1971. The response of six building shapes to turbulent wind. Philosophical Transactions of the Royal Society of London. Series A, Mathematical and Physical Sciences 269, 385–394.

Davenport, A., 1988. The response of supertall buildings to wind. second century of the skyscraper (edited by cs beedle), council on tall buildings and the urban habitat.

Davenport, A.G., 1961a. The application of statistical concepts to the wind loading of structures. Proceedings of the Institution of Civil Engineers 19, 449–472.

Davenport, A.G., 1961b. The spectrum of horizontal gustiness near the ground in high winds. Quarterly Journal of the Royal Meteorological Society 87, 194–211.

Davenport, A.G., 1964. Note on the distribution of the largest value of a random function with application to gust loading. Proceedings of the Institution of Civil Engineers 28, 187–196.

Davenport, A.G., 1965. The buffeting of structures by gusts, in: Proc. of Conference on'Wind Effects on Structures', NPL, 1965 (ICWE-1), HMSO. p. 1.

Davenport, A.G., 1967. Gust loading factors. Journal of the Structural Division 93, 11–34.

Davenport, A.G., Isyumov, N., 1967. The application of the boundary layer wind tunnel to the prediction of wind loading, in: Proceedings of the International Research Seminar: Wind Effects on Buildings and Structures. Ottawa, Canada. September, pp. 11–15.

De Silva, V., Tenenbaum, J.B., 2004. Sparse multidimensional scaling using landmark points. Technical Report. technical report, Stanford University.

Deaves, D., Harris, R., 1978. A mathematical model of the structure of strong winds. CIRIA Report 76, Const. Ind. Research and Inf. Assoc. .

Degroote, J., Bathe, K.J., Vierendeels, J., 2009. Performance of a new partitioned procedure versus a monolithic procedure in fluid–structure interaction. Computers & Structures 87, 793–801.

Demirdžić, I., Perić, M., 1988. Space conservation law in finite volume calculations of fluid flow. International journal for numerical methods in fluids 8, 1037–1050.

Demirdžić, I., Perić, M., 1990. Finite volume method for prediction of fluid flow in arbitrarily shaped domains with moving boundaries. International journal for numerical methods in fluids 10, 771–790.

Deodatis, G., 1996a. Non-stationary stochastic vector processes: seismic ground motion applications. Probabilistic Engineering Mechanics 11, 149–167.

Deodatis, G., 1996b. Simulation of ergodic multivariate stochastic processes. Journal of engineering mechanics 122, 778–787.

Deparis, S., Fernández, M.A., Formaggia, L., 2003. Acceleration of a fixed point algorithm for fluid-structure interaction using transpiration conditions. ESAIM: Mathematical Modelling and Numerical Analysis 37, 601–616.

Dhamankar, N.S., Blaisdell, G.A., Lyrintzis, A.S., 2018. Overview of turbulent inflow boundary conditions for large-eddy simulations. Aiaa Journal 56, 1317–1334.

Di Mare, L., Klein, M., Jones, W., Janicka, J., 2006. Synthetic turbulence inflow conditions for large-eddy simulation. Physics of Fluids 18, 025107.

Di Paola, M., 1998. Digital simulation of wind field velocity. Journal of Wind Engineering and Industrial Aerodynamics 74, 91–109.

Di Paola, M., Gullo, I., 2001. Digital generation of multivariate wind field processes. Probabilistic Engineering Mechanics 16, 1–10.

Ding, Q., Zhu, L., Xiang, H., 2006. Simulation of stationary gaussian stochastic wind velocity field. Wind and Structures 9, 231–243.

Ding, Q., Zhu, L., Xiang, H., 2011. An efficient ergodic simulation of multivariate stochastic processes with spectral representation. Probabilistic Engineering Mechanics 26, 350–356.

Dowell, E.H., Curtiss, H., Scanlan, R.H., Sisto, F., 2021. A modern course in aeroelasticity. Springer.

Driest, E.V., 1956. On turbulent flow near a wall. Journal of the aeronautical sciences 23, 1007–1011.

Elshaer, A., Aboshosha, H., Bitsuamlak, G., El Damatty, A., Dagnew, A., 2016. Les evaluation of wind-induced responses for an isolated and a surrounded tall building. Engineering Structures 115, 179–195.

Elshaer, A., Bitsuamlak, G., 2018. Multiobjective aerodynamic optimization of tall building openings for wind-induced load reduction. Journal of Structural Engineering 144, 04018198.

Elshaer, A., Bitsuamlak, G., El Damatty, A., 2017a. Enhancing wind performance of tall buildings using corner aerodynamic optimization. Engineering Structures 136, 133–148.

Elshaer, A., Gairola, A., Adamek, K., Bitsuamlak, G., 2017b. Variations in wind load on tall buildings due to urban development. Sustainable cities and society 34, 264–277.

ESDU, 2001a. Strong winds in the atmospheric boundary layer. part 1: hourly-mean wind speeds. Data Item 82026 1.

ESDU, 2001b. Strong winds in the atmospheric boundary layer. part 2: discrete gust speeds. Data Item 83045 1.

ESDU-85020, 2001. Characteristics of atmospheric turbulence near the ground. Part II: single point data for strong winds (neutral atmosphere).. volume 1.

Farhat, C., Lesoinne, M., 2000. Two efficient staggered algorithms for the serial and parallel solution of three-dimensional nonlinear transient aeroelastic problems. Computer methods in applied mechanics and engineering 182, 499–515.

Farhat, C., Park, K., Dubois-Pelerin, Y., 1991. An unconditionally stable staggered algorithm for transient finite element analysis of coupled thermoelastic problems. Computer methods in applied mechanics and engineering 85, 349–365.

Farhat, C., Van der Zee, K.G., Geuzaine, P., 2006. Provably second-order time-accurate loosely-coupled solution algorithms for transient nonlinear computational aeroelasticity. Computer methods in applied mechanics and engineering 195, 1973–2001.

Felippa, C.A., Park, K.C., 1980. Staggered transient analysis procedures for coupled mechanical systems: formulation. Computer Methods in Applied Mechanics and Engineering 24, 61–111.

Felippa, C.A., Park, K.C., Farhat, C., 2001. Partitioned analysis of coupled mechanical systems. Computer methods in applied mechanics and engineering 190, 3247–3270.

Ferziger, J.H., Peric, M., 2012. Computational methods for fluid dynamics. Springer Science & Business Media.

Ferziger, J.H., Perić, M., Street, R.L., 2002. Computational methods for fluid dynamics. volume 3. Springer.

Franke, J., 2006. Recommendations of the cost action c14 on the use of cfd in predicting pedestrian wind environment, in: The fourth international symposium on computational wind engineering, Yokohama, Japan, pp. 529–532.

Franke, J., Hellsten, A., Schlunzen, K.H., Carissimo, B., 2011. The cost 732 best practice guideline for cfd simulation of flows in the urban environment: a summary. International Journal of Environment and Pollution 44, 419–427.

Frigo, M., Johnson, S.G., 2005a. The design and implementation of FFTW3. Proceedings of the IEEE 93, 216–231. Special issue on "Program Generation, Optimization, and Platform Adaptation".

Frigo, M., Johnson, S.G., 2005b. The design and implementation of fftw3. Proceedings of the IEEE 93, 216–231.

Fureby, C., 2008. Towards the use of large eddy simulation in engineering. Progress in Aerospace Sciences 44, 381–396.

Gairola, A., Bitsuamlak, G., 2019. Numerical tornado modeling for common interpretation of experimental simulators. Journal of Wind Engineering and Industrial Aerodynamics 186, 32–48.

Gallinger, T., Kupzok, A., Israel, U., Bletzinger, K., Wüchner, R., 2009. A computational environment for membrane-wind interaction, in: International Workshop on Fluid-Structure Interaction. Theory, Numerics and Applications, kassel university press GmbH. p. 97.

Gao, Y., Wu, Y., Li, D., Liu, H., Zhang, N., 2012. An improved approximation for the spectral representation method in the simulation of spatially varying ground motions. Probabilistic Engineering Mechanics 29, 7–15.

Gerbeau, J.F., Vidrascu, M., Frey, P., 2005. Fluid–structure interaction in blood flows on geometries based on medical imaging. Computers & Structures 83, 155–165.

Gousseau, P., Blocken, B., Stathopoulos, T., Van Heijst, G., 2011. Cfd simulation of near-field pollutant dispersion on a high-resolution grid: a case study by les and rans for a building group in downtown montreal. Atmospheric Environment 45, 428–438.

Greenshields, C., Weller, H., 2022a. Notes on Computational Fluid Dynamics: General Principles. CFD Direct Ltd, Reading, UK.

Greenshields, C., Weller, H., 2022b. Notes on computational fluid dynamics: General principles. CFD Direct Ltd.: Reading, UK .

Greenshields, C.J., et al., 2015. Openfoam user guide. OpenFOAM Foundation Ltd, version 3, 47.

Gresho, P.M., Sani, R.L., 1987. On pressure boundary conditions for the incompressible navier-stokes equations. International Journal for Numerical Methods in Fluids 7, 1111–1145.

Grötzbach, 1987. Direct numerical and large eddy simulation of turbulent channel flows. In Encyclopedia of Fluid Mechanics 1, 1337–1391.

Gungor, A., Sillero, J., Jiménez, J., 2012. Pressure statistics from direct simulation of turbulent boundary layer, in: Seventh int conf compt fluid dyn (Hawaii), Citeseer. pp. 1–6.

Halton, J.H., 1960. On the efficiency of certain quasi-random sequences of points in evaluating multi-dimensional integrals. Numerische Mathematik 2, 84–90.

Hammersley, J.M., 1960. Monte carlo methods for solving multivariable problems. Annals of the New York Academy of Sciences 86, 844–874.

He, L., Zhang, H., 2018. Kernel k-means sampling for nyström approximation. IEEE Transactions on Image Processing 27, 2108–2120.

Hémon, P., Santi, F., 2007. Simulation of a spatially correlated turbulent velocity field using biorthogonal decomposition. Journal of wind engineering and industrial aerodynamics 95, 21–29.

Hewitt, S., Margetts, L., Revell, A., Pankaj, P., Levrero-Florencio, F., 2019. Openfpci: A parallel fluid–structure interaction framework. Computer Physics Communications 244, 469–482.

Ho, T., Surry, D., Morrish, D., Kopp, G., 2005. The uwo contribution to the nist aerodynamic database for wind loads on low buildings: Part 1. archiving format and basic aerodynamic data. Journal of Wind Engineering and Industrial Aerodynamics 93, 1–30.

Holmes, J., 1975. Pressure fluctuations on a large building and along-wind structural loading. Journal of Wind Engineering and Industrial Aerodynamics 1, 249–278.

Holmes, J.D., 2007. Wind loading of structures. CRC press.

Hou, G., Wang, J., Layton, A., 2012. Numerical methods for fluid-structure interaction—a review. Communications in Computational Physics 12, 337–377.

Huang, G., Liao, H., Li, M., 2013. New formulation of cholesky decomposition and applications in stochastic simulation. Probabilistic Engineering Mechanics 34, 40–47.

Huang, S., Li, Q., Wu, J., 2010. A general inflow turbulence generator for large eddy simulation. Journal of Wind Engineering and Industrial Aerodynamics 98, 600–617.

Huang, S., Li, Q.S., Xu, S., 2007. Numerical evaluation of wind effects on a tall steel building by cfd. Journal of Constructional Steel Research 63, 612–627.

Hübner, B., Walhorn, E., Dinkler, D., 2004. A monolithic approach to fluid–structure interaction using space–time finite elements. Computer methods in applied mechanics and engineering 193, 2087–2104.

Hughes, T.J., Liu, W.K., Zimmermann, T.K., 1981. Lagrangian-eulerian finite element formulation for incompressible viscous flows. Computer methods in applied mechanics and engineering 29, 329–349.

Irwin, P., 1982. Model studies of the dynamic response of tall buildings to wind, in: Canadian Society for Civil Engineering Annual Conference,, Edmonton, Alberta, pp. 285–302.

Irwin, P., Denoon, R., Scott, D., 2013. Wind tunnel testing of high-rise buildings. Routledge.

Irwin, P.A., 2009. Wind engineering challenges of the new generation of super-tall buildings. Journal of Wind Engineering and Industrial Aerodynamics 97, 328–334.

Issa, R.I., 1986. Solution of the implicitly discretised fluid flow equations by operator-splitting. Journal of computational physics 62, 40–65.

Isymov, N., 1982. The aeroelastic modeling of tall buildings. Wind tunnel modeling for civil engineering applications , 373–456.

Jarrin, N., Benhamadouche, S., Laurence, D., Prosser, R., 2006. A synthetic-eddy-method for generating inflow conditions for large-eddy simulations. International Journal of Heat and Fluid Flow 27, 585–593.

Jasak, H., 1996. Error analysis and estimation for the finite volume method with applications to fluid flows. Ph.D. thesis. Imperial College London (University of London).

Jasak, H., 2009. Dynamic mesh handling in openfoam, in: 47th AIAA aerospace sciences meeting including the new horizons forum and aerospace exposition, p. 341.

Jasak, H., Tukovic, Z., 2006. Automatic mesh motion for the unstructured finite volume method. Transactions of FAMENA 30, 1–20.

Jasak, H., Tuković, Ž., 2010. Dynamic mesh handling in openfoam applied to fluid-structure interaction simulations, in: Proceedings of the V European Conference on Computational Fluid Dynamics ECCOMAS CFD 2010, pp. 1–19.

Jørgensen, N.G., Koss, H., Bennetsen, J.C., 2012. Embedded-les and experiment of turbulent boundary layer flow around a floor-mounted cube, in: The Seventh International Colloquium on Bluff Body Aerodynamics and Applications, pp. 1–10.

Kahsay, M.T., Bitsuamlak, G.T., Tariku, F., 2021. Thermal zoning and window optimization framework for high-rise buildings. Applied Energy 292, 116894.

Kakimpa, B., Hargreaves, D., Owen, J., Martinez-Vazquez, P., Baker, C., Sterling, M., Quinn, A., 2010. Cfd modelling of free-flight and auto-rotation of plate type debris. Wind and Structures 13, 169.

Kareem, A., 2008. Numerical simulation of wind effects: a probabilistic perspective. Journal of Wind Engineering and Industrial Aerodynamics 96, 1472–1497.

Kim, J.W., Haeri, S., 2015. An advanced synthetic eddy method for the computation of aerofoil–turbulence interaction noise. Journal of Computational Physics 287, 1–17.

Kim, Y., Castro, I.P., Xie, Z.T., 2013. Divergence-free turbulence inflow conditions for large-eddy simulations with incompressible flow solvers. Computers & Fluids 84, 56–68.

Klein, M., Sadiki, A., Janicka, J., 2003. A digital filter based generation of inflow data for spatially developing direct numerical or large eddy simulations. Journal of computational Physics 186, 652–665.

Kondo, K., Murakami, S., Mochida, A., 1997. Generation of velocity fluctuations for inflow boundary condition of les. Journal of Wind Engineering and Industrial Aerodynamics 67, 51–64.

Kornev, N., Hassel, E., 2007. Method of random spots for generation of synthetic inhomogeneous turbulent fields with prescribed autocorrelation functions. Communications in numerical methods in engineering 23, 35–43.

Kornev, N., Kröger, H., Hassel, E., 2008. Synthesis of homogeneous anisotropic turbulent fields with prescribed second-order statistics by the random spots method. Communications in numerical methods in engineering 24, 875–877.

Kraichnan, R.H., 1970. Diffusion by a random velocity field. The physics of fluids 13, 22–31.

Kröger, H., Kornev, N., 2018. Generation of divergence free synthetic inflow turbulence with arbitrary anisotropy. Computers & Fluids 165, 78–88.

Kumar, S., Mohri, M., Talwalkar, A., 2009. Sampling techniques for the nystrom method, in: Artificial Intelligence and Statistics, pp. 304–311.

Kumar, S., Mohri, M., Talwalkar, A., 2012. Sampling methods for the nyström method. The Journal of Machine Learning Research 13, 981–1006.

Küttler, U., Wall, W.A., 2008. Fixed-point fluid–structure interaction solvers with dynamic relaxation. Computational mechanics 43, 61–72.

Lamberti, G., García-Sánchez, C., Sousa, J., Gorlé, C., 2018. Optimizing turbulent inflow conditions for large-eddy simulations of the atmospheric boundary layer. Journal of Wind Engineering and Industrial Aerodynamics 177, 32–44.

Lamberti, G., Gorlé, C., 2020. Sensitivity of les predictions of wind loading on a high-rise building to the inflow boundary condition. Journal of Wind Engineering and Industrial Aerodynamics 206, 104370.

Lamberti, G., Gorlé, C., 2021. A multi-fidelity machine learning framework to predict wind loads on buildings. Journal of Wind Engineering and Industrial Aerodynamics 214, 104647.

Lee, S., Lele, S.K., Moin, P., 1992. Simulation of spatially evolving turbulence and the applicability of taylor's hypothesis in compressible flow. Physics of Fluids A: Fluid Dynamics 4, 1521–1530.

Lewellen, W., Lewellen, D., Sykes, R., 1997. Large-eddy simulation of a tornado's interaction with the surface. Journal of the atmospheric sciences 54, 581–605.

Li, J., Li, C., Chen, S., 2011. Spline-interpolation-based fft approach to fast simulation of multivariate stochastic processes. Mathematical Problems in Engineering 2011.

Li, Y., Kareem, A., 1991. Simulation of multivariate nonstationary random processes by fft. Journal of Engineering Mechanics 117, 1037–1058.

Li, Y., Kareem, A., 1993. Simulation of multivariate random processes: Hybrid dft and digital filtering approach. Journal of Engineering Mechanics 119, 1078–1098.

Li, Y., Kareem, A., 1995. Stochastic decomposition and application to probabilistic dynamics. Journal of engineering mechanics 121, 162–174.

Li, Y.C., Cheng, C.M., Lo, Y.L., Fang, F.M., Zheng, D.q., 2015. Simulation of turbulent flows around a prism in suburban terrain inflow based on random flow generation method simulation. Journal of Wind Engineering and Industrial Aerodynamics 146, 51–58.

Lieblein, J., 1976. Efficient methods of extreme-value methodology. Technical Report. Technical Analysis Div., Institute for Applied Technology, National Bureau: of Standards, Washington, D.C. (United States).

Lilly, D.K., 1967. The representation of small-scale turbulence in numerical simulation experiments. IBM Form , 195–210.

Liu, R., Jain, V., Zhang, H., 2006. Sub-sampling for efficient spectral mesh processing, in: Computer Graphics International Conference, Springer. pp. 172–184.

Lu, C., Li, Q., Huang, S., Chen, F., Fu, X., 2012. Large eddy simulation of wind effects on a long-span complex roof structure. Journal of Wind Engineering and Industrial Aerodynamics 100, 1–18.

Lumley, J., Panofsky, H., 1964. The structure of atmospheric turbulence, intersci. Monogr. Texts Phys. Astron 12, 19–23.

Lund, T.S., Wu, X., Squires, K.D., 1998. Generation of turbulent inflow data for spatially-developing boundary layer simulations. Journal of computational physics 140, 233–258.

Mahaffy, J., Chung, B., Song, C., Dubois, F., Graffard, E., Ducros, F., Heitsch, M., Scheuerer, M., Henriksson, M., Komen, E., et al., 2007. Best practice guidelines for the use of CFD in nuclear reactor safety applications. Technical Report. Organisation for Economic Co-Operation and Development.

Mann, J., 1998. Wind field simulation. Probabilistic engineering mechanics 13, 269–282.

Marshall, J., Imregun, M., 1996. An analysis of the aeroelastic behaviour of a typical fan-blade with emphasis on the flutter mechanism. volume 78767. American Society of Mechanical Engineers.

Mathey, F., Cokljat, D., Bertoglio, J.P., Sergent, E., 2006. Assessment of the vortex method for large eddy simulation inlet conditions. Progress in Computational Fluid Dynamics, An International Journal 6, 58–67.

McKenna, F.T., 1997. Object-oriented finite element programming: frameworks for analysis, algorithms and parallel computing. University of California, Berkeley.

Melaku, A., Bitsuamlak, G., Elshaer, A., Aboshosha, H., 2017. Synthetic inflow turbulence generation methods for les study of tall building aerodynamics, in: 13th Americas Conference on Wind Engineering, pp. 1–16.

Melaku, A.F., Bitsuamlak, G.T., 2021. A divergence-free inflow turbulence generator using spectral representation method for large-eddy simulation of abl flows. Journal of Wind Engineering and Industrial Aerodynamics 212, 104580.

Melaku, A.F., Doddipatla, L.S., Bitsuamlak, G.T., 2022. Large-eddy simulation of wind loads on a roof-mounted cube: application for interpolation of experimental aerodynamic data. Journal of Wind Engineering and Industrial Aerodynamics 231, 105230.

Melbourne, W., 1980. Comparison of measurements on the caarc standard tall building model in simulated model wind flows. Journal of Wind Engineering and Industrial Aerodynamics 6, 73–88.

Moeng, C.H., Wyngaard, J.C., 1988. Spectral analysis of large-eddy simulations of the convective boundary layer. Journal of Atmospheric Sciences 45, 3573–3587.

Morokoff, W.J., Caflisch, R.E., 1994. Quasi-random sequences and their discrepancies. SIAM Journal on Scientific Computing 15, 1251–1279.

Morooka, C.K., Yokoo, I.H., et al., 1997. Numerical simulation and spectral analysis of irregular sea waves. International Journal of Offshore and Polar Engineering 7.

Murakami, S., 1990. Computational wind engineering. Journal of wind engineering and industrial aerodynamics 36, 517–538.

Murakami, S., 1993. Comparison of various turbulence models applied to a bluff body, in: Computational Wind Engineering 1. Elsevier, pp. 21–36.

Murakami, S., 1998. Overview of turbulence models applied in cwe–1997. Journal of Wind Engineering and Industrial Aerodynamics 74, 1–24.

Newberry, C.W., 1967. The nature of gust loading on tall building, in: Proc. 2nd International Conf. on Wind Effects on Buildings and Structures, pp. 399–428.

Newmark, N.M., 1959. A method of computation for structural dynamics. Journal of the engineering mechanics division 85, 67–94.

Nicoud, F., Ducros, F., 1999. Subgrid-scale stress modelling based on the square of the velocity gradient tensor. Flow, turbulence and Combustion 62, 183–200.

Nozawa, K., Tamura, T., 2002. Large eddy simulation of the flow around a low-rise building immersed in a rough-wall turbulent boundary layer. Journal of Wind Engineering and Industrial Aerodynamics 90, 1151–1162.

Nozu, T., Tamura, T., Okuda, Y., Sanada, S., 2008. Les of the flow and building wall pressures in the center of tokyo. Journal of Wind Engineering and Industrial Aerodynamics 96, 1762–1773.

Nozu, T., Tamura, T., Takeshi, K., Akira, K., 2015. Mesh-adaptive les for wind load estimation of a high-rise building in a city. Journal of Wind Engineering and Industrial Aerodynamics 144, 62–69.

Obasaju, E., 1992. Measurement of forces and base overturning moments on the caarc tall building model in a simulated atmospheric boundary layer. Journal of Wind Engineering and Industrial Aerodynamics 40, 103–126.

Oberkampf, W.L., Trucano, T.G., 2002. Verification and validation in computational fluid dynamics. Progress in aerospace sciences 38, 209–272.

Pamiès, M., Weiss, P.E., Garnier, E., Deck, S., Sagaut, P., 2009. Generation of synthetic turbulent inflow data for large eddy simulation of spatially evolving wall-bounded flows. Physics of Fluids 21, 045103.

Park, S., Park, S., Yeo, D., 2018. Introductory tutorial for DAD: Design examples of high-rise building for wind. US Department of Commerce, National Institute of Standards and Technology.

Patruno, L., de Miranda, S., 2020. Unsteady inflow conditions: A variationally based solution to the insurgence of pressure fluctuations. Computer Methods in Applied Mechanics and Engineering 363, 112894.

Patruno, L., Ricci, M., 2018. A systematic approach to the generation of synthetic turbulence using spectral methods. Computer Methods in Applied Mechanics and Engineering 340, 881–904.

Pellegrini, F., 1994. Static mapping by dual recursive bipartitioning of process architecture graphs, in: Proceedings of IEEE Scalable High Performance Computing Conference, IEEE. pp. 486–493.

Peng, L., Huang, G., Kareem, A., Li, Y., 2016. An efficient space–time based simulation approach of wind velocity field with embedded conditional interpolation for unevenly spaced locations. Probabilistic Engineering Mechanics 43, 156–168.

Piomelli, U., 1999. Large-eddy simulation: achievements and challenges. Progress in aerospace sciences 35, 335–362.

Piomelli, U., Balaras, E., 2002. Wall-layer models for large-eddy simulations. Annual review of fluid mechanics 34, 349–374.

Piperno, S., Farhat, C., 2001. Partitioned procedures for the transient solution of coupled aeroelastic problems–part ii: energy transfer analysis and three-dimensional applications. Computer methods in applied mechanics and engineering 190, 3147–3170.

Piperno, S., Farhat, C., Larrouturou, B., 1995. Partitioned procedures for the transient solution of coupled aroelastic problems part i: Model problem, theory and two-dimensional application. Computer methods in applied mechanics and engineering 124, 79–112.

Poletto, R., Craft, T., Revell, A., 2013. A new divergence free synthetic eddy method for the reproduction of inlet flow conditions for les. Flow, turbulence and combustion 91, 519–539.

Poletto, R., Revell, A., Craft, T.J., Jarrin, N., 2011. Divergence free synthetic eddy method for embedded les inflow boundary conditions, in: TSFP Digital Library Online, Begel House Inc.. pp. 1–6.

Porté-Agel, F., Wu, Y.T., Lu, H., Conzemius, R.J., 2011. Large-eddy simulation of atmospheric boundary layer flow through wind turbines and wind farms. Journal of Wind Engineering and Industrial Aerodynamics 99, 154–168.

Priestley, M.B., 1981. Spectral analysis and time series: probability and mathematical statistics. ACADEMIC PRESS, INC.

Razak, A.A., Hagishima, A., Ikegaya, N., Tanimoto, J., 2013. Analysis of airflow over building arrays for assessment of urban wind environment. Building and Environment 59, 56–65.

Ricci, M., Patruno, L., De Miranda, S., 2017. Wind loads and structural response: benchmarking les on a low-rise building. Engineering Structures 144, 26–42.

Ricci, M., Patruno, L., Kalkman, I., de Miranda, S., Blocken, B., 2018. Towards les as a design tool: Wind loads assessment on a high-rise building. Journal of Wind Engineering and Industrial Aerodynamics 180, 1–18.

Romero, A., Galvín, P., Domínguez, J., 2013. 3d non-linear time domain fem–bem approach to soil–structure interaction problems. Engineering Analysis with Boundary Elements 37, 501–512.

Rosenblueth, E., 1951. A basis for aseismic design.

Ryzhakov, P., Rossi, R., Idelsohn, S., Onate, E., 2010. A monolithic lagrangian approach for fluid–structure interaction problems. Computational mechanics 46, 883–899.

Sagaut, P., Deck, S., 2009. Large eddy simulation for aerodynamics: status and perspectives. Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences 367, 2849–2860.

Schumann, U., 1975. Subgrid scale model for finite difference simulations of turbulent flows in plane channels and annuli. Journal of computational physics 18, 376–404.

Shah, K.B., Ferziger, J.H., 1997. A fluid mechanicians view of wind engineering: Large eddy simulation of flow past a cubic obstacle. Journal of wind engineering and industrial aerodynamics 67, 211–224.

Shinozuka, M., 1971. Simulation of multivariate and multidimensional random processes. The Journal of the Acoustical Society of America 49, 357–368.

Shinozuka, M., Deodatis, G., 1988. Stochastic process models for earthquake ground motion. Probabilistic engineering mechanics 3, 114–123.

Shinozuka, M., Deodatis, G., 1991a. Simulation of stochastic processes by spectral representation. Applied Mechanics Reviews .

Shinozuka, M., Deodatis, G., 1991b. Stochastic wave models for stationary and homogeneous seismic ground motion. Structural Safety 10, 235–246.

Shinozuka, M., Jan, C.M., 1972. Digital simulation of random processes and its applications. Journal of sound and vibration 25, 111–128.

Shinozuka, M., Yun, C.B., Seya, H., 1990. Stochastic methods in wind engineering. Journal of Wind Engineering and Industrial Aerodynamics 36, 829–843.

Silva, V., Tenenbaum, J., 2002. Global versus local methods in nonlinear dimensionality reduction. Advances in neural information processing systems 15.

Simiu, E., Gabbai, R.D., Fritz, W.P., 2008. Wind-induced tall building response: a time-domain approach. Wind & structures 11, 427–440.

Simiu, E., Scanlan, R.H., 1996. Wind effects on structures: Fundamentals and application to design. Book published by John Willey & Sons Inc 605.

Slone, A., Pericleous, K., Bailey, C., Cross, M., 2002. Dynamic fluid–structure interaction using finite volume unstructured mesh procedures. Computers & structures 80, 371–390.

Smagorinsky, J., 1963. General circulation experiments with the primitive equations: I. the basic experiment. Monthly weather review 91, 99–164.

Smirnov, A., Shi, S., Celik, I., 2001. Random flow generation technique for large eddy simulations and particle-dynamics modeling. Journal of fluids engineering 123, 359–371.

Sobol, I.M., 1967. On the distribution of points in a cube and the approximate evaluation of integrals. Zhurnal Vychislitel'noi Matematiki i Matematicheskoi Fiziki 7, 784–802.

Solari, G., Carassale, L., 2000. Modal transformation tools in structural dynamics and wind engineering. Wind and Structures 3, 221–241.

Solari, G., Piccardo, G., 2001. Probabilistic 3-d turbulence modeling for gust buffeting of structures. Probabilistic Engineering Mechanics 16, 73–86.

Solari, G., Tubino, F., 2002. A turbulence model based on principal components. Probabilistic engineering mechanics 17, 327–335.

Song, Y., Chen, J., Peng, Y., Spanos, P.D., Li, J., 2018. Simulation of nonhomogeneous fluctuating wind speed field in two-spatial dimensions via an evolutionary wavenumber-frequency joint power spectrum. Journal of Wind Engineering and Industrial Aerodynamics 179, 250–259.

Spalding, D., 1961. A single formula for the law of the wall. Journal of Applied Mechanics 28, 455–458.

Spanos, P., Zeldin, B., 1998. Monte carlo treatment of random fields: a broad perspective. Applied Mechanics Reviews 51, 219–237.

Stathopoulos, T., 1997. Computational wind engineering: Past achievements and future challenges. Journal of Wind Engineering and Industrial Aerodynamics 67, 509–532.

Stein, K., Benney, R., Kalro, V., Tezduyar, T.E., Leonard, J., Accorsi, M., 2000. Parachute fluid–structure interactions: 3-d computation. Computer Methods in Applied Mechanics and Engineering 190, 373–386.

Sullivan, P.P., McWilliams, J.C., Moeng, C.H., 1994. A subgrid-scale model for large-eddy simulation of planetary boundary-layer flows. Boundary-Layer Meteorology 71, 247–276.

Tabor, G.R., Baba-Ahmadi, M., 2010. Inlet conditions for large eddy simulation: A review. Computers & Fluids 39, 553–567.

Tamura, T., 2008. Towards practical use of les in wind engineering. Journal of Wind Engineering and Industrial Aerodynamics 96, 1451–1471.

Tamura, T., Kawai, H., Bale, R., Onishi, K., Tsubokura, M., Kondo, K., Nozu, T., 2015. Analysis of wind turbulence in canopy layer at large urban area using hpc database, in: ICUC9-9th International Conference on Urban Climate Jointly with 12th Symposium on the Urban Environment, pp. 1–6.

Tamura, T., Kondo, K., Kataoka, H., Ono, Y., Kawai, H., 2017. Application of les to wind loading estimation on buildings, in: 9th Asia Pacific Conference on Wind Engineering, APCWE 2017, The University of Auckland. pp. 1–4.

Tamura, T., Nozawa, K., Kondo, K., 2008. Aij guide for numerical prediction of wind loads on buildings. Journal of Wind Engineering and Industrial Aerodynamics 96, 1974–1984.

Tamura, T., Okuda, Y., Kishida, T., Nakamura, O., Miyashita, K., Katsumura, A., Tamari, M., 2010. Les for aerodynamic characteristics of a tall building inside a dense city district. CWE2010, pp1-8 5.

Tamura, T., Ono, Y., 2003. Les analysis on aeroelastic instability of prisms in turbulent flow. Journal of wind engineering and industrial aerodynamics 91, 1827–1846.

Tanaka, H., Tamura, Y., Ohtake, K., Nakai, M., Kim, Y.C., Bandi, E.K., 2013. Aerodynamic and flow characteristics of tall buildings with various unconventional configurations. International Journal of High-Rise Buildings 2, 213–228.

Tao, T., Wang, H., Yao, C., He, X., Kareem, A., 2017. Efficacy of interpolation-enhanced schemes in random wind field simulation over long-span bridges. Journal of Bridge Engineering 23, 04017147.

Tao, T., Wang, H., Yao, C., He, X., Kareem, A., 2018. Efficacy of interpolation-enhanced schemes in random wind field simulation over long-span bridges. Journal of Bridge Engineering 23, 04017147.

Taylor, G.I., 1938. The spectrum of turbulence. Proceedings of the Royal Society of London. Series A-Mathematical and Physical Sciences 164, 476–490.

Tennekes, H., 1973. The logarithmic wind profile. Journal of the Atmospheric Sciences 30, 234–238.

ASCE 7-16, 2017. Minimum design loads and associated criteria for buildings and other structures, in: ASCE/SEI 7-16, American Society of Civil Engineers, Reston, VA.. pp. 245–387.

Tezduyar, T.E., Sathe, S., Schwaab, M., Pausewang, J., Christopher, J., Crabtree, J., 2008. Fluid–structure interaction modeling of ringsail parachutes. Computational Mechanics 43, 133–142.

Thomas, P., Lombard, C., 1979. Geometric conservation law and its application to flow computations on moving grids. AIAA journal 17, 1030–1037.

Thomas, T., Williams, J., 1999. Generating a wind environment for large eddy simulation of bluff body flows. Journal of Wind Engineering and Industrial Aerodynamics 82, 189–208.

Thordal, M.S., Bennetsen, J.C., Koss, H.H.H., 2019. Review for practical application of cfd for the determination of wind load on high-rise buildings. Journal of Wind Engineering and Industrial Aerodynamics 186, 155–168.

Tomas, J., Pourquie, M., Jonker, H., 2015. The influence of an obstacle on flow and pollutant dispersion in neutral and stable boundary layers. Atmospheric Environment 113, 236–246.

Tominaga, Y., Mochida, A., Murakami, S., Sawaki, S., 2008a. Comparison of various revised k–ε models and les applied to flow around a high-rise building model with 1: 1: 2 shape placed within the surface boundary layer. Journal of Wind Engineering and Industrial Aerodynamics 96, 389–411.

Tominaga, Y., Mochida, A., Yoshie, R., Kataoka, H., Nozu, T., Yoshikawa, M., Shirasawa, T., 2008b. Aij guidelines for practical applications of cfd to pedestrian wind environment around buildings. Journal of wind engineering and industrial aerodynamics 96, 1749–1761.

Tominaga, Y., Stathopoulos, T., 2011. Cfd modeling of pollution dispersion in a street canyon: Comparison between les and rans. Journal of Wind Engineering and Industrial Aerodynamics 99, 340–348.

Tubino, F., Solari, G., 2005. Double proper orthogonal decomposition for representing and simulating turbulence fields. Journal of engineering mechanics 131, 1302–1312.

Tucker, M., Challenor, P.G., Carter, D., 1984. Numerical simulation of a random sea: a common error and its effect upon wave group statistics. Applied ocean research 6, 118–122.

Tucker, P.G., Lardeau, S., 2009. Applied large eddy simulation.

UNDESA, 2018. 68% of the world population projected to live in urban areas by 2050 says un. United Nations Department of Economic and Social Affairs (UNDESA) URL: https://www.un.org/development/desa/en/news/population/2018-revision-of-world-urbanization-prospects.html.

Van Driest, E.R., 1956. On turbulent flow near a wall. Journal of the aeronautical sciences 23, 1007–1011.

Van Leer, B., 1979. Towards the ultimate conservative difference scheme. v. a second-order sequel to godunov's method. Journal of computational Physics 32, 101–136.

Vanmarcke, E., 2010. Random fields: analysis and synthesis. World Scientific.

Vasaturo, R., Kalkman, I., Blocken, B., van Wesemael, P., 2018. Large eddy simulation of the neutral atmospheric boundary layer: performance evaluation of three inflow methods for terrains with different roughness. Journal of Wind Engineering and Industrial Aerodynamics 173, 241–261.

Vermeire, B.C., Orf, L.G., Savory, E., 2011. A parametric study of downburst line near-surface outflows. Journal of wind engineering and industrial aerodynamics 99, 226–238.

Vickery, B.J., 1990. Experimental techniques for the determination of the dynamic responses of structures to wind. Meccanica 25, 147–158.

de Villiers, E., 2006. The Potential of Large Eddy Simulation for the Modelling of Wall Bounded Flows. Ph.D. thesis. Imperial College London.

Vuorinen, V., Chaudhari, A., Keskinen, J.P., 2015. Large-eddy simulation in a complex hill terrain enabled by a compact fractional step openfoam® solver. Advances in Engineering Software 79, 70–80.

Wang, H., Zhou, Y., 2009. The finite-length square cylinder near wake. Journal of Fluid Mechanics 638, 453–490.

Wang, S., Zhang, Z., 2013. Improving cur matrix decomposition and the nyström approximation via adaptive sampling. The Journal of Machine Learning Research 14, 2729–2769.

Wang, Y., Chen, X., 2020. Simulation of approaching boundary layer flow and wind loads on high-rise buildings by wall-modeled les. Journal of Wind Engineering and Industrial Aerodynamics 207, 104410.

Warsido, W.P., 2013. Reducing uncertainties in estimation of wind effects on tall buildings using aerodynamic wind tunnel tests. Ph.D. thesis. Florida International University.

Warsido, W.P., Bitsuamlak, G.T., 2015. Synthesis of wind tunnel and climatological data for estimating design wind effects: A copula based approach. Structural Safety 57, 8–17.

Weller, H.G., Tabor, G., Jasak, H., Fureby, C., 1998. A tensorial approach to computational continuum mechanics using object-oriented techniques. Computers in physics 12, 620–631.

Willcox, K., Paduano, J., Peraire, J., Hall, K., 1999. Low order aerodynamic models for aeroelastic control of turbomachines, in: 40th Structures, Structural Dynamics, and Materials Conference and Exhibit, p. 1467.

Williams, C.K., Seeger, M., 2001. Using the nyström method to speed up kernel machines, in: Advances in neural information processing systems, pp. 682–688.

Wilson, E., Habibullah, A., 1987. Static and dynamic analysis of multi-story buildings, including p-delta effects. Earthquake spectra 3, 289–298.

Wu, J., Liu, P., Li, Q., 2007. Effects of amplitude-dependent damping and time constant on wind-induced responses of super tall building. Computers & structures 85, 1165–1176.

Wu, X., 2017. Inflow turbulence generation methods. Annual Review of Fluid Mechanics 49, 23–49.

Wüchner, R., Kupzok, A., Bletzinger, K.U., 2007. A framework for stabilized partitioned analysis of thin membrane–wind interaction. International journal for numerical methods in fluids 54, 945–963.

Xie, Z.T., Castro, I.P., 2008. Efficient generation of inflow conditions for large eddy simulation of street-scale flows. Flow, turbulence and combustion 81, 449–470.

Xie, Z.T., Castro, I.P., 2009. Large-eddy simulation for flow and dispersion in urban streets. Atmospheric Environment 43, 2174–2185.

Yamada, T., Hong, G., Kataoka, S., Yoshimura, S., 2016. Parallel partitioned coupling analysis system for large-scale incompressible viscous fluid–structure interaction problems. Computers & Fluids 141, 259–268.

Yan, B., Li, Q., 2015. Inflow turbulence generation methods with large eddy simulation for wind effects on tall buildings. Computers & Fluids 116, 158–175.

Yan, B., Li, Q., 2016. Large-eddy simulation of wind effects on a super-tall building in urban environment conditions. Structure and Infrastructure Engineering 12, 765–785.

Yang, J.N., 1972. Simulation of random envelope processes. Journal of Sound and Vibration 21, 73–85.

Yang, J.N., 1973. On the normality and accuracy of simulated random processes. Journal of Sound and Vibration 26, 417–428.

Yang, W., Chang, T., Chang, C., 1997. An efficient wind field simulation technique for bridges. Journal of Wind Engineering and Industrial Aerodynamics 67, 697–708.

Yazdchi, M., Khalili, N., Valliappan, S., 1999. Dynamic soil–structure interaction analysis via coupled finite-element–boundary-element method. Soil Dynamics and Earthquake Engineering 18, 499–517.

Yeo, D., Simiu, E., 2011. High-rise reinforced concrete structures: Database-assisted design for wind. Journal of Structural Engineering 137, 1340–1349.

Yoshikawa, M., Tamura, T., 2015. Cfd wind-resistant design of tall building in actual urban area using unstructured-grid les, in: IABSE Conference: Elegance in structures, Nara, Japan, 13-15 May 2015, pp. 486–487.

Yoshizawa, A., 1986. Statistical theory for compressible turbulent shear flows, with the application to subgrid modeling. The Physics of fluids 29, 2152–2164.

Yu, Y., Yang, Y., Xie, Z., 2018. A new inflow turbulence generator for large eddy simulation evaluation of wind effects on a standard high-rise building. Building and Environment 138, 300–313.

Yuan, C., Norford, L., Britter, R., Ng, E., 2016. A modelling-mapping approach for fine-scale assessment of pedestrian-level wind in high-density cities. Building and Environment 97, 152–165.

Zhang, K., Kwok, J.T., 2010. Clustered nyström method for large scale manifold learning and dimension reduction. IEEE Transactions on Neural Networks 21, 1576–1587.

Zhang, L.l., Li, J., Peng, Y., 2008. Dynamic response and reliability analysis of tall buildings subject to wind loading. Journal of Wind Engineering and Industrial Aerodynamics 96, 25–40.

Zhang, X., Wegner, J., Haddow, J., 1999. Three-dimensional dynamic soil–structure interaction analysis in the time domain. Earthquake engineering & structural dynamics 28, 1501–1524.

Zhang, Y., Cao, S., Cao, J., 2022. An improved consistent inflow turbulence generator for les evaluation of wind effects on buildings. Building and Environment , 109459.

Zhang, Y., Habashi, W., Khurram, R., RANS, H., 2012. Les method for fsi simulations of tall buildings, in: World Congress on Advances in Civil, Environmental, and Materials Research (ACEM12), pp. 3048–3059.

Zhang, Y., Habashi, W.G., Khurram, R.A., 2015. Predicting wind-induced vibrations of high-rise buildings using unsteady cfd and modal analysis. Journal of Wind Engineering and Industrial Aerodynamics 136, 165–179.

Appendix A

Numerical implementation of DFSR and CDRFG methods

In the current study, the codes for the DFSR and CDRFG methods were implemented in C++. For faster execution, parts of the code that can be executed concurrently for both methods are written in parallel utilizing the Open MPI (Message Passing Interface) library. Parallelizing the CDRFG method is seamless because the generation of the velocity time-series at one point is independent of the other. For the DFSR method, however, performing the Cholesky factorization of the CPSD matrix in parallel is challenging; therefore, the code is divided into two pieces: serial and parallel parts. The serial part of the code performs the Cholesky factorization of the CPSD matrix to get the $\mathbf{H}_{u_i}(\omega)$ matrix in Eq.(2.9). Once the $\mathbf{H}_{u_i}(\omega)$ matrix is computed, the parallel part of the code is invoked to generate the velocity time-series using the FFT algorithm. The Cholesky factorization of the CPSD matrix given in Eq.(2.9) is performed by a highly optimized linear algebra package LAPACK (Anderson et al., 1999b). Similarly, to perform the FFT given in Eq.(2.12), the Fastest Fourier Transform in the West (FFTW) package developed by Frigo and Johnson (2005a) is adopted providing the aforementioned $O(N \log N)$ speed gain. Finally, the generated velocity field is exported into an OpenFOAM readable inflow data for LES use. The implemented code for the DFSR method is available at https://github.com/GBitsuamlak/DFSR together with an illustrative example showing how to use it for LES in OpenFOAM.

Figure A.1 compares the computational cost of DFSR and CDRFG methods for different durations with 10^4 points at $\Delta t = 0.002$. The number of the frequency intervals (N) is set to 4096 for both DFSR and CDRFG techniques. The simulations are run on 32 processors with a Xeon(R) CPU E5-4620 @ 2.2 GHz device having 128GB of memory. For the DFSR method, the execution time shown in Figure A.1 includes the time consumed in performing both the interpolated Cholesky factorization with $\tilde{N} = 30$ frequencies and wind field generation using

the FFT technique. In Figure A.1 the time taken for the Cholesky factorization is shown with a dotted line. As depicted in Figure A.1, for a shorter duration, the CDRFG method takes less CPU time compared to the DFSR method. However, when the duration gets longer, the DFSR method becomes more efficient because of the FFT technique. It should be noted that the CPU hours shown in Figure A.1 are execution times using 32 processors.

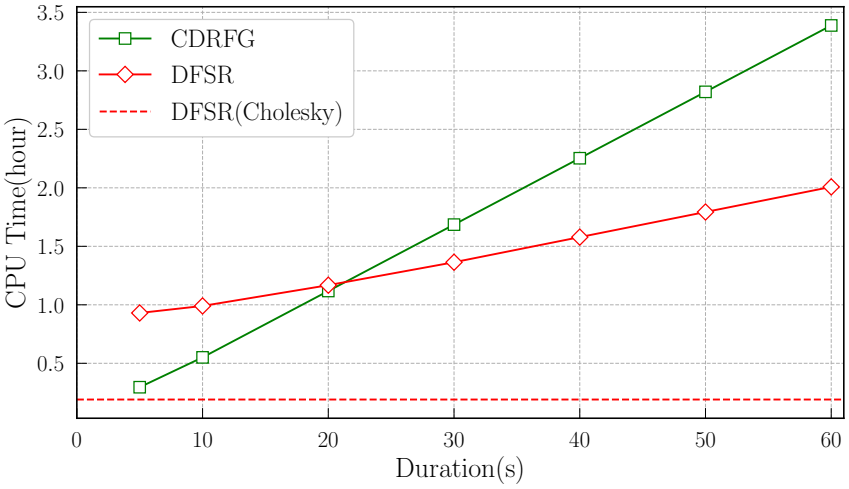


Figure A.1: Comparison of execution times for different duration using 32 processors.

Appendix B

Upper bound of the Nyström approximation error

As seen in Eq. (3.31), the approximation error for Nyström method results from the reconstruction error of the block CPSD matrix $\mathbf{S}_{22}(\omega)$, and can be expressed in Frobenius norm as (Arcolano and Wolfe, 2010):

$$\|\mathbf{S}^0(\omega) - \widetilde{\mathbf{S}}(\omega)\|_F = \|\mathbf{S}_{22}(\omega) - \mathbf{S}_{21}(\omega)\mathbf{W}^\dagger(\omega)\mathbf{S}_{21}^\mathsf{T}(\omega)\|_F, \tag{B.1}$$

where the matrices $\mathbf{W}^\dagger(\omega)$ and $\mathbf{S}_{21}$ are defined in Eq. (3.25). The expression on the right hand of Eq. (B.1) is often referred to as Schur-complement of the block matrix $\mathbf{W}(\omega)$, and the approximation error is entirely characterized by it.

Now, let us appeal to the special structure of the CPSD matrix, i.e., it has low coherence for high frequencies. This makes the relative reconstruction error for the CPSD matrix maximum at the high-frequency end. Then, when $\omega \to \infty$, the matrix $\mathbf{S}^0(\omega)$ becomes nearly a diagonal matrix holding only the auto-PSD functions, and the block matrix $\mathbf{S}_{21}(\omega)$ approaches a zero matrix since it is formed from the off-diagonal entries of $\mathbf{S}^0(\omega)$. Hence, the reconstruction error in Eq. (B.1) is bounded by

$$\|\mathbf{S}^0(\omega) - \widetilde{\mathbf{S}}(\omega)\|_F \leq \|\mathbf{S}_{22}(\omega)\|_F \approx \sqrt{\sum_{i=1}^{n-c} [\mathbf{S}_{22}(\omega)]_{ii}^2} \quad , \tag{B.2}$$

For simplicity, if we assume homogeneous turbulence, we have the same auto-PSD functions along the diagonal of the CPSD matrix. Finally, an upper limit to the relative reconstruction

error is given by

$$\epsilon_{\mathbf{nys}}(\omega) = \frac{\|\mathbf{S}^0(\omega) - \widetilde{\mathbf{S}}(\omega)\|_F}{\|\mathbf{S}^0(\omega)\|_F} \leq \frac{\|\mathbf{S}_{22}(\omega)\|_F}{\|\mathbf{S}^0(\omega)\|_F} \approx \sqrt{\frac{(n-c)}{n}} \tag{B.3}$$

Actually, this error bound is generally crude for most cases of practical interest, especially when considering the entire frequency range. However, it clearly serves as a weak proof for the convergence of the Nyström approximation of the CPSD matrix.

Appendix C

Numerical implementation of NY-POD method

The simulation procedure was implemented in C++. For the eigen-decomposition of the CPSD and for all matrix-related computations, we used the LAPACK (Anderson et al., 1999a) package. The velocity time series is simulated using the FFTW (Frigo and Johnson, 2005b) library. Parts of the implemented C++ code that take significant computing time were multi-threaded using OpenMP (Open Multi-Processing). Listing C.1 provides the implemented source code.

```
/*--------------------------------------------------------------------------*\
This code generates velocity time history using spectral representation for
a stationary Gaussian multivariate stochastic process. The cross-power spectral
matrix is approximated using the Nystrom method. The simulation process is not
ergodic, and a random-phase-angle approach is utilized. The time history of
each component of the process is computed using the FFT technique. The generated
field is saved to a file with rows representing time and columns for the points.
\*--------------------------------------------------------------------------*/

#ifndef SRM_H
#define SRM_H

#include <iostream>
#include <vector>
#include <string>
#include "dictionary.h"

using namespace std;


const double pi=3.141592653589793;

class SRM
{
    //Private data
    private:

        //Number points in the simulation
        int npoints;

        //Number of columns to sample for Nystrom approx
        int ncols;

        //Number of modes for the POD time series simulation
        //Should always be less than or equal to m or n
        int nmodes;
```

```
    //Number of frequency intervals
    int N;

    //Number of time steps
    int Nt;

    //Upper cut-off frequency
    double fmax;

    //Time step
    double dt;

    //Frequency step
    double df;

    //Period of the simulation
    double T;

    //Seed for the random number generator
    int seed;

    //Number of sample functions to generate for
    //time-series generation
    int nsamples;

    //Max number of threads
    int max_num_threads;

    //Coherency decay coefficients for u-component of the velocity
    vector<double> Cu;

    //Point coordinates
    vector<vector<double>> points;

    //Mean velocity profile
    vector<double> Uav;

    //Turbulence intensity profile
    vector<double> I;

    //Integral length scale profile
    vector<double> L;

    //Number of columns to be sampled for NY-POD
    vector<vector<int>> cols;

    //Simulation type
    string sim_type;

    //Sampling type
    string sample_type;

    //FFT simulation type
    string ergodic;

    //Name of the case simulated.
    string case_name;

    //Add mean or not
    string add_mean;

    //Column average
    string avg_columns;


public:

    //Constructor
    SRM(dictionary& dict);

    //Reads point data and Uav, I , L from a file stream
    void read_point_data();
```

```
//Reads indices of sampled points
//Used only for 2D case
void read_sample_points();

//Prints a given matrix
void print_matrix(int nrow, int ncol, double* matrix);

//Prints a vector
void print_vector(int nrow, double* vect);

//Return the von Karman spectrum for the longitudinal component of the velocity.
double von_k_spectrum(double freq, double Uav, double I, double L);

//Calculates the coherency function for any given two point p1 and p2.
double coherency(double freq, double Uav, vector<double>& p1, vector<double>& p2);

//Calculate the PSD at all points for a given frequency.
void calc_psd_all(double freq, double* S0);

// Calculates the cross-power spectrum matrix for a given frequency.
// Computes only the lower part of the matrix because it is symmetric
void calc_xpsd_matrix(double freq, double* S);

//Subtract two given matrices
void matrix_sub(int nrow, int ncol, double* m1, double* m2, double* result);

//Add two given matrices
void matrix_add(int nrow, int ncol, double* m1, double* m2, double* result);

//Geometrically uniform column sampling for 1D case
//n: number of points (total number of points)
//m: number of columns to be sampled
void uniform_sample_1D(int n, int m);

//Geometrically uniform column sampling for 2D case
//n: number of points (total number of points)
//m: number of columns to be sampled
void uniform_sample_2D(int n, int m);

//Random column sampling
//n: number of points (total number of points)
//m: number of columns to be sampled
void random_sample(int n, int m);

//Check sampling method and sample columns for NY-POD
//n: number of points (total number of points)
//m: number of columns to be sampled
void sample_columns(int n, int m);

//Sample the column and intersection matrix for Nystrom approximation
//n: number of points or dim of S
//m: number of columns to be sampled
//S: XPSD matrix with dimension [n x n]
//C: Sampled column matrix (dim = [n x m])
//W: Sampled interesection matrix with dim [m x m]
void sample_xpsd_matrix(int n, int m, double* S, double *C, double* W);

//Performs eigen-decomposition of the cross-spectral density matrix.
// n : matrix dim
// D : diagonal matrix
// V : eigenvector
// S : XPSD matrix
void eigen_decompose(int n, double*S, double*V, double*D);

//Returns the eigen-decomposition H = D^{1/2}*V using eigen decomposition.
// n : matrix dim
// k : rank of approximation
// S : [n x n] XPSD matrix
// H : [n x k] the decomposed matrix H = D^{1/2}*V
void eigen_decompose_H(int n, int k, double*S, double*H);

//Returns the cholesky decomposition of S = H*H^T using eigen decomposition.
// n : matrix dim
// S : [n x n] XPSD matrix
```

```
// H : [n x n] the decomposed matrix H = H*H^T
void cholesky_decompose_H(int n, double*S, double*H);

//Performs k-rank approximation of the CPSD matrix.
//V:  eigenvectors of the matrix [n x m]
//D:  eigenvalues of the matrix  [m]
//k:  rank of the approximation.
//Sk: k-rank approximation of the CPSD matrix [n x n]
void low_rank_approx(int n, int m, int k, double*V, double*D, double*Sk);

//Calculate the decomposion S = H*H^T. Where H = VD^(1/2).
//V:  eigenvectors of the matrix [n x m]
//D:  eigenvalues of the matrix  [m]
//H:  matrix [n x m]
void get_H(int n, int m, double*V, double*D, double*H);

//Returns the approximate eigenvectors and eigenvalues for CPSD matrix using Nystrom approximation.
//Here m stands for the number of columns to be sampled.
//Dm is the approximate eigenvalues
//Vm is the approximate eigenvector
//S is the full CPSD matrix
void nystrom_approx(int n, int m, double*S, double*Vm, double*Dm);

//Returns the approximate matrix H = C*V*D^{-1/2} using Nystrom approximation.
//n: number of points or size of S.
//m: number of columns to be sampled.
//k: rank of the approximation.
//S: [n x n] the full CPSD matrix
//H: [n x k] the decomposed matrix H = C*V*D^{-1/2}
void nystrom_approx_H(int n, int m, int k, double* S, double* H);

//Calculates the Frobenius/Eculidian norm of the error given the original matrix and
//the low-rank approximation of the same matrix
double frobenius_norm(int n, double* S, double* Sk);

//Generate matrix of random phase angles from uniform distribution
//Range: [0, 2*pi]
//k: number of columns
//Phi is [k by N] matrix
void generate_phi(int k, double* phi);

//Writes the decomposed matrix data for all frequencies for 'POD' based simulation
//  -eigenvalues
//  -eigenvectors
void write_pod_data();

//Writes the decomposed matrix data for all frequencies for Nystrom based simulation
//  -eigenvalues
//  -eigenvectors
void write_nys_data();

//Writes the intermediat data.
void write_matrix_data();

//Writes a given 1D matrix into a file
//row-major format. 'nrow' represents the number of rows of rows.
void write_1D_data(const char* fname, int nrow, vector<double>& data);

//Writes a one dimentional integer data
void write_1D_data(const char* fname, int nrow, vector<int>& data);

//Writes a given 2D matrix into a file
//row major format. 'nrow' and 'ncol' represent the number of rows
//and columns, respectively.
void write_2D_data(const char* fname, int nrow, int ncol, vector<double*>& data);

//Writes a given 2D matrix into a file
//row major format. 'nrow' and 'ncol' represent the number of rows
//and columns, respectively.
void write_2D_data(const char* fname, int nrow, int ncol, double* data);

//Writes a given 3D matrix into a file
//nf: Number of idividual 2D slices
//row major format. 'nrow' and 'ncol' represent the number of rows
```

```
        //and columns individual 2D slices, respectively.
        void write_3D_data(const char* fname, int nf, int nrow, int ncol, vector<double*>& data);

        //Writes a given 3D matrix into a file
        //nf: Number of individual 2D slices
        //row major format. 'nrow' and 'ncol' represent the number of rows
        //and columns individual 2D slices, respectively.
        void write_3D_data_compact(const char* fname, int nf, int m, int nrow, int ncol, vector<double*>& data);

        //Generates velocity time-series for a given H matrix as ergodic time series
        //with a double indexed frequency
        //n: number of points
        //k: number of modes to consider
        //H: decomposed matrix for each frequency
        //U: Array that will hold the velocity time series
        void fft_U_ergodic(int n, int k, vector<double*>& H, double* U);

        //Generates nonergodic velocity time-series for a given H matrix
        //n: number of points
        //k: number of modes to consider
        //H: decomposed matrix for each frequency
        //U: Array that will hold the velocity time series
        void fft_U(int n, int k, vector<double*>& H, double* U);

        //Test the cpsd matrix at a selected frequency
        void test_cpsd_matrix();

        //Reconstrcuts the cpsd matrix for each frequency and
        //calculate the reconstruction L2-norm of the error
        void write_reconst_error(int n_modes, int delta_k);

        //Calculates the spectral reconstruction error
        void write_spectral_error(int m, int k);

        //Generates the final velocity field
        void generate_velocity();

        //Generates the final velocity field
        //m: number of columns to sample
        //k: number of modes
        //sim: type of the simulation (POD or NY-POD)
        // void generate_velocity(int m, int k, string sim);

        //Sets m, k, sime_type parameters.
        void set_params(int m, int k, string sim);

        //A function to generate sample velocity time-series at each story
        //for tall building buffeting analysis.
        //The following parameters are changed to generate different sampels.
        //  -velocity at roof height
        //  -turbulence intensity
        //  -Length scale
        //  -Seed used to generate the phase anngles
        void generate_velocity_tall_building();
};

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

/*-------------------------------------------------------------------------*\
Source file: SRM.cpp
\*-------------------------------------------------------------------------*/

#include <iostream>
#include <string>
#include <fstream>
#include <sstream>
#include <cstring>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <chrono>
#include <time.h>
#include <random>
#include "mkl.h"
```

```
#include "fftw3.h"
#include "SRM.h"
#include "dictionary.h"


using namespace std;
using namespace std::chrono;

SRM::SRM(dictionary& dict)
{
  npoints  = dict.read_int_entry("nPoints");
  ncols  = dict.read_int_entry("nColumns");
  nmodes  = dict.read_int_entry("nModes");
  N = dict.read_int_entry("N");
  seed = dict.read_int_entry("seed");
  nsamples = dict.read_int_entry("nSamples");
  fmax = dict.read_scalar_entry("fMax");
  T = dict.read_scalar_entry("T");

  Cu.resize(3, 0.0);
  Cu[0] = dict.read_scalar_entry("Cux");
  Cu[1] = dict.read_scalar_entry("Cuy");
  Cu[2] = dict.read_scalar_entry("Cuz");
  case_name = dict.read_string_entry("caseName");

  sim_type = dict.read_string_entry("simType");
  sample_type = dict.read_string_entry("sampleType");
  ergodic = dict.read_string_entry("ergodic");
  max_num_threads = dict.read_int_entry("numThreads");
  add_mean = dict.read_string_entry("addMean");
  avg_columns = dict.read_string_entry("avgColumns");

  dt = 1.0/(2.0*fmax);
  df = fmax/N;
  Nt = T/dt;

  read_point_data();

  //Print input params
  cout << "\n"<< endl;
  cout << "==================Input␣Parameters================\n"<< endl;
  cout << "Case␣name␣=␣" << case_name << endl;
  cout << "Number␣of␣points(n)␣=␣" << npoints << endl;
  cout << "Number␣of␣columns␣sampled(m)␣=␣" << ncols << endl;
  cout << "Number␣of␣modes(K)␣=␣" << nmodes << endl;
  cout << "Number␣of␣frequency␣intervals(N)␣=␣" << N << endl;
  cout << "Number␣of␣time␣steps␣=␣" << Nt << endl;
  cout << "Max.␣frequency(fMax)␣=␣" << fmax << endl;
  cout << "Time␣step(dt)␣=␣" << dt << endl;
  cout << "Frequency␣step(df)␣=␣" << df << endl;
  cout << "Simulation␣duration(T)␣=␣" << T << endl;
  cout << "Simulation␣period(T0)␣=␣" << 2.0/df << endl;
  cout << "Coherency␣decay␣coeff(Cy)␣=␣" << Cu[1] << endl;
  cout << "Coherency␣decay␣coeff(Cz)␣=␣" << Cu[2] << endl;
  cout << "Simulation␣type␣=␣" << sim_type << endl;
  cout << "Sampling␣method␣=␣" << sample_type << endl;
  cout << "Ergodic␣time-series␣=␣" << ergodic << endl;
  cout << "Add␣mean␣=␣" << add_mean << endl;
  cout << "Average␣columns␣=␣" << avg_columns << endl;
  cout << "Random␣number␣seed␣=␣" << seed << endl;
  cout << "Number␣of␣sample␣functions␣=␣" << nsamples << endl;
  cout << "Max.␣number␣of␣OpenMP␣threads␣=␣" << max_num_threads << endl;
  cout << "\n=================================================="<< endl;

  sample_columns(npoints, ncols);


  //Set number of threads
  // mkl_set_num_threads_local(max_num_threads);
  mkl_set_num_threads(max_num_threads);
}

void SRM::read_point_data()
{
```

```
  ifstream point_data;
  string file_name = "input/points_n" +  to_string(npoints);
  point_data.open (file_name);

  vector<double> pointi;
  pointi.resize(3, 0.0);

  double Uavi, Ii, Li;

  if(point_data.is_open())
  {
    cout << "Reading␣point␣data␣and␣ABL␣profile␣from:␣" << file_name << endl;

    // printf("x(m)\t\ty(m)\t\tz(m)\t\tU(m/s)\t\tI(%%)\t\tL(m)\n");

    while(point_data >> pointi[0] >> pointi[1] >> pointi[2] >> Uavi >> Ii >> Li)
    {
      points.push_back(pointi);
      Uav.push_back(Uavi);
      I.push_back(Ii);
      L.push_back(Li);
    }

    if(npoints != points.size())
    {
      printf("Number␣of␣points␣do␣not␣match.\n");
      exit(EXIT_FAILURE);
    }

    npoints = points.size();

    point_data.close(); // close file
  }
  else
  {
    // fp null means no file to read from
    printf("Cannot␣read␣the␣input␣file.\n");
    exit(EXIT_FAILURE);
  }
}

void SRM::read_sample_points()
{
  ifstream sample_data;
  string file_name = "input/samples_n" +  to_string(npoints) + "_m" +  to_string(ncols);

  sample_data.open(file_name);

  cols.clear();

  int pointi;

  string line;

  if(sample_data.is_open())
  {
      cout<< "Reading␣indexes␣of␣sampled␣points␣from:␣" << file_name << endl;

      while(getline(sample_data, line, '\n'))
      {
        //create a temporary vector that will contain all the columns
        std::vector<int> avg_points;

        std::istringstream ss(line);

        //read int by int
        while(ss >> pointi)
        {
          avg_points.push_back(pointi);
          // cout << pointi << "\t";
        }

        //add each row to the main array
        cols.push_back(avg_points);
```

```
        // cout << endl;
      }

      if(ncols != cols.size())
      {
        printf("Number␣of␣columns␣sampled␣do␣not␣match.\n");
        exit(EXIT_FAILURE);
      }

      cout << "Number␣of␣samples:␣" << ncols << endl;
      cout << "Number␣of␣averaging␣columns:␣" << cols[0].size() << endl;

      sample_data.close(); // close file
  }
  else
  {
    // fp null means no file to read from
    printf("Cannot␣read␣the␣sampled␣points␣file.\n");
    exit(EXIT_FAILURE);
  }
}

void SRM::print_matrix(int nrow, int ncol, double* matrix)
{
  for (int i=0; i<nrow; i++)
  {
    for(int j=0; j<ncol; j++)
    {
      printf("%.3f\t",matrix[i*ncol + j]);
    }
    printf("\n");
  }
}

void SRM::print_vector(int nrow,  double* vect)
{
  for (int i=0; i<nrow; i++)
  {
      printf("%.3f\t",vect[i]);
      printf("\n");
  }
}

double SRM::von_k_spectrum(double freq, double Uav, double I, double L)
{
  return 4.0*pow(I*Uav, 2.0)*(L/Uav) / pow(1.0 + 70.8*pow(freq*L/Uav, 2.0), 5.0/6.0);
}

double SRM::coherency(double freq, double Uav, vector<double>& p1, vector<double>& p2)
{
  return exp(-freq*sqrt(pow(Cu[0]*(p2[0] - p1[0]), 2.0) + pow(Cu[1]*(p2[1] - p1[1]), 2.0) + pow(Cu[2]*(p2[2] - p1[2]), 2.0))/
       Uav);
}

void SRM::calc_psd_all(double freq, double* S0)
{
  for(int i=0; i < npoints; i++)
  {
    S0[i] = von_k_spectrum(freq, Uav[i], I[i], L[i]);
  }
}

void SRM::calc_xpsd_matrix(double freq, double*S)
{
  double s1, s2;

  for (int i = 0; i < npoints; i++)
  {
    s1 = von_k_spectrum(freq, Uav[i], I[i], L[i]);

    for (int j = 0; j <= i; j++)
    {
      s2 = von_k_spectrum(freq, Uav[j], I[j], L[j]);
      S[i*npoints +j] = sqrt(s1*s2)*coherency(freq, 0.5*(Uav[i] + Uav[j]), points[i], points[j]);
```

```
      S[j*npoints +i] = S[i*npoints +j];
    }
  }
}

void SRM::matrix_sub(int nrow, int ncol, double* m1, double* m2, double* result)
{
  for(int i=0; i<nrow; i++)
  {
    for(int j=0; j<ncol; j++)
    {
      result[i*ncol + j] = m1[i*ncol + j] - m2[i*ncol + j];
    }
  }
}

void SRM::matrix_add(int nrow, int ncol, double* m1, double* m2, double* result)
{
  for(int i=0; i<nrow; i++)
  {
    for(int j=0; j<ncol; j++)
    {
      result[i*ncol + j] = m1[i*ncol + j] + m2[i*ncol + j];
    }
  }
}

void SRM::uniform_sample_1D(int n, int m)
{
  double dn = (n + 0.00001)/m;
  cols.clear();

  for (int i = 0; i < m; i++)
  {
    vector<int> index(1,int(dn*(i + 0.5)));
    cols.push_back(index);
    // printf("Column = %i\n", cols[i] + 1);
  }
}

void SRM::uniform_sample_2D(int n, int m)
{
  int nx = int(sqrt(n + 1.0e-6));
  int ns = int(sqrt(m + 1.0e-6));

  double ds = (nx + 1.0e-6)/ns;

  int idx = -1;
  int idy = -1;

  cols.clear();


  for (int i = 0; i < ns; i++)
  {
    for (int j = 0; j < ns; j++)
    {
        idx = int((i + 0.4999999)*ds);
        idy = int((j + 0.4999999)*ds);

        vector<int> index(1, idx*nx + idy) ;

        cols.push_back(index);
      // printf("Column = %i\n", idx*nx + idy + 1);
    }
  }
  // exit(0);
}

void SRM::random_sample(int n, int m)
{
  //Seed the random number generator
  srand(seed);
```

```
  cols.resize(m, vector<int>(1,-1));

  //Select the columns to be sampled from a uniform distribution
  for (int i = 0; i < m; i++)
  {
    int rand_col = rand() % n;
    bool checked = false;
    while (!checked)
    {
      bool repeated = false;

      for(int j = 0; j<i ;j++)
      {
        if(rand_col == cols[j][0])
        {
          repeated = true;
          break;
        }
      }

      if (repeated)
      {
        rand_col = rand()%n;
      }
      else
      {
        vector<int> index(1, rand_col);
        cols[i] = index ;
        checked = true;
      }
    }
  }

  int temp  = 0;
  //Sort the columns in ascending order
  for (int i = 0; i < m; ++i)
  {
    for (int j = i + 1; j < m; ++j)
    {
      if (cols[i][0]> cols[j][0])
      {
        temp =  cols[i][0];
        cols[i][0] = cols[j][0];
        cols[j][0] = temp;
      }
    }
    // printf("Column = %i\n", cols[i] + 1);
  }
}


void SRM::sample_columns(int n, int m)
{
  if(sim_type == "NYPOD")
  {
    //Sample random
    if(sample_type=="R")
    {
      random_sample(n, m);
    }
    //Sample uniform
    else if(sample_type=="U")
    {
      read_sample_points();
    }
    else
    {
      // exit if not 'R' or 'U
      cout<<"Sampling␣type:␣" << sample_type << "is␣unknown!" << endl;
      exit(EXIT_FAILURE);
    }
  }
}
```

```
void SRM::eigen_decompose(int n, double* S, double* V, double* D)
{
  double ErrTol = 1.0e-6;
  int* m = (int *)calloc(n, sizeof(int));
  int* is = (int *)calloc(2*n, sizeof(int));
  double* S_copy = (double *)calloc(n*n, sizeof(double));

  //Copy the CPSD, S matrix because the LAPACKE_dsyevr function modifies
  //it if the original array is passed in the argument.
  for(int i = 0; i < n; i++)
  {
    for(int j = 0; j <=i; j++)
    {
      S_copy[i*n + j] = S[i*n + j];
    }
  }
  mkl_set_num_threads(max_num_threads);

  int infoEig = LAPACKE_dsyevr(LAPACK_ROW_MAJOR, 'V', 'I', 'L', n, S_copy, n, 0.0, 1.0, 1, n, ErrTol, m, D, V, n, is);

  if(infoEig != 0)
  {
    printf("The algorithm failed to compute eigenvalue decomposition\n");
    exit(EXIT_FAILURE);
  }

  //Free-up memory
  free(m);
  free(S_copy);
  free(is);
}

void SRM::eigen_decompose_H(int n, int k, double* S, double* H)
{
  double ErrTol = 1.0e-6;
  int* m_s = (int *)calloc(n, sizeof(int));
  int* is = (int *)calloc(2*n, sizeof(int));
  double* S_copy = (double *)calloc(n*n, sizeof(double));
  double* V = (double *)calloc(n*n, sizeof(double));
  double* D = (double *)calloc(n, sizeof(double));
  mkl_set_num_threads(max_num_threads);

  //Copy the CPSD, S matrix because the LAPACKE_dsyevr function modifies
  //it if the original array is passed in the argument.
  for(int i = 0; i < n; i++)
  {
    for(int j = 0; j <=i; j++)
    {
      S_copy[i*n + j] = S[i*n + j];
    }
  }

  int infoEig = LAPACKE_dsyevr(LAPACK_ROW_MAJOR, 'V', 'I', 'L', n, S_copy, n, 0.0, 1.0, 1, n, ErrTol, m_s, D, V, n, is);

  if(infoEig != 0)
  {
    printf("The algorithm failed to compute eigenvalue decomposition\n");
    exit(EXIT_FAILURE);
  }

  double sqrtD = 0.0;

  for(int j = n-1; j >= n-k; j--)
  {
    //Discard those with negative eigen value (usually very small)
    //For the time-being only absolute value is taken but in the future
    //they can explicitly set to zero.
    sqrtD = sqrt(fabs(D[j]));

    for(int i = 0; i < n; i++)
    {
      H[i*k + (n-1-j)] = V[i*n + j]*sqrtD;
    }
  }
```

```
  //Free-up memory
  free(m_s);
  free(is);
  free(S_copy);
  free(V);
  free(D);
}

void SRM::cholesky_decompose_H(int n, double* S, double* H)
{
  double* S_copy = (double *)calloc(n*(n + 1)/2, sizeof(double));
  mkl_set_num_threads(max_num_threads);

  //Copy the CPSD to a packed storage S_copy matrix for LAPACKE_dpptrf function
  for(int i = 0; i < n; i++)
  {
    for(int j = 0; j <=i; j++)
    {
      S_copy[j + i*(i + 1)/2] = S[i*n + j];
    }
  }

  int infoChol = LAPACKE_dpptrf(LAPACK_ROW_MAJOR, 'L', n , S_copy);

  if(infoChol != 0)
  {
    printf("The␣algorithm␣failed␣to␣compute␣Cholesky␣decomposition\n");
    exit(EXIT_FAILURE);
  }

  for(int i = 0; i < n; i++)
  {
    for(int j = 0; j <= i; j++)
    {
      H[i*n + j] = S_copy[j + i*(i + 1)/2];
    }
  }

  //Free-up memory
  free(S_copy);
}

void SRM::low_rank_approx(int n, int m, int k, double* V, double* D, double* Sk)
{
  if(k > m)
  {
    printf("k␣cannot␣be␣greater␣than␣m␣for␣low-rank␣approximation!\n");
    exit(EXIT_FAILURE);
  }

  //Temporary matrix products to reconstruct the matrix
  double *P = (double *)calloc(n*m, sizeof(double));

  //Calculate P = Vk*Dk
  //Multiply each column by corresponding eigen value.
  for(int i = 0; i < n; i++)
  {
    for(int j = m-k; j < m; j++)
    {
      P[i*m + j] = V[i*m + j]*D[j];
    }
  }
  mkl_set_num_threads(max_num_threads);

  //Calculate Sk -> P*Vk^T
  cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasTrans, n, n, m, 1.0, P, m, V, m, 0.0, Sk, n);

  //Free-up memory
  free(P);
}

void SRM::get_H(int n, int m, double* V, double* D, double* H)
{
```

```
  double sqrtD = 0.0;

  for(int j = 0; j < m; j++)
  {
    sqrtD = sqrt(fabs(D[j]));

    for(int i = 0; i < n; i++)
    {
      H[i*m + j] = sqrtD*V[i*m + j];
    }
  }
}

void SRM::nystrom_approx(int n, int m, double* S, double* Vm, double* Dm)
{
  //Matrix formed by selected columns
  double *C = (double *)calloc(n*m, sizeof(double));

  //Matrix formed by the intersection of selected columns
  //with the corresponding rows.
  double *W = (double *)calloc(m*m, sizeof(double));

  //Sample cpsd matrix
  sample_xpsd_matrix(n, m, S, C, W);

  // print_matrix(m, m, W);
  // print_matrix(n, m, C);

  //eigenvalues and eigenvectors of matrix A.
  double *D = (double *)calloc(m, sizeof(double));
  double *Q = (double *)calloc(m*m, sizeof(double));
  int* m_w = (int *)calloc(m, sizeof(int));
  int* is = (int *)calloc(2*m, sizeof(int));
  double ErrTol = 1.0e-6;
  mkl_set_num_threads(max_num_threads);

  int infoEig = LAPACKE_dsyevr(LAPACK_ROW_MAJOR, 'V', 'I', 'L', m, W, m, 0.0, 1.0, 1, m, ErrTol, m_w, D, Q, m, is);

  if(infoEig != 0)
  {
    printf("The␣algorithm␣failed␣to␣compute␣eigenvalue␣decomposition\n");
    exit(EXIT_FAILURE);
  }

  //Form a diagonal matrix as inverse of D
  double *D_inv = (double *)calloc(m*m, sizeof(double)); //k^th rank diagonal matrix containing eigenvalues on the diagonal.

  double scale = (n + 1e-20)/m;

  //Calculate the diagonal matrix D_inv as inverse of D
  for(int i = 0; i < m; i++)
  {
    D_inv[i*m + i] = 1.0/D[i];
    Dm[i] = scale*D[i];
  }

  //Temporary matrix to hold products
  double *P = (double *)calloc(m*m, sizeof(double));

  //Calculate P -> Q*D_inv
  cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, m, m, m, 1.0, Q, m, D_inv, m, 0.0, P, m);

  //Calculate Vm -> C*P
  cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, n, m, m, sqrt(1.0/scale), C, m, P, m, 0.0, Vm, m);

  //Free-up memory
  free(C);
  free(W);
  free(D_inv);
  free(D);
  free(Q);
  free(m_w);
  free(is);
  free(P);
```

```
}

void SRM::nystrom_approx_H(int n, int m, int k, double* S, double* H)
{
  //Matrix formed by selected columns
  double *C = (double *)calloc(n*m, sizeof(double));

  //Matrix formed by the intersection of selected columns
  //with the corresponding rows.
  double *W = (double *)calloc(m*m, sizeof(double));

  //Sample cpsd matrix
  sample_xpsd_matrix(n, m, S, C, W);

  // print_matrix(m, m, W);
  // print_matrix(n, m, C);

  //eigenvalues and eigenvectors of matrix A.
  double *D = (double *)calloc(m, sizeof(double));
  double *Q = (double *)calloc(m*m, sizeof(double));
  int* m_w = (int *)calloc(m, sizeof(int));
  int* is = (int *)calloc(2*m, sizeof(int));
  double ErrTol = 1.0e-6;
  mkl_set_num_threads(max_num_threads);

  int infoEig = LAPACKE_dsyevr(LAPACK_ROW_MAJOR, 'V', 'I', 'L', m, W, m, 0.0, 1.0, 1, m, ErrTol, m_w, D, Q, m, is);

  if(infoEig != 0)
  {
    printf("The␣algorithm␣failed␣to␣compute␣eigenvalue␣decomposition\n");
    exit(EXIT_FAILURE);
  }

  //Temporary matrix to hold products
  double *P = (double *)calloc(n*m, sizeof(double));

  //Calculate P -> C*Q
  cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, n, m, m, 1.0, C, m, Q, m, 0.0, P, m);

  double sqrtD_inv = 0.0;

  for(int j = m-1; j >= m-k; j--)
  {
    //Avoid the negative eigenvalues taking abs
    //Since the matrix is PSD matrix this values are very close to zero
    sqrtD_inv = 1.0/sqrt(fabs(D[j]));

    for(int i = 0; i < n; i++)
    {
      H[i*k + (m-1-j)] = P[i*m + j]*sqrtD_inv;
    }
  }

  //Free-up memory
  free(C);
  free(W);
  free(D);
  free(Q);
  free(m_w);
  free(is);
  free(P);
}

void SRM::sample_xpsd_matrix(int n, int m, double* S, double *C, double* W)
{
  double sum_spec = 0.0;

  //Form elements of matrix C
  if(avg_columns=="on")
  {
    for(int i = 0; i < n; i++)
    {
      for(int j = 0; j < m; j++)
      {
```

```
        sum_spec = 0.0;

        //Take average of the columns
        for(int ci=0; ci < cols[j].size(); ci++)
        {
          sum_spec += S[i*n + cols[j][ci]];
        }

        C[i*m + j] = sum_spec/cols[j].size();
      }
    }
  }
  else
  {
    for(int i = 0; i < n; i++)
    {
      for(int j = 0; j < m; j++)
      {
          C[i*m + j] = S[i*n + cols[j][0]];
      }
    }
  }

  //Form elements of matrix W
  for(int i = 0; i < m; i++)
  {
    for(int j = 0; j < m; j++)
    {
      W[i*m + j] = C[cols[i][0]*m + j];
    }
  }
}

double SRM::frobenius_norm(int n, double* S, double* Sk)
{
  //Stores the error matrix.
  double *S_err = (double *)calloc(n*n, sizeof(double));

  //Calculates the error matrix
  for ( int i  = 0; i < n; i++)
  {
    for ( int j = 0; j < n; j++)
    {
      S_err[i*n + j] = S[i*n + j] - Sk[i*n + j];
    }
  }

  //Lapack subroutine to compute the Frobenius or Eculidian norm.
  return LAPACKE_dlange(LAPACK_ROW_MAJOR, 'F', n, n, S_err, n);

  //Free-up memory
  free(S_err);
}

void SRM::generate_phi(int k, double* phi)
{
  //Seed the random number generator
  std::default_random_engine generator(seed);
  std::uniform_real_distribution<double> distribution(0.0, 1.0);

  for(int j = 0; j < k; j++)
  {
    for(int l = 0; l < N; l++)
    {
      phi[j*N + l] = 2.0*pi*distribution(generator);
      //printf("phi = %f\n", phi[i*N + l]);
    }
  }
}

void SRM::fft_U(int n, int k, vector<double*>& H, double* U)
{
  double sqrt2df = sqrt(2.0*df);
  double Hij = 0.0;
```

```
  double pidfdt = 2.0*pi*dt*df/2.0;
  int M = 2*N;

  fftw_complex* B = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * M);
  fftw_complex* C = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * M);

  fftw_plan pln;

  double *phi = (double*) calloc(k*N, sizeof(double));
  double *cosPhi = (double*) calloc(k*N, sizeof(double));
  double *sinPhi = (double*) calloc(k*N, sizeof(double));

  double *cost = (double*) calloc(Nt, sizeof(double));
  double *sint = (double*) calloc(Nt, sizeof(double));

  for(int t = 0; t < Nt; t++)
  {
    cost[t] = cos(t*pidfdt);
    sint[t] = sin(t*pidfdt);
  }

  //Generate the phase angels
  generate_phi(k, phi);

  for(int j = 0; j < k; j++)
  {
    for(int l = 0; l < N; l++)
    {
      cosPhi[j*N + l] = sqrt2df*cos(phi[j*N + l]);
      sinPhi[j*N + l] = sqrt2df*sin(phi[j*N + l]);
    }
  }

  for(int i = 0; i < n; i++)
  {
          for(int j = 0; j < k; j++)
          {
            for(int l = 0; l < N; l++)
            {
                // Hij = fabs(H[l][i*k + j]);
                Hij = H[l][i*k + j];
                B[l][0] = Hij*cosPhi[j*N + l];
                B[l][1] = Hij*sinPhi[j*N + l];
            }
            for(int l = N; l < M; l++)
            {
                B[l][0] = 0.0;
                B[l][1] = 0.0;
            }

            pln = fftw_plan_dft_1d(M, B, C, FFTW_BACKWARD, FFTW_ESTIMATE);

            fftw_execute(pln);
            fftw_destroy_plan(pln);

            //It's assumed that Nt < M*N
            for(int t = 0; t < Nt; t++)
            {
                U[t*n + i] += (C[t%M][0]*cost[t] - C[t%M][1]*sint[t]);
            }
          }
    // cout << "Generation completed for " << (i + 1) << " points out of " << n << endl;
  }


  fftw_free(B);
  fftw_free(C);
  free(phi);
  free(cosPhi);
  free(sinPhi);
  free(cost);
  free(sint);
}
```

```
void SRM::fft_U_ergodic(int n, int k, vector<double*> & H, double* U)
{
  double sqrt2df = sqrt(2.0*df);
  double Hij = 0.0;
  double freq = 0.0;
  int M = 2*N;

  fftw_complex* B = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * M);
  fftw_complex* C = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * M);

  fftw_plan pln = NULL;

  double *phi = (double*) calloc(k*N, sizeof(double));
  double *cosPhi = (double*) calloc(k*N, sizeof(double));
  double *sinPhi = (double*) calloc(k*N, sizeof(double));

  //Generate the phase angels
  generate_phi(k, phi);

  for(int j = 0; j < k; j++)
  {
    for(int l = 0; l < N; l++)
    {
      cosPhi[j*N + l] = sqrt2df*cos(phi[j*N + l]);
      sinPhi[j*N + l] = sqrt2df*sin(phi[j*N + l]);
    }
  }

  for(int i = 0; i < n; i++)
  {
          for(int j = 0; j < k; j++)
          {
            for(int l = 0; l < N; l++)
            {
                Hij = H[l][i*k + j];
                B[l][0] = Hij*cosPhi[j*N + l];
                B[l][1] = Hij*sinPhi[j*N + l];
            }
            for(int l = N; l < M; l++)
            {
                B[l][0] = 0.0;
                B[l][1] = 0.0;
            }

            pln = fftw_plan_dft_1d(M, B, C, FFTW_BACKWARD, FFTW_ESTIMATE);

            fftw_execute(pln);
            fftw_destroy_plan(pln);

            freq = 2.0*pi*dt*(j + 1.0)*df/n;

            //It's assumed that Nt < M*N
            for(int t = 0; t < Nt; t++)
            {
                U[t*n + i] += (C[t%M][0]*cos(t*freq) - C[t%M][1]*sin(t*freq));
            }
          }
    cout << "Generation completed for" << (i + 1) << "points out of" << n << endl;
  }

  fftw_free(B);
  fftw_free(C);
  free(phi);
  free(cosPhi);
  free(sinPhi);
}

void SRM::write_1D_data(const char* fname, int nrow, vector<double>& data)
{
  cout << "Writing file to:" << fname << endl;

  FILE *output_file = fopen(fname,"w");

  for(int i = 0; i < nrow; i++)
```

```
  {
    fprintf(output_file,"%.6e\n", data[i]);
  }

  fclose(output_file);
}

void SRM::write_1D_data(const char* fname, int nrow, vector<int>& data)
{
  cout << "Writing␣file␣to:␣" << fname << endl;

  FILE *output_file = fopen(fname,"w");

  for(int i = 0; i < nrow; i++)
  {
    fprintf(output_file,"%i\n", data[i]);
  }

  fclose(output_file);
}

void SRM::write_2D_data(const char* fname, int nrow, int ncol, vector<double*>& data)
{
  cout << "Writing␣file␣to:␣" << fname << endl;

  FILE *output_file = fopen(fname,"w");

  for(int i = 0; i < nrow; i++)
  {
    for(int j = 0; j < ncol; j++)
    {
      fprintf(output_file,"%.6e\t", data[i][j]);
    }
    fprintf(output_file,"\n");
  }

  fclose(output_file);
}

void SRM::write_2D_data(const char* fname, int nrow, int ncol, double* data)
{
  cout << "Writing␣file␣to:␣" << fname << endl;

  FILE *output_file = fopen(fname,"w");

  for(int i = 0; i < nrow; i++)
  {
    for(int j = 0; j < ncol; j++)
    {
      fprintf(output_file,"%.6e\t", data[i*ncol + j]);
    }
    fprintf(output_file,"\n");
  }

  fclose(output_file);
}

void SRM::write_3D_data(const char* fname, int nf, int nrow, int ncol, vector<double*>& data)
{
  cout << "Writing␣file␣to:␣" << fname << endl;

  FILE *output_file = fopen(fname,"w");

  for(int f = 0; f < nf; f++)
  {
    for(int i = 0; i < nrow; i++)
    {
      for(int j = 0; j < ncol; j++)
      {
        fprintf(output_file,"%.6e\t", data[f][i*ncol + j]);
      }
      fprintf(output_file,"\n");
    }
    fprintf(output_file,"\n");
```

```
  }

  fclose(output_file);
}

void SRM::write_3D_data_compact(const char* fname, int nf, int m, int nrow, int ncol, vector<double*>& data)
{
  cout << "Writing␣file␣to:␣" << fname << endl;

  FILE *output_file = fopen(fname,"w");

  for(int f = 0; f < nf; f++)
  {
    for(int i = 0; i < nrow; i++)
    {
      for(int j = ncol-m; j < ncol; j++)
      {
        fprintf(output_file,"%.6e\t", data[f][i*ncol + j]);
      }
      fprintf(output_file,"\n");
    }
    fprintf(output_file,"\n");
  }

  fclose(output_file);
}

void SRM::write_pod_data()
{
  int n = npoints;
  int m = ncols;

  vector<double>F(N, 0.0); //Array of frequency
  vector<double*>S0(N);   //Array of XPSD matrix
  vector<double*>Dpod(N); //Array of eigen values
  vector<double*>Vpod(N); //Array of eigen vectors

  double* S = (double*) calloc(n*n, sizeof(double));

  for(int f = 0; f < N; f++)
  {
    F[f] = df*(f + 0.5);

    S0[f] = (double*) calloc(n, sizeof(double));
    Dpod[f] = (double*) calloc(n, sizeof(double));
    Vpod[f] = (double*) calloc(n*n, sizeof(double));

    calc_psd_all(F[f], S0[f]);

    calc_xpsd_matrix(F[f], S);

    eigen_decompose(n, S, Vpod[f], Dpod[f]);

    cout << "Completed␣for␣f␣=␣" << F[f] << endl;
  }

  write_1D_data(("output/F_" + case_name).c_str(), N, F);
  write_2D_data(("output/S0_" + case_name).c_str(), N, n, S0);
  write_2D_data(("output/D_pod_m" + to_string(m) + "_" + case_name).c_str(), N, n, Dpod);
  write_3D_data_compact(("output/V_pod_m" + to_string(m) + "_" + case_name).c_str(), N, m, n, n, Vpod);

  //Free-up memory
  free(S);
}

void SRM::write_nys_data()
{
  int n = npoints;
  int m = ncols;

  vector<double>F(N,0.0);  //Array of frequency
  vector<double*>S0(N);    //Array of XPSD matrix
  vector<double*>Dnys(N);  //Array of eigen values
  vector<double*>Vnys(N);  //Array of eigen vectors
```

```
  double* S = (double*) calloc(n*n, sizeof(double));

  for(int f = 0; f < N; f++)
  {
    F[f] = df*(f + 0.5);

    S0[f] = (double*) calloc(n, sizeof(double));
    Dnys[f] = (double*) calloc(n, sizeof(double));
    Vnys[f] = (double*) calloc(n*n, sizeof(double));

    calc_psd_all(F[f], S0[f]);

    calc_xpsd_matrix(F[f], S);

    nystrom_approx(n, m, S, Vnys[f], Dnys[f]);

    cout << "Completed␣for␣f␣=␣" << F[f] << endl;
  }

  write_1D_data(("output/F_" + case_name).c_str(), N, F);
  write_2D_data(("output/S0_" + case_name).c_str(), N, n, S0);
  write_2D_data(("output/D_nys_m" + to_string(m) + "_" + case_name).c_str(), N, m, Dnys);
  write_3D_data_compact(("output/V_nys_m" + to_string(m) + "_" + case_name).c_str(), N, m, n, m, Vnys);

  //Free-up memory
  free(S);
}

void SRM::write_matrix_data()
{
  if(sim_type=="POD")
  {
    write_pod_data();
  }
  else if(sim_type == "NYPOD")
  {
    write_nys_data();
  }
}

void SRM::write_reconst_error(int n_modes, int delta_k)
{
  int n = npoints;
  int m = ncols;

  vector<int>modes(n_modes, 0);

  for(int i = 0; i < n_modes; i++)
  {
    modes[i] = (i + 1)*delta_k;
  }

  vector<double>F(N,0.0); //Array of frequency
  vector<double*>Norm_pod(N);   //Array Frobinues norm of k-rank approx for POD
  vector<double*>Norm_nys(N);   //Array Frobinues norm of k-rank approx for NY-POD
  vector<double>Norm_S(N);   //Frobinues norm of the whole XPSD matrix at each frequency

  double* S = (double*) calloc(n*n, sizeof(double));

  //Zero matrix
  double* S_zero = (double*) calloc(n*n, sizeof(double));

  //Reconstructed matrix
  double* Sk_pod = (double*) calloc(n*n, sizeof(double));
  double* Sk_nys = (double*) calloc(n*n, sizeof(double));

  //Eigen decomposition
  double *Dpod  = (double*) calloc(n, sizeof(double));
  double *Vpod  = (double*) calloc(n*n, sizeof(double));
  double *Dnys  = (double*) calloc(m, sizeof(double));
  double *Vnys  = (double*) calloc(n*m, sizeof(double));

  for(int f = 0; f < N; f++)
```

```
  {
    F[f] = df*(f + 0.5);

    calc_xpsd_matrix(F[f], S);

    eigen_decompose(n, S, Vpod, Dpod);

    nystrom_approx(n, m, S, Vnys, Dnys);

    Norm_pod[f] = (double*) calloc(n_modes, sizeof(double));
    Norm_nys[f] = (double*) calloc(n_modes, sizeof(double));

    Norm_S[f] = frobenius_norm(n, S, S_zero);

    for(int i = 0; i < n_modes; i++)
    {
      low_rank_approx(n, n, modes[i], Vpod, Dpod, Sk_pod);

      low_rank_approx(n, m, modes[i], Vnys, Dnys, Sk_nys);

      Norm_pod[f][i] = frobenius_norm(n, S, Sk_pod);
      Norm_nys[f][i] = frobenius_norm(n, S, Sk_nys);
    }

    cout << "Completed for f = " << F[f] << endl;
  }

  write_1D_data(("output/modes_" + case_name).c_str(), n_modes, modes);
  write_1D_data(("output/Norm_S0_" + case_name).c_str(), N, Norm_S);
  write_2D_data(("output/Norm_pod_m" + to_string(m) + "_" + case_name).c_str(), N, n_modes, Norm_pod);
  write_2D_data(("output/Norm_nys_m" + to_string(m) + "_" + case_name).c_str(), N, n_modes, Norm_nys);

  //Free-up memory
  free(S);
  free(Sk_pod);
  free(Sk_nys);
  free(S_zero);
  free(Dpod);
  free(Vpod);
  free(Dnys);
  free(Vnys);
}

void SRM::write_spectral_error(int m, int k)
{
  int n = npoints;
  // int m = ncols;
  // int k = nmodes;
  ncols = m;
  nmodes = k;

  // cout << "k = " << k << "\tm = " << m << endl;

  vector<double>F(N,0.0); //Array of frequency
  double* S = (double*) calloc(n*n, sizeof(double));

  //Reconstructed matrix
  double* Sk_pod = (double*) calloc(n*n, sizeof(double));
  double* Sk_nys = (double*) calloc(n*n, sizeof(double));

  double* Sk_area_pod = (double*) calloc(n*n, sizeof(double));
  double* Sk_area_nys = (double*) calloc(n*n, sizeof(double));
  double* S_area = (double*) calloc(n*n, sizeof(double));

  //Eigen decomposition
  double *Dpod  = (double*) calloc(n, sizeof(double));
  double *Vpod  = (double*) calloc(n*n, sizeof(double));
  double *Dnys  = (double*) calloc(m, sizeof(double));
  double *Vnys  = (double*) calloc(n*m, sizeof(double));

  for(int f = 0; f < N; f++)
  {
    F[f] = df*(f + 0.5);
```

```
    //Calculate the XPSD matrix
    calc_xpsd_matrix(F[f], S);
    matrix_add(n, n, S, S_area, S_area);

    //Eigen decompose
    eigen_decompose(n, S, Vpod, Dpod);
    low_rank_approx(n, n, k, Vpod, Dpod, Sk_pod);
    matrix_add(n, n, Sk_pod, Sk_area_pod, Sk_area_pod);

    //Nystrom approx
    nystrom_approx(n, m, S, Vnys, Dnys);
    low_rank_approx(n, m, k, Vnys, Dnys, Sk_nys);
    matrix_add(n, n, Sk_nys, Sk_area_nys, Sk_area_nys);

    cout << "Completed for f = " << F[f] << endl;
  }

  write_2D_data(("output/S_area_pod_m" + to_string(m) + "_k" + to_string(k) + "_" + case_name).c_str(), n, n, Sk_area_pod);
  write_2D_data(("output/S_area_nys_m" + to_string(m) + "_k" + to_string(k) + "_" + case_name).c_str(), n, n, Sk_area_nys);
  write_2D_data(("output/S_area_" + case_name).c_str(), n, n, S_area);

  //Free-up memory
  free(S);
  free(Sk_pod);
  free(Sk_nys);
  free(Sk_area_pod);
  free(Sk_area_nys);
  free(Dpod);
  free(Vpod);
  free(Dnys);
  free(Vnys);
}

void SRM::generate_velocity()
{
  int n = npoints;
  int m = ncols;
  int k = nmodes;

  cout << "m = " << m << endl;
  cout << "k = " << k << endl;
  cout << "sim type = " << sim_type << endl;

  if (sim_type=="CHO")
  {
    m = n;
    k = n;
  }

  double decomp_time = 0.0;
  double fft_time = 0.0;

  vector<double*>H(N);

  //CPSD matrix
  double* S = (double*) calloc(n*n, sizeof(double));

  //Decompose for each frequency
  for(int f = 0; f < N; f++)
  {
    H[f] = (double*) calloc(n*k, sizeof(double));

    calc_xpsd_matrix(df*(f + 0.5), S);

    auto start_decomp = high_resolution_clock::now();

    if(sim_type=="POD")
    {
      eigen_decompose_H(n, k, S, H[f]);
    }
    else if(sim_type=="NYPOD")
    {
      nystrom_approx_H(n, m, k, S, H[f]);
    }
```

```
  else if(sim_type=="CHO")
  {
    cholesky_decompose_H(n, S, H[f]);
  }
  else
  {
    printf("Simulation␣type␣is␣not␣known!\n");
    exit(EXIT_FAILURE);
  }

  auto end_decomp = high_resolution_clock::now();

  decomp_time += 1.0e-6*duration_cast<microseconds>(end_decomp - start_decomp).count();

  cout << "Completed␣for␣f␣=␣" << df*(f + 0.5)
                               <<",␣" << 100.0*(f + 1.0)/N
                               << "%,␣CPU␣time:␣" << decomp_time << endl;

}

cout << "Decomposition␣elapsed␣time␣=␣" << decomp_time << endl;

srand(seed);

//Loop to generate sample functions with different realization.
for(int si=0; si < nsamples; si++)
{
  auto start_fft = high_resolution_clock::now();

  //Velocity time series data
  double* U = (double*) calloc(Nt*n, sizeof(double));

  //Have a different seed number for each realization.
  seed = rand();

  //Generate the time-series using IFFT
  if(ergodic=="off")
  {
    fft_U(n, k, H, U); //non-ergodic
  }
  else if(ergodic=="on")
  {
    //This function need to checked further(not fully ergodic!)
    fft_U_ergodic(n, k, H, U); //ergodic
  }

  auto end_fft = high_resolution_clock::now();

  fft_time =  1.0e-6*duration_cast<microseconds>(end_fft - start_fft).count();

  cout << "FFT␣elapsed␣time␣=␣" << fft_time << endl;
  cout << "Total␣elapsed␣time␣=␣" << decomp_time + fft_time << endl;

  //Add mean
  if (add_mean=="on")
  {
    for(int i=0; i < n; i++)
    {
      for(int t=0; t < Nt; t++)
      {
        U[t*n + i] += Uav[i];
      }
    }
  }

  //Write the data
  if(sim_type=="POD")
  {
    write_2D_data(("output/U_pod_k" + to_string(k) + "_" + case_name + to_string(si)).c_str(), Nt, n, U);
  }
  else if(sim_type=="NYPOD")
  {
    write_2D_data(("output/U_nys_m" + to_string(m) + "_k" + to_string(k) + "_" + case_name + to_string(si)).c_str(), Nt, n,
          U);
```

```
    }
    else if(sim_type=="CHO")
    {
      write_2D_data(("output/U_cho_k" + to_string(k) + "_" + case_name + to_string(si)).c_str(), Nt, n, U);
    }

    free(U);
  }

  for(int f = 0; f < N; f++)
  {
    free(H[f]);
  }

  free(S);
}

void SRM::test_cpsd_matrix()
{
  int n = npoints;
  int m = ncols;
  int k = nmodes;

  double *S  = (double*) calloc(n*n, sizeof(double)); //XPSD matrix.
  double *D  = (double*) calloc(n, sizeof(double));
  double *V  = (double*) calloc(n*n, sizeof(double));
  double *Dm  = (double*) calloc(m, sizeof(double));
  double *Vm  = (double*) calloc(n*m, sizeof(double));
  double *Sk  = (double*) calloc(n*n, sizeof(double));
  double *Sm  = (double*) calloc(n*n, sizeof(double));
  double *H  = (double*) calloc(n*n, sizeof(double));
  double *Hm  = (double*) calloc(n*m, sizeof(double));
  double *Hk_pod  = (double*) calloc(n*k, sizeof(double));
  double *Hk_nys  = (double*) calloc(n*k, sizeof(double));

  double freq = 1.0;

  //Form the XPSD matrix
  calc_xpsd_matrix(freq, S);

  //Print the original matrix
  cout << "\nXPSD matrix at f = " << freq << endl;
  print_matrix(n, n, S);

  write_2D_data(("output/S0_f" + to_string(freq) + "_" + case_name).c_str(), n, n, S);

  //Perform eigendecomposition
  eigen_decompose(n, S, V, D);

  //Get low rank approximation from
  //Eigen decomposition
  low_rank_approx(n, n, m, V, D, Sk);

  //Print eigen-low rank approx
  cout << "\nEigen decompositon" << endl;
  cout << "Eigen vectors:" << endl;
  print_matrix(n, n, Sk);
  printf("Error norm  = %lf\n", frobenius_norm(n, S, Sk));
  cout << "Eigen values:" << endl;
  print_vector(n, D);

  cout << "\nH-matrix :" << endl;
  get_H(n, n, V, D, H);
  eigen_decompose_H(n, k, S, Hk_pod);

  cout << "\nMethod 1" << endl;
  print_matrix(n, n, H);

  cout << "\nMethod 2" << endl;
  print_matrix(n, k, Hk_pod);

  //Get nystrom approx of the eigen decomp
  nystrom_approx(n, m, S, Vm, Dm);
```

```
  //Get low rank based on nystrom
  low_rank_approx(n, m, m, Vm, Dm, Sm);

  //Print nystrom approx of the xpsd
  cout << "\nNystrom decompositon" << endl;
  cout << "Eigen vectors:" << endl;
  print_matrix(n, n, Sm);
  printf("Error norm = %lf\n", frobenius_norm(n, S, Sm));
  cout << "Eigen values:" << endl;
  print_vector(m, Dm);

  //Perform Nystrom approximation of H matrix
  get_H(n, m, Vm, Dm, Hm);
  nystrom_approx_H(n, m, k, S, Hk_nys);

  cout << "\nNystrom approximation of H matrix" << endl;
  cout << "\nApprox 1" << endl;
  print_matrix(n, k, Hk_nys);
  cout << "\nApprox 2" << endl;
  print_matrix(n, m, Hm);
}

//Sets m, k, sime_type parameters.
void SRM::set_params(int m, int k, string sim)
{
  ncols = m;
  nmodes = k;
  sim_type = sim;

  sample_columns(npoints, ncols);
}

void SRM::generate_velocity_tall_building()
{
  //Copy wind profiles originals
  vector<double> UavOrig;
  vector<double> IuOrig;
  vector<double> LuOrig;

  UavOrig.resize(npoints, 0.0);
  IuOrig.resize(npoints, 0.0);
  LuOrig.resize(npoints, 0.0);

  for(int p=0; p < npoints; p++)
  {
    UavOrig[p] = Uav[p];
    IuOrig[p] = I[p];
    LuOrig[p] = L[p];
  }

  std::default_random_engine generator(seed);

  const int nUav = 1;
  const int nIu =  1;
  const int nLu =  1;

  double scaleUav, scaleIu, scaleLu;

  const double coefVar = 0.1;

  //Specify mean and std of the profiles
  std::normal_distribution<double> distUav(1.0, coefVar);
  std::normal_distribution<double> distIu(1.0, coefVar);
  std::normal_distribution<double> distLu(1.0, 2.0*coefVar);

  std::uniform_int_distribution<> distSeed(0, 1000000);

  int count = 1;

  for(int i=0; i<nUav; i++)
  {
    for(int j=0; j<nIu; j++)
    {
      for(int k=0; k<nLu; k++)
```

```
    {
      do
      {
        scaleUav = distUav(generator);
        scaleIu = distIu(generator);
        scaleLu = distLu(generator);
      }
      while(scaleUav < 0.2 || scaleIu < 0.2 || scaleLu < 0.2);

      for(int p=0; p < npoints; p++)
      {
        Uav[p] = scaleUav*UavOrig[p];
        I[p] = scaleIu*IuOrig[p];
        L[p] = scaleLu*LuOrig[p];
      }

      seed = distSeed(generator);
      case_name = "Sample" + to_string(count);

      cout <<"Case␣name:␣" << case_name << endl;
      cout <<"Seed␣:␣" << seed << endl;
      cout <<"scaleUav␣:␣" << scaleUav << endl;
      cout <<"scaleIu␣:␣" << scaleIu << endl;
      cout <<"scaleLu␣:␣" << scaleLu << endl;

      generate_velocity();

      count++;

      cout <<"\n++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++\n" << endl;
    }
  }
 }

 Uav = UavOrig;
 I = IuOrig;
 L = LuOrig;
}
```

Listing C.1: Selected source code of NY-POD and POD methods

Appendix D

Source code of the windFSI framework

Here sample header files from the implemented framework are given in Listings D.1, D.2, D.3. The full C++ source code for the `windFSI` framework will be made available at https://github.com/GBitsuamlak/windFSI. The GitHub repository contains detailed documentation and installation instructions. Also, the OpenFOAM tutorial case files used to set up the wind-induced vibration of a tall building will be provided. The source code is organized into three core components corresponding to the structural, fluid, and dynamic mesh subsystems.

D.1 Structural subsystem

```
/*
Class
    Foam::structuralMotion

Description
A multi-degree of freedom(MDOF) motion for a deformable body. The structural deformation has a Lagrangian reference frame.
     The deformation of the structure is always measured from an undeformed state. At each time step, the displacement of
     the structure is applied to the mesh. The time-integrator for the motion is run-time selectable with options for
     Newmark-Beta(implicit), and 4-th order Runge-Kutta schemes.
*/
namespace Foam
{
    namespace FSI
    {
        class structuralSolver;
        class structuralMotion
        {
            friend class structuralSolver;

            structuralProperties strProps_;
            structuralMotionState motionState0_;
            structuralMotionState motionState_;
            vectorField floorCenters_;
            vectorField displacement_;
            vectorField velocity_;
            vectorField acceleration_;
            point origin_;
            vector strDir_;
            Switch report_;
            autoPtr<structuralSolver> solver_;

            void updateStructuralMotion();
```

```
            inline const scalarField& gd() const;
            inline const scalarField& gv() const;
            inline const scalarField& ga() const;
            inline const scalarField& gF() const;
            inline scalarField& gd();
            inline scalarField& gv();
            inline scalarField& ga();
            inline scalarField& gF();

            public:
            TypeName("structuralMotion");
            structuralMotion();
            structuralMotion(const dictionary& dict, const dictionary& stateDict);
            structuralMotion(const dictionary& dict);
            structuralMotion(const structuralMotion&);
            virtual ~structuralMotion();
            inline const structuralMotionState& state() const;
            inline const structuralMotionState& state0() const;
            inline const fileName& dataPath() const;
            inline const point& origin() const;
            inline const point centerOfRotation() const;
            inline const vector& strDir() const;
            inline label nStorey() const;
            inline label nMode() const;
            inline label nModeUsed() const;
            inline const scalarField& storeyHeights() const;
            inline const vectorField& floorCenters() const;
            inline const vectorField& masses() const;
            inline const List<vectorField>& modeShapes() const;
            inline const vectorField& displacement() const;
            inline const vectorField& velocity() const;
            inline const vectorField& acceleration() const;
            inline bool report() const;
            void update(bool firstIter,const vectorField& fGlobal,scalar deltaT,scalar deltaT0);
            void status() const;
            inline point transform(const point& initialPoints) const;
            tmp<pointField> transform(const pointField& initialPoints) const;
            tmp<pointField> transform(const pointField& initialPoints,const scalarField& scale) const;
            void write(Ostream&) const;
            bool read(const dictionary& dict);
            };
        }
    }

/*****************************************************************************************************************************************/
/*-------------------------------------------------------------------------*\
Source file: structuralMotion.C
\*-------------------------------------------------------------------------*/
#include "structuralMotion.H"
#include "structuralSolver.H"

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

namespace Foam
{
    namespace FSI
    {
        defineTypeNameAndDebug(structuralMotion, 0);
    }
}

// * * * * * * * * * * * * * * * Private Member Functions  * * * * * * * * * * * * * //
void Foam::FSI::structuralMotion::updateStructuralMotion()
{
    displacement_ = Zero;
    velocity_ = Zero;
    acceleration_ = Zero;

    forAll(displacement_, i)
    {
        for(label m=0; m < nMode(); m++)
        {
            displacement_[i] += modeShapes()[m][i]*gd()[m];
            velocity_[i] += modeShapes()[m][i]*gv()[m];
```

```
            acceleration_[i] += modeShapes()[m][i]*ga()[m];
        }
    }
}

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

Foam::FSI::structuralMotion::structuralMotion()
:
    strProps_(),
    motionState0_(0),
    motionState_(0),
    floorCenters_(0),
    displacement_(0),
    velocity_(0),
    acceleration_(0),
    origin_(vector::zero),
    strDir_(vector::zero),
    report_(false),
    solver_(nullptr)
{}

Foam::FSI::structuralMotion::structuralMotion
(
    const dictionary& dict
)
:
    strProps_(dict.subDict("structuralProperties")),
    motionState0_(strProps_.nMode()),
    motionState_(strProps_.nMode()),
    floorCenters_(strProps_.nStorey(), vector::zero),
    displacement_(strProps_.nStorey(), vector::zero),
    velocity_(strProps_.nStorey(), vector::zero),
    acceleration_(strProps_.nStorey(), vector::zero),
    origin_(dict.lookupOrDefault("origin", vector::zero)),
    strDir_(dict.lookupOrDefault("strDir", vector(0.0, 0.0, 1.0))),
    report_(dict.lookupOrDefault<Switch>("report", false)),
    solver_(structuralSolver::New(dict.subDict("structuralSolver"), *this))
{

    floorCenters_ = storeyHeights()*strDir_/mag(strDir_);

    updateStructuralMotion();
}

Foam::FSI::structuralMotion::structuralMotion
(
    const dictionary& dict,
    const dictionary& stateDict
)
:
    strProps_(dict.subDict("structuralProperties")),
    motionState0_(stateDict, strProps_.nMode()),
    motionState_(stateDict, strProps_.nMode()),
    floorCenters_(strProps_.nStorey(), vector::zero),
    displacement_(strProps_.nStorey(), vector::zero),
    velocity_(strProps_.nStorey(), vector::zero),
    acceleration_(strProps_.nStorey(), vector::zero),
    origin_(dict.lookupOrDefault("origin", vector::zero)),
    strDir_(dict.lookupOrDefault("strDir", vector(0.0, 0.0, 1.0))),
    report_(dict.lookupOrDefault<Switch>("report", false)),
    solver_(structuralSolver::New(dict.subDict("solver"), *this))
{

    floorCenters_ = storeyHeights()*strDir_/mag(strDir_) + origin();

    updateStructuralMotion();
}


Foam::FSI::structuralMotion::structuralMotion
(
    const structuralMotion& sM
)
```

```
:
    strProps_(sM.strProps_),
    motionState0_(sM.motionState0_),
    motionState_(sM.motionState_),
    floorCenters_(sM.floorCenters_),
    displacement_(sM.displacement_),
    velocity_(sM.velocity_),
    acceleration_(sM.acceleration_),
    origin_(sM.origin_),
    strDir_(sM.strDir_),
    report_(sM.report_)
{}


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

Foam::FSI::structuralMotion::~structuralMotion()
{}


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

void Foam::FSI::structuralMotion::update
(
    bool firstIter,
    const vectorField& fGlobal,
    scalar deltaT,
    scalar deltaT0
)
{

    // Save the old-time motion state for the first time step(outer iteration)
    if(firstIter)
    {
        motionState0_ = motionState_;
    }

    //Store the inner iteration values for convergence check
    motionState_.gd0() = motionState_.gd();

    //Solve the structure only on the master processor.
    if (Pstream::master())
    {
        solver_->solve(fGlobal, deltaT, deltaT0);
    }

    //Scatter modal displacements
    Pstream::scatter(motionState_);

    //Calculate the structural responses in Global co-ord system.
    updateStructuralMotion();

    if(report_)
    {
        status();
    }
}


void Foam::FSI::structuralMotion::status() const
{
    Info<< "Structural response at the tip:" << nl
        << "    Displacement: " << displacement().last() << nl
        << "    Velocity: " << velocity().last() << nl
        << "    Acceleration: " << acceleration().last() << nl
        << endl;
}


Foam::tmp<Foam::pointField> Foam::FSI::structuralMotion::transform
(
    const pointField& initialPoints
) const
{
```

```
    tmp<pointField> tpoints(new pointField(initialPoints));
    pointField& points = tpoints.ref();

    vector dispi(vector::zero);

    forAll(points, pointi)
    {

        //Interpolate translation and rotation for a point using storey data
        dispi = interpolateXY<vector>
        (
            initialPoints[pointi] & strDir_,
            storeyHeights(),
            displacement_
        );

        // Calculate the transformation septernion from the initial state
        septernion s
        (
            vector(dispi.x(), dispi.y(), 0.0),
            quaternion(strDir_, dispi.z())
        );

        //Calculate the transformation
        //invTransformPoint works counter-clock-wise
        points[pointi] = s.invTransformPoint(initialPoints[pointi]);
    }

    return tpoints;
}


Foam::tmp<Foam::pointField> Foam::FSI::structuralMotion::transform
(
    const pointField& initialPoints,
    const scalarField& scale
) const
{


    tmp<pointField> tpoints(new pointField(initialPoints));
    pointField& points = tpoints.ref();

    vector dispi(vector::zero);

    forAll(points, pointi)
    {
        // Move non-stationary points
        if (scale[pointi] > small)
        {
            // Use solid-body motion where scale = 1
            if (scale[pointi] > 1 - small)
            {
                points[pointi] = transform(initialPoints[pointi]);
            }
            // Slerp septernion interpolation
            else
            {
                dispi = interpolateXY<vector>
                (
                    initialPoints[pointi] & strDir_,
                    storeyHeights(),
                    displacement_
                );

                // Calculate the transformation septernion given current state
                septernion s
                (
                    vector(dispi.x(), dispi.y(), 0.0),
                    quaternion(strDir_, dispi.z())
                );

                septernion ss(slerp(septernion::I, s, scale[pointi]));
```

```
                points[pointi] = ss.invTransformPoint(initialPoints[pointi]);
            }
        }
    }

    return tpoints;
}

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

/*
Class:
    Foam::structuralProperties

Description:
    This class holds the structural properties of the building model. For this version of the code, the structure is assumed
        to be just a multi-story building. The masses are concentrated on each floor, and the floors are assumed to be a
        rigid diaphragm. Only properties that are relevant for modal analysis are stored.
*/

namespace Foam
{
    namespace FSI
    {
        class structuralProperties
        {
            private:
                fileName dataDir_;
                label nStorey_;
                const label nStoreyDOF_ = 3;
                label nDOF_;
                label nMode_;
                label nModeUsed_;
                scalarField storeyHeights_;
                vectorField masses_;
                scalarField modalFrequencies_;
                scalarField modalDampingRatios_;
                List<vectorField> modeShapes_;
                tensor orientation_;
                scalarField gM_;
                scalarField gC_;
                scalarField gK_;
                word freqUnit_;
                scalar massScale_;
                scalar freqScale_;

            void calcGeneralizedProperties();

            public:
                inline const fileName& dataDir() const;
                inline label nStorey() const;
                inline label nDOF() const;
                inline label nStoreyDOF() const;
                inline label nMode() const;
                inline label nModeUsed() const;
                inline const scalarField& storeyHeights() const;
                inline const vectorField& masses() const;
                inline const scalarField& modalFrequencies() const;
                inline const scalarField& modalDampingRatios() const;
                inline const List<vectorField>& modeShapes() const;
                inline const scalarField& gM() const;
                inline const scalarField& gC() const;
                inline const scalarField& gK() const;
                inline const word& freqUnit() const;

                TypeName("structuralProperties");
                structuralProperties();
                structuralProperties(const dictionary& dict);
                virtual ~structuralProperties();
                bool read(const dictionary& dict);
                void write(Ostream& os) const;

        };
```

```
    }
}

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

/*---------------------------------------------------------------------------*\
Source file: structuralProperties.C
\*---------------------------------------------------------------------------*/
#include "structuralProperties.H"
#include "mathematicalConstants.H"


namespace Foam
{
    namespace FSI
    {
        defineTypeNameAndDebug(structuralProperties, 0);
    }
}

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

Foam::FSI::structuralProperties::structuralProperties()
:
    dataDir_("constant/structuralData"),
    nStorey_(0),
    nDOF_(0),
    nMode_(0),
    nModeUsed_(0),
    storeyHeights_(0),
    masses_(0),
    modalFrequencies_(0),
    modalDampingRatios_(0),
    modeShapes_(0),
    orientation_(tensor::I),
    freqUnit_(),
    massScale_(1.0),
    freqScale_(1.0)
{
}

Foam::FSI::structuralProperties::structuralProperties(const dictionary& dict)
:
    dataDir_(dict.lookupOrDefault<word>("dataDir", "constant/structuralData")),
    nStorey_(dict.lookup<label>("nStorey")),
    nDOF_(nStorey_*nStoreyDOF_),
    nMode_(dict.lookup<label>("nMode")),
    nModeUsed_(dict.lookupOrDefault<label>("nModeUsed",nMode_)),
    storeyHeights_(IFstream(dataDir_/"storeyHeights")()),
    masses_(IFstream(dataDir_/"masses")()),
    modalFrequencies_(IFstream(dataDir_/"modalFrequencies")()),
    modalDampingRatios_(IFstream(dataDir_/"modalDampingRatios")()),
    modeShapes_(IFstream(dataDir_/"modeShapes")()),
    orientation_(dict.lookupOrDefault("orientation", tensor::I)),
    freqUnit_(dict.lookupOrDefault<word>("freqUnit", "Hz")),
    massScale_(dict.lookupOrDefault<scalar>("massScale", 1.0)),
    freqScale_(dict.lookupOrDefault<scalar>("freqScale", 1.0))
{

    if
    (
        nStorey_ != storeyHeights_.size() ||
        nStorey_ != masses_.size() ||
        nMode_ != modalFrequencies_.size() ||
        nMode_ != modalDampingRatios_.size() ||
        nMode_ != modeShapes_.size()
    )
    {
        FatalErrorInFunction
            << "Dimensions␣do␣not␣match!␣Check␣the␣inputs␣in:␣" << dict.name()
            <<  exit(FatalError);
    }

    Info << "\n***************␣Structural␣Properties␣***************\n" << nl;
```

```
    Info << "Structural␣data␣path:␣" << dataDir_ << nl;
    Info << "Number␣of␣DOFs␣per␣storey:␣" << nStoreyDOF_ << nl;
    Info << "Number␣of␣storeys:␣" << nStorey_ << nl;
    Info << "Total␣number␣of␣DOFs␣:␣" << nDOF_ << nl;
    Info << "Number␣of␣modes:␣" << nMode_ << nl;
    Info << "Modal␣truncation:␣" << nModeUsed_ << nl;
    Info << "Modal␣frequencies:␣" << modalFrequencies_ << nl;
    Info << "Orientation␣of␣the␣structure:␣" << orientation_ << nl;
    Info << "Frequency␣unit:␣" << freqUnit_ << nl;
    Info << "Modal␣damping␣ratios:␣" << modalDampingRatios_ << nl;
    Info << "Mass␣scaling␣factor:␣" << massScale_ << nl;
    Info << "Frequency␣scaling␣factor:␣" << freqScale_ << nl;


    //Calculate structural properties
    calcGeneralizedProperties();

    //Generalized properties
    Info <<"Generalized␣masses:␣" << gM_ << nl;
    Info <<"Generalized␣damping:␣" << gC_ << nl;
    Info <<"Generalized␣stiffness:␣" << gK_ << nl;
}


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

Foam::FSI::structuralProperties::~structuralProperties()
{}


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

void Foam::FSI::structuralProperties::calcGeneralizedProperties()
{

    //Initialize generalized properties
    gM_ = scalarField(nMode_, 0.0);
    gC_ = scalarField(nMode_, 0.0);
    gK_ = scalarField(nMode_, 0.0);

    //- Calculate the generalized structural properties
    scalar HzToRad = 1.0;

    if(freqUnit_=="Hz")
    {
            HzToRad = constant::mathematical::twoPi;
    }

    for(label i=0; i < nMode_; i++)
    {
        for(label j=0; j < nStorey_; j++)
        {
            //Transform the mode shapes with the orientation tensor
            //This feature could be used to simulate different wind directions
            modeShapes_[i][j] =  orientation_ & modeShapes_[i][j];

            for(label k=0; k < nStoreyDOF_; k++)
            {
                gM_[i] += masses_[j][k]*Foam::pow(modeShapes_[i][j][k], 2.0);
            }
        }

        gM_[i]*=massScale_;
    }

    gC_ = 2.0*modalDampingRatios_*(freqScale_*modalFrequencies_*HzToRad)*gM_;
    gK_ = gM_*Foam::pow(freqScale_*modalFrequencies_*HzToRad, 2.0);
}

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

/*
Class
    Foam::structuralSolver
```

```
Description
    An abstract class that represents a structure solver. This solver is based on modal analysis. Only the generalized modal
        coordinates are used for the solution. The solver advances the solution by a one-time step for a given force vector
        .
*/

namespace Foam
{
    namespace FSI
    {
        class structuralSolver
        {
            protected:
                structuralMotion& body_;
                scalar relaxationFactor_;
                inline scalarField& gd();
                inline scalarField& gv();
                inline scalarField& ga();
                inline scalarField& gF();
                inline scalarField& gd0();
                inline scalarField& gv0();
                inline scalarField& ga0();
                inline scalarField& gF0();
                inline const scalarField& gM() const;
                inline const scalarField& gC() const;
                inline const scalarField& gK() const;
                inline const List<vectorField>& modeShapes() const;
                inline label nMode() const;
                inline label nStoreyDOF() const;
                inline scalar calcAcceleration(scalar d, scalar v, scalar f, label mode);
                inline scalar calcGeneralizedForce (const vectorField& fGlobal, label mode);

            public:
                TypeName("structuralSolver");
                declareRunTimeSelectionTable
                (
                    autoPtr,
                    structuralSolver,
                    dictionary,
                    (
                        const dictionary& dict,
                        structuralMotion& body
                    ),
                    (dict, body)
                );

                structuralSolver(const dictionary& dict, structuralMotion& body);
                virtual ~structuralSolver();
                static autoPtr<structuralSolver> New(const dictionary& dict,structuralMotion& body);
                virtual void solve(const vectorField& fGlobal, scalar deltaT, scalar deltaT0) = 0;
        };
    }
}

/*****************************************************************************************************************************************/
/*-------------------------------------------------------------------------*\
Source file: structuralSolver.C
\*-------------------------------------------------------------------------*/
#include "structuralSolver.H"

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

namespace Foam
{
    namespace FSI
    {
        defineTypeNameAndDebug(structuralSolver, 0);
        defineRunTimeSelectionTable(structuralSolver, dictionary);
    }
}


// * * * * * * * * * * * * * * * * * * Constructors  * * * * * * * * * * * * * * * * * //
```

```
Foam::FSI::structuralSolver::structuralSolver(const dictionary& dict, structuralMotion& body)
:
    body_(body),
    relaxationFactor_(dict.lookupOrDefault<scalar>("relaxationFactor", 1.0))
{}


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

Foam::FSI::structuralSolver::~structuralSolver()
{}

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

/*
Class
    Foam::structuralSolvers::NewmarkBeta

Description
    NewmarkBeta 2nd-order time-integrator for structural deformation motion.

    Example specification in dynamicMeshDict:
    solver
    {
        type    NewmarkBeta;
        gamma   0.5;
        beta    0.25;
    }
*/


namespace Foam
{
    namespace FSI
    {
        namespace structuralSolvers
        {
            class NewmarkBeta
            :
                public structuralSolver
            {
                private:
                    const scalar beta_;
                    const scalar gamma_;

                public:
                    TypeName("NewmarkBeta");
                    NewmarkBeta (const dictionary& dict, structuralMotion& body);
                    virtual ~NewmarkBeta();
                    virtual void solve (const vectorField& fGlobal, scalar deltaT, scalar deltaT0);
            };
        }
    }
}

/*****************************************************************************************************************************************/
/*-------------------------------------------------------------------------*\
Source file: NewmarkBeta.C
\*-------------------------------------------------------------------------*/

#include "NewmarkBeta.H"
#include "addToRunTimeSelectionTable.H"

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

namespace Foam
{
    namespace FSI
    {
        namespace structuralSolvers
        {
            defineTypeNameAndDebug(NewmarkBeta, 0);
            addToRunTimeSelectionTable(structuralSolver, NewmarkBeta, dictionary);
```

```
        }
    }
}


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

Foam::FSI::structuralSolvers::NewmarkBeta::NewmarkBeta
(
    const dictionary& dict,
    structuralMotion& body
)
:
    structuralSolver(dict, body),
    beta_(dict.lookupOrDefault<scalar>("beta", 0.25)),
    gamma_(dict.lookupOrDefault<scalar>("gamma", 0.5))
{}


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

Foam::FSI::structuralSolvers::NewmarkBeta::~NewmarkBeta()
{}


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

void Foam::FSI::structuralSolvers::NewmarkBeta::solve
(
    const vectorField& fGlobal,
    scalar deltaT,
    scalar deltaT0
)
{
  //Taken from Chopra third edition

  scalar gfNew, kHat, a, b, pHat, dd, dv, da, dNew, vNew, aNew;

  for(label i=0; i < nMode(); i++)
  {
      //Calculate generalized force for the new time step
      gfNew = calcGeneralizedForce(fGlobal, i);

      kHat = gK()[i] + gamma_*gC()[i]/(beta_*deltaT) + gM()[i]/(beta_*deltaT*deltaT);

      a = gM()[i]/(beta_*deltaT) + gamma_*gC()[i]/beta_;

      b = gM()[i]/(2.0*beta_) + deltaT*gC()[i]*(gamma_/(2.0*beta_)-1.0);

      pHat = gfNew - gF0()[i] + a*gv0()[i] + b*ga0()[i];

      dd = pHat/kHat;

      dv = dd*gamma_/(beta_*deltaT) - gamma_*gv0()[i]/beta_ + deltaT*ga0()[i]*(1.0- gamma_/(2.0*beta_));

      da = dd/(beta_*deltaT*deltaT) - gv0()[i]/(beta_*deltaT) - ga0()[i]/(2.0*beta_);

      //Calculate the next time step
      dNew = gd0()[i] + dd;
      vNew = gv0()[i] + dv;
      aNew = ga0()[i] + da;

      //Apply relaxation to the new values
      gd()[i] = gd()[i] + relaxationFactor_*(dNew - gd()[i]);
      gv()[i] = gv()[i] + relaxationFactor_*(vNew - gv()[i]);
      ga()[i] = ga()[i] + relaxationFactor_*(aNew - ga()[i]);

      gF()[i] = gfNew;
  }
}

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

/*
```

```
Class
    Foam::structuralSolvers::RungeKutta

Description
    4th-Order Runge-Kutta methods(explicit method).

    Example specification in dynamicMeshDict:
    solver
    {
        type    RungeKutta;
    }
*/

namespace Foam
{
    namespace FSI
    {
        namespace structuralSolvers
        {

            class RungeKutta
            :
                public structuralSolver
            {
                public:
                    TypeName("RungeKutta");
                    RungeKutta (const dictionary& dict, structuralMotion& body);
                    virtual ~RungeKutta();
                    virtual void solve ( const vectorField& fGlobal, scalar deltaT, scalar deltaT0 );
            };
        }
    }
}

/*****************************************************************************************************************************************/
/*-------------------------------------------------------------------------*\
Source file: RungeKutta.C
\*-------------------------------------------------------------------------*/

#include "RungeKutta.H"
#include "addToRunTimeSelectionTable.H"

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

namespace Foam
{
    namespace FSI
    {
        namespace structuralSolvers
        {
            defineTypeNameAndDebug(RungeKutta, 0);
            addToRunTimeSelectionTable(structuralSolver, RungeKutta, dictionary);
        }
    }
}


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

Foam::FSI::structuralSolvers::RungeKutta::RungeKutta
(
    const dictionary& dict,
    structuralMotion& body
)
:
    structuralSolver(dict, body)
{}


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

Foam::FSI::structuralSolvers::RungeKutta::~RungeKutta()
{}
```

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

void Foam::FSI::structuralSolvers::RungeKutta::solve
(
    const vectorField& fGlobal,
    scalar deltaT,
    scalar deltaT0
)
{
    //Algorithm taken from:
    //https://www.compadre.org/PICUP/resources/Numerical-Integration/
    scalar gfAvg, gfNew, k1d, k2d, k3d, k4d, k1v, k2v, k3v, k4v, dNew, vNew;

    for(label i=0; i < nMode(); i++)
    {
        //Calculate generalized force for the new time step
        gfNew = calcGeneralizedForce(fGlobal,i);

        gfAvg = 0.5*(gF0()[i] + gfNew);

        k1v = calcAcceleration(gd0()[i], gv0()[i], gF0()[i],i)*deltaT;
        k1d = gv0()[i]*deltaT;

        k2v = calcAcceleration(gd0()[i] + 0.5*k1d, gv0()[i] + 0.5*k1v, gfAvg, i)*deltaT;
        k2d = (gv0()[i] + 0.5*k1v)*deltaT;

        k3v = calcAcceleration(gd0()[i] + 0.5*k2d, gv0()[i] + 0.5*k2v, gfAvg, i)*deltaT;
        k3d = (gv0()[i] + 0.5*k2v)*deltaT;

        k4v = calcAcceleration(gd0()[i] + k3d, gv0()[i] + k3v, gfNew, i)*deltaT;
        k4d = (gv0()[i] + k3v)*deltaT;

        //Calculate the next time step
        dNew = gd0()[i] + (k1d + 2.0*k2d + 2.0*k3d +k4d)/6.0;
        vNew = gv0()[i] + (k1v + 2.0*k2v + 2.0*k3v +k4v)/6.0;

        //Apply relaxation to the previous iteration(or time step)
        gd()[i] = gd()[i] + relaxationFactor_*(dNew - gd()[i]);
        gv()[i] = gv()[i] + relaxationFactor_*(vNew - gv()[i]);

        //Solve acceleration from equilibrium
        ga()[i] = calcAcceleration(gd()[i], gv()[i], gfNew, i);
        gF()[i] = gfNew;
    }
}

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

/*
Class
    Foam::functionObjects::structuralResponseState

Description
    Writes the structural response state.

    Example of function object specification:
    structuralResponseState
    {
        type           structuralResponseState;
        libs           ("libstructuralResponseState.so");
    }
*/

namespace Foam
{
    namespace FSI
    {
        class structuralMotion;
    }

    namespace functionObjects
    {
        using namespace Foam::FSI;
```

```
        class structuralResponseState
        :
            public fvMeshFunctionObject,
            public logFiles
        {
            private:
                const structuralMotion& motion() const;

            protected:
                virtual void writeFileHeader(const label i = 0);

            public:
                TypeName("structuralResponseState");
                structuralResponseState(const word& name, const Time& runTime, const dictionary& dict);
                structuralResponseState(const structuralResponseState&) = delete;
                virtual ~structuralResponseState();
                virtual bool read(const dictionary&);
                virtual bool execute();
                virtual bool write();
                void operator=(const structuralResponseState&) = delete;
        };
    }
}

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

/*---------------------------------------------------------------------------*\
Source file: structuralResponseState.C
\*---------------------------------------------------------------------------*/

#include "structuralResponseState.H"
#include "dynamicMotionSolverFvMesh.H"
#include "structuralMotionMeshSolver.H"
#include "unitConversion.H"
#include "addToRunTimeSelectionTable.H"

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

namespace Foam
{
    namespace functionObjects
    {
        defineTypeNameAndDebug(structuralResponseState, 0);

        addToRunTimeSelectionTable
        (
            functionObject,
            structuralResponseState,
            dictionary
        );
    }
}

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

Foam::functionObjects::structuralResponseState::structuralResponseState
(
    const word& name,
    const Time& runTime,
    const dictionary& dict
)
:
    fvMeshFunctionObject(name, runTime, dict),
    logFiles(obr_, name)
{
    read(dict);
    resetName(typeName);
}


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

Foam::functionObjects::structuralResponseState::~structuralResponseState()
```

```
{}


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

bool Foam::functionObjects::structuralResponseState::read(const dictionary& dict)
{
    fvMeshFunctionObject::read(dict);
    // angleFormat_ = dict.lookupOrDefault<word>("angleFormat", "radians");

    return true;
}


void Foam::functionObjects::structuralResponseState::writeFileHeader(const label)
{
    OFstream& file = this->file();

    writeHeader(file, "Structural␣Response␣State");

    const vectorField& storeyPoints = motion().floorCenters();

    label nStorey = storeyPoints.size();

    writeHeaderValue(file, "storeys", nStorey);

    writeCommented(file, "x␣co-ords␣␣:");

    forAll(storeyPoints, pointi)
    {
        file << tab << storeyPoints[pointi].x();
    }
    file << nl;

    writeCommented(file, "y␣co-ords␣␣:");
    forAll(storeyPoints, pointi)
    {
        file << tab << storeyPoints[pointi].y();
    }
    file << nl;

    writeCommented(file, "z␣co-ords␣␣:");

    forAll(storeyPoints, pointi)
    {
        file << tab << storeyPoints[pointi].z();
    }

    file << nl;

    writeCommented(file, "Time");

    const word storeyResponseTypes("[displacement,velocity,acceleration]");
    for (label j = 0; j < nStorey; j++)
    {
        const word jn('(' + Foam::name(j) + ')');
        const word f("storeyResponse" + jn + storeyResponseTypes);

        file << tab << f;
    }
}


bool Foam::functionObjects::structuralResponseState::execute()
{
    return true;
}


const Foam::FSI::structuralMotion&
Foam::functionObjects::structuralResponseState::motion() const
{
    const dynamicMotionSolverFvMesh& mesh =
        refCast<const dynamicMotionSolverFvMesh>(obr_);
```

```
    return (refCast<const structuralMotion>(mesh.motion()));
}


bool Foam::functionObjects::structuralResponseState::write()
{
    logFiles::write();

    if (Pstream::master())
    {
        const structuralMotion& strMotion = this->motion();

        const vectorField& disp = strMotion.displacement();
        const vectorField& vel = strMotion.velocity();
        const vectorField& acc = strMotion.acceleration();

        writeTime(file());

        forAll(disp, i)
        {
            file()
                << tab << setw(1) << '('
                << disp[i] << setw(1) << '␣'
                << vel[i]  << setw(1) << '␣'
                << acc[i]  << setw(3) << ')';
        }

        file() << endl;
    }

    return true;
}

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

Listing D.1: Selected source code of `windFSI` framework for the structural subsystem

D.2 Fluid subsystem

```
/*****************************************************************************************************************************/
/*
Class
    Foam::fsiControl

Description
    FSI control class to supply convergence information/checks for the FSI loop. Used for controlling the fluid, dynamic mesh
        and structural subsystems
*/

namespace Foam
{
    namespace FSI
    {
        class structuralMotion;

        enum class fsiCouplingAlgorithms
        {
            CSS,    // Conventional Serial Staggered algorithm
            FPI     // Fixed point iteration algorithm
        };

        extern const NamedEnum<fsiCouplingAlgorithms, 2> fsiCouplingAlgorithmNames;
        extern const fsiCouplingAlgorithms defaultFsiCouplingAlgorithm;
```

```
        class fsiControl
        :
            public regIOobject
        {
            private:
                const structuralMotion& motion() const;

            protected:
                label maxFsiIter_;
                label fsiIter_;
                fsiCouplingAlgorithms algorithm_;
                scalar tolerance_;
                scalar relativeError_;
                scalar referenceDisplacement_;
                virtual bool writeData(Ostream&) const;
                void calcFsiError();

            public:
                TypeName("fsiControl");
                fsiControl(const objectRegistry& registry, const Time& time, const dictionary& dict);
                virtual ~fsiControl();
                virtual bool read(const dictionary& dict);
                inline label maxFsiIter() const;
                inline bool firstFsiIter() const;
                inline bool finalFsiIter() const;
                bool run();
                bool loop();
        };
    }
}

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

/*---------------------------------------------------------------------------*\
Source file: fsiControl.C
\*---------------------------------------------------------------------------*/

#include "fsiControl.H"
#include "dynamicMotionSolverFvMesh.H"
#include "structuralMotionMeshSolver.H"

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

template<>
const char* Foam::NamedEnum
<
    Foam::FSI::fsiCouplingAlgorithms,
    2
>::names[] =
{
    "CSS",
    "FPI"
};

const Foam::NamedEnum<Foam::FSI::fsiCouplingAlgorithms, 2>
    Foam::FSI::fsiCouplingAlgorithmNames;

const Foam::FSI::fsiCouplingAlgorithms

Foam::FSI::defaultFsiCouplingAlgorithm
(
    Foam::FSI::fsiCouplingAlgorithms::CSS
);

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

namespace Foam
{
    namespace FSI
    {
        defineTypeNameAndDebug(fsiControl, 0);
    }
}
```

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

bool Foam::FSI::fsiControl::writeData(Ostream&) const
{
    NotImplemented;
    return false;
}

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

Foam::FSI::fsiControl::fsiControl
(
    const objectRegistry& registry,
    const Time& time,
    const dictionary& dict
)
:
    regIOobject
    (
        IOobject
        (
            typeName,
            time.timeName(),
            registry,
            IOobject::NO_READ,
            IOobject::NO_WRITE
        )
    ),
    maxFsiIter_(-1),
    fsiIter_(0),
    algorithm_(),
    tolerance_(1e-6),
    relativeError_(1.0),
    referenceDisplacement_(1.0)
{
    read(dict);
}
// * * * * * * * * * * * * * * * * * * Destructor  * * * * * * * * * * * * * * * * * //

Foam::FSI::fsiControl::~fsiControl()
{}


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

const Foam::FSI::structuralMotion&
Foam::FSI::fsiControl::motion() const
{
    const dynamicMotionSolverFvMesh& mesh =
        refCast<const dynamicMotionSolverFvMesh>(this->db());

    return (refCast<const structuralMotion>(mesh.motion()));
}


bool Foam::FSI::fsiControl::loop()
{
    if(firstFsiIter())
    {
        fsiIter_++;
        return true;
    }
    else
    {
        calcFsiError();

        if(finalFsiIter())
        {
            fsiIter_ = 0;
            return false;
        }

        if(relativeError_ < tolerance_)
        {
```

```
            fsiIter_ = 0;
            return false;
        }

        fsiIter_++;
        return true;
    }
}

bool Foam::FSI::fsiControl::read(const dictionary& dict)
{
    maxFsiIter_ = dict.lookupOrDefault<label>("maxFsiIteration", 5);

    algorithm_ = fsiCouplingAlgorithmNames
                 [
                     dict.lookupOrDefault
                     (
                         "couplingAlgorithm",
                         word(fsiCouplingAlgorithmNames[defaultFsiCouplingAlgorithm])
                     )
                 ];

    if(algorithm_ == fsiCouplingAlgorithms::CSS)
    {
        maxFsiIter_ = 1;
    }

    tolerance_ = dict.lookupOrDefault<scalar>("fsiTolerance", 1e-6);
    referenceDisplacement_ = dict.lookupOrDefault<scalar>("referenceDisplacement", 1.0);

    fsiIter_ = 0;

    return true;
}

void Foam::FSI::fsiControl::calcFsiError()
{
    //Calculate the relative error norm

    const structuralMotion & motion = this->motion();

    relativeError_ = sum(mag(motion.state().gd() - motion.state().gd0()));

    relativeError_ /= referenceDisplacement_;

    Info << "FSI: relative error: sum of all generalized displacements = "
         << relativeError_ << endl;
}

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

Application
    pimpleFsiFoam

Description
    Transient FSI solver for incompressible, turbulent flow of Newtonian fluids
    and small displacement structural deformation with moving mesh.

    Turbulence modeling is generic, i.e. laminar, RAS or LES may be selected.

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

#include "fvCFD.H"
#include "dynamicFvMesh.H"
#include "singlePhaseTransportModel.H"
#include "kinematicMomentumTransportModel.H"
#include "pimpleControl.H"
#include "fsiControl.H"
#include "CorrectPhi.H"
#include "fvOptions.H"
#include "localEulerDdtScheme.H"
#include "fvcSmooth.H"

#include "dynamicMotionSolverFvMesh.H"
```

```
#include "displacementMotionSolver.H"

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

using namespace Foam::FSI;

int main(int argc, char *argv[])
{
    #include "postProcess.H"

    #include "setRootCaseLists.H"
    #include "createTime.H"
    #include "createDynamicFvMesh.H"
    #include "initContinuityErrs.H"
    #include "createDyMControls.H"
    #include "createFsiControls.H"
    #include "createFields.H"
    #include "createUfIfPresent.H"

    turbulence->validate();

    if (!LTS)
    {
        #include "CourantNo.H"
        #include "setInitialDeltaT.H"
    }

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

    Info<< "\nStarting time loop\n" << endl;

    while (pimple.run(runTime))
    {
        #include "readDyMControls.H"

        if (LTS)
        {
            #include "setRDeltaT.H"
        }
        else
        {
            #include "CourantNo.H"
            #include "setDeltaT.H"
        }

        runTime++;

        Info<< "Time = " << runTime.timeName() << nl << endl;

        while(fsi.loop())
        {
            //Solve the structure and move the mesh
            mesh.update();

            //Correct flux if the mesh is moving
            if (mesh.changing())
            {
                MRF.update();

                if (correctPhi)
                {
                    // Calculate absolute flux
                    // from the mapped surface velocity
                    phi = mesh.Sf() & Uf();

                    #include "correctPhi.H"

                    // Make the flux relative to the mesh motion
                    fvc::makeRelative(phi, U);
                }

                if (checkMeshCourantNo)
                {
                    #include "meshCourantNo.H"
```

```
                }
            }

            // --- Pressure-velocity PIMPLE corrector loop
            while (pimple.loop())
            {
                #include "UEqn.H"

                // --- Pressure corrector loop
                while (pimple.correct())
                {
                    #include "pEqn.H"
                }

                if (pimple.turbCorr())
                {
                    laminarTransport.correct();
                    turbulence->correct();
                }
            }
        }

        runTime.write();

        Info<< "ExecutionTime = " << runTime.elapsedCpuTime() << " s"
            << "  ClockTime = " << runTime.elapsedClockTime() << " s"
            << nl << endl;
    }

    Info<< "End\n" << endl;

    return 0;
}

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

/*
Class
    Foam::functionObjects::storeyForces

Description
    Calculates the storey forces and moments by integrating the pressure and skin-friction storey forces over a given list of
          patches (structural surface).

    Example of function object specification:
    storeyForces1
    {
        type        storeyForces;
        libs        ("libforces.so");
        ...
        log         yes;
        patches     (building);

        storeyData
        {
            nStorey     60;
            direction   (0 0 1);
        }
    }


*/
namespace Foam
{
    namespace functionObjects
    {
        class storeyForces
        :
            public fvMeshFunctionObject,
            public logFiles
        {
            protected:
                enum class fileID
                {
```

```
                    mainFile = 0,
                    storeysFile = 1
                };

                List<Field<vector>> force_;
                List<Field<vector>> storeyForce_;
                List<Field<vector>> moment_;
                labelHashSet patchSet_;
                word pName_;
                word UName_;
                word rhoName_;
                Switch directForceDensity_;
                word fDName_;
                scalar rhoRef_;
                scalar pRef_;
                coordinateSystem coordSys_;
                bool localSystem_;
                bool porosity_;
                label nStorey_;
                Switch readStoreyHeights_;
                List<scalar> storeyHeights_;
                vector storeyDir_;
                scalar storeyDx_;
                scalar storeyMin_;
                List<point> storeyPoints_;
                bool initialised_;

                using logFiles::file;

                Ostream& file(const fileID fid)
                {
                    return logFiles::file(label(fid));
                }
                wordList createFileNames(const dictionary& dict) const;
                virtual void writeFileHeader(const label i);
                void initialise();
                tmp<volSymmTensorField> devTau() const;
                tmp<volScalarField> mu() const;
                tmp<volScalarField> rho() const;
                scalar rho(const volScalarField& p) const;
                void calcStoreyForces (const vectorField& Md, const vectorField& fN, const vectorField& fT, const vectorField
                     & fP, const vectorField& d);
                void writeForces();
                void writeStoreyForces();


            public:
                TypeName("storeyForces");
                storeyForces (const word& name, const Time& runTime, const dictionary& dict);
                storeyForces (const word& name, const objectRegistry& obr, const dictionary&);
                storeyForces(const storeyForces&) = delete;
                virtual ~storeyForces();
                virtual bool read(const dictionary&);
                virtual void calcStoreyForcesMoments();
                virtual vector forceEff() const;
                virtual vector momentEff() const;
                virtual vectorField storeyForcesEff() const;
                virtual bool execute();
                virtual bool write();
                void operator=(const storeyForces&) = delete;
        };
    }
}
/*****************************************************************************************************************************************/

/*---------------------------------------------------------------------------*\
Source file: storeyForces.C
\*---------------------------------------------------------------------------*/

#include "storeyForces.H"
#include "fvcGrad.H"
#include "porosityModel.H"
#include "kinematicMomentumTransportModel.H"
#include "fluidThermoMomentumTransportModel.H"
```

```
#include "addToRunTimeSelectionTable.H"

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

namespace Foam
{
    namespace functionObjects
    {
        defineTypeNameAndDebug(storeyForces, 0);

        addToRunTimeSelectionTable(functionObject, storeyForces, dictionary);
    }
}


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

Foam::wordList Foam::functionObjects::storeyForces::createFileNames
(
    const dictionary& dict
) const
{
    DynamicList<word> names(1);

    const word forceType(dict.lookup("type"));

    // Name for file(fileID::mainFile=0)
    names.append(forceType+ "_base");

    const dictionary& storeyDict(dict.subDict("storeyData"));
    label nb = storeyDict.lookup<label>("nStorey");
    if (nb > 0)
    {
        // Name for file(fileID::storeysFile=1)
        names.append(forceType + "_storey");
    }

    return move(names);
}


void Foam::functionObjects::storeyForces::writeFileHeader(const label i)
{
    switch (fileID(i))
    {
        case fileID::mainFile:
        {
            // force data

            writeHeader(file(i), "Forces");
            writeHeaderValue(file(i), "CofR", coordSys_.origin());
            writeCommented(file(i), "Time");

            const word forceTypes("(pressure␣viscous␣porous)");
            file(i)
                << "storeyForces" << forceTypes << tab
                << "storeyMoments" << forceTypes;

            if (localSystem_)
            {
                file(i)
                    << tab
                    << "localForces" << forceTypes << tab
                    << "localMoments" << forceTypes;
            }

            break;
        }
        case fileID::storeysFile:
        {
            // storey data

            writeHeader(file(i), "Storey␣forces");
            writeHeaderValue(file(i), "storeys", nStorey_);
```

```
            writeHeaderValue(file(i), "start", storeyMin_);
            writeHeaderValue(file(i), "delta", storeyDx_);
            writeHeaderValue(file(i), "direction", storeyDir_);

            vectorField storeyPoints(nStorey_);
            writeCommented(file(i), "x␣co-ords␣␣:");
            forAll(storeyPoints, pointi)
            {
                // storeyPoints[pointi] = (storeyMin_ + (pointi + 1)*storeyDx_)*storeyDir_;
                storeyPoints[pointi] = (storeyMin_ + pointi*storeyDx_)*storeyDir_;
                file(i) << tab << storeyPoints[pointi].x();
            }
            file(i) << nl;

            writeCommented(file(i), "y␣co-ords␣␣:");
            forAll(storeyPoints, pointi)
            {
                file(i) << tab << storeyPoints[pointi].y();
            }
            file(i) << nl;

            writeCommented(file(i), "z␣co-ords␣␣:");
            forAll(storeyPoints, pointi)
            {
                file(i) << tab << storeyPoints[pointi].z();
            }
            file(i) << nl;

            writeCommented(file(i), "Time");

            const word storeyForceTypes("[pressure,viscous,porous]");
            for (label j = 0; j < nStorey_; j++)
            {
                const word jn('(' + Foam::name(j) + ')');
                const word f("storeyForces" + jn + storeyForceTypes);
                // const word m("storeyMoments" + jn + storeyForceTypes);

                file(i)<< tab << f;
            }
            if (localSystem_)
            {
                for (label j = 0; j < nStorey_; j++)
                {
                    const word jn('(' + Foam::name(j) + ')');
                    const word f("localForces" + jn + storeyForceTypes);
                    // const word m("localMoments" + jn + storeyForceTypes);

                    file(i)<< tab << f;
                }
            }

            break;
        }
        default:
        {
            FatalErrorInFunction
                << "Unhandled␣file␣index:␣" << i
                << abort(FatalError);
        }
    }

    file(i)<< endl;
}


void Foam::functionObjects::storeyForces::initialise()
{
    if (initialised_)
    {
        return;
    }

    if (directForceDensity_)
    {
```

```
        if (!obr_.foundObject<volVectorField>(fDName_))
        {
            FatalErrorInFunction
                << "Could not find " << fDName_ << " in database."
                << exit(FatalError);
        }
    }
    else
    {
        if
        (
            !obr_.foundObject<volVectorField>(UName_)
         || !obr_.foundObject<volScalarField>(pName_)

        )
        {
            FatalErrorInFunction
                << "Could not find " << UName_ << ", " << pName_
                << exit(FatalError);
        }

        if
        (
            rhoName_ != "rhoInf"
         && !obr_.foundObject<volScalarField>(rhoName_)
        )
        {
            FatalErrorInFunction
                << "Could not find " << rhoName_
                << exit(FatalError);
        }
    }

    initialised_ = true;
}


Foam::tmp<Foam::volSymmTensorField>
Foam::functionObjects::storeyForces::devTau() const
{
    typedef compressible::momentumTransportModel cmpTurbModel;
    typedef incompressible::momentumTransportModel icoTurbModel;

    if (obr_.foundObject<cmpTurbModel>(momentumTransportModel::typeName))
    {
        const cmpTurbModel& turb =
            obr_.lookupObject<cmpTurbModel>(momentumTransportModel::typeName);

        return turb.devTau();
    }
    else if (obr_.foundObject<icoTurbModel>(momentumTransportModel::typeName))
    {
        const incompressible::momentumTransportModel& turb =
            obr_.lookupObject<icoTurbModel>(momentumTransportModel::typeName);

        return rho()*turb.devSigma();
    }
    else if (obr_.foundObject<fluidThermo>(fluidThermo::dictName))
    {
        const fluidThermo& thermo =
            obr_.lookupObject<fluidThermo>(fluidThermo::dictName);

        const volVectorField& U = obr_.lookupObject<volVectorField>(UName_);

        return -thermo.mu()*dev(twoSymm(fvc::grad(U)));
    }
    else if
    (
        obr_.foundObject<transportModel>("transportProperties")
    )
    {
        const transportModel& laminarT =
            obr_.lookupObject<transportModel>("transportProperties");
```

```
        const volVectorField& U = obr_.lookupObject<volVectorField>(UName_);

        return -rho()*laminarT.nu()*dev(twoSymm(fvc::grad(U)));
    }
    else if (obr_.foundObject<dictionary>("transportProperties"))
    {
        const dictionary& transportProperties =
             obr_.lookupObject<dictionary>("transportProperties");

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

        const volVectorField& U = obr_.lookupObject<volVectorField>(UName_);

        return -rho()*nu*dev(twoSymm(fvc::grad(U)));
    }
    else
    {
        FatalErrorInFunction
            << "No valid model for viscous stress calculation"
            << exit(FatalError);

        return volSymmTensorField::null();
    }
}


Foam::tmp<Foam::volScalarField> Foam::functionObjects::storeyForces::mu() const
{
    if (obr_.foundObject<fluidThermo>(basicThermo::dictName))
    {
        const fluidThermo& thermo =
             obr_.lookupObject<fluidThermo>(basicThermo::dictName);

        return thermo.mu();
    }
    else if
    (
        obr_.foundObject<transportModel>("transportProperties")
    )
    {
        const transportModel& laminarT =
            obr_.lookupObject<transportModel>("transportProperties");

        return rho()*laminarT.nu();
    }
    else if (obr_.foundObject<dictionary>("transportProperties"))
    {
        const dictionary& transportProperties =
             obr_.lookupObject<dictionary>("transportProperties");

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

        return rho()*nu;
    }
    else
    {
        FatalErrorInFunction
            << "No valid model for dynamic viscosity calculation"
            << exit(FatalError);

        return volScalarField::null();
    }
}
```

```
Foam::tmp<Foam::volScalarField> Foam::functionObjects::storeyForces::rho() const
{
    if (rhoName_ == "rhoInf")
    {
        return volScalarField::New
        (
            "rho",
            mesh_,
            dimensionedScalar(dimDensity, rhoRef_)
        );
    }
    else
    {
        return(obr_.lookupObject<volScalarField>(rhoName_));
    }
}


Foam::scalar Foam::functionObjects::storeyForces::rho(const volScalarField& p) const
{
    if (p.dimensions() == dimPressure)
    {
        return 1.0;
    }
    else
    {
        if (rhoName_ != "rhoInf")
        {
            FatalErrorInFunction
                << "Dynamic␣pressure␣is␣expected␣but␣kinematic␣is␣provided."
                << exit(FatalError);
        }

        return rhoRef_;
    }
}


void Foam::functionObjects::storeyForces::calcStoreyForces
(
    const vectorField& Md,
    const vectorField& fN,
    const vectorField& fT,
    const vectorField& fP,
    const vectorField& d
)
{
    if (nStorey_ == 1)
    {
        force_[0][0] += sum(fN);
        force_[1][0] += sum(fT);
        force_[2][0] += sum(fP);
        moment_[0][0] += sum(Md^fN);
        moment_[1][0] += sum(Md^fT);
        moment_[2][0] += sum(Md^fP);
    }
    else
    {
        scalarField dd((d & storeyDir_) - storeyMin_);

        forAll(dd, i)
        {
            // label storeyi = min(max(floor(dd[i]/storeyDx_), 0), force_[0].size() - 1);

            //Make the floors the center of application
            label storeyi = min(max(floor((dd[i] + storeyDx_/2.0)/storeyDx_), 0), force_[0].size() - 1);

            //Exclude the bottom half of the first floor
            // if (dd[i] > storeyDx_/2.0)

            force_[0][storeyi] += fN[i];
            force_[1][storeyi] += fT[i];
            force_[2][storeyi] += fP[i];
```

```
            moment_[0][storeyi] += Md[i]^fN[i];
            moment_[1][storeyi] += Md[i]^fT[i];
            moment_[2][storeyi] += Md[i]^fP[i];
        }
    }
}


void Foam::functionObjects::storeyForces::writeForces()
{
    Log << type() << " " << name() << " write:" << nl
        << "    sum of storey forces:" << nl
        << "        pressure : " << sum(storeyForce_[0]) << nl
        << "        viscous  : " << sum(storeyForce_[1]) << nl
        << "        porous   : " << sum(storeyForce_[2]) << nl
        << endl;

    writeTime(file(fileID::mainFile));

    file(fileID::mainFile) << tab << setw(1) << '('
        << sum(storeyForce_[0]) << setw(1) << ' '
        << sum(storeyForce_[1]) << setw(1) << ' '
        << sum(storeyForce_[2]) << setw(3) << ')';

    if (localSystem_)
    {
        vectorField localForceN(coordSys_.localVector(storeyForce_[0]));
        vectorField localForceT(coordSys_.localVector(storeyForce_[1]));
        vectorField localForceP(coordSys_.localVector(storeyForce_[2]));

        file(fileID::mainFile) << tab << setw(1) << '('
            << sum(localForceN) << setw(1) << ' '
            << sum(localForceT) << setw(1) << ' '
            << sum(localForceP) << setw(3) << ')';
    }

    file(fileID::mainFile) << endl;
}


void Foam::functionObjects::storeyForces::writeStoreyForces()
{
    if (nStorey_ == 1)
    {
        return;
    }
    writeTime(file(fileID::storeysFile));

    forAll(storeyForce_[0], i)
    {
        file(fileID::storeysFile)
            << tab << setw(1) << '('
            << storeyForce_[0][i] << setw(1) << ' '
            << storeyForce_[1][i] << setw(1) << ' '
            << storeyForce_[2][i] << setw(3) << ')';
    }

    if (localSystem_)
    {
        List<Field<vector>> lf(3);
        lf[0] = coordSys_.localVector(storeyForce_[0]);
        lf[1] = coordSys_.localVector(storeyForce_[1]);
        lf[2] = coordSys_.localVector(storeyForce_[2]);

        forAll(lf[0], i)
        {
            file(fileID::storeysFile)
                << tab << setw(1) << '('
                << lf[0][i] << setw(1) << ' '
                << lf[1][i] << setw(1) << ' '
                << lf[2][i] << setw(3) << ')';
        }
    }
```

```
    file(fileID::storeysFile) << endl;
}


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

Foam::functionObjects::storeyForces::storeyForces
(
    const word& name,
    const Time& runTime,
    const dictionary& dict
)
:
    fvMeshFunctionObject(name, runTime, dict),
    logFiles(obr_, name),
    force_(3),
    storeyForce_(3),
    moment_(3),
    patchSet_(),
    pName_(word::null),
    UName_(word::null),
    rhoName_(word::null),
    directForceDensity_(false),
    fDName_(""),
    rhoRef_(vGreat),
    pRef_(0),
    coordSys_(),
    localSystem_(false),
    porosity_(false),
    nStorey_(1),
    readStoreyHeights_(false),
    storeyHeights_(0),
    storeyDir_(Zero),
    storeyDx_(0.0),
    storeyMin_(great),
    storeyPoints_(),
    initialised_(false)
{
    read(dict);
    resetNames(createFileNames(dict));
}


Foam::functionObjects::storeyForces::storeyForces
(
    const word& name,
    const objectRegistry& obr,
    const dictionary& dict
)
:
    fvMeshFunctionObject(name, obr, dict),
    logFiles(obr_, name),
    force_(3),
    storeyForce_(3),
    moment_(3),
    patchSet_(),
    pName_(word::null),
    UName_(word::null),
    rhoName_(word::null),
    directForceDensity_(false),
    fDName_(""),
    rhoRef_(vGreat),
    pRef_(0),
    coordSys_(),
    localSystem_(false),
    porosity_(false),
    nStorey_(1),
    readStoreyHeights_(false),
    storeyHeights_(0),
    storeyDir_(Zero),
    storeyDx_(0.0),
    storeyMin_(great),
    storeyPoints_(),
    initialised_(false)
```

```
{
    read(dict);
    resetNames(createFileNames(dict));
}


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

Foam::functionObjects::storeyForces::~storeyForces()
{}


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

bool Foam::functionObjects::storeyForces::read(const dictionary& dict)
{
    fvMeshFunctionObject::read(dict);

    initialised_ = false;

    Log << type() << " " << name() << ":" << nl;

    directForceDensity_ = dict.lookupOrDefault("directForceDensity", false);

    const polyBoundaryMesh& pbm = mesh_.boundaryMesh();

    patchSet_ = pbm.patchSet(wordReList(dict.lookup("patches")));

    if (directForceDensity_)
    {
        // Optional entry for fDName
        fDName_ = dict.lookupOrDefault<word>("fD", "fD");
    }
    else
    {
        // Optional entries U and p
        pName_ = dict.lookupOrDefault<word>("p", "p");
        UName_ = dict.lookupOrDefault<word>("U", "U");
        rhoName_ = dict.lookupOrDefault<word>("rho", "rho");

        // Reference density needed for incompressible calculations
        if (rhoName_ == "rhoInf")
        {
            dict.lookup("rhoInf") >> rhoRef_;
        }

        // Reference pressure, 0 by default
        pRef_ = dict.lookupOrDefault<scalar>("pRef", 0.0);
    }

    coordSys_.clear();

    // Centre of rotation for moment calculations
    // specified directly, from coordinate system, or implicitly (0 0 0)
    if (!dict.readIfPresent<point>("CofR", coordSys_.origin()))
    {
        coordSys_ = coordinateSystem(obr_, dict);
        localSystem_ = true;
    }

    dict.readIfPresent("porosity", porosity_);
    if (porosity_)
    {
        Log << "    Including porosity effects" << endl;
    }
    else
    {
        Log << "    Not including porosity effects" << endl;
    }

    if (dict.found("storeyData"))
    {
        const dictionary& storeyDict(dict.subDict("storeyData"));
        storeyDict.lookup("nStorey") >> nStorey_;
```

```
readStoreyHeights_ = storeyDict.lookupOrDefault<Switch>("readStoreyData","off");

if(readStoreyHeights_)
{
    // storeyDict.lookup("storeyDataPath") >> storeyDataPath_;
    storeyDict.lookup("storeyHeights") >> storeyHeights_;

    // IFstream storeyFile(storeyDataPath_/"storeyHeights")

    // storeyFile >> storeyHeights_;

    if(storeyHeights_.size() != nStorey_)
    {
        FatalIOErrorInFunction(dict)
            << "Number␣of␣storeys␣do␣not␣match!"
            << exit(FatalIOError);
    }
}

if (nStorey_ < 0)
{
    FatalIOErrorInFunction(dict)
        << "Number␣of␣storeys␣(nStorey)␣must␣be␣zero␣or␣greater"
        << exit(FatalIOError);
}
else if ((nStorey_ == 0) || (nStorey_ == 1))
{
    nStorey_ = 1;
    forAll(force_, i)
    {
        force_[i].setSize(1);
        storeyForce_[i].setSize(1);
        moment_[i].setSize(1);
    }
}

if (nStorey_ > 1)
{
    storeyDict.lookup("direction") >> storeyDir_;
    storeyDir_ /= mag(storeyDir_);

    storeyMin_ = great;
    scalar storeyMax = -great;
    forAllConstIter(labelHashSet, patchSet_, iter)
    {
        label patchi = iter.key();
        const polyPatch& pp = pbm[patchi];
        // scalarField d(pp.faceCentres() & storeyDir_);
        scalarField d(pp.localPoints() & storeyDir_); //use patch points
        storeyMin_ = min(min(d), storeyMin_);
        storeyMax = max(max(d), storeyMax);
    }

    reduce(storeyMin_, minOp<scalar>());
    reduce(storeyMax, maxOp<scalar>());

    // slightly boost storeyMax so that region of interest is fully
    // within bounds
    storeyMax = 1.0001*(storeyMax - storeyMin_) + storeyMin_;

    //Here the nStorey is treated as number of floor levels of the bldg
    //including the ground and roof. So we subtract 1
    storeyDx_ = (storeyMax - storeyMin_)/scalar(nStorey_-1);

    // create the storey points used for writing
    storeyPoints_.setSize(nStorey_);

    forAll(storeyPoints_, i)
    {
        // storeyPoints_[i] = (i + 0.5)*storeyDir_*storeyDx_;
        storeyPoints_[i] = i*storeyDir_*storeyDx_;
    }
```

```
            // check if the storey heights are within the bound and
            // update storeyPoints_ with the storey height information
            if(readStoreyHeights_)
            {
                if(storeyHeights_[nStorey_-1] > (storeyMax - storeyMin_))
                {
                    FatalIOErrorInFunction(dict)
                        << "To top storey is beyond the patch point bound!"
                        << exit(FatalIOError);
                }

                storeyPoints_= storeyDir_*storeyHeights_;
            }

            // allocate storage for forces and moments
            forAll(force_, i)
            {
                force_[i].setSize(nStorey_);
                storeyForce_[i].setSize(nStorey_);
                moment_[i].setSize(nStorey_);
            }
        }
    }

    if (nStorey_ == 1)
    {
        // allocate storage for storeyForces and moments
        force_[0].setSize(1);
        force_[1].setSize(1);
        force_[2].setSize(1);
        storeyForce_[0].setSize(1);
        storeyForce_[1].setSize(1);
        storeyForce_[2].setSize(1);
        moment_[0].setSize(1);
        moment_[1].setSize(1);
        moment_[2].setSize(1);
    }

    return true;
}

void Foam::functionObjects::storeyForces::calcStoreyForcesMoments()
{
    initialise();

    force_[0] = Zero;
    force_[1] = Zero;
    force_[2] = Zero;

    storeyForce_[0] = Zero;
    storeyForce_[1] = Zero;
    storeyForce_[2] = Zero;

    moment_[0] = Zero;
    moment_[1] = Zero;
    moment_[2] = Zero;

    if (directForceDensity_)
    {
        const volVectorField& fD = obr_.lookupObject<volVectorField>(fDName_);

        const surfaceVectorField::Boundary& Sfb =
            mesh_.Sf().boundaryField();

        forAllConstIter(labelHashSet, patchSet_, iter)
        {
            label patchi = iter.key();

            vectorField Md
            (
                mesh_.C().boundaryField()[patchi] - coordSys_.origin()
            );

            scalarField sA(mag(Sfb[patchi]));
```

```
            // Normal force = surfaceUnitNormal*(surfaceNormal & forceDensity)
            vectorField fN
            (
                Sfb[patchi]/sA
               *(
                    Sfb[patchi] & fD.boundaryField()[patchi]
                )
            );

            // Tangential force (total force minus normal fN)
            vectorField fT(sA*fD.boundaryField()[patchi] - fN);

            //- Porous force
            vectorField fP(Md.size(), Zero);

            calcStoreyForces(Md, fN, fT, fP, mesh_.C().boundaryField()[patchi]);
        }
    }
    else
    {
        const volScalarField& p = obr_.lookupObject<volScalarField>(pName_);

        const surfaceVectorField::Boundary& Sfb =
            mesh_.Sf().boundaryField();

        tmp<volSymmTensorField> tdevTau = devTau();
        const volSymmTensorField::Boundary& devTaub
            = tdevTau().boundaryField();

        // Scale pRef by density for incompressible simulations
        scalar pRef = pRef_/rho(p);

        forAllConstIter(labelHashSet, patchSet_, iter)
        {
            label patchi = iter.key();

            vectorField Md
            (
                mesh_.C().boundaryField()[patchi] - coordSys_.origin()
            );

            vectorField fN
            (
                rho(p)*Sfb[patchi]*(p.boundaryField()[patchi] - pRef)
            );

            vectorField fT(Sfb[patchi] & devTaub[patchi]);

            vectorField fP(Md.size(), Zero);

            calcStoreyForces(Md, fN, fT, fP, mesh_.C().boundaryField()[patchi]);
        }
    }

    if (porosity_)
    {
        const volVectorField& U = obr_.lookupObject<volVectorField>(UName_);
        const volScalarField rho(this->rho());
        const volScalarField mu(this->mu());

        const HashTable<const porosityModel*> models =
            obr_.lookupClass<porosityModel>();

        if (models.empty())
        {
            WarningInFunction
                << "Porosity effects requested, but no porosity models found "
                << "in the database"
                << endl;
        }

        forAllConstIter(HashTable<const porosityModel*>, models, iter)
        {
```

```
            // non-const access required if mesh is changing
            porosityModel& pm = const_cast<porosityModel&>(*iter());

            vectorField fPTot(pm.force(U, rho, mu));

            const labelList& cellZoneIDs = pm.cellZoneIDs();

            forAll(cellZoneIDs, i)
            {
                label zoneI = cellZoneIDs[i];
                const cellZone& cZone = mesh_.cellZones()[zoneI];

                const vectorField d(mesh_.C(), cZone);
                const vectorField fP(fPTot, cZone);
                const vectorField Md(d - coordSys_.origin());

                const vectorField fDummy(Md.size(), Zero);

                calcStoreyForces(Md, fDummy, fDummy, fP, d);
            }
        }
    }

    Pstream::listCombineGather(force_, plusEqOp<vectorField>());
    Pstream::listCombineGather(moment_, plusEqOp<vectorField>());
    Pstream::listCombineScatter(force_);
    Pstream::listCombineScatter(moment_);

    forAll(force_, i)
    {
        storeyForce_[i] = force_[i];

        forAll(force_[i], j)
        {
            //Store the storey forces(Fx, Fy, Mz)
            storeyForce_[i][j].z() = moment_[i][j] & storeyDir_;
        }
    }
}

//I added this, do not exist before
Foam::vectorField Foam::functionObjects::storeyForces::storeyForcesEff() const
{
    //You may need only pressure forces
    // Everything: pressure + viscous + porous
    return storeyForce_[0] + storeyForce_[1] + storeyForce_[2];
}

Foam::vector Foam::functionObjects::storeyForces::forceEff() const
{
    return sum(force_[0]) + sum(force_[1]) + sum(force_[2]);
}


Foam::vector Foam::functionObjects::storeyForces::momentEff() const
{
    return sum(moment_[0]) + sum(moment_[1]) + sum(moment_[2]);
}


bool Foam::functionObjects::storeyForces::execute()
{
    return true;
}


bool Foam::functionObjects::storeyForces::write()
{
    calcStoreyForcesMoments();

    if (Pstream::master())
    {
        logFiles::write();
```

```
        writeForces();

        writeStoreyForces();

        Log << endl;
    }

    return true;
}


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

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

Listing D.2: Selected source code of `windFSI` framework for the fluid subsystem

D.3 Dynamic mesh subsystem

```
/*
Class
    Foam::structuralMotionMeshSolver

Description
    Structurally deformable-body mesh motion solver for fvMesh. Applies SLERP interpolation of movement as a function of
         distance to the object surface.

SourceFiles
    structuralMotionMeshSolver.C

*/

namespace Foam
{
    namespace FSI
    {
        class structuralMotionMeshSolver
        :
            public displacementMotionSolver,
            public structuralMotion
        {
            private:
                wordReList patches_;
                const labelHashSet patchSet_;
                const scalar di_;
                const scalar do_;
                Switch test_;
                Switch twoWayCoupled_;
                scalar rhoInf_;
                word rhoName_;
                pointScalarField scale_;
                label curTimeIndex_;
                mutable volVectorField cellDisplacement_;

            public:
                TypeName("structuralMotionMeshSolver");
                structuralMotionMeshSolver(const polyMesh&, const dictionary& dict);
                structuralMotionMeshSolver(const structuralMotionMeshSolver&);
                ~structuralMotionMeshSolver();
                const structuralMotion& motion() const;
                virtual tmp<pointField> curPoints() const;
                virtual void solve();
                virtual bool write() const;
                void operator=(const structuralMotionMeshSolver&) = delete;
        };
    }
```

```
}

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

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

Source file: structuralMotionMeshSolver.C

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

#include "structuralMotionMeshSolver.H"
#include "pointVolInterpolation.H"
#include "addToRunTimeSelectionTable.H"
#include "polyMesh.H"
#include "vectorField.H"
#include "pointPatchDist.H"
#include "pointConstraints.H"
#include "uniformDimensionedFields.H"
#include "storeyForces.H"
#include "mathematicalConstants.H"

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

namespace Foam
{
    namespace FSI
    {
        defineTypeNameAndDebug(structuralMotionMeshSolver, 0);

        addToRunTimeSelectionTable
        (
            motionSolver,
            structuralMotionMeshSolver,
            dictionary
        );
    }
}


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

Foam::FSI::structuralMotionMeshSolver::structuralMotionMeshSolver
(
    const polyMesh& mesh,
    const dictionary& dict
)
:
    displacementMotionSolver(mesh, dict, typeName),
    structuralMotion
    (
        coeffDict(),
        IOobject
        (
            "structuralMotionState",
            mesh.time().timeName(),
            "uniform",
            mesh
        ).typeHeaderOk<IOdictionary>(true)
      ? IOdictionary
        (
            IOobject
            (
                "structuralMotionState",
                mesh.time().timeName(),
                "uniform",
                mesh,
                IOobject::READ_IF_PRESENT,
                IOobject::NO_WRITE,
                false
            )
        )
      : coeffDict() //Otherwise use initial disp from dynamicDict
    ),
    patches_(wordReList(coeffDict().lookup("patches"))),
```

```
    patchSet_(mesh.boundaryMesh().patchSet(patches_)),
    di_(coeffDict().lookup<scalar>("innerDistance")),
    do_(coeffDict().lookup<scalar>("outerDistance")),
    test_(coeffDict().lookupOrDefault<Switch>("test", false)),
    twoWayCoupled_(coeffDict().lookupOrDefault<Switch>("twoWayCoupled", true)),
    rhoInf_(1.0),
    rhoName_(coeffDict().lookupOrDefault<word>("rho", "rho")),
    scale_
    (
        IOobject
        (
            "motionScale",
            mesh.time().timeName(),
            mesh,
            IOobject::NO_READ,
            IOobject::NO_WRITE,
            false
        ),
        pointMesh::New(mesh),
        dimensionedScalar(dimless, 0)
    ),
    curTimeIndex_(-1),
    cellDisplacement_
    (
        IOobject
        (
            "cellDisplacement",
            mesh.time().timeName(),
            mesh,
            IOobject::NO_READ,
            IOobject::NO_WRITE
        ),
        refCast<const fvMesh>(mesh.thisDb()),
        dimensionedVector("cellDisplacement", dimLength, vector::zero),
        pointDisplacement().boundaryField().types()
    )

{
    if (rhoName_ == "rhoInf")
    {
        rhoInf_ = coeffDict().lookup<scalar>("rhoInf");
    }

    // Calculate scaling factor everywhere
    // ------------------------------------

    {
        const pointMesh& pMesh = pointMesh::New(mesh);

        pointPatchDist pDist(pMesh, patchSet_, points0());

        // Scaling: 1 up to di then linear down to 0 at do away from patches
        scale_.primitiveFieldRef() =
            min
            (
                max
                (
                    (do_ - pDist.primitiveField())/(do_ - di_),
                    scalar(0)
                ),
                scalar(1)
            );

        // Convert the scale function to a cosine
        scale_.primitiveFieldRef() =
            min
            (
                max
                (
                    0.5
                  - 0.5
                   *cos(scale_.primitiveField()
                   *Foam::constant::mathematical::pi),
                    scalar(0)
```

```
                ),
                scalar(1)
            );

        pointConstraints::New(pMesh).constrain(scale_);
        scale_.write();
    }
}


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

Foam::FSI::structuralMotionMeshSolver::~structuralMotionMeshSolver()
{}


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

Foam::tmp<Foam::pointField>
Foam::FSI::structuralMotionMeshSolver::curPoints() const
{
    return points0() + pointDisplacement_.primitiveField();
}


void Foam::FSI::structuralMotionMeshSolver::solve()
{
    const Time& t = mesh().time();

    if (mesh().nPoints() != points0().size())
    {
        FatalErrorInFunction
            << "The number of points in the mesh seems to have changed." << endl
            << "In constant/polyMesh there are " << points0().size()
            << " points; in the current mesh there are " << mesh().nPoints()
            << " points." << exit(FatalError);
    }

    // Store the motion state at the beginning of the time-step
    bool firstIter = false;

    if (curTimeIndex_ != t.timeIndex())
    {
        // newTime(); //might be necessary (assigns state0 -> state)
        curTimeIndex_ = t.timeIndex();
        firstIter = true;
    }

    dimensionedVector g("g", dimAcceleration, Zero);

    if (mesh().foundObject<uniformDimensionedVectorField>("g"))
    {
        g = mesh().lookupObject<uniformDimensionedVectorField>("g");
    }
    else if (coeffDict().found("g"))
    {
        coeffDict().lookup("g") >> g;
    }

    // scalar ramp = min(max((t.value() - 5)/10, 0), 1);
    scalar ramp = 1.0;

    if (test_)
    {
        //Test with a lateral force equivalent to self-weight
        // const vector g = coeffDict().lookup<vector>("g");

        update
        (
            firstIter,
            ramp*cmptMultiply(masses(), g.value()),
            t.deltaTValue(),
            t.deltaT0Value()
        );
```

```
    }
    else
    {
        dictionary storeyForcesDict;

        storeyForcesDict.add("type", functionObjects::storeyForces::typeName);
        storeyForcesDict.add("patches", patches_);
        storeyForcesDict.add("rhoInf", rhoInf_);
        storeyForcesDict.add("rho", rhoName_);

        //If it's coupled, the origin shifts with the object
        if (twoWayCoupled_)
        {
            storeyForcesDict.add("CofR", centerOfRotation());
        }
        else
        {
            storeyForcesDict.add("CofR", origin());
        }

        // Info << "Center of rotation: " << centerOfRotation() << endl;
        // Info << "Origin: " << origin() << endl;

        dictionary storeySubDict;
        storeySubDict.add("nStorey", nStorey());
        storeySubDict.add("direction", strDir());
        storeySubDict.add("cumulative", "no");

        //Add the storey subdict
        storeyForcesDict.add("storeyData", storeySubDict);


        functionObjects::storeyForces f("storeyForces", t, storeyForcesDict);

        f.calcStoreyForcesMoments();

        update
        (
            firstIter,
            ramp*(f.storeyForcesEff() + cmptMultiply(masses(), g.value())),
            t.deltaTValue(),
            t.deltaT0Value()
        );
    }

    if(twoWayCoupled_)
    {
        // Update the displacements
        pointDisplacement_.primitiveFieldRef() =
            transform(points0(), scale_) - points0();

        // Displacement has changed. Update boundary conditions
        pointConstraints::New
        (
            pointDisplacement_.mesh()
        ).constrainDisplacement(pointDisplacement_);
    }

    // Interpolate cellDisplacement from pointDisplacement
    const pointMesh& pMsh = pointMesh::New(mesh());
    const fvMesh& fvMsh = refCast<const fvMesh>(mesh().thisDb());

    pointVolInterpolation pvi(pMsh, fvMsh);

    cellDisplacement_ = pvi.interpolate(pointDisplacement_);
}


bool Foam::FSI::structuralMotionMeshSolver::write() const
{
    IOdictionary dict
    (
        IOobject
        (
```

```
            "structuralMotionState",
            mesh().time().timeName(),
            "uniform",
            mesh(),
            IOobject::NO_READ,
            IOobject::NO_WRITE,
            false
        )
    );

    state().write(dict);

    return
        dict.regIOobject::writeObject
        (
            IOstream::ASCII,
            IOstream::currentVersion,
            mesh().time().writeCompression(),
            true
        )
     && displacementMotionSolver::write();
}
```

Listing D.3: Selected source code of `windFSI` framework for the moving mesh subsystem

Appendix E

Usage of the windFSI framework

Since the `pimpleFsiFoam` solver is developed as a standard OpenFOAM application, it offers all the functionalities of a typical transient solver, including parallel execution. However, here we focus on input parameters specific to the FSI framework. Most of the input parameters are specified in `constant/dynamicMeshDict`.

```
/*--------------------------------*- C++ -*----------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     | Website:  https://openfoam.org
    \\  /    A nd           | Version:  8
     \\/     M anipulation  |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      dynamicMeshDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

dynamicFvMesh       dynamicMotionSolverFvMesh;
motionSolverLibs    ("libwindFSI.so");
motionSolver        structuralMotionMeshSolver;

patches             (CAARC);
origin              (0 0 0);
innerDistance       45.0;
outerDistance       180.0;

```

```
26  rho                 rhoInf;
27  rhoInf              1.225;
28  report              on;
29  twoWayCoupled       on;
30
31  structuralSolver
32  {
33      type                NewmarkBeta;
34      beta                0.25;
35      gamma               0.5;
36      relaxationFactor    1.0;
37  }
38
39  structuralProperties
40  {
41      dataDir         "constant/structuralData"
42      nStorey         60;
43      nMode           6;
44      nModeUsed       6;
45      orientation     (1 0 0 0 1 0 0 0 1);
46      freqUnit        Hz;
47  }
48
49  fsiControl
50  {
51      couplingAlgorithm       CSS;
52      tolerance               1e-7;
53      referenceDisplacement   45.0;
54      maxFsiIteration         5;
55  }
```

Listing E.1: Sample definition of `dynamicMeshDict` dictionary for aeroelastic modeling of tall buildings

Listing E.1 shows a sample `dynamicMeshDict` file used to set up an aeroelastic simulation of a tall building. The compiled FSI library is loaded as a shared object library specifying `"libwindFSI.so"`. Then, for the mesh motion solver, we select `structuralMotionMeshSolver`. Parameters related to the structural solver are defined in `structuralSolver` sub-dictionary. The modal properties of the structure, such as the elevation of each storey, mass distribution, natural frequencies, damping ratios, and mode shape vectors, are read from text files located in `constant/structuralData` directory. Finally, parameters that control the FSI iterations, including the type of coupling algorithm, error tolerance, and maximum FSI iteration, are

specified in `fsiControl` sub-dictionary given in Listing E.1. Regarding the fluid solver, all parameters used to control `pimpleFoam` solver are equally applicable for the FSI simulation too. Finally, once the input parameters are specified, the FSI simulation can be run in parallel by calling:

```
mpirun -np <nProc> pimpleFsiFoam -parallel
```

Curriculum Vitae

Name:	Abiy Fantaye Melaku
Post-Secondary Education and Year:	University of Western Ontario London, Canada 2016 - 2022
Degrees:	**Ph.D in Wind Engineering and Scientific Computing**
	Chungbuk National University Cheongju-si, South Korea 2014 - 2016 **M.Sc in Computational Structural Engineering**
	Adama Science and Technology University Adama, Ethiopia 2008 - 2013 **B.Sc in Civil Engineering**
Honours and Awards:	Ontario Trillium Scholarship 2016-2020
Related Work Experience:	Teaching Assistant The University of Western Ontario 2016-2022
	Research Intern FM Global Research, Norwood, Massachusetts, USA Sept. 2017 - Dec. 2017

Publications:

Journal papers published:

1. **Melaku, A. F.** & Bitsuamlak, G. T. (2021). A divergence-free inflow turbulence generator using spectral representation method for large-eddy simulation of ABL flows. *Journal of Wind Engineering and Industrial Aerodynamics*.

2. **Melaku, A. F.** , Doddipatla L. S. & Bitsuamlak, G. T. (2022). Large-eddy simulation of wind loads on a roof-mounted cube: application for interpolation of experimental aerodynamic data. *Journal of Wind Engineering and Industrial Aerodynamics*.

3. **Melaku, A. F.** & Jung, K. S. (2017). Evaluation of welded joints of vertical stiffener to web under fatigue load by hotspot stress method. *International Journal of Steel Structures*.

4. **Melaku, A. F.**, Geleta, T. N. & Jung, K. S. (2015). Application of object-oriented finite element method in structural mechanics. *Journal of the Institute of Construction Technology*.

Journal papers under review:

1. **Melaku, A. F.**, & Bitsuamlak, G. T. LES for predicting wind loads and responses: prospect for wind-resistant tall building design. *Journal of Wind Engineering and Industrial Aerodynamics*.

2. **Melaku, A. F.** & Bitsuamlak, G. T. Computationally efficient simulation of multivariate wind velocity field using a low-rank representation of the cross-power spectral density matrix. *Journal of Engineering Mechanics*.

Journal papers in preparation:

1. **Melaku, A. F.**, Jansse J. & Bitsuamlak, G. T. Wall-modeled large-eddy simulation for evaluating wind loads on a tall building located in a city center: comparison with experimental data. **In preparation** for *Engineering Structures*.

2. **Melaku, A. F.** & Bitsuamlak, G. T. windFSI: An open-source fluid-structure interaction framework for aeroelastic modeling of flexible structures. **In preparation** for *Advances in Engineering Software*.

Conference publications and presentations:

1. **Melaku, A. F.**, & Bitsuamlak G. T. (2022). A high-fidelity fluid-structure interaction framework for computational aeroelastic modeling of flexible structures. In *2022 SimCenter Symposium*, Texas Advanced Computing Center(TACC), The University of Texas at Austin, Texas, USA.

2. **Melaku, A. F.**, Janssen J., McDonald P., Sundholm N. & Bitsuamlak G. T. (2022). Large-eddy simulation of wind loads on a tall building located in a city center: comparison with experimental data In *Engineering Mechanics Institute Conference 2022*, Johns Hopkins University, Baltimore, Maryland, USA.

3. **Melaku, A. F.** & Bitsuamlak G. T. (2022). Predicting the dynamic response of a tall building using large-eddy simulation and time-domain analysis In *The 14th Americas Conference on Wind Engineering*, Texas Tech University , Lubbock, Texas, USA.

4. **Melaku, A. F.** & Bitsuamlak G. T. (2022). Computationally efficient large-scale wind velocity field simulation using Nyström based spectral decomposition In *The 14th Americas Conference on Wind Engineering*, Texas Tech University , Lubbock, Texas, USA.

5. Geleta, T. N., **Melaku, A. F.**,Doddipatla, L. S. & Bitsuamlak G. T. (2022). Comparison of surface pressure and near-surface flow field of TTU building between LES and PIV measurements In *The 14th Americas Conference on Wind Engineering*, Texas Tech University , Lubbock, Texas, USA.

6. **Melaku, A. F.**, Doddipatla, L. S. & Bitsuamlak G. T. (2021). Large-eddy Simulation of Wind Loads on a Roof-mounted Cube: A Means to Interpolate Experimental Data In *The 6th American Association for Wind Engineering Workshop*, Clemson University, Clemson, SC, USA.

7. Geleta, T. N., Elshaer, A., **Melaku, A. F.** & Bitsuamlak, G. T. (2018). Computational wind load evaluation of low-rise buildings with complex roofs using LES. In *The 7th International Symposium on Computational Wind Engineering 2018.*, Seoul, Republic of Korea.

8. **Melaku, A. F.**, Bitsuamlak, G. T., Elshaer & A., Aboshosha, H. (2017). Synthetic inflow turbulence generation methods for LES study of tall building aerodynamics. In *The 13th Americas Conference on Wind Engineering (13ACWE)*, Gainesville, Florida, USA.

9. Adamek K., **Melaku, A. F.**, Bitsuamlak, G. T. & Sadeghpour, F. (2017). Wind safety assessment during high rise building construction. In *Annual Conference of the Canadian Society for Civil Engineering (CSCE)*, Vancouver, BC, Canada.